3D Optical Microscopy Method Based on Plane-Sweep Reconstruction

April 29, 2013

by

Max Liberman

Submitted in partial fulfillment of the requirements of the degree of Bachelor of Science with Honors in the School of Engineering at Brown University.

Prepared under the direction of Prof. Gabriel Taubin, Advisor

Advisor's Signature

Honors Chair's Signature

Reader's Signature

3D Optical Microscopy Method Based On Plane-Sweep Reconstruction

Max Liberman
Advisor: Professor Gabriel Taubin

April 29, 2013

Abstract

We present the design and implementation of a nondestructive method for 3D optical microscopy based on a simple active lighting strategy. In our 3D scanning system, a custom-designed optical projector is computer-controlled to sweep a plane of light across the scanned object. A high-resolution camera outfitted with an industrial inspection lens is synchronized with the sweeping of the active lighting system to capture its motion across the breadth of the object. The sequence of images is used to generate a colored 3D point cloud representation of the surface of the object. We further present a software interface used to efficiently capture and process data with our scanning system.

1 Introduction

The principles of three-dimensional reconstruction using active lighting have been applied in many fields to compute accurate spatial information for objects of interest. Transforming real-world objects into precise computer models can be an extremely valuable tool, from analyzing ancient artifacts to reverse engineering complex structures. Much work has been done to develop low-cost and accurate methods of 3D reconstruction using equipment readily available to the general public. However, little has been done to apply these methods to small-scale objects not easily captured by standard cameras.

Several imaging techniques have been used to generate 3D reconstructions of small objects down to the microscopic scale [1, 2, 3, 4, 5]. However, the majority of these methods require exorbitantly priced equipment that make it an unrealistic investment for most 3D scanning applications in biological, biomedical, or bioengineering research. Moreover, conventional microscopy methods such as confocal microscopy are only useful for thinly sliced sections of tissue, and are thus completely destructive to the subject of the 3D reconstruction.

We propose a nondestructive method for 3D optical microscopy based on a simple active lighting strategy. In our proposed method, a custom-designed optical projector is computer-controlled to sweep a plane of light across the object. An industrial inspection lens is applied to a high-resolution camera, which is synchronized with the projector motion to capture the light plane's progression across the breadth of the object. The sequence of images is used to generate a colored 3D point cloud representation of the scanned object. We further present a software interface used to efficiently capture and process data with our proposed system.

2 Theory

2.1 Pinhole Camera Model

The pinhole camera model is a mathematical approximation used to describe the projection of a point in 3-D space onto the 2-D image plane of a camera. A single ray of light is emitted by a visible point with a world position of $p = [\,x, y, z\,]^T$. It travels through the camera aperture, approximated as a point (or

pinhole), and intersects the image plane at pixel coordinates $p' = [\,x', y', 1\,]^T$. The relationship between p and its projection onto the image plane is described by

$$\lambda p' = K(Rp + T), \tag{1}$$

where λ is an arbitrary positive scalar, K is a 3×3 matrix describing the intrinsic parameters of the camera, and $[\,R, T\,]$ are the rotation and translation matrices describing the transformation from world coordinates to camera coordinates [6].

We can rearrange this equation to match the definition of a ray:

$$p = (-R^T T) + \lambda(R^T K^{-1} p') = u + \lambda v. \tag{2}$$

This form makes it clear that p lies on a ray that originates at point $u = -R^T T$ and points in the direction described by the vector $v = R^T K^{-1} p'$. The only unknown is the scaling factor λ which represents the distance along vector v at which p is located. The essence of any 3-D reconstruction method is to accurately determine this scaling factor so as to calculate the position of p in world coordinates from its projection on the image plane.

2.2 Ray-Plane Intersection

Suppose we orient a camera to capture images of an object. A light source is used to project a straight line along its optical axis, creating a plane of light that slices through the scene (Figure 1). This plane

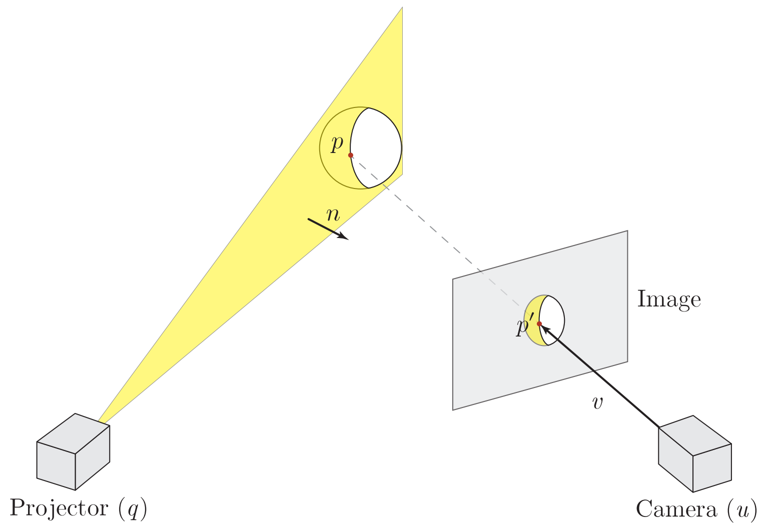


Figure 1: Reconstruction by ray-plane intersection.

is implicitly defined as

$$P = \left\{ p : n^T (p - q) = 0 \right\} \tag{3}$$

where n is the normal vector of the plane, and q is some reference point on the plane. The intersection of this projected plane of light and the surface of our object is a curve, and a portion of this curve is captured by our camera in an image.

Let's examine one such point that lies on the intersection. Because we know that this point p not only lies along the ray defined in Equation (2), but also on the plane defined in Equation (3), the following conclusion results:

$$n^T (u + \lambda v - q) = 0 \Rightarrow \lambda = \frac{n^T (q - u)}{n^T v}. \tag{4}$$

Using the value for λ calculated in Equation (4), we simply evaluate Equation (2) to compute the position of p in world coordinates. We then repeat this procedure for the remaining points on the curve captured in the image. If the light plane is slowly swept across the span of the object and imaged at each step, a high-resolution 3-D point cloud can be generated for the camera-facing portion of the object.

2.3 Calibration

We've identified how to extract depth information, or 3-D position, from an image using ray-plane intersection, but thus far we've assumed that the parameters that describe both the pinhole model and the projected light plane have been given. In the implementation of this system, we will have to calculate these parameters ourselves before we can proceed to 3-D reconstruction.

Camera Calibration. Much work has been done in the field of computer vision on methods of camera calibration. Zhang developed a technique for intrinsic and extrinsic camera calibration from several images of a planar geometric pattern [7]. The methods described by Zhang can be used with simple checkerboard calibration targets to compute approximations for K, R, and T for the camera, allowing us to calculate u and v in Equation (4).

Light Plane Calibration. Commonly used methods of swept-plane reconstruction use orthogonal reference planes in the background of the captured images to calibrate the light plane normal vector for each frame [8]. Due to the depth-of-focus limitations that arise when imaging at the millimeter scale, a new method for light plane calibration was necessary.

Figure 2 shows the optics of a light plane projector with a lens of focal length f. Light rays travel through a quarter-circle-shaped hole of radius h_s and are projected onto a plane orthogonal to the optical axis with a magnified radius of h_p. If the optical axis is tilted by an angle θ with respect to the

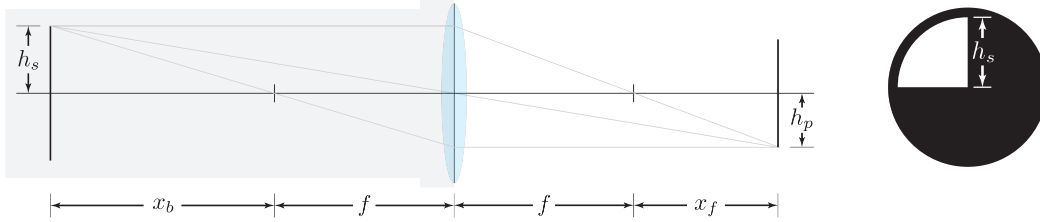


Figure 2: Left: Optical parameters of the light plane projector. Right: Cross-sectional view of the projector slide (white indicates absence of material).

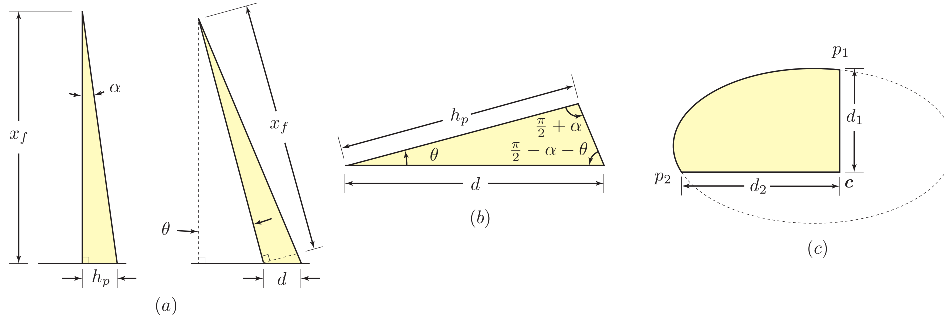


Figure 3: (a) The geometry of the projected slide with and without angular offset. (b) A detail view of the geometry of the distorted slide length. (c) The projected quarter-circle slide pattern, as seen by the camera.

plane's normal, as seen in Figure 3(a), the projected length of h_s is distorted to a new length d. Simple trigonometric analysis of the angles shown in Figure 3(b) allows us to calculate this distorted length as a function of the angle of rotation:

$$d = \frac{\cos(\alpha)}{\cos(\alpha+\theta)}\, h_p, \text{ where } \alpha = \arctan\left[\frac{h_s}{f}\right]. \tag{5}$$

We design the slide and its distance from the lens to yield an appropriate scale on the projected length h_p. From Equation (5), we can express θ in terms of these design parameters and the measured projection length d:

$$\theta = \arccos\left[\frac{h_p \cos\alpha}{d}\right] - \alpha = \arccos\left[\frac{h_p\, h_s}{d\sqrt{f^2+h_s^2}}\right] - \arctan\left[\frac{h_s}{f}\right]. \tag{6}$$

We project the quarter-circle slide pattern onto a planar surface parallel to the camera's image plane, which we use as a reference plane described by $z = 0$ in world coordinates. The captured image is similar in geometry to Figure 3(c). Assuming the camera has been intrinsically and extrinsically calibrated, we can calculate the positions in world coordinates of the three corners, p_1, p_2, and c. Using the measured values $d_1 = \|p_1 - c\|$ and $d_2 = \|p_2 - c\|$, we can use Equation (6) to compute the corresponding rotation angles θ_2 and θ_1, respectively. Let us define unit vectors for the orthogonal axes whose origin is at c:

$$\hat{\mathrm{e}}_1 = \frac{p_1 - c}{d_1} \tag{7a}$$

$$\hat{\mathrm{e}}_2 = \frac{p_2 - c}{d_2} \tag{7b}$$

$$\hat{\mathrm{e}}_3 = \hat{\mathrm{e}}_1 \times \hat{\mathrm{e}}_2 \tag{7c}$$

Let $\hat{\mathrm{v}}_1$ be the rotation of $\hat{\mathrm{e}}_1$ about $\hat{\mathrm{e}}_2$ by angle θ_2. Likewise, let $\hat{\mathrm{v}}_2$ be the rotation of $\hat{\mathrm{e}}_2$ about $\hat{\mathrm{e}}_1$ by angle θ_1. These rotated unit vectors can be expressed as:

$$\hat{\mathrm{v}}_1 = \cos(\theta_2)\,\hat{\mathrm{e}}_1 + \sin(\theta_2)\,\hat{\mathrm{e}}_3 \tag{8a}$$

$$\hat{\mathrm{v}}_2 = \cos(\theta_1)\,\hat{\mathrm{e}}_2 + \sin(\theta_1)\,\hat{\mathrm{e}}_3 \tag{8b}$$

The unit vector orthogonal to $\hat{\mathrm{v}}_1$ and $\hat{\mathrm{v}}_2$ is coaxial with the optical axis of the projector, so the position of the focal point shown in Figure 3(a) can be expressed in world coordinates:

$$F = c + x_f(\hat{\mathrm{v}}_1 \times \hat{\mathrm{v}}_2) = c + T_F. \tag{9}$$

The quarter-circle edge parallel to $\hat{\mathrm{e}}_1$ will be used as the intersecting light plane of Figure 1. Thus from Equation (9) we have three points on the light plane, which we can use to calculate the normal vector:

$$n = (F - c) \times (p_1 - c). \tag{10}$$

By taking c as the light plane's reference point q, we have the remaining two variables in Equation (4) to solve for λ and reconstruct the world position of p from its image coordinate p' using Equation (2).

3 Plane-Sweep Reconstruction Algorithm

The reconstruction process proceeds similarly to other plane-sweep reconstruction methods (see [8]). Assuming the camera has been properly calibrated (see [9, 7]), we take K, R, and T as given, and proceed to set up the object in the field of view of the camera-projector system. A plane of light is swept across the object while images are synchronously captured; the faster the images are captured, the denser the resulting 3D reconstruction will be. For a captured sequence of M grayscale images, each of dimensions $w \times h$ pixels, we begin by computing the maximum and minimum intensities of each pixel's M values, stored as two grayscale images I^{max} and I^{min}, respectively. This function is shown in Algorithm 1.

At this point we apply a contrast constraint to our pixel set. Any areas of the images in shadow will not change significantly in pixel intensity during the image sequence. Likewise, if the target object

Algorithm 1 Pixel Maximum & Minimum Intensities

```
I^max ← 0
I^min ← inf
for m = 1 → M do
    I ← image m
    for i, j = 1 → w, h do
        if ( I[i, j] > I^max[i, j] ) then
            I^max[i, j] ← I[i, j]
        else if ( I[i, j] < I^min[i, j] ) then
            I^min[i, j] ← I[i, j]
        end if
    end for
end for
```

has been placed on a dark, non-reflective surface, this background should exhibit less intensity variation than the object itself. To reduce both the computational load and the noise of the reconstruction, we disregard pixels whose values of I^{min} and I^{max} differ by less than some minimum contrast value, I^*. We carry this information through the rest of the reconstruction process in the form of a matrix of logical values C. The creation of this pixel mask is detailed in Algorithm 2.

Algorithm 2 Pixel Contrast Constraint

```
C ← true
for i, j = 1 → w, h do
    if ( I^max[i, j] − I^min[i, j] < I* ) then
        C[i, j] ← false
    end if
end for
```

We then define the per-pixel shadow threshold, S, as the per-pixel average of I^{min} and I^{max}. We assume that a pixel is in shadow until it reaches this value, at which point it is directly illuminated by the projector light plane. A loop iterates through the M images, keeping track of those pixels that have reached their threshold in a matrix of logical values, B. If a pixel intensity $I[\,i, j\,]$ is detected as being greater than or equal to its shadow threshold at frame m and $B[\,i, j\,]$ is false, then the pixel intensity must have reached its shadow threshold at some time on the interval $(\,m-1, m\,]$. This value is interpolated to sub-frame resolution, and stored in the matrix T. Algorithm 3 details this step of the reconstruction process. Note that the contrast constraint matrix C is used to only evaluate pixels that meet the minimum contrast requirement.

Now each pixel has been assigned a time, in units of frames, at which it is being intersected by our projector light plane (refer back to Figure 1). If we can describe the behavior of the projector light plane over time, we can compute the parameters of the light plane in Equation (3) from the values stored in T. This is exactly how we proceed. First, we assume that the active lighting system is subject only to linear translation as it sweeps across the object (this assumption will hold based on the design); thus, the normal vector of its projected light plane does not change over the course of an image sequence—only its reference point q is time-dependent. So we can estimate the light plane normal vector using the mathematics of Section 2.3. Assuming the points c, p_1, and p_2 have been identified and transformed to world coordinates, Equations (7) to (10) can be evaluated sequentially to compute the normal of the light plane, n.

Now assume we've identified the point c of the quarter-circle pattern in a subset of our image frames. We've limited the movement of the active lighting system to linear translation. As a result, we can compute a linear equation to describe the position of the corner c as a function of frame t, on the interval $[\,1, N\,]$. This can be achieved with simple linear regression, and we'll call the resulting function $corner(t)$.

We now have all the tools we need to evaluate Equation (4) and reconstruct a 3D position from any recoverable pixel's coordinates, $P'[\,i, j\,]$. For every pixel for which $C[\,i, j\,]$ is true, we use the position

Algorithm 3 Pixel Shadow Threshold Crossings

```
for i, j = 1 → w, h do
    S[i, j] ← 1/2 ( I^min[i, j] + I^max[i, j] )
end for
B ← false
for m = 2 → M do
    I ← image m
    for ( i, j = 1 → w, h s.t. C[i, j] is true ) do
        if ( I[i, j] ≥ S[i, j] and B[i, j] is false ) then
            T[i, j] ← t  s.t. I^t[i, j] = S[i, j],  t ∈ (m − 1, m]
            B[i, j] ← true
        end if
    end for
end for
```

of our light plane's reference point computed from $corner(t)$ along with our calibrated values for n, K, R, and T, to solve Equation (4). Using the result, we evaluate Equation (2) to reconstruct a position in world coordinates, $P[i,j]$, for each recoverable pixel. Algorithm 4 details this final portion of the reconstruction process.

Algorithm 4 3D Pixel Reconstruction

```
u ← −R^T T
for ( i, j = 1 → w, h  s.t. C[i, j] is true ) do
    t ← T[i, j]
    c ← corner(t)
    l ← n · (c − u)
    v ← R^T K^{−1} P′[i, j]
    l ← l / (n · v)
    P[i, j] ← u + l ∗ v
end for
```

Finally, define N as the number of true values in C. N is generally much smaller than the total number of pixels; thus, the result P is more efficiently stored in a $3 \times N$ matrix. We refer to this matrix of recovered positions as the point cloud.

4 Hardware

The 3D scanning system was assembled from a combination of mechanical, electrical, and optical products combined with custom-designed parts. The drawings for any custom parts, as well as the assembly diagrams, are included in Appendix A. These drawings were used by the Brown University Joint Engineering Physics Instrument Shop to manufacture the parts. The scanning system can be functionally subdivided into three modules: the imaging system, the active lighting system, and the staging system. The hardware components of each of these modules will now be discussed in detail.

4.1 Imaging Components

For our imaging system, we purchased an industrial inspection lens with an adjustable magnification of 0.7-4.5X and minimal distortion. This large magnifying lens was secured with a machined aluminum collar into an AmScope microscope stand, which can be manually adjusted with focusing knobs. Onto the standard C-mount attachment of the lens we mounted a Point Grey Research Flea®2 Firewire camera with a maximum resolution of 1024×768 pixels captured at 15 frames per second. Figure 4 shows a diagram of the imaging system setup.

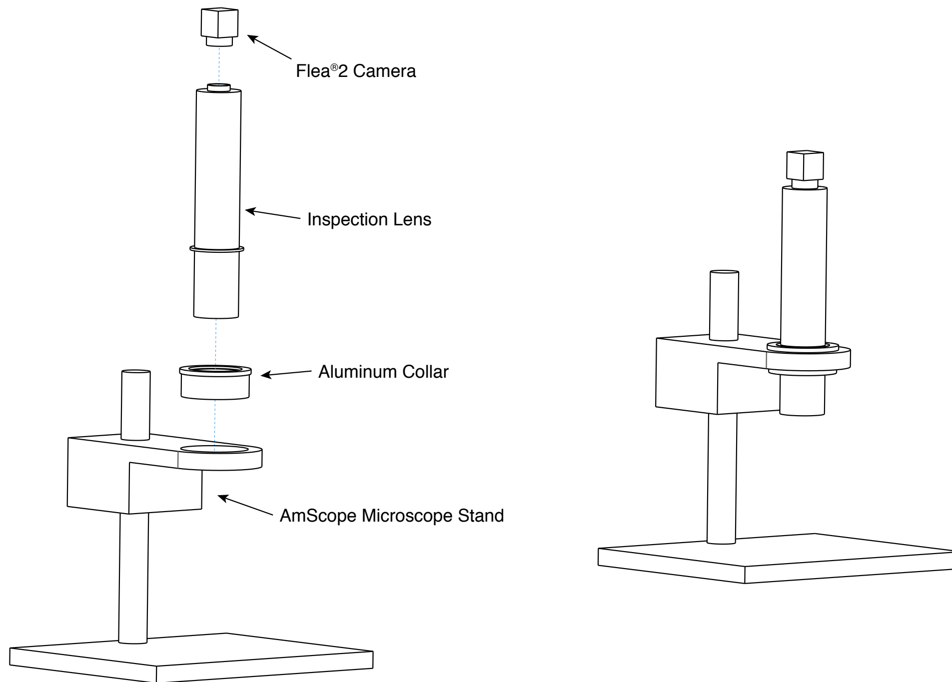


Figure 4: A schematic of the microscopic imaging system, in an exploded (*left*) and assembled (*right*) form.

4.2 Active Lighting Components

We designed a low-cost slide projector (Figure 5) based around an off-the-shelf MagLite® flashlight. The filament of the flashlight is positioned at the focal point of a 9.0-mm collimating lens with a focal length of 12.0 mm, which reorients the light rays such that they are parallel to the optical axis. A circular aluminum slide with the quarter-circle pattern cut out of its cross-section creates a hard edge in the collimated light. The focusing lens, with a focal length of 50 mm, is separated from the slide by a threaded extension tube, whose length determines the magnification of the projected slide pattern.

In order to accomplish the sweeping of the light plane the slide projector was mounted on a low-friction THK linear translation stage. A small Haydon™ linear actuator, or stepper motor, was attached to the sliding stage to drive its translation an axis parallel to the camera's image plane, with a step size of 0.001 inches. Once the orientation of the projector was tuned such that the straight edge of the light pattern was focused on the object, the projector would simply travel along the stage's line of motion, sweeping the shadow plane over the breadth of the scanned object. Figure 6 shows a diagram of the complete active lighting system: slide projector, translation stage, and stepper motor. Appendix A.2 contains the engineering drawings for the custom-made aluminum components of the active lighting system.

For precise and repeatable translation of the projector, computer control was integrated into the design with the addition of two components: an Arduino Uno microcontroller board and an EasyDriver

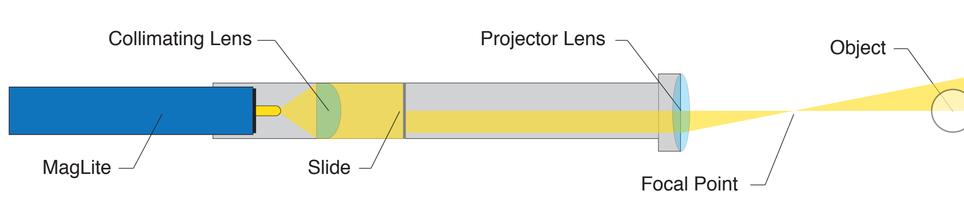


Figure 5: Schematic of the optical components of the custom slide projector.

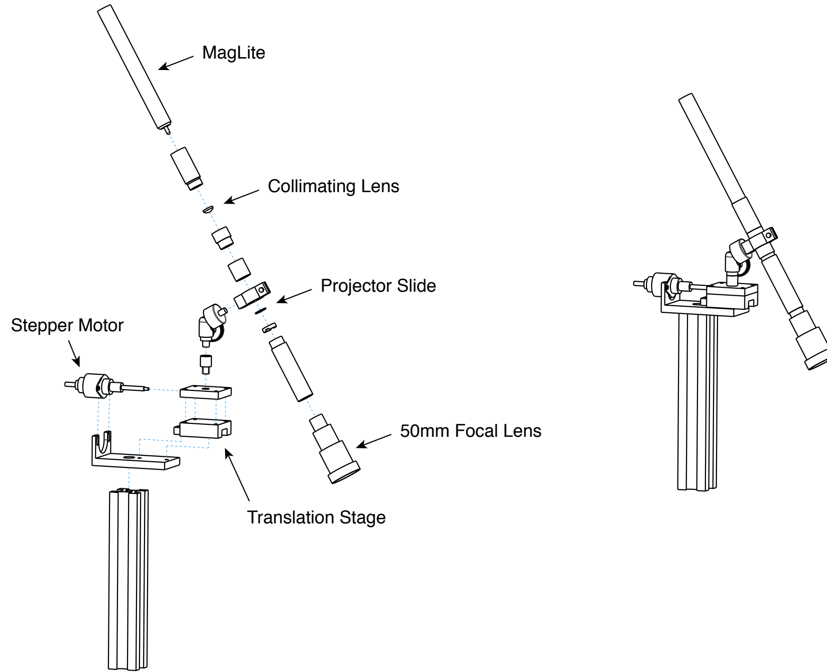


Figure 6: A schematic of the complete active lighting system, in exploded (*left*) and assembled (*right*) form.

chip.

The Arduino Uno is a microcontroller board with both analog and digital input and output. Its processor is programmable with the downloadable Arduino software that loads code based on the Processing programming language into the Arduino Uno's memory [10]. The Arduino board can be connected to a computer via USB, and uses the standard USB COM drivers such that communication with the Arduino from a host computer is simple to initiate.

The EasyDriver chip is a stepper motor driver powered by a 12V wall adapter that provides microresolution to bipolar stepper motors [11], allowing 8 microsteps per nominal 0.001-inch step of the Haydon$^{\mathrm{TM}}$ stepper. The four wires of motor are soldered to its output leads. It has two digital inputs, referred to as STEP and DIR. The STEP input reacts to the rising edge of its digital signal, and causes the stepper motor to move one step. The DIR signal controls the direction of the stepping, and remains high for forward, and low for backward.

The Arduino is used as the interface between the EasyDriver chip and the scanning software on the host computer (see Figure 7). Commands are sent via serial communication through a COM port on the host computer, and interpreted by the programmed Arduino board. Two digital outputs are wired to the STEP and DIR inputs of the EasyDriver chip. By appropriately setting its digital outputs high or low based on the incoming commands, the Arduino can precisely control the stepping of the motor as needed by the scanning software.

4.3 Staging Components

To be able to precisely place the scanning target within the small field of view of the imaging system, a micrometer positioning stage was added to the microscope stand. The Deltron 301-XY ball positioning stage offers precise linear travel in both the X and Y axes of the microscope image plane, with positioning gradations of 0.001 inches. A thin rectangular piece of aluminum was covered in non-reflective black foil and mounted on the positioning stage to provide a dark backdrop to the imaging system. Figure 8 shows the complete 3D scanning system, with a summary of the imaging, lighting, and staging components, as well as a photograph of the actual 3D scanning system setup.

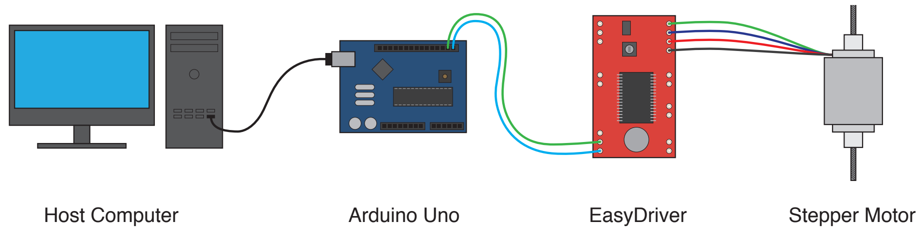


Figure 7: A schematic of the stepper motor control interface.

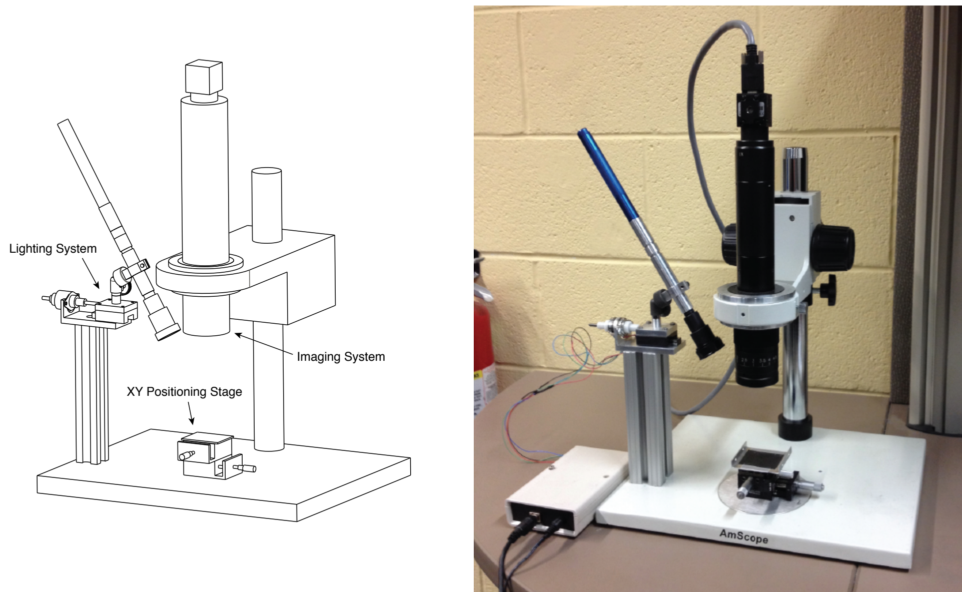


Figure 8: A schematic summarizing the complete 3D scanning system (*left*); a photograph of the actual 3D scanning setup (*right*).

5 Software

To allow for easy and user-friendly use of the scanning system, a standalone software application was implemented in Java. The entire 3D scanning process can be handled using this software's user interface, from the image capturing to the calibration to the final reconstruction. The complete source code for this application is included in Appendix B. The main modules of the application will be described briefly here.

5.1 Camera Control Module

Point Grey Research, the manufacturer of the scanning system's camera, offers a software development kit (SDK), the FlyCapture® SDK, featuring application programming interfaces (APIs) for C and C++ programming languages [12]. The Flea®2 camera is connected to the host computer via 6-pin FireWire cable, and the FlyCapture® SDK allows the software to issue configuration or image capture commands directly to the camera.

The open-source JavaCV package [13] provides Java wrappers to common computer vision libraries, including the FlyCapture® SDK and OpenCV, allowing the C and C++ libraries to be used in Java applications. This package is incorporated into the software application to facilitate the control of the Flea®2 camera from the Java-implemented user interface.

At application startup, the host computer's FireWire bus is queried for valid camera connections. If

one is found, a `Flea2` object (see Appendix B.1) is initialized and acts as the interface between the main software application and the camera control API for the remainder of application execution.

5.2 Stepper Control Module

As detailed in Section 4.2, an Arduino Uno microcontroller board is used to control the digital logic levels that govern the direction and stepping of the stepper motor. Two of the Arduino Uno's digital outputs are used—one for the EasyDriver's STEP input, and the other for its DIR input.

The Arduino processor runs its software as a cyclic executive program. In other words, it performs only one task over and over from the moment it boots up until it powers down. In this case, the task is simple. At each cycle, the serial communication buffer is checked for content. If the buffer contains one of three defined command characters, it performs one of the three corresponding actions: (1) set the STEP output to logical high, then logical low; (2) set the DIR output to logical high; or (3) set the DIR output to logical low. These commands have the effect of actuating one microstep, setting the step direction to forward, and setting the step direction to backward, respectively.

This simple command system is used by the Java application to control the sweeping of the plane during the scanning process. If the Arduino Uno is detected on one of the host computer's standard COM ports at application startup, a `Stepper` object (see Appendix B.2) is created, and acts as the interface between the main application and the low-level serial command output to the Arduino Uno.

5.3 Data Analysis Module

The numerous computational methods necessary throughout the scanning system's execution, glossed over in Section 3, are assembled into a common library in `DataAnalysis.java` (see Appendix B.3). Most of these methods take advantage of OpenCV's extensive library of matrix manipulation and image processing functions, made available in Java with JavaCV. They can be further subdivided into three smaller sections, as follows.

Camera Calibration. The estimation of the scanning camera's intrinsic and extrinsic parameters uses the OpenCV `calib3d` module to process images of a checkerboard calibration target. From these processed images, estimates of K, R, and T in Equation (1) are computed and written to an application-specific calibration file.

Projector Calibration. This section implements the projector calibration algorithms described in Section 3, using images of the slide pattern projected onto an empty stage. Estimates for n and the parameters of the linear function we called $corner(t)$ are computed and written to the same application-specific calibration file.

Reconstruction. Using the calibration parameters computed for the camera and the projector, the images captured during the scan are processed and used to reconstruct a 3D point cloud of the scanned object, implementing Algorithm 4. These results are then written to a PLY-formatted file and can be viewed in a mesh processing software system such as MeshLab [14].

5.4 Graphical User Interface

The application includes a graphical user interface (GUI) that uses these three fundamental modules to perform the various tasks of the 3D scanning pipeline at the request of the user. The GUI is divided into three main control panels, each to accomplish one of three tasks: scanning, calibration, and reconstruction.

Scanning Interface. Figure 9 shows the user interface for the scanning control panel. The source code for this control panel can be found in Appendix B.4.

It is here that the user can configure the camera sensor parameters; the gain, shutter, and brightness settings can be changed in realtime if the *PREVIEW* button is enabled, which initiates a live image stream from the camera at a frame rate of 15 fps.

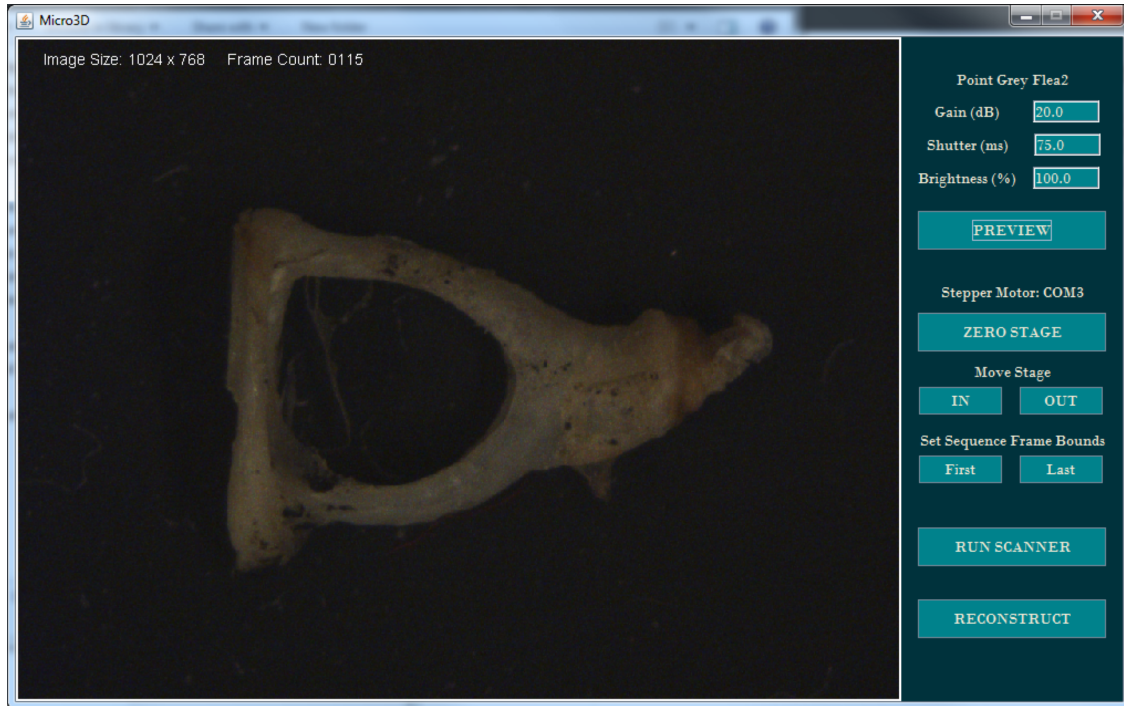


Figure 9: The user interface to control the image capture process of the scanning system.

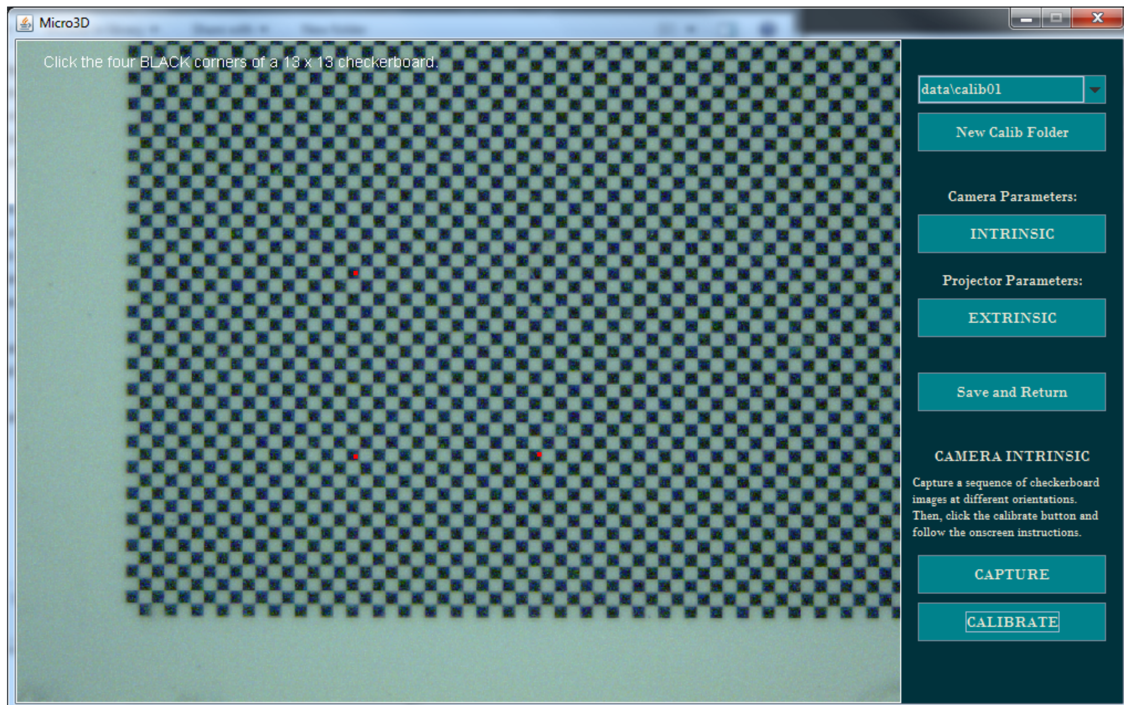


Figure 10: The user interface to control the calibration of the scanning system.

Once the camera sensor is configured appropriately for the lighting conditions of the scanning environment, the active lighting system must be initiated. The stepper motor should be zeroed at its most distal position to establish its translation limits. With the target object completely in the field of view (as shown in Figure 9), the illuminated projector is positioned along its linear translation axis using the *IN* and *OUT* buttons under the *Move Stage* label. The desired position of the projected slide pattern in the first and last frames are set with the corresponding buttons under the *Set Sequence Frame Bounds* label. For the first frame, the leading edge of the projector pattern should be completely to the left of the target object, but still visible in the viewport. Likewise, for the last frame, the leading edge of the projector pattern should be completely to the right of the target object, but still visible in the viewport.

Once these scanning setup procedures are completed, the user simply clicks *RUN SCANNER*. The stage is positioned at its user defined starting position, and the scanning process begins. The translation stage is incrementally moved a predefined number of substeps, and an image is captured of the light pattern projected onto the object. This process continues until the active lighting system has reached its user defined ending position, saving each image it captures to file.

Once the last captured image is saved to file, the user is presented with the first and last frames of the sequence, and asked to indicate the three corners of the projected slide pattern in each. This allows for the calculation of the projector light plane normal, n, and the computation of a linear function $corner(t)$ that returns the position of the projector pattern corner over time, as described in Section 3.

Calibration Interface. Figure 10 shows the user interface for the calibration control panel. The source code for this control panel can be found in Appendix B.5.

It is here that the intrinsic camera calibration is executed. The calibration target is placed on the stage, and manually brought into focus by the user, by adjusting the focusing knobs of the microscope stand holding the imaging system. Our calibration target was a small checkerboard pattern of square side 0.100 mm, shown as captured by the live camera stream in Figure 10. To begin a new calibration session, the user clicks the *New Calib Folder* button. This creates a new directory for the calibration images to be stored. The user manually adjusts the positioning of the calibration target on the stage, and clicks the *CAPTURE* button at each orientation to save a calibration image to file. Once the user is satisfied with the captured calibration images, he clicks *CALIBRATE* to interactively calibrate the camera from the captured images. Each image is displayed, and the user encloses a grid of 13 × 13 checkerboard squares by clicking on its four corners. The square corners within this grid are then automatically detected and used to iteratively estimate the intrinsic parameters of the imaging system, K, R, and T.

The calibration panel also includes functionality for the extrinsic calibration of the active lighting system, which is also performed during the scanning process as explained above.

When the camera and projector have been successfully calibrated, the user clicks the *Save and Return* button to return to the reconstruction panel and perform the final steps of the reconstruction process.

Reconstruction Interface. Figure 11 shows the user interface for the reconstruction control panel, executed on an image sequence captured of a human stapes bone (see Section 6). Within this panel, the algorithms of Section 3 implemented in `DataAnalysis.java` are used to process the captured image sequence. The source code for this control panel can be found in Appendix B.6.

The reconstruction process can be completed in a few simple steps. First, the directory containing the entire sequence of scanned images is selected. Clicking the *Max & Min Intensity* button loads the images and computes I^{max} and I^{min} (Algorithm 1), displaying them in the interface. Next, the user defines the appropriate contrast requirement that must be exceeded in order for pixels to be considered recoverable. Clicking *Contrast Mask* will compute and display the corresponding pixel mask (C in Algorithm 2). Once an appropriate pixel mask is achieved, clicking *Shadow Edge* will compute for each recoverable pixel the interpolated floating point frame value at with the leading edge of the shadow crosses that pixel (T in Algorithm 3). These results are visualized on the interface in grayscale, remapping the values from the frame interval $[1, N]$ to the color interval $[\text{black}, \text{white}]$. Finally, the user clicks *RECONSTRUCT*. The shadow edge results are used to reconstruct a 3D point cloud from the scan sequence (P in Algorithm 4). The point cloud is automatically output in PLY file format to be visualized or post-processed in appropriate software.

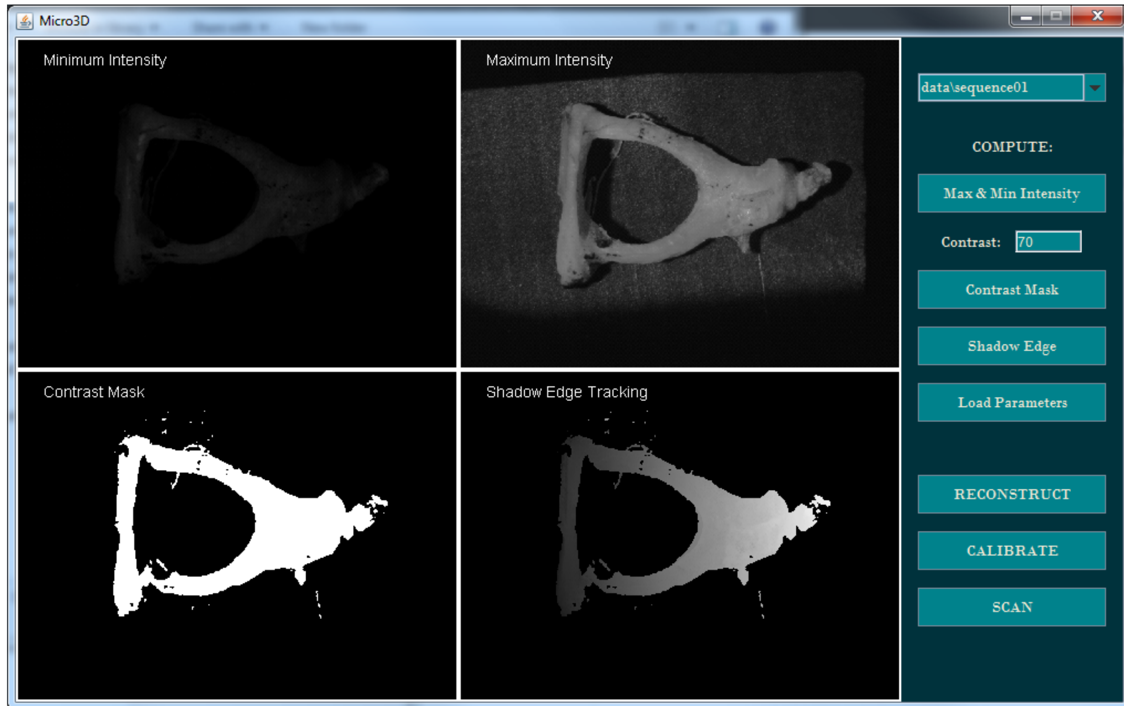


Figure 11: The user interface to control the reconstruction from the captured images.

6 Testing Results

In order to test the complete scanning system, we chose to scan a human stapes bone. The stapes is one of three bones of the middle ear, and is the smallest bone of the human body. Fairly accurate three-dimensional models of the stapes have been developed using stacked images of thinly sliced histological sections; however, the process of decalcifying the bone, along with the labor involved with the delicate tissue microscopy, can draw this process out by days or weeks. Such three-dimensional information could be used to develop more accurate models of the stapes bone and the middle ear mechanics.

Scanning. The stapes specimen, measuring just over 4 mm along its long axis, was swept by the projector light plane in 527 frames. Figure 12 shows some characteristic frames from the image sequence.

Figure 12: Three characteristic frames from the sequence of 527 images: frame 1 (*left*); frame 203 (*middle*); and frame 527 (*right*).

Calibration. The scanning system was interactively calibrated using the software application, as explained above. The camera calibration parameters were estimated to be

$$K = \begin{bmatrix} 11560. & 0.0 & 692.1 \\ 0.0 & 11530. & 591.6 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \quad (11)$$

$$R = \begin{bmatrix} 0.9998 & 0.001851 & 0.02065 \\ -0.002296 & 0.9998 & 0.02157 \\ -0.02061 & -0.02162 & 0.9996 \end{bmatrix} \quad (12)$$

$$T = \begin{bmatrix} -2.049 \\ -2.543 \\ 76.88 \end{bmatrix}, \quad (13)$$

where spatial units are expressed in terms of mm.

The projector normal was estimated to be

$$n = \begin{bmatrix} -0.5461 \\ -0.0055 \\ 0.8377 \end{bmatrix}, \quad (14)$$

while the time-dependent corner-positioning function was computed as

$$corner(t) = \begin{bmatrix} -0.00939 \\ 0.00087 \\ 0.0 \end{bmatrix} t + \begin{bmatrix} 0.2851 \\ -1.224 \\ 0.0 \end{bmatrix}. \quad (15)$$

Shadow Threshold Crossings. Figures 13 and 14 show the computed pixel intensity minimum and maximum images, respectively. A value of 70 (out of 255) was found to be a good contrast constraint, yielding a visually appropriate pixel mask C (see Figure 15). Black pixels are disregarded to compute the shadow threshold crossings. Figure 16 shows the visualization of these threshold crossing interpolations. The darker a pixel intensity is, the earlier it was directly illuminated by the sweeping light plane.

Reconstruction. Finally, the calibration results above were used to reconstruct a 3D point cloud of the surface of the stapes bone from these shadow threshold crossing values. Based on the contrast constraint, 75 151 points out of a possible 786 432 pixels were reconstructed. Figure 17 shows two renderings of this point cloud visualized in the MeshLab software.

This 3D point data can be used as input to one of several surface reconstruction algorithms available in MeshLab that output a triangle mesh. The Ball-Pivoting Algorithm ([15]) was applied to the stapes point cloud, and yielded a triangle mesh composed of 137 778 faces. Figure 18 shows two renderings of the resulting triangle mesh after applying Taubin smoothing ([16]).

The 3D mesh generated from our scanning system is fairly representative of the geometry of the stapes bone from visual inspection. It seems to accurately capture depth information for large scale features on the stapes surface. However, high frequency noise in the point cloud data leads to the pitted appearance of the final mesh. Moreover, the thinner portions of the stapes bone "stirrup" appear to be quite flat in the 3D reconstruction, when in reality they are more cylindrical.

Some of these reconstruction errors can be attributed to the components of the 3D scanner. The low-cost active lighting system was assembled from aluminum parts machined in-house; the Joint Engineering Physics Instrument Shop does not have the equipment required to machine such small-scale parts to a very high precision, and thus assumptions about the geometry of the optical projector—especially the slide edges—may deviate from their actual dimensions a significant amount. The calibration target for the intrinsic camera calibration was simply a cheap demo of the optical targets of Applied Image Inc.; a precisely manufactured target would have cost in excess of $1000.

Some of the error can also be attributed to the material properties of the target object. The reconstruction theory presented in Section 2 assumes a perfect Lambertian surface where an illuminated point emits diffuse light based on its facing ratio to a light source. However, as can be seen by examining the images of the captured sequence, the bone exhibits both specular highlights and subsurface scattering, leading to a significant amount of noise in the recovered data. Because the "stirrups" of the stapes are

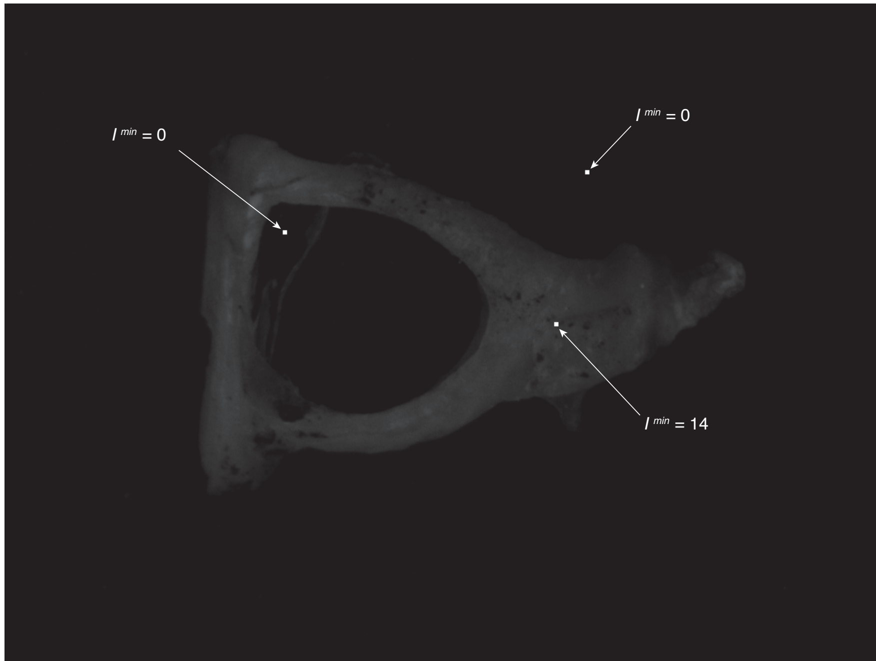


Figure 13: Minimum per-pixel intensity values I^{min}, with some characteristic values in different image regions labeled.

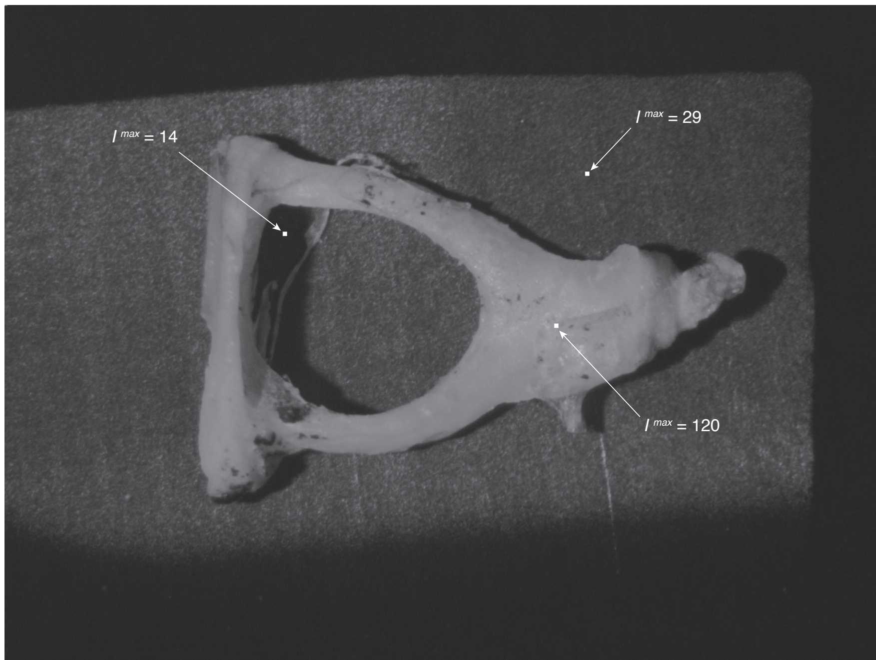


Figure 14: Maximum per-pixel intensity values I^{max}, with some characteristic values in different image regions labeled.

Figure 15: Pixel mask C indicating recoverable pixels, with some characteristic values of pixel contrast in different image regions labeled.

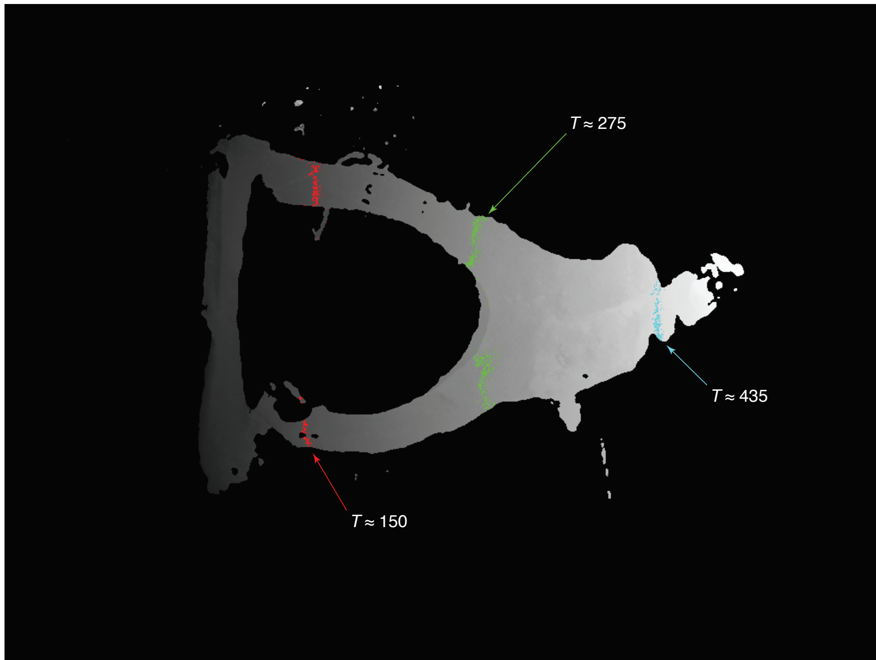


Figure 16: Visualization of shadow threshold crossing time T, with three rounded frame values shown in color.

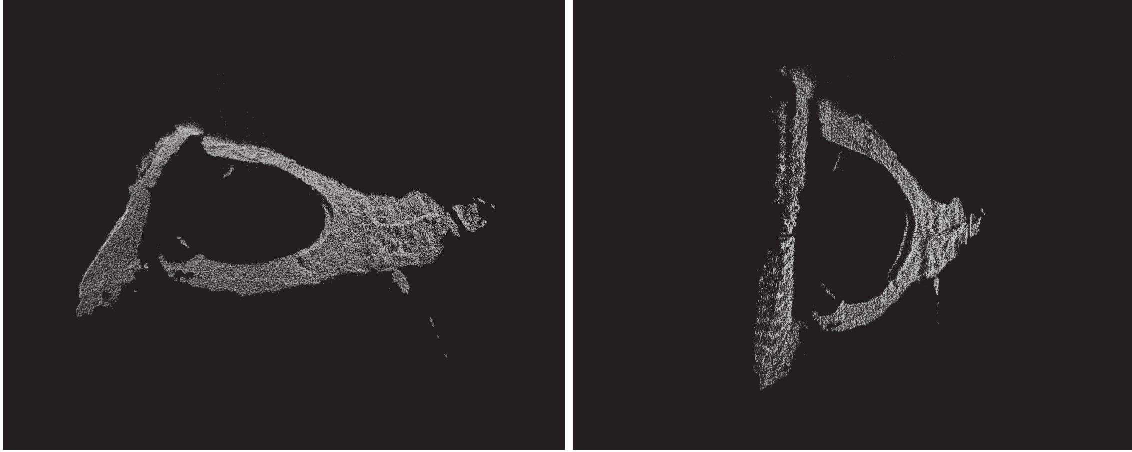


Figure 17: Renderings of the stapes bone point cloud, with 75 151 vertices, in the MeshLab software.

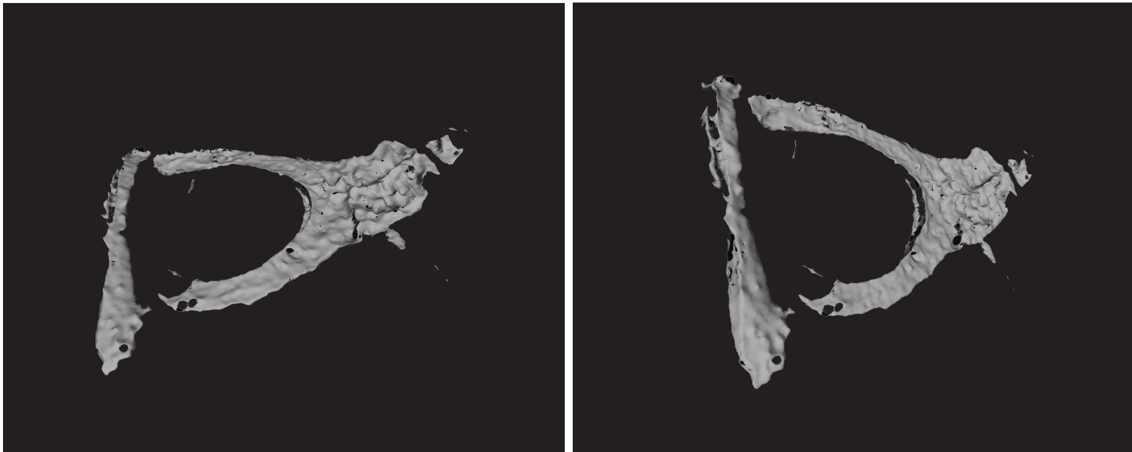


Figure 18: Renderings of the reconstructed stapes bone surface, with 75 151 vertices and 137 778 faces, in the MeshLab software. The surface was reconstructed using the Ball-Pivoting Algorithm with a ball radius of 0.0156 mm and an angle threshold of $90°$. Taubin smoothing was applied to the initial reconstructed surface, with $\lambda = 0.5$ and $\mu = -0.53$ for 50 iterations.

so thin, the subsurface scattering has more of an effect, and the hard light plane edge seems to "bleed" into the material around it. This leads to significant noise in the recovered shadow crossing thresholds, and could explain why the reconstructed surface appears flattened around these areas.

7 Conclusion

The results of our 3D scanning system are promising, given the small budget used to purchase and machine its components. The software provides a user-friendly graphical interface to make it easy to calibrate the system, scan a sample, or reconstruct captured data. The whole process can be completed in just minutes, and the resulting high-resolution point clouds can be easily imported to open-source mesh processing software like MeshLab for surface reconstruction and other analysis methods. With a larger budget, these principles could be implemented using much more precisely machined parts, and would most certainly yield even better results.

8 Continuing Work

One of the main drawbacks of the 3D scanning system design presented here is its inflexibility in capturing full surface scans—i.e., to scan several perspectives of the object and merge the results. The target object must be placed on the flat aluminum stage and imaged as is. For an object with a certain amount of planarity, like our stapes bone, this limits the scan to really only two perspectives: the "top" and the "bottom". For objects with more amorphous surface shapes, it might even prove to be impossible to keep the object still in the desired orientation during the course of a scan.

To overcome these limitations, continuing work is being done to augment the staging system of the scanner with a rotating "cage" that holds the object in a microvise as it is scanned. Then, the cage can simply be rotated about the cage axis and rescanned from a different perspective. This process could even be optimized with the inclusion of another stepper motor, to automatically rotate the cage, scan, and repeat as necessary until a sufficient portion of the object's surface has been scanned. A prototype for this cage has been built, and is shown in Figure 19. However, the vise design remains to be implemented well enough to hold the small, delicate objects that would be scanned by such a system.

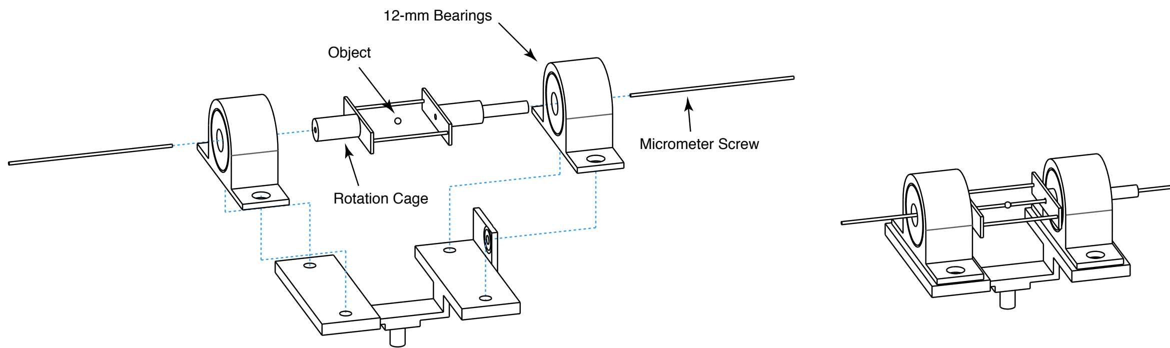


Figure 19: A schematic of the complete active lighting system, in exploded (*left*) and assembled (*right*) form.

Acknowledgements

I wish to thank Professor Gabriel Taubin for his advice and supervision throughout this project. Thanks are due as well to Professor Jerry Daniels for his valuable comments and questions. This project would not have been possible without the skill and effort of Charles Vickers and Michael Packer in the Joint Engineering Physics Instrument Shop. Finally, I would like to thank my family for their continuing support in all my academic endeavors.

References

[1] D. Gabor, "Microscopy by reconstructed wave-fronts," *Sciences*, vol. 197, no. 1051, pp. 454–487, 1949.

[2] S. W. Paddock, "Confocal laser scanning microscopy," *BioTechniques*, vol. 27, pp. 992–1004, 1999.

[3] K.-W. Kim, Kyu-Gyeom, J.-H. Kim, J.-H. Min, H.-S. Lee, and J. whoan Lee, "Development of the 3D volumetric micro-CT scanner for preclinical animals," *3D Research*, vol. 2, pp. 1–6, 2011.

[4] J.-J. Lee, D. Shin, B.-G. Lee, and H. Yoo, "3D optical microscopy method based on synthetic aperture integral imaging," *3D Research*, vol. 3, pp. 1–6, 2012.

[5] J. Singh, C. C. Hoe, T. H. S. Jason, C. Nanguang, C. S. Premachandran, C. J. R. Sheppard, and M. Olivo, "Optical coherent tomography (OCT) bio-imaging using 3D scanning micromirror," *Endoscopic Microscopy II*, 2007.

[6] Y. Ma, S. Soatto, J. Kosecka, and S. S. Sastry, *An Invitation to 3-D Vision: From Images to Geometric Models*, 1st ed. Springer, ISBN: 978-0-387-00893-6, 2004.

[7] Z. Zhang, "A flexible new technique for camera calibration." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, pp. 1330–1334, 2000.

[8] J.-Y. Bouguet and P. Perona, "3d photography using shadows in dual-space geometry," *International Journal of Computer Vision*, vol. 35, no. 2, pp. 129–149, 1999.

[9] J.-Y. Bouguet, *Camera Calibration Toolbox for Matlab®*, Updated July 2010.

[10] Arduino, *Arduino Uno.* http://arduino.cc/en/Main/ArduinoBoardUno.

[11] B. Schmalz, *EasyDriver Stepper Motor Driver.* http://www.schmalzhaus.com/EasyDriver.

[12] Point Grey Research, *FlyCapture® SDK.* http://ww2.ptgrey.com/sdk/flycap.

[13] S. Audet, *JavaCV: Java Interface to OpenCV and more.* https://code.google.com/p/javacv/.

[14] Instituto di Scienza e Technologie dell'Informazione, *MeshLab.* http://meshlab.sourceforge.net/.

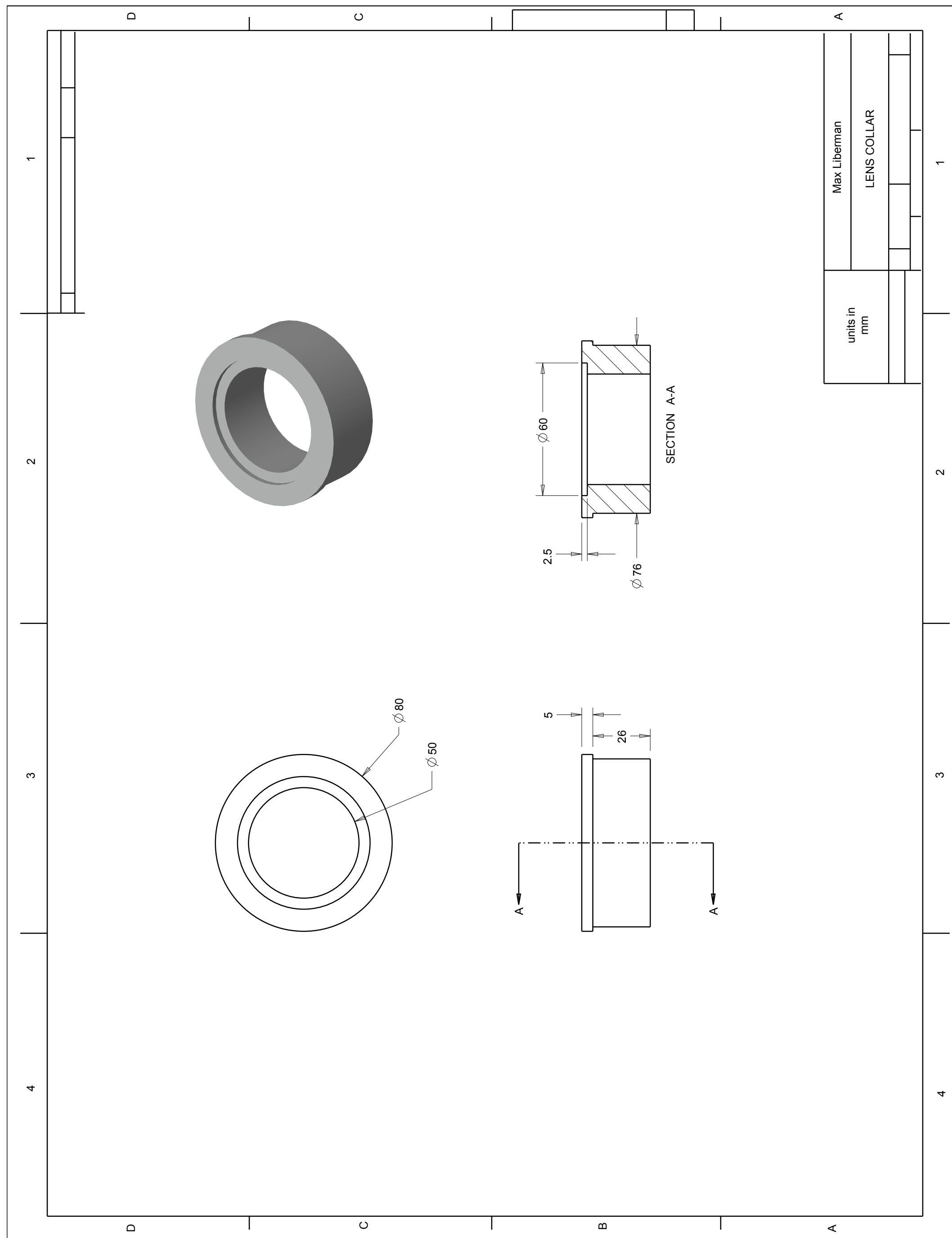
[15] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin, "The ball-pivoting algorithm for surface reconstruction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 4, pp. 349–359, 1999.

[16] G. Taubin, "Curve and surface smoothing without shrinkage," *Computer Vision. Fifth International Conference on*, pp. 852–857, 1995.

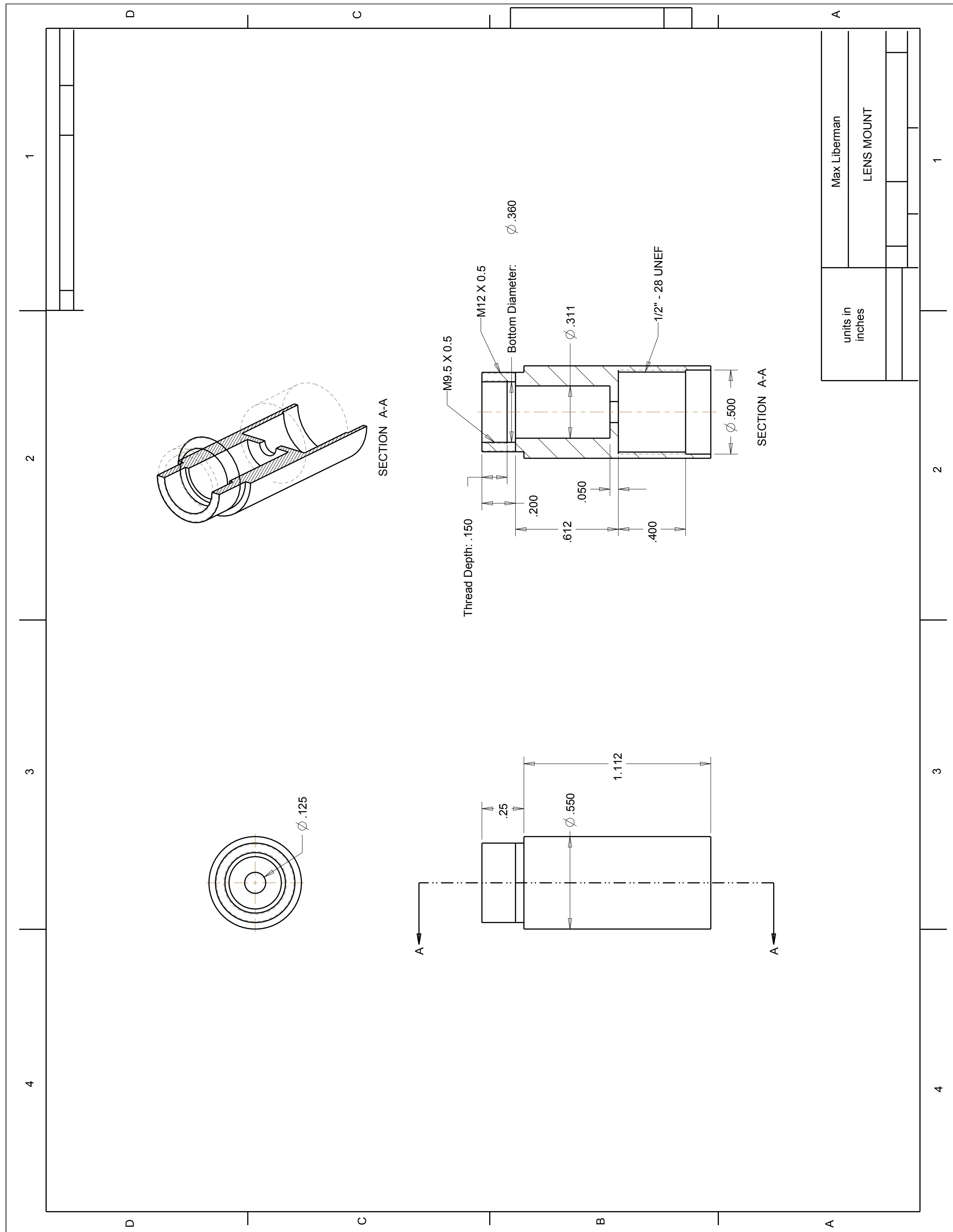
Appendix

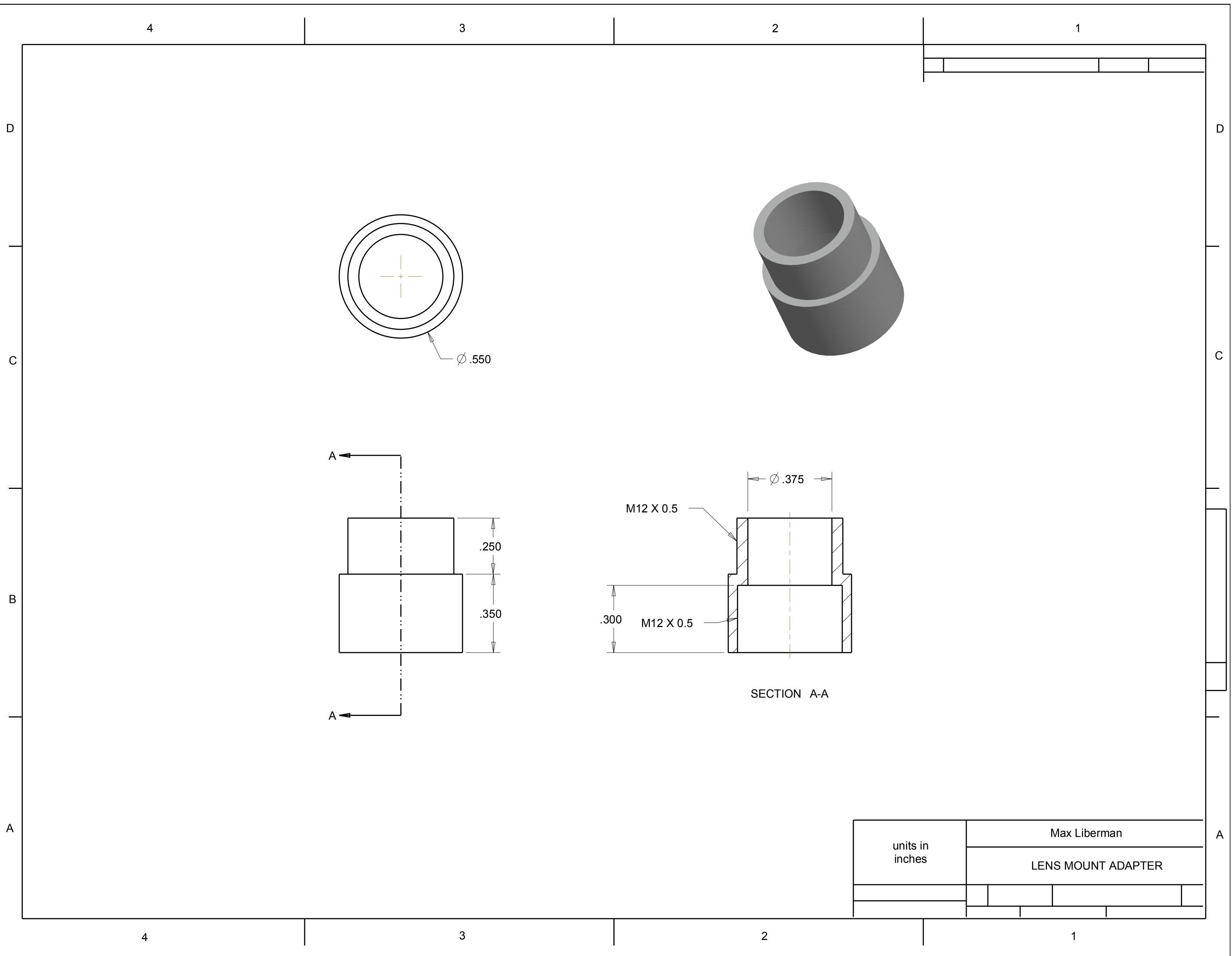
A Engineering Drawings

A.1 Imaging Components

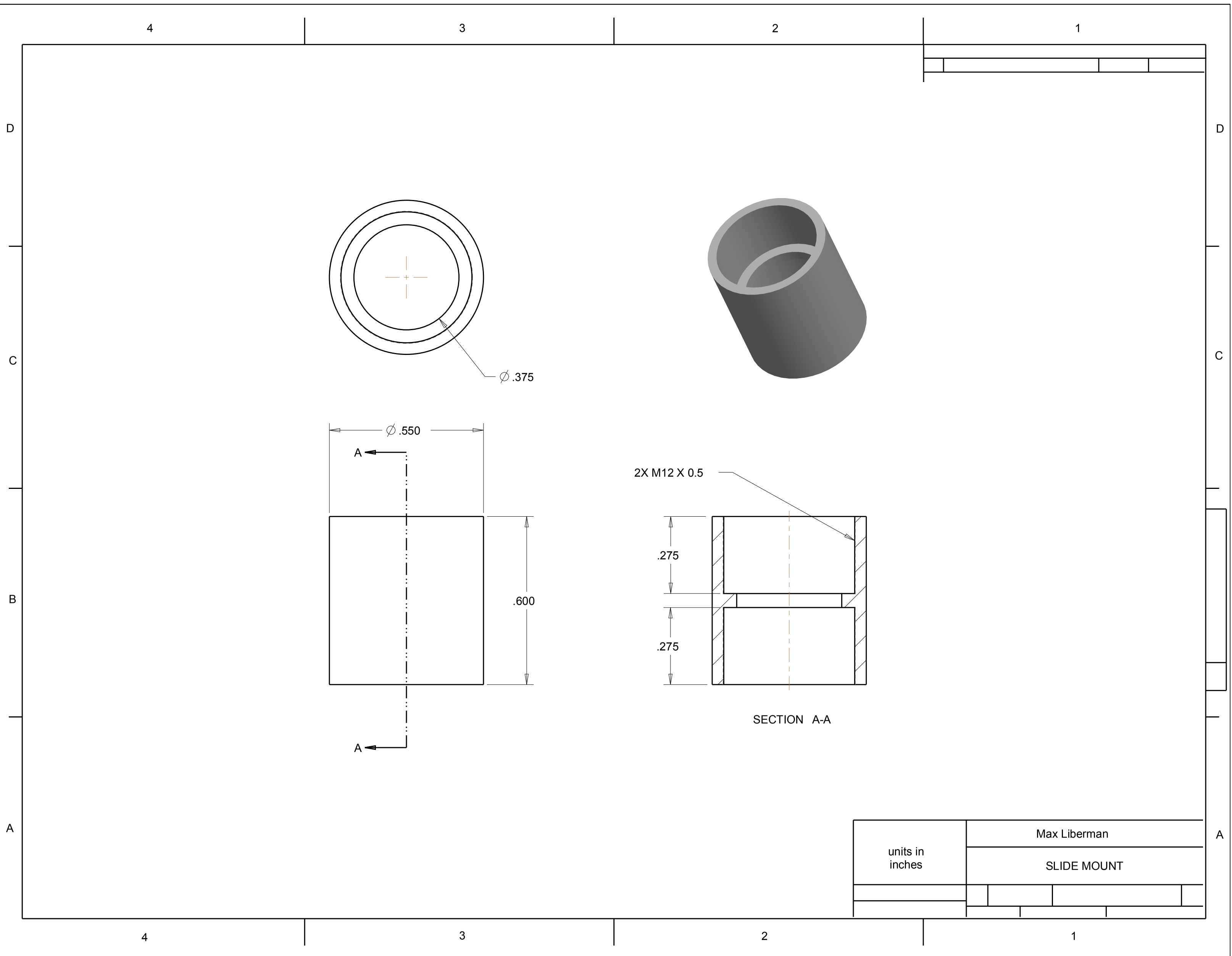


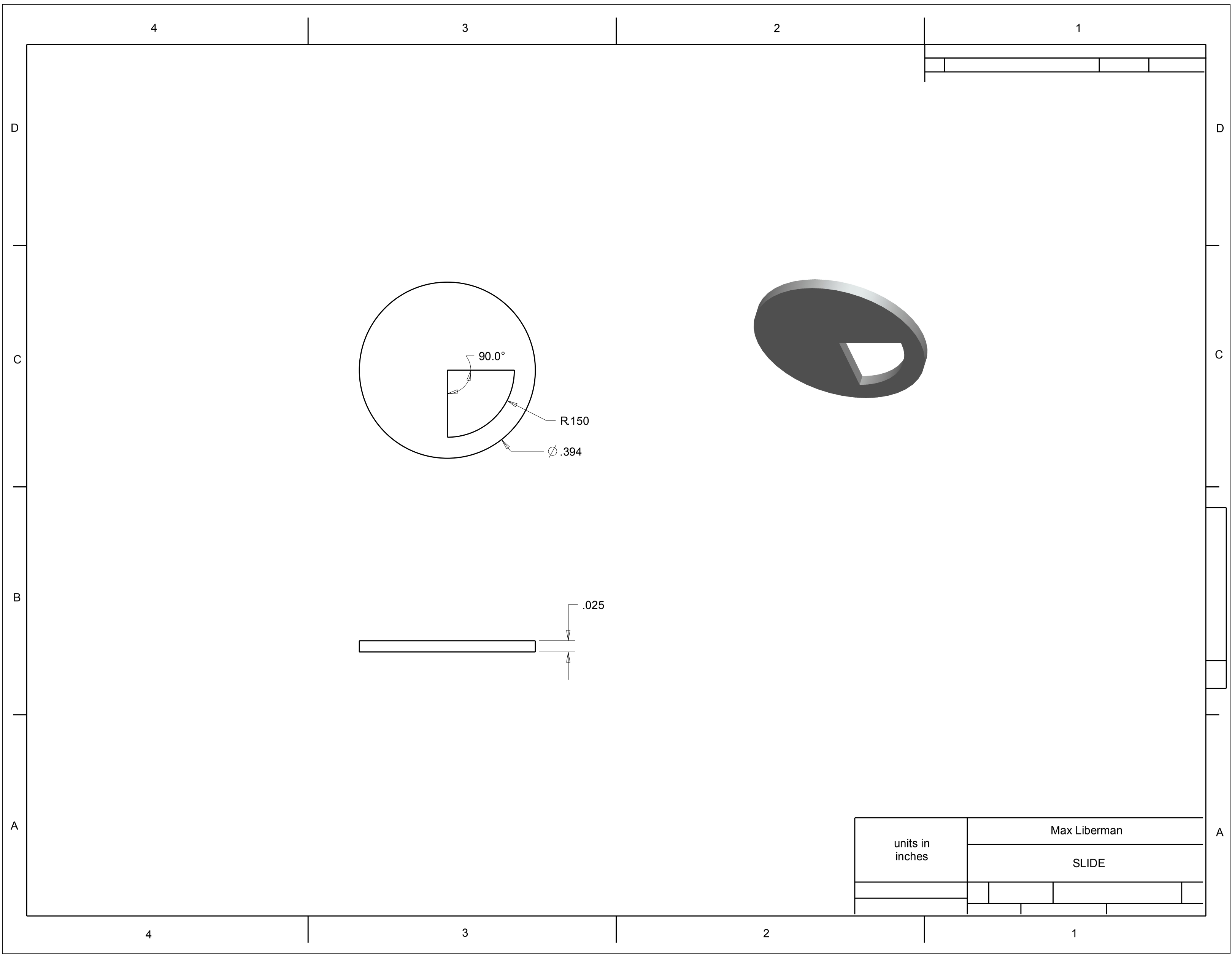
A.2 Active Lighting Components

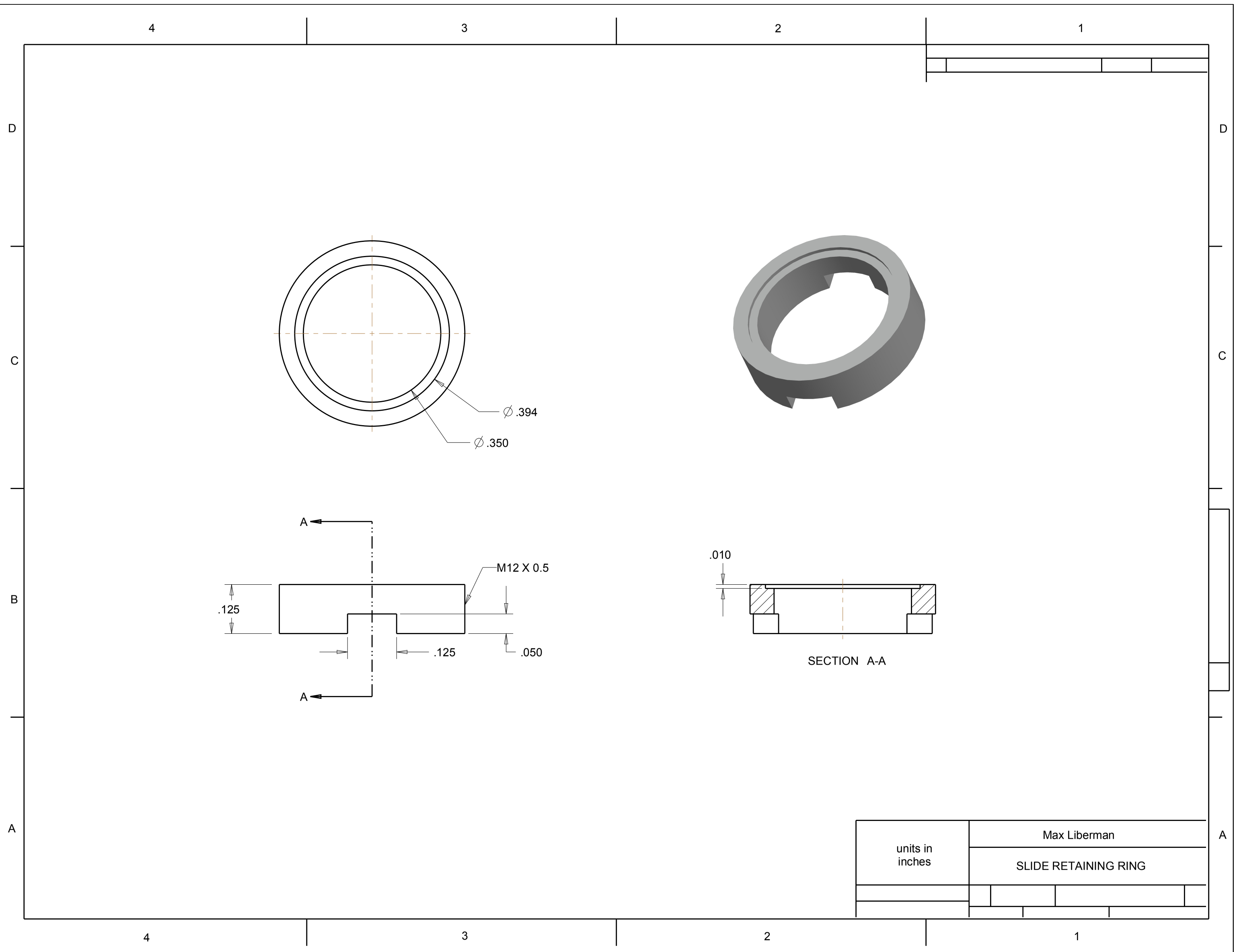


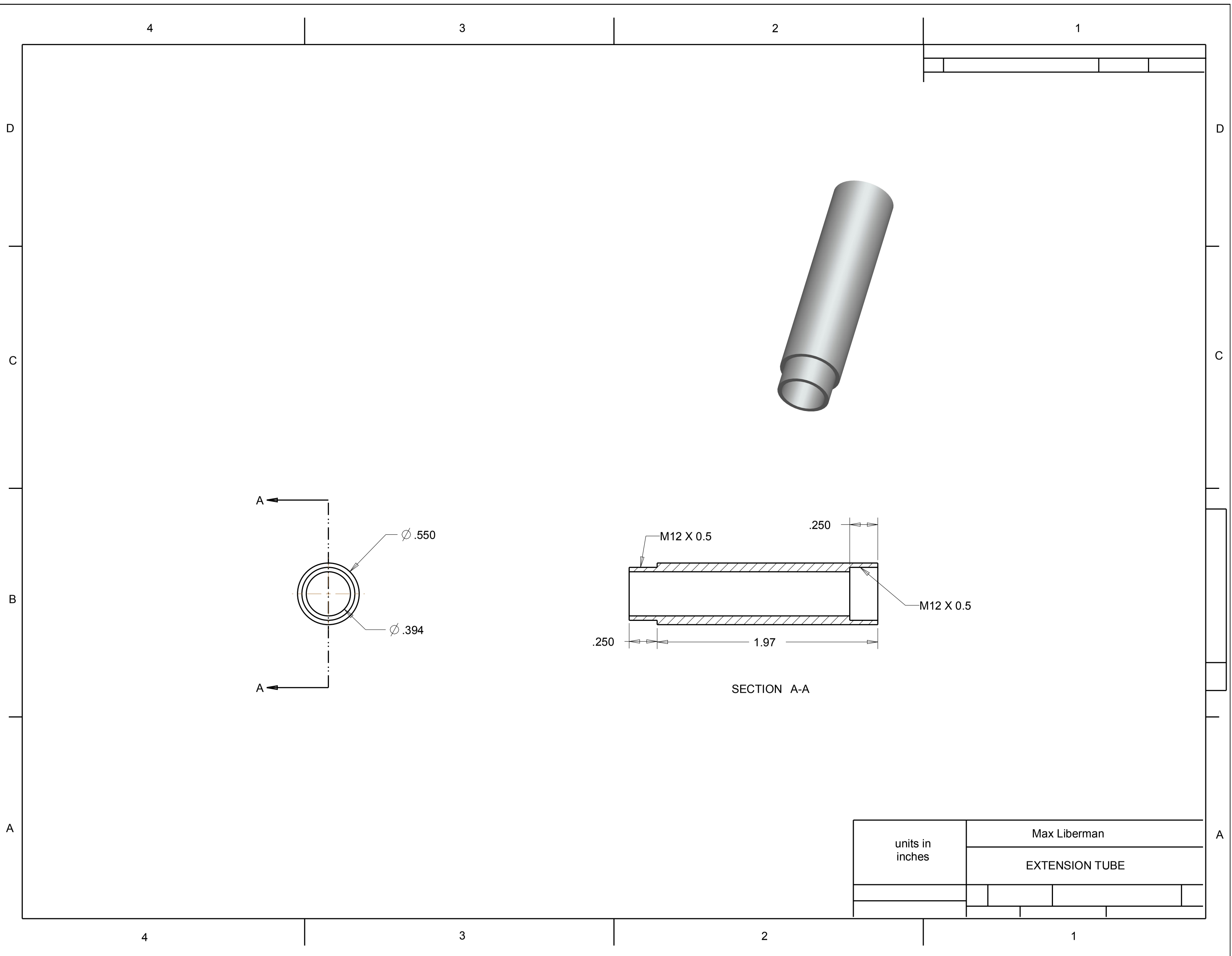


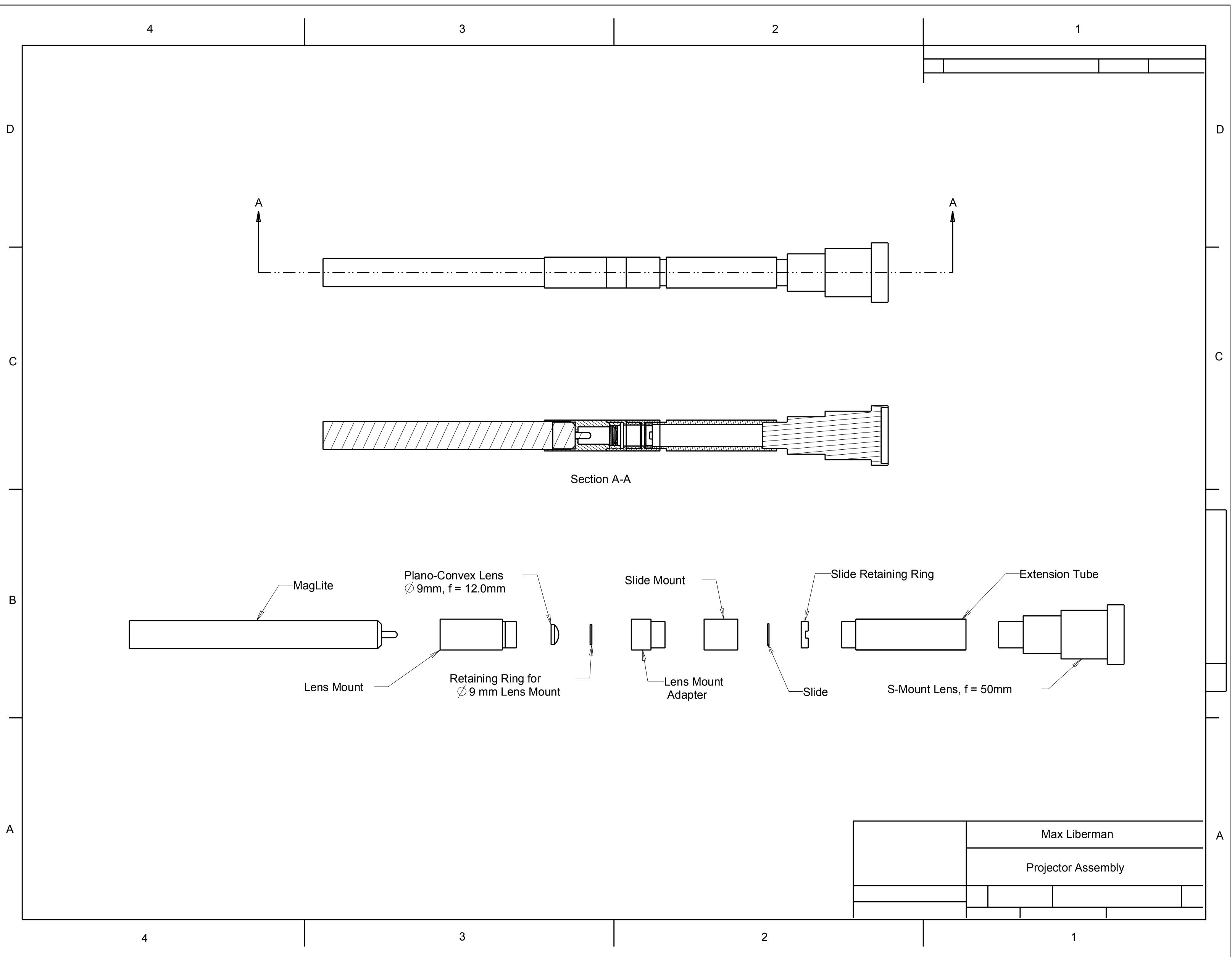
Max Liberman
LENS MOUNT ADAPTER
units in
inches
SECTION A-A
⌀ .375
M12 X 0.5
M12 X 0.5
.300
.250
.350
⌀ .550
A
A

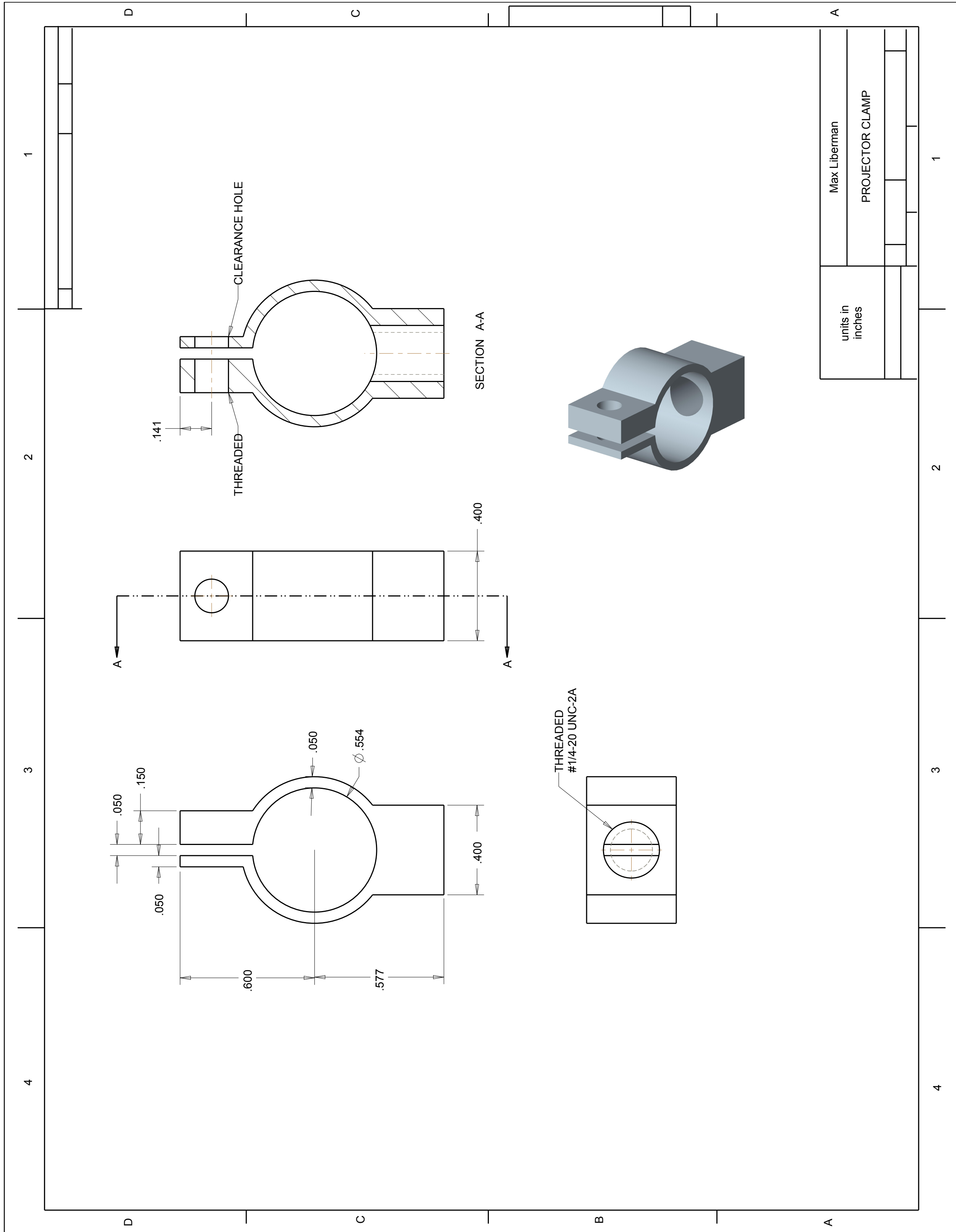

⌀.375
⌀.550
A
A
.600
2X M12 X 0.5
.275
.275
SECTION A-A
units in inches
Max Liberman
SLIDE MOUNT

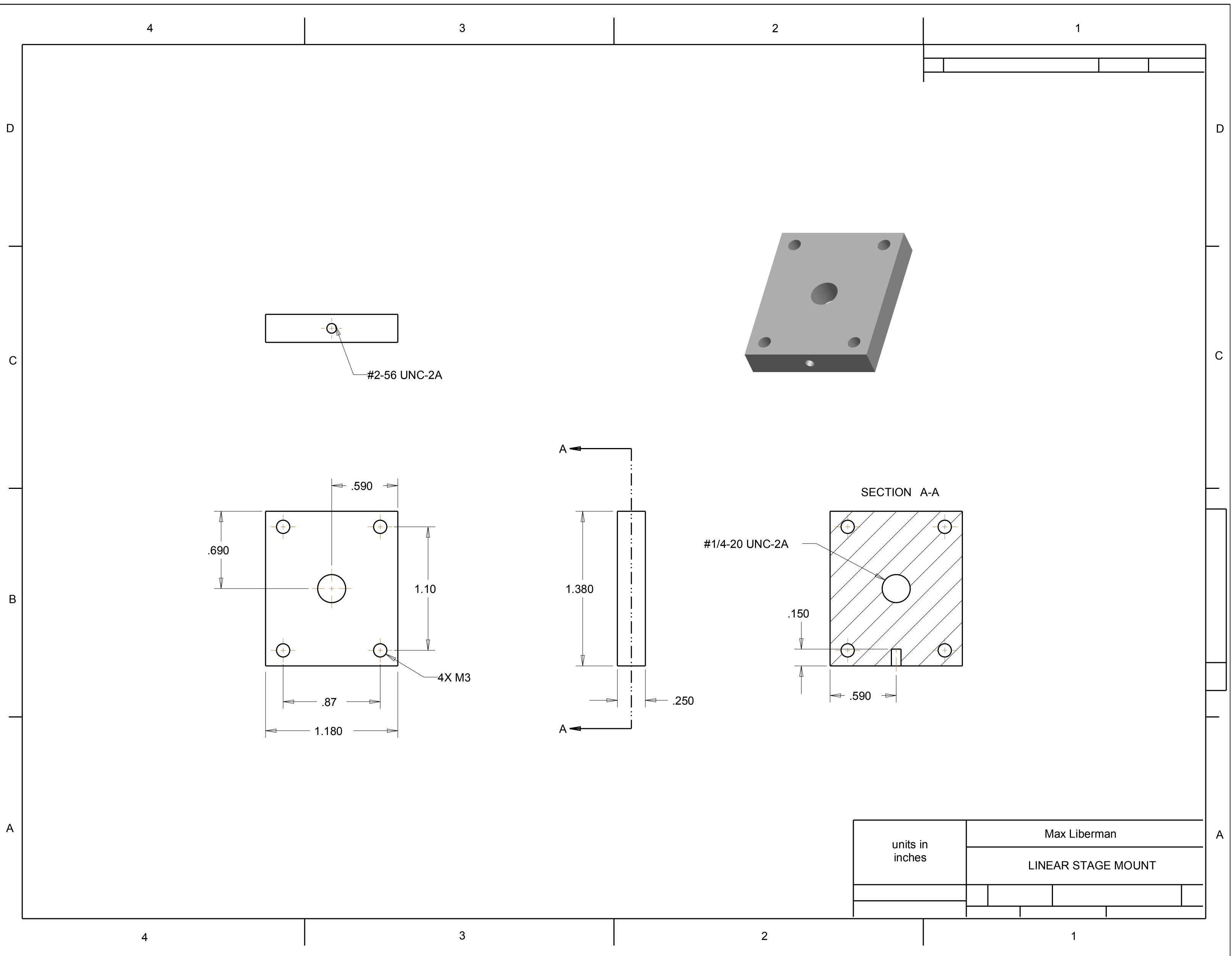

90.0°
R.150
⌀ .394
.025
units in inches
Max Liberman
SLIDE

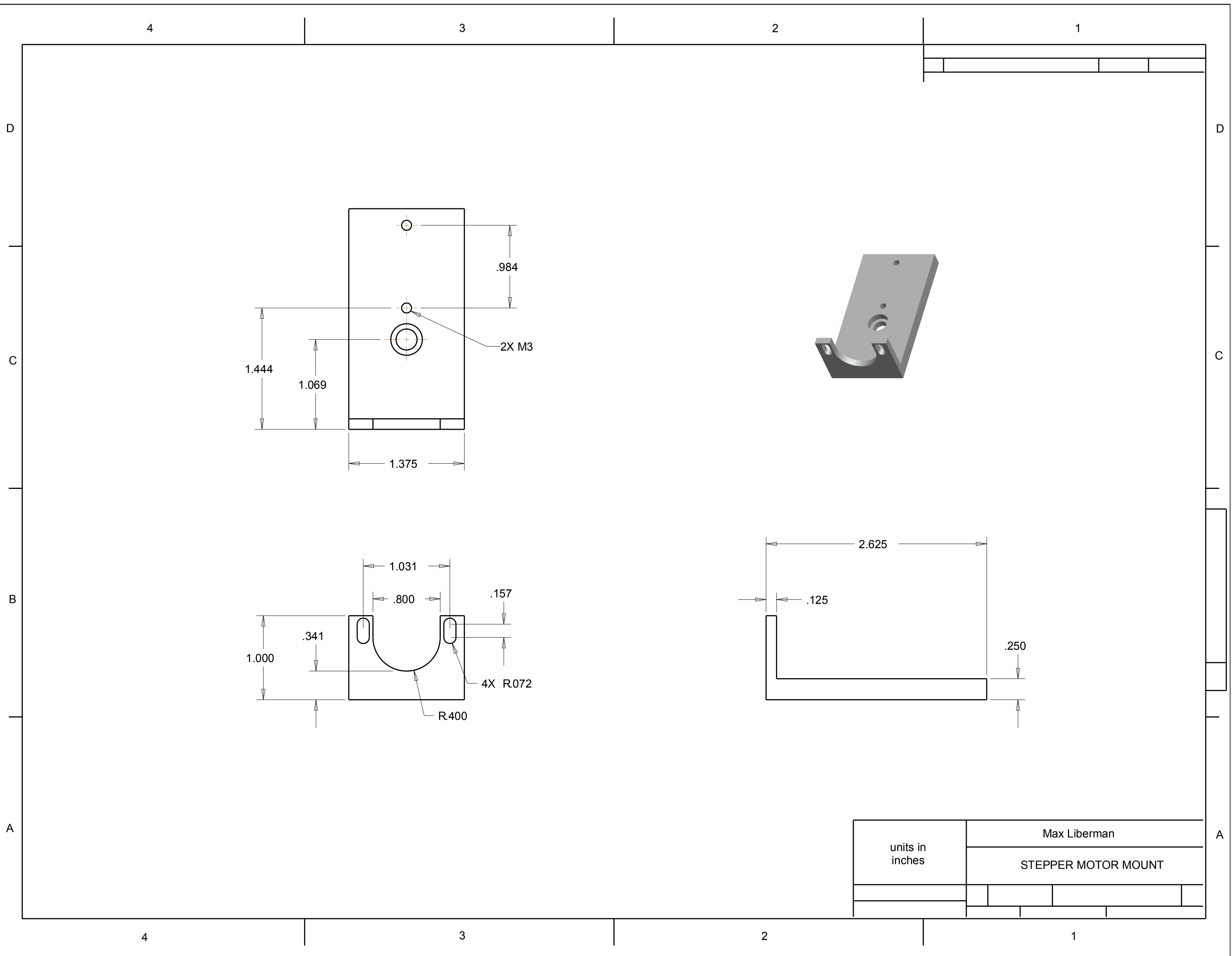

⌀.394
⌀.350
A
M12 X 0.5
.125
.125
.050
A
.010
SECTION A-A
units in inches
Max Liberman
SLIDE RETAINING RING

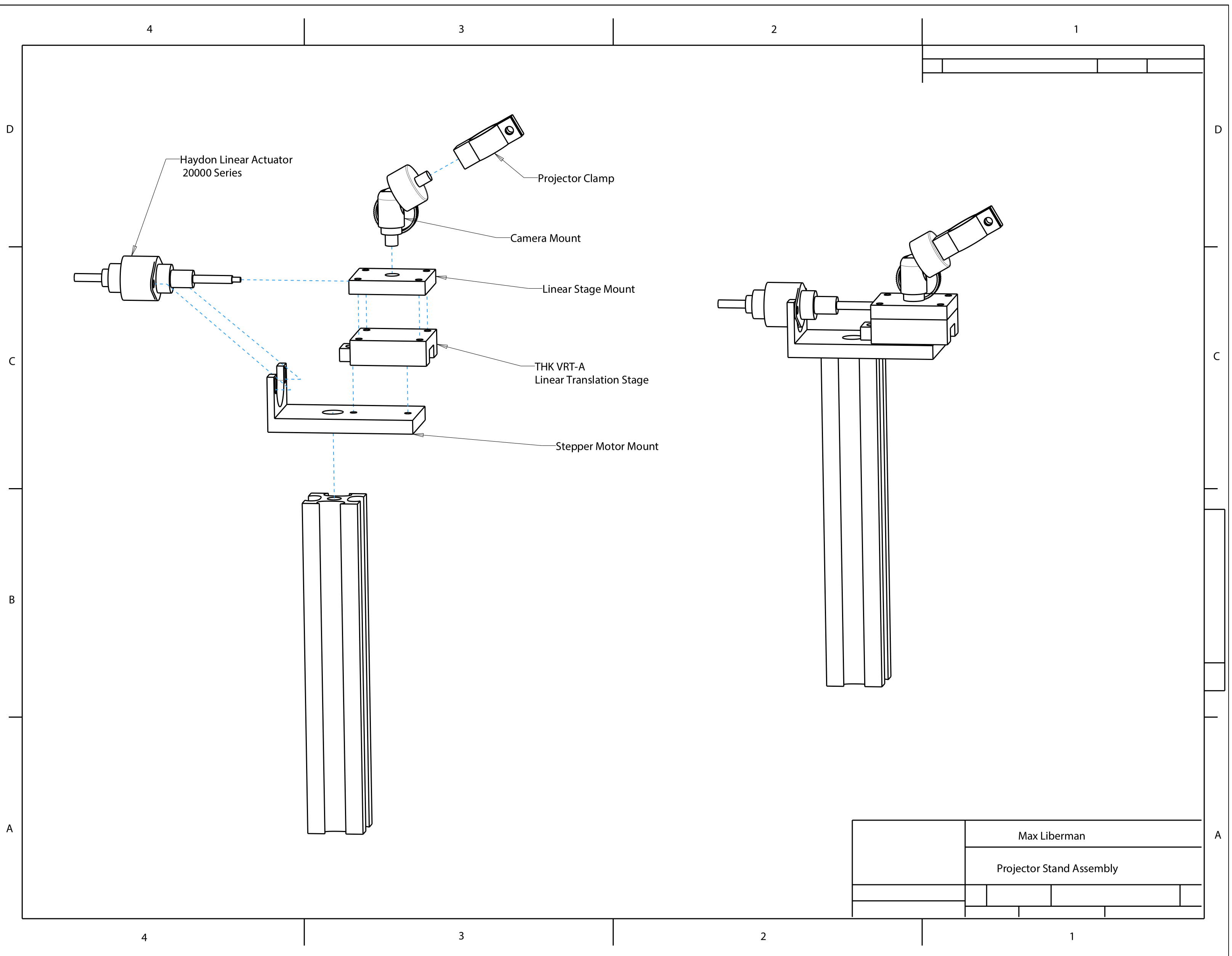

A
∅ .550
∅ .394
A
M12 X 0.5
.250
M12 X 0.5
.250
1.97
SECTION A-A
units in inches
Max Liberman
EXTENSION TUBE


A
A
Section A-A
MagLite
Plano-Convex Lens
∅ 9mm, f = 12.0mm
Slide Mount
Slide Retaining Ring
Extension Tube
Lens Mount
Retaining Ring for
∅ 9 mm Lens Mount
Lens Mount
Adapter
Slide
S-Mount Lens, f = 50mm
Max Liberman
Projector Assembly

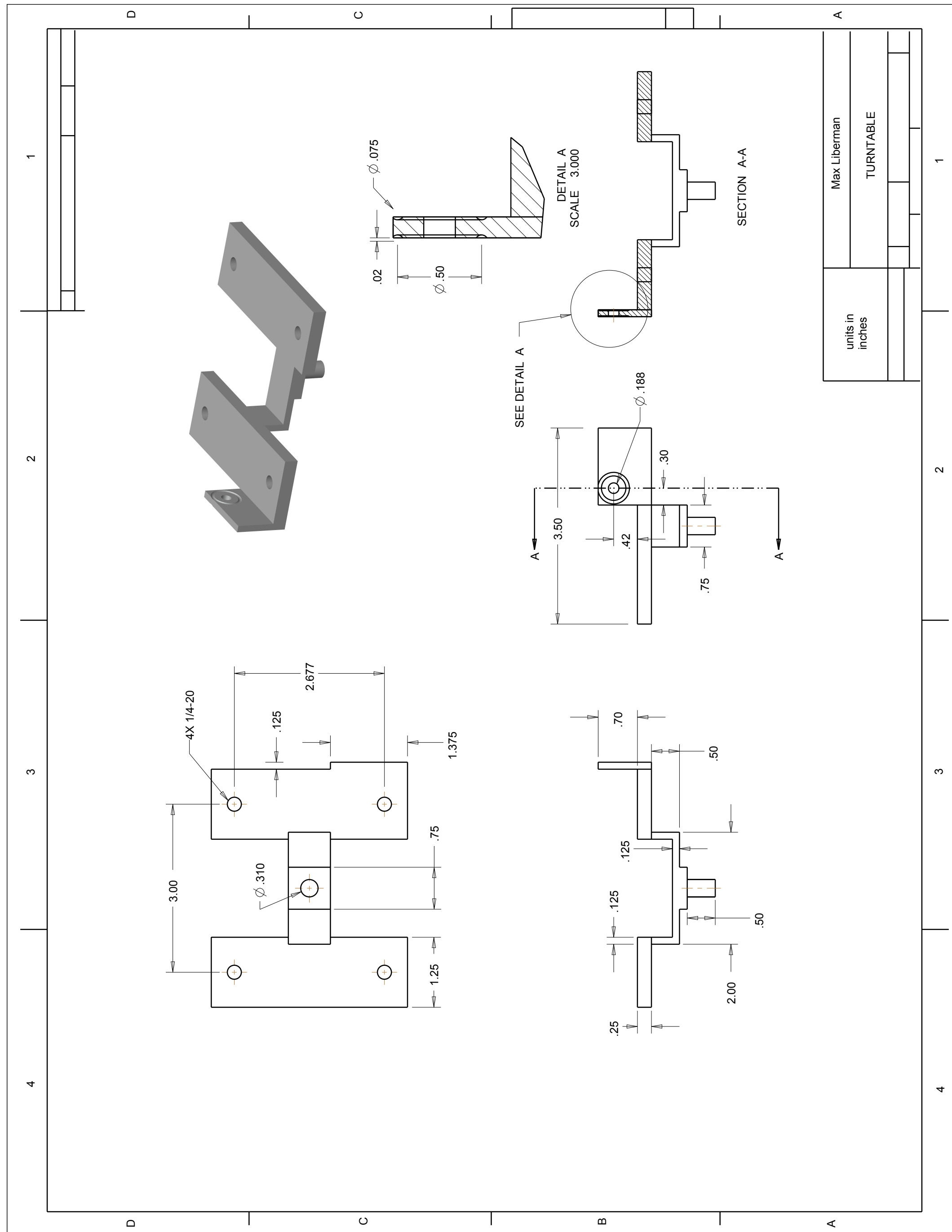

CLEARANCE HOLE
THREADED
.141
SECTION A-A
.400
A
A
.050
.150
.050
.050
⌀.554
.400
.600
.577
THREADED
#1/4-20 UNC-2A
Max Liberman
PROJECTOR CLAMP
units in
inches

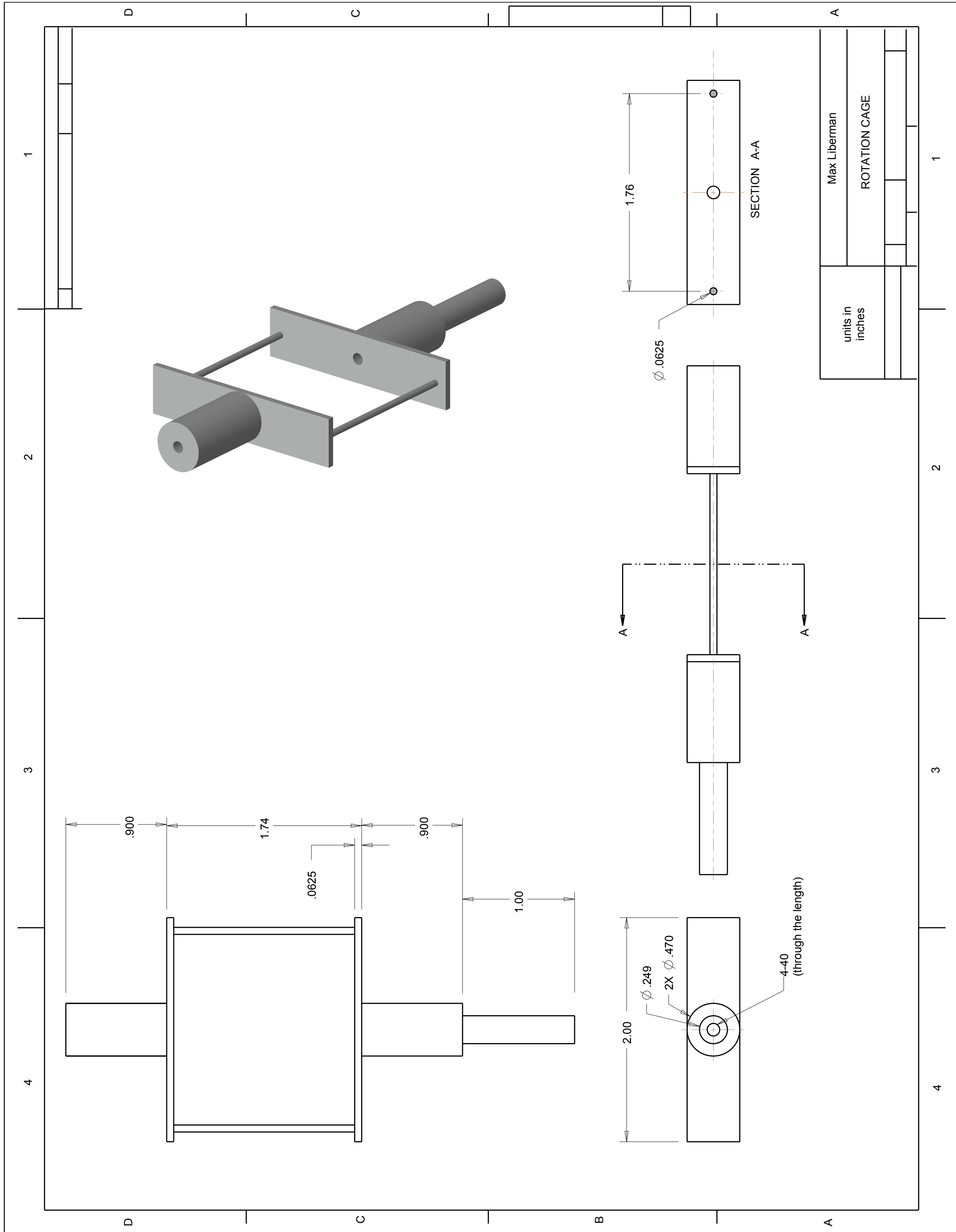

#2-56 UNC-2A
.590
.690
1.10
4X M3
.87
1.180
A
A
1.380
.250
SECTION A-A
#1/4-20 UNC-2A
.150
.590
units in inches
Max Liberman
LINEAR STAGE MOUNT


.984
2X M3
1.444
1.069
1.375
1.031
.800
.157
.341
1.000
4X R.072
R.400
2.625
.125
.250
units in inches
Max Liberman
STEPPER MOTOR MOUNT


Haydon Linear Actuator
20000 Series
Projector Clamp
Camera Mount
Linear Stage Mount
THK VRT-A
Linear Translation Stage
Stepper Motor Mount
Max Liberman
Projector Stand Assembly

A.3 Rotation Components




Max Liberman
ROTATION CAGE
units in inches
SECTION A-A
1.76
∅.0625
A
A
.900
1.74
.900
.0625
1.00
2.00
∅.249
2X ∅.470
4-40
(through the length)

B Software Source Code

B.1 Camera Control Module

Listing 1: `Flea2.java` — The `Flea2` class, an interface between the application and the FlyCapture® SDK.

```
/*
 * import packages
 */

public class Flea2 {

        private FlyCaptureFrameGrabber2 grabber;
        private String cameraName;

        public Flea2(int imW, int imH, int frameRate) {
                // try to create a Flea2 object by querying the FireWire bus.
                try {

                        // the FlyCaptureFrameGrabber2 class is a slightly edited version
                        // of the original FlyCaptureFrameGrabber class included in the
                        // JavaCV source code. It simply adds the ability to change
                        // individual parameters of the Flea2 camera.
                        grabber = new FlyCaptureFrameGrabber2(0);

                        // set the camera parameters
                        grabber.setFrameRate(frameRate);
                        grabber.setImageMode(ImageMode.COLOR);
                        grabber.setImageWidth(imW);
                        grabber.setImageHeight(imH);
                        grabber.setCameraAbsProperty(PGRFlyCapture.FLYCAPTURE_SHUTTER, 10);
                        String[] devices = FlyCaptureFrameGrabber2.getDeviceDescriptions();
                        cameraName = devices[0];
                        System.out.println("Connected to " + cameraName);

                } catch (Exception e) {
                        System.out.println("Could not connect to PGR camera device.");
                }
        }

        // start the FlyCaptureFrameGrabber
        public void start() {
                try {
                        grabber.start();
                } catch (Exception e) {
                        System.out.println("Error starting frame capture.");
                }
        }

        // stop the FlyCaptureFrameGrabber
        public void stop() {
                try {
                        grabber.stop();
                } catch (Exception e) {
                        System.out.println("Error stopping frame capture.");
                }
        }

        // set the frame size of the captured images
        public void setFrameSize(int width, int height) {
                try {
                        grabber.stop();
                        grabber.setImageWidth(width);
                        grabber.setImageHeight(height);
                        grabber.start();
                } catch (Exception e) {
                        System.out.println("Error setting frame size.");
                }
        }

        // set the camera sensor gain
        public void setGain(float value) {
                grabber.setCameraAbsProperty(PGRFlyCapture.FLYCAPTURE_GAIN, value);
        }
```

```java
        // set the camera shutter speed, in ms
        public void setShutter(float value) {
                grabber.setCameraAbsProperty(PGRFlyCapture.FLYCAPTURE_SHUTTER, value);
        }

        // set the brightness percent
        public void setBrightness(float value) {
                grabber.setCameraAbsProperty(PGRFlyCapture.FLYCAPTURE_BRIGHTNESS, value);
        }

        public String getCameraName() {
                return cameraName;
        }

        // release the FlyCaptureFrameGrabber object at program termination
        public void release() {
                try {
                        grabber.release();
                } catch (Exception e) {
                        System.out.println("Error releasing Flea2.");
                }
        }

        // get the current frame in the Flea2's buffer
        public BufferedImage getFrame() {
                try {
                        IplImage iplImage = grabber.grab();
                        return _getBufferedImage(iplImage);
                } catch (Exception e) {
                        System.out.println("Error retrieving frame.");
                        return null;
                }

        }

        // helper method to convert the OpenCV IplImage object returned
        // by grab() into a BufferedImage object.
        private BufferedImage _getBufferedImage(IplImage image) {
                BufferedImage finalImage = image.getBufferedImage();
                WritableRaster raster = finalImage.getRaster();
                int r, b, x, y;
                for (y = 0; y < image.height(); y++) {
                        for (x = 0; x < image.width(); x++) {
                                r = raster.getSample(x, y, 0);
                                b = raster.getSample(x, y, 2);
                                raster.setSample(x, y, 0, b);
                                raster.setSample(x, y, 2, r);
                        }
                }
                return finalImage;
        }

}
```

B.2 Stepper Control Module

Listing 2: `Stepper.java` — The `Stepper` class, an interface between the application and the Arduino Uno controlling the stepper motor, via a serial communication port on the host computer.

```
/*
 * import packages
 */

public class Stepper {

        private                 SerialPort          serialPort      = null;
        private                 String              port            = null;
        private                 OutputStream        output          = null;
        private static final    int                 TIME_OUT        = 2000;
        private static final    int                 DATA_RATE       = 9600;
        private                 boolean             live            = false;
        public static final     int                 DIR_OUT         = 0;
        public static final     int                 DIR_IN          = 1;
        private                 int                 _location       = 0;
        private                 int                 _direction      = 1;
        public static final     int                 TRAVEL          = 4500;

        // initialize a COM port serial communication if a valid port is found
        public Stepper(String com) {

                // get the host computer COM port identifiers
                port = com;
                CommPortIdentifier portId = null;
                Enumeration portEnum = CommPortIdentifier.getPortIdentifiers();

                // iterate through the valid COM ports
                while (portEnum.hasMoreElements()) {
                        CommPortIdentifier currPortId = (CommPortIdentifier) portEnum.
                            nextElement();
                        String currPortName = currPortId.getName();

                        // if the port is equal to the target port, break
                        if (currPortName.equals(port)) {
                                portId = currPortId;
                                break;
                        }
                }

                // if the target port was not found, log an error message and return
                if (portId == null) {
                        System.out.println("Could not establish communication with " + port);
                        return;
                }

                // try to open serial communication to the target COM port
                try {
                        serialPort = (SerialPort) portId.open(this.getClass().getName(),
                            TIME_OUT);

                        // configure the serial port for communication with Arduino Uno
                        serialPort.setSerialPortParams(DATA_RATE,
                                                       SerialPort.DATABITS_8,
                                                       SerialPort.STOPBITS_1,
                                                       SerialPort.PARITY_NONE);
                        output = serialPort.getOutputStream();
                        live = true;

                        // "dead" period before communication with serial port can occur
                        Thread.sleep(1500);

                } catch (Exception e) {
                        System.err.println(e.toString());
                }
        }

        // return stepper to its zero position and close its serial port
        public void close() {
                goTo(0);
```

```
		if (serialPort != null) {
			serialPort.close();
		}
	}

	public boolean isValid() {
		return live;
	}

	// The stepper stores its current location as an integer variable
	// of microsteps. The zero position is considered the extreme of the
	// stepper's travel range. To send the stepper to a certain location,
	// it checks against its current location and continues stepping until
	// it reaches the target.
	public void goTo(int l) {

		// check if the target location exceeds the stepper's travel range,
		// or if the stepper is already at its target location.
		if (l < 0 || l > TRAVEL || l == _location) {
			return;
		}

		// if the current location is greater than the target location, set
		// the direction in and step until the target is reached.
		if (_location > l) {
			setDirection(DIR_IN);
			while (_location > l) {
				stepSafe();
			}
		}

		// if the current location is less than the target location, set
		// the direction out and step until the target is reached.
		else {
			setDirection(DIR_OUT);
			while (_location < l) {
				stepSafe();
			}
		}
	}

	/*
	 * The Arduino board is set to loop, waiting for one incoming byte.
	 * The first options pair corresponds to the STEP_PIN command (0 for
	 * LOW, 1 for HIGH). The second option pair corresponds to the
	 * DIR_PIN command (2 for LOW, 3 for HIGH).
	 */

	// a safe stepping method that checks the target location against its
	// travel bounds before stepping.
	public void stepSafe() {

		// step only if the position after a step in its current direction
		// remains within its travel bounds
		if (_location + _direction >= 0 && _location + _direction <= TRAVEL) {

			// write a digital pulse to the STEP_PIN of the Arduino, and
			// update the stepper's location.
			try {
				output.write(1);
				output.write(0);
				_location += _direction;
			} catch (Exception e) {
				System.out.println("Unable to write to " + port + ".");
			}
		}
	}

	// an unsafe stepping method that doesn't check the target location against
	// its travel bounds before stepping.
	public void stepUnsafe() {

		// write a digital pulse to the STEP_PIN of the Arduino, and
		// update the stepper's location.
		try {
			output.write(1);
			output.write(0);
```

```
				_location += _direction;
			} catch (Exception e) {
				System.out.println("Unable to write to " + port + ".");
			}
	}

	public void zeroLocation() {
		_location = 0;
	}

	public int getLocation() {
		return _location;
	}

	// Set the direction of the stepper motor. The directions are represented
	// in the Java application as 0 and 1 for out and in, respectively, but
	// the corresponding Arduino commands are 2 and 3, respectively.
	public void setDirection(int d) {

		// convert the target direction to the Arduino command integer
		int dir = 2+d;

		// write the direction to the serial communication stream, and
		// update the local variable.
		try {
			output.write(dir);
			if (d == DIR_OUT) {
				_direction = 1;
			} else {
				_direction = -1;
			}
		} catch (IOException e) {
			System.out.println("Unable to write to " + port + ".");
		}
	}

}
```

B.3 Data Analysis Module

Listing 3: `DataAnalysis.java` — The `DataAnalysis` library class, containing all the functions used to process the input images, reconstruct 3D data, and output the results.

```
/*
 * import packages
 */

public class DataAnalysis {

	// Compute the minimum and maximum intensities of the image
	// sequence contained in the directory argument.
	public static void computeMinMax(File dir, CvMat min, CvMat max) {

		// Find all the appropriately named files in the directory,
		// i.e., the image files of the sequence.
		File files[] = dir.listFiles(new FilenameFilter() {
			public boolean accept(File path, String name) {
				return (name.toLowerCase().startsWith("frame"));
			}
		});

		// Initialize the minimum and maximum intensity matrices
		cvSet(min, cvRealScalar(255), null);
		cvSet(max, cvRealScalar(0), null);

		// Iterate through the image files and update the minimum
		// and maximum intensity matrices
		for (int i = 0; i < files.length; i++) {
			updateMinMax(_loadImageAs8BitMat(files[i]), min, max);
		}
	}

	// Helper method to update the minimum and maximum intensity matrices
	// given a new frame and the current minimum and maximum intensity
	// matrices.
	public static void updateMinMax(CvMat frame, CvMat min, CvMat max) {

		// Get the image dimensions.
		int cols = frame.cols();
		int rows = frame.rows();

		// Create a mask for the pixels in the frame that are smaller than
		// the corresponding minimum pixel values.
		CvMat minMask = cvCreateMat(rows, cols, CV_8UC1);
		cvCmp(frame, min, minMask, CV_CMP_LT);

		// Set the appropriate minimum intensity matrix values to their
		// new values.
		cvSub(min, min, min, minMask);
		cvAdd(min, frame, min, minMask);

		// Create a mask for the pixels in the frame that are larger than
		// the corresponding maximum pixel values.
		CvMat maxMask = cvCreateMat(rows, cols, CV_8UC1);
		cvCmp(frame, max, maxMask, CV_CMP_GT);

		// Set the appropriate maximum intensity matrix values to their
		// new values.
		cvSub(max, max, max, maxMask);
		cvAdd(max, frame, max, maxMask);

	}

	// For each pixel, compute the interpolated frame value at which the light plane
	// of the projector initially passes over it, store them in the crossover matrix.
	// The threshold matrix stores the average of the minimum and maximum intensities
	// for each pixel, which is used as the threshold value to determine when the
	// light plane has crossed a pixel. The mask matrix is a bitmask that zeros out
	// the pixels that don't meet the minimum contrast requirement between the
	// minimum and maximum image intensities.
	public static void computeTemporalEdge(File dir, CvMat threshold, CvMat mask, CvMat
	    crossover) {
```

```
        // Find all the appropriately named image files in the directory.
        File files[] = dir.listFiles(new FilenameFilter() {
                public boolean accept(File path, String name) {
                        return (name.toLowerCase().startsWith("frame"));
                }
        });

        // Get the image dimensions.
        int rows = threshold.rows();
        int cols = threshold.cols();
        int N = files.length;

        // Initialize the crossover matrix to 0
        cvSet(crossover, cvRealScalar(0), null);

        // Create a bitmask to keep track of which pixels have been
        // assigned frame values. Initialize to logical 1.
        CvMat empty = cvCreateMat(rows, cols, CV_8UC1);
        cvSet(empty, cvRealScalar(255), null);

        // Matrices to hold the current frame and previous frame
        CvMat frameMat;
        CvMat framePreMat = _loadImageAs32BitMat(files[0]);

        // Temporary matrices to hold intermediate interpolation data.
        CvMat above = cvCreateMat(rows, cols, CV_8UC1);
        CvMat interpolateMask = cvCreateMat(rows, cols, CV_8UC1);
        CvMat buffer1 = cvCreateMat(rows, cols, CV_32FC1);
        CvMat buffer2 = cvCreateMat(rows, cols, CV_32FC1);

        // Iterate through the images
        for (int i = 1; i < N; i++) {

                // Load the image and compare it to the shadow threshold
                frameMat = _loadImageAs32BitMat(files[i]);
                cvCmp(frameMat, threshold, above, CV_CMP_GT);

                // Store a bitmask for the pixels that are above the threshold
                // and haven't been assigned a crossover value yet
                cvAnd(above, empty, interpolateMask, null);
                cvAnd(interpolateMask, mask, interpolateMask, null);

                // Interpolate the floating point frame value at which the pixel
                // was exactly equal to its threshold.
                cvSub(threshold, framePreMat, buffer1, interpolateMask);
                cvSub(frameMat, framePreMat, buffer2, interpolateMask);
                cvDiv(buffer1, buffer2, buffer1, 1.0);

                // Add the values into the crossover matrix.
                cvAddS(buffer1, cvRealScalar(i), crossover, interpolateMask);

                // Update the empty matrix
                cvSubS(empty, cvRealScalar(255), empty, interpolateMask);

        }

        // Clamp the frame values between 1 and N
        CvMat clampMask = cvCreateMat(rows, cols, CV_8UC1);
        cvInRangeS(crossover, cvRealScalar(1), cvRealScalar(N), clampMask);
        cvNot(clampMask, clampMask);
        cvSet(crossover, cvRealScalar(0), clampMask);

}

// Helper method to load an 8-bit grayscale image from file
private static CvMat _loadImageAs8BitMat(File file) {
        IplImage image = cvLoadImage(file.getPath());
        IplImage grayImage = cvCreateImage(image.cvSize(), IPL_DEPTH_8U, 1);
        cvCvtColor(image, grayImage, CV_RGB2GRAY);
        return grayImage.asCvMat();
}

// Helper method to load a 32-bit RGBA image from file
private static CvMat _loadImageAs32BitMat(File file) {
        IplImage image = cvLoadImage(file.getPath());
        IplImage grayImage = cvCreateImage(image.cvSize(), IPL_DEPTH_8U, 1);
```

```java
        cvCvtColor(image, grayImage, CV_RGB2GRAY);
        CvMat grayFloat = cvCreateMat(image.height(), image.width(), CV_32FC1);
        grayFloat.put(grayImage.asCvMat().get());
        return grayFloat;
}

// Use the crossover matrix and the calibration file to compute the reconstruction
// of the scanned object
public static void computeReconstruction(CvMat crossover, Calibration calibration,
    File imageDir) {

        // Compute how many pixels are recoverable, and create a coordinate array
        int N = _countElementsAboveThreshold(crossover, 1);
        int[] imageCoords = new int[2*N];
        double[] crossoverArray = new double[N];
        _getPixelCoordinatesAndCrossover(crossover, imageCoords, crossoverArray);

        // Convert the image coordinates of the recoverable pixels to pixel rays.
        double[] pixelRays = pixelCoordToRay(imageCoords, calibration);

        // Use the calibration file to get the projector plane normal and the
        // function to track its pattern corner, and thus its reference points.
        double[] cornerFunction = _computeCornerFunction(imageDir, calibration);
        double[] normal = _getProjectorNormal(imageDir, calibration);
        double[] referencePoints = _getPlaneReferencePoints(crossoverArray,
            cornerFunction, calibration);

        // Convert the double arrays for the projector normals and reference points
        // into CvMat objects.
        CvMat nT = cvCreateMat(1, 3, CV_32FC1);
        nT.put(normal);
        CvMat qT = cvCreateMat(N, 3, CV_32FC1);
        qT.put(referencePoints);
        CvMat q = cvCreateMat(3, N, CV_32FC1);
        cvTranspose(qT, q);
        CvMat T = cvCreateMat(3, 1, CV_32FC1);
        T.put(calibration.getCameraT());
        CvMat Trep = cvCreateMat(3, N, CV_32FC1);
        cvRepeat(T, Trep);
        CvMat num = cvCreateMat(1, N, CV_32FC1);

        // Compute the numerator of the lambda equation for the recoverable pixels.
        cvSub(q, Trep, q, null);
        cvMatMulAdd(nT, q, null, num);

        // Get the camera calibration parameters and convert the pixel ray array
        // into CvMat objects.
        CvMat R = cvCreateMat(3, 3, CV_32FC1);
        R.put(calibration.getCameraR());
        CvMat xT = cvCreateMat(N, 3, CV_32FC1);
        xT.put(pixelRays);
        CvMat x = cvCreateMat(3, N, CV_32FC1);
        cvTranspose(xT, x);
        CvMat Rx = cvCreateMat(3, N, CV_32FC1);
        cvMatMulAdd(R, x, null, Rx);

        // Compute the denominator of the lambda equation for the recoverable pixels.
        CvMat den = cvCreateMat(1, N, CV_32FC1);
        cvMatMulAdd(nT, Rx, null, den);

        // Compute the lambda values for the recoverable pixels.
        CvMat lambdas = cvCreateMat(1, N, CV_32FC1);
        cvDiv(num, den, lambdas, 1);

        // Use the lambda values to compute the 3D position of the
        // recoverable pixel values.
        CvMat xP = cvCreateMat(3, N, CV_32FC1);
        CvMat lambdasRep = cvCreateMat(3, N, CV_32FC1);
        cvRepeat(lambdas, lambdasRep);
        cvMul(lambdasRep, Rx, xP, 1);
        cvAdd(xP, Trep, xP, null);
        CvMat outputX = cvCreateMat(N, 3, CV_32FC1);
        cvTranspose(xP, outputX);

        System.out.println(N + " points reconstructed.");

        // Output the point cloud in PLY file format.
```

```
		_saveToPly(outputX, imageDir);
	}

	// Helper method to output the point cloud in PLY file format.
	private static void _saveToPly(CvMat pointCloud, File imageDir) {

		// Get the dimensions of the point cloud matrix, and its raw data.
		int rows = pointCloud.rows();
		int cols = pointCloud.cols();
		double[] data = pointCloud.get();

		// Create a new PLY file based on the PLY files that already
		// exist in the image directory.
		File files[] = imageDir.listFiles(new FilenameFilter() {
			public boolean accept(File path, String name) {
				return (name.toLowerCase().startsWith("reconstruction"));
			}
		});
		String fileName = "";
		boolean done = false;
		for (int i = 0; i < files.length+1; i++) {
			fileName = String.format("reconstruction%02d.ply", i+1);
			for (int j = 0; j < files.length; j++) {
				String dir = files[j].getName();
				if (fileName.equals(dir)) {
					break;
				}
				if (j == files.length-1) done = true;
			}
			if (done) break;
		}
		File plyFile = new File(imageDir.getPath() + File.separator + fileName);
		System.out.println("Saving to " + plyFile.getPath());

		// write the PLY file.
		try {
			PrintWriter out = new PrintWriter(new FileWriter(plyFile));

			// standard PLY file header
			out.print("ply\n"+
					"format ascii 1.0\n"+
					"element vertex " + rows + "\n"+
					"property float x\n"+
					"property float y\n"+
					"property float z\n"+
					"element face 0\n"+
					"property list uchar int vertex_indices\n"+
					"end_header\n");

			// print the 3D point locations, one vertex per line
			for (int i = 0, row = 0; row < rows; row++) {
				for (int col = 0; col < cols; col++, i++) {
					out.print(String.format("%6.6g\t", data[i]));
				}
				out.print("\n");
			}
			out.close();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	// Helper method to count the elements of a matrix above a certain threshold.
	private static int _countElementsAboveThreshold(CvMat matrix, double t) {

		CvMat nonzero = cvCreateMat(matrix.rows(), matrix.cols(), CV_8UC1);
		cvSet(nonzero, cvRealScalar(0), null);
		cvCmpS(matrix, t, nonzero, CV_CMP_GE);
		cvSet(nonzero, cvRealScalar(1), nonzero);

		return (int)cvSum(nonzero).val(0);
	}

	// Helper function to compute the linear function that describes the translation
	// of the projector in terms of frame number. This function imports the calibration
	    file
	// built before.
```

```java
	private static double[] _computeCornerFunction(File file, Calibration calibration) {
		double[] cornerFunction = new double[4];

		try {
			Scanner scanner = new Scanner(new File(file.getPath() + File.separator
				 + "projector.txt"));
			while(!scanner.hasNextInt()) { if(scanner.hasNext()) { scanner.next();
				 } }
			int frame1 = scanner.nextInt();
			int x1 = scanner.nextInt();
			int y1 = scanner.nextInt();
			while(!scanner.hasNextInt()) { if(scanner.hasNext()) { scanner.next();
				 } }
			int frame2 = scanner.nextInt();
			int x2 = scanner.nextInt();
			int y2 = scanner.nextInt();
			scanner.close();

			System.out.println(x1 + " " + y1 + " " + x2 + " " + y2);

			int[] points = {x1, y1, x2, y2};
			double[] worldPoints = pointsOnWorldHorizontal(points, calibration);

			cornerFunction[0] = ((worldPoints[3] - worldPoints[0])/(frame2 -
				frame1));
			cornerFunction[1] = worldPoints[0] - cornerFunction[0]*frame1;
			cornerFunction[2] = ((worldPoints[4] - worldPoints[1])/(frame2 -
				frame1));
			cornerFunction[3] = worldPoints[1] - cornerFunction[0]*frame1;

		} catch (IOException e) {
			e.printStackTrace();
		}

		return cornerFunction;
	}

	// Helper method to compute the projector normal. This function imports the
		calibration file
	// built before.
	private static double[] _getProjectorNormal(File imageDir, Calibration calibration) {

		// read the calibration file
		try {
			Scanner scanner = new Scanner(new File(imageDir.getPath() + File.
				separator + "projector.txt"));
			scanner.nextLine();
			scanner.nextLine();
			while(!scanner.hasNextInt()) { if(scanner.hasNext()) { scanner.next();
				 } }
			int[] corners = new int[6];
			for (int i = 0; i < 6; i++) {
				corners[i] = scanner.nextInt();
			}
			scanner.close();
			double[] cornerPoints = pointsOnWorldHorizontal(corners, calibration);
			calibrateProjector(calibration, cornerPoints);
		} catch (IOException e) {
			e.printStackTrace();
		}

		// Compute the normal of the light plane.
		CvMat v1 = cvCreateMat(3, 1, CV_32FC1);
		double[] cornerT = calibration.getProjCornerT();
		v1.put(cornerT);
		CvMat v2 = cvCreateMat(3, 1, CV_32FC1);
		double[] focalT = calibration.getProjFocalT();
		v2.put(focalT);
		CvMat normalM = cvCreateMat(3, 1, CV_32FC1);
		cvCrossProduct(v1, v2, normalM);

		return normalM.get();

	}

	// Helper method to compute the reference points for the projector plane based on
	// its translation function and the camera calibration.
```

```
private static double[] _getPlaneReferencePoints(double[] crossover, double[] cFunc,
     Calibration calibration) {
        int N = crossover.length;
        double[] points = new double[3*N];
        double frame;
        for (int i = 0; i < N; i++) {
                frame = crossover[i];
                points[3*i] = cFunc[0]*frame+cFunc[1];
                points[3*i+1] = cFunc[2]*frame+cFunc[3];
                points[3*i+2] = 0.0;
        }
        return points;
}

// Perform a bilinear transformation to remap an arbitrary quadrilateral portion
// of an image onto a rectangle.
public static IplImage zoomToRect(double[] corners, IplImage fullColor, CvMat trans) {

        // Get image dimensions.
        double width = (double)fullColor.width();
        double height = (double)fullColor.height();

        // Get the corners of the quadrilateral image portion.
        CvPoint2D32f src = new CvPoint2D32f(4);
        src.put(corners);
        CvPoint2D32f dst = new CvPoint2D32f(4);
        double[] imCorners = {0.0, height, 0.0, 0.0, width, 0.0, width, height};
        dst.put(imCorners);

        // Get the transformation to map these four corners to the four
        // corners of a rectangle.
        cvGetPerspectiveTransform(src, dst, trans);

        // Apply the transformation to the color image, and convert to gray.
        IplImage zoomColor = cvCreateImage(fullColor.cvSize(), IPL_DEPTH_8U, 3);
        cvWarpPerspective(fullColor, zoomColor, trans, CV_INTER_LINEAR +
             CV_WARP_FILL_OUTLIERS, cvScalarAll(0));
        IplImage zoomGray = cvCreateImage(zoomColor.cvSize(), IPL_DEPTH_8U, 1);
        cvCvtColor(zoomColor, zoomGray, CV_RGB2GRAY);
        cvReleaseImage(zoomColor);

        return zoomGray;
}

// Once the calibration corners are detected, invert them back to image
// coordinates so they can be drawn on the UI panel.
public static void invertCorners(CvPoint2D32f corners, CvMat forwardTrans) {
        int N = corners.capacity();
        CvMat initialCorners = cvCreateMat(N, 1, CV_32FC2);
        initialCorners.put(corners.get());
        CvMat finalCorners = cvCreateMat(N, 1, CV_32FC2);

        CvMat backTrans = cvCreateMat(3, 3, CV_32FC1);
        cvInvert(forwardTrans, backTrans, CV_LU);

        cvPerspectiveTransform(initialCorners, finalCorners, backTrans);
        corners.put(finalCorners.get());
}

// Helper method to compute the sum of a vector's elements.
private static int _vectorSum(int[] vector) {
        int sum = 0;
        for (int i = 0; i < vector.length; i++) {
                sum += vector[i];
        }
        return sum;
}

// Calibrate the intrinsic and extrinsic parameters of the camera.
public static void calibrateCamera(Calibration calibration, double[] imagePoints,
     double[] objectPoints, int[] pointCounts, Dimension imageSize) {

        int successes = 0;
        for (int i = 0; i < pointCounts.length; i++) {
                if (pointCounts[i] != 0) {
                        successes++;
                }
```

```
        }

        // create the matrices to store the image coordinates and spatial coordinates
            of
        // the checkerboard corners.
        int corners = _vectorSum(pointCounts);
        CvMat iPts = cvCreateMat(corners, 1, CV_32FC2);
        iPts.put(Arrays.copyOf(imagePoints, corners*2));
        CvMat oPts = cvCreateMat(corners, 1, CV_32FC3);
        oPts.put(Arrays.copyOf(objectPoints, corners*3));
        CvMat pCts = cvCreateMat(successes, 1, CV_32SC1);
        for (int i = 0; i < successes; i++) {
                pCts.put(i, 0, pointCounts[i]);
        }

        // create an initial estimate for the intrinsic parameter matrix
        CvMat intrinsicMatrix = cvCreateMat(3, 3, CV_32FC1);
        intrinsicMatrix.put(0, 0, 1.0);                                        //
            focal length x;
        intrinsicMatrix.put(0, 1, 0.0);                                        //
            skew;
        intrinsicMatrix.put(0, 2, imageSize.width/2);           // center x;
        intrinsicMatrix.put(1, 0, 0.0);
        intrinsicMatrix.put(1, 1, 1.0);                                        //
            focal length y;
        intrinsicMatrix.put(1, 2, imageSize.height/2);          // center y;
        intrinsicMatrix.put(2, 0, 0.0);
        intrinsicMatrix.put(2, 1, 0.0);
        intrinsicMatrix.put(2, 1, 1.0);
        CvMat distortionCoeffs = cvCreateMat(4, 1, CV_32FC1);

        CvMat rVecs = cvCreateMat(successes, 1, CV_32FC3);
        CvMat tVecs = cvCreateMat(successes, 1, CV_32FC3);

        // use the OpenCV calib3d function to calibrate the camera parameters.
        cvCalibrateCamera2(oPts, iPts, pCts, cvSize(imageSize.width, imageSize.height)
            , intrinsicMatrix, distortionCoeffs, rVecs, tVecs, 0);

        // Get the results as arrays
        double[] cameraK = intrinsicMatrix.get();
        double[] distCoeff = distortionCoeffs.get();

        double[] rVec = new double[3];
        rVecs.get(0, rVec);
        CvMat rotVec = cvCreateMat(1, 1, CV_32FC3);
        rotVec.put(rVec);
        CvMat rotMat = cvCreateMat(3, 3, CV_32FC1);
        cvRodrigues2(rotVec, rotMat, null);
        double[] cameraR = rotMat.get();

        double[] tVec = new double[3];
        tVecs.get(0, tVec);

        // store the results in the calibration object
        calibration.setCameraK(cameraK);
        calibration.setCameraDistortion(distCoeff);
        calibration.setCameraR(cameraR);
        calibration.setCameraT(tVec);
}

// Calibrate the project using the user defined projector pattern corners.
public static void calibrateProjector(Calibration calibration, double[] corners) {

        // Get the three slide pattern corners
        CvMat nineOclock = cvCreateMat(3, 1, CV_32FC1);
        CvMat sixOclock = cvCreateMat(3, 1, CV_32FC1);
        CvMat center = cvCreateMat(3, 1, CV_32FC1);
        for (int i = 0; i < 3; i++) {
                double[] point = {corners[3*i], corners[3*i+1], corners[3*i+2]};
                switch(i) {
                case 0:
                        nineOclock.put(point);
                        break;
                case 1:
                        center.put(point);
                        break;
                case 2:
```

```
                        sixOclock.put(point);
                        break;
                    }
            }

            // Calculate the T_f vector using the projector calibration method
            double d1 = cvNorm(nineOclock, center, CV_L2, null);
            double d2 = cvNorm(sixOclock, center, CV_L2, null);

            double focal = 50.0;
            double hs = .15*25.4;
            double xb = 102.32;
            double hp = hs*focal/xb;
            double xf = (hp/hs)*focal;
            double alpha = Math.atan(hs/focal);

            double ratio1 = hp/d1; if (ratio1 > 1) ratio1 = 1/ratio1;
            double ratio2 = hp/d2; if (ratio2 > 1) ratio2 = 1/ratio2;

            double theta1 = Math.acos(ratio1*Math.cos(alpha))-alpha;
            double theta2 = Math.acos(ratio2*Math.cos(alpha))-alpha;

            CvMat e1 = cvCreateMat(3, 1, CV_32FC1);
            CvMat e2 = cvCreateMat(3, 1, CV_32FC1);
            CvMat e3 = cvCreateMat(3, 1, CV_32FC1);

            CvMat p1 = cvCreateMat(3, 1, CV_32FC1);
            CvMat p2 = cvCreateMat(3, 1, CV_32FC1);

            cvSub(nineOclock, center, p1, null);
            cvSub(sixOclock, center, p2, null);

            cvConvertScale(p1, e1, 1.0/d1, 0);
            cvConvertScale(p2, e2, 1.0/d2, 0);
            cvCrossProduct(e1, e2, e3);

            CvMat v1 = cvCreateMat(3, 1, CV_32FC1);
            CvMat v2 = cvCreateMat(3, 1, CV_32FC1);
            CvMat v3 = cvCreateMat(3, 1, CV_32FC1);

            cvConvertScale(e1, v1, Math.cos(theta1), 0);
            cvConvertScale(e2, v2, Math.cos(theta2), 0);
            cvScaleAdd(e3, cvRealScalar(Math.sin(theta1)), v1, v1);
            cvScaleAdd(e3, cvRealScalar(Math.sin(theta2)), v2, v2);
            cvCrossProduct(v1, v2, v3);

            System.out.println(cvNorm(v1, null, CV_L2, null) + " " + cvNorm(v2, null,
                 CV_L2, null));

            if (v3.get(2) < 0) {
                    cvConvertScale(v3, v3, -xf, 0);
            } else {
                    cvConvertScale(v3, v3, xf, 0);
            }

            calibration.setProjCornerT(p2.get());
            calibration.setProjFocalT(v3.get());


    }

    // Helper method to get the points on the z=0 world plane given their
    // projection onto the image plane.
    public static double[] pointsOnWorldHorizontal(int[] points, Calibration calibration)
         {
            int N = points.length/2;
            double[] rays = pixelCoordToRay(points, calibration);

            CvMat raysMat = cvCreateMat(N, 3, CV_32FC1);
            raysMat.put(rays);
            CvMat raysMatT = cvCreateMat(3, N, CV_32FC1);
            cvTranspose(raysMat, raysMatT);

            CvMat R = cvCreateMat(3, 3, CV_32FC1);
            R.put(calibration.getCameraR());
            CvMat T1 = cvCreateMat(3, 1, CV_32FC1);
```

```
        T1.put(calibration.getCameraT());
        CvMat T = cvCreateMat(3, N, CV_32FC1);
        cvRepeat(T1, T);

        CvMat rotatedRaysT = cvCreateMat(3, N, CV_32FC1);
        cvMatMulAdd(R, raysMatT, null, rotatedRaysT);

        CvMat rotatedZs = cvCreateMat(1, N, CV_32FC1);
        cvGetRow(rotatedRaysT, rotatedZs, 2);
        CvMat translateZs = cvCreateMat(1, N, CV_32FC1);
        cvGetRow(T, translateZs, 2);
        cvConvertScale(translateZs, translateZs, -1.0, 0);

        CvMat lambdas = cvCreateMat(1, N, CV_32FC1);
        cvDiv(translateZs, rotatedZs, lambdas, 1);

        CvMat lambdaMat = cvCreateMat(3, N, CV_32FC1);
        cvRepeat(lambdas, lambdaMat);

        cvConvertScale(translateZs, translateZs, -1.0, 0);
        CvMat worldPointsMatT = cvCreateMat(3, N, CV_32FC1);
        cvMul(lambdaMat, rotatedRaysT, rotatedRaysT, 1);
        cvAdd(rotatedRaysT, T, worldPointsMatT, null);

        CvMat worldPointsMat = cvCreateMat(N, 3, CV_32FC1);
        cvTranspose(worldPointsMatT, worldPointsMat);

        double[] newPoints = worldPointsMat.get();

        return newPoints;
}

// Helper method to transform 2D pixel coordinates into 3D pixel rays
// originating at the pinhole model's aperture.
public static double[] pixelCoordToRay(int[] points, Calibration calibration) {
        int N = points.length/2;
        double[] pixels = new double[N*3];

        // Build the N x 3 pixel coordinate matrix;
        for (int i = 0; i < N; i++) {
                pixels[3*i] = (double)points[2*i];
                pixels[3*i+1] = (double)points[2*i+1];
                pixels[3*i+2] = 1.0;
        }

        // Create the inverse K matrix
        CvMat invK = cvCreateMat(3, 3, CV_32FC1);
        invK.put(calibration.getCameraInvK());

        // Create the pixel matrix, transpose for multiplication;
        CvMat pixelMat = cvCreateMat(N, 3, CV_32FC1);
        pixelMat.put(pixels);
        CvMat pixelMatT = cvCreateMat(3, N, CV_32FC1);
        cvTranspose(pixelMat, pixelMatT);

        // Multiply K^(-1)*X and transpose the result
        CvMat raysMatT = cvCreateMat(3, N, CV_32FC1);
        cvMatMulAdd(invK, pixelMatT, null, raysMatT);
        CvMat raysMat = cvCreateMat(N, 3, CV_32FC1);
        cvTranspose(raysMatT, raysMat);

        double[] rays = raysMat.get();

        return rays;
}

// Helper method to get the image coordinates and crossover frame from the crossover
    matrix.
private static void _getPixelCoordinatesAndCrossover(CvMat crossoverMat, int[]
    imageCoords, double[] crossover) {

        int width = crossoverMat.cols();
        int height = crossoverMat.rows();

        double[] crossoverData = crossoverMat.get();
        double c;
        int count = 0;
```

```
		int x, y, i;
		for(i = 0, y = 0; y < height; y++) {
			for (x = 0; x < width; x++, i++) {
				c = crossoverData[i];
				if (c >= 1) {
					crossover[count] = c;
					imageCoords[count*2] = x;
					imageCoords[count*2+1] = y;
					count++;
				}
			}
		}
	}
}
```

B.4 Scanning Control Panel

Listing 4: `ImageScanPanel.java` — The `ImageScanPanel` class controls the user interface for the scanning control panel. For brevity, code involving component layout is elided.

```
/*
 * import packages
 */

public class ImageScanPanel extends JPanel {

        private App                     _application            = null;

        private JPanel                  _controlPanel           = null;
        private ImagePanel              _imagePanel             = null;

        private Flea2                   _camera                 = null;
        private int                     _width                  = 1024;
        private int                     _height                 = 768;

        private float                   _gain                   = 20.0f;
        private float[]                 _gainBnds               = {0.0f, 20.0f};
        private float                   _shutter                = 75.0f;
        private float[]                 _shutterBnds            = {0.0f, 75.0f};
        private float                   _brightness             = 0.0f;
        private float[]                 _brightnessBnds         = {0.0f, 100.0f};

        private int                     _itemHeight             = 20;
        private int                     _vSpace                 = 10;
        private JTextField              _gainField              = null;
        private JTextField              _shutterField           = null;
        private JTextField              _brightnessField        = null;
        private JButton                 _previewButton          = null;
        private Timer                   _previewTimer           = null;
        private boolean                 _isStreaming            = false;
        private int                     _frame                  = 0;

        private Stepper                 _stepper                = null;
        private String                  _stepperPort            = "COM3";
        private Timer                   _stepperTimer           = null;
        private int                     _stepperLow             = -1;
        private int                     _stepperHigh            = -1;

        private JButton                 _zeroButton             = null;
        private JButton                 _moveInButton           = null;
        private JButton                 _moveOutButton          = null;
        private JButton                 _setLowButton           = null;
        private JButton                 _setHighButton          = null;

        private JButton                 _runButton              = null;
        private boolean                 _saveToDisk             = false;
        private File                    _imageDir               = null;
        private JButton                 _reconstructButton      = null;

        private Timer                   _scanTimer              = null;
        private int                     _scanTick               = 70;
        private int                     _slowSteps              = 3;
        private int                     _fastSteps              = 10;

        public ImageScanPanel(App app, Flea2 camera) {
                super();
                _application = app;
                this.setBackground(Appearance.BACKGROUND);
                this.setLayout(new BoxLayout(this, BoxLayout.LINE_AXIS));
                this.setPreferredSize(new Dimension(1000, 600));
                _imagePanel = new ImagePanel();
                this.add(_imagePanel);
                _camera = camera;
                _addControlPanel();
        }

        public File getImageDir() {
                return _imageDir;
        }
```

```
	// Update the image panel with the image and a frame number
	private void _updateDisplay(BufferedImage image, int frame) {

		String label = "Image Size: " + _width + " x " + _height
		+ String.format("     Frame Count: %04d", frame);
		_imagePanel.setImage(image, label);

		// During the scanning process, this branch is evaluated
		// to save each captured image to disk.
		if (_saveToDisk) {
			String fileName = _imageDir.getPath() + File.separator + String.format
				("frame%04d.bmp", frame);
			File outputFile = new File(fileName);
			try {
				ImageIO.write(image, "bmp", outputFile);
			} catch (IOException e) {
				e.printStackTrace();
			}
		}
	}

	// Prepare for a new scan by creating an image directory
	private void _prepareForScan() {

		// Find all the data folders currently on the disk,
		// and create a new one by finding the first
		// available suffixed number, i.e. sequence04.
		File path = new File("data/");
		File files[] = path.listFiles(new FilenameFilter() {
			public boolean accept(File path, String name) {
				return (name.toLowerCase().startsWith("sequence"));
			}
		});
		String dirName = "";
		boolean done = false;
		for (int i = 0; i < files.length+1; i++) {
			dirName = String.format("sequence%02d", i+1);
			for (int j = 0; j < files.length; j++) {
				String dir = files[j].getName();
				if (dirName.equals(dir)) {
					break;
				}
				if (j == files.length-1) done = true;
			}
			if (done) break;
		}
		_imageDir = new File("data" + File.separator + dirName);
		boolean success = _imageDir.mkdirs();
		if (success) System.out.println("Directory created: " + _imageDir.getPath());

		// set the save flag to true
		_saveToDisk = true;
		_application.setImageDirectory(_imageDir);
	}

	// set the camera's default parameters on initialization
	private void _initializeCamera() {
		_camera.setGain(_gain);
		_camera.setShutter(_shutter);
		_camera.setBrightness(_brightness);
		_camera.start();
	}

	// close the camera and stepper motor communication ports.
	// called before program termination.
	public void closePorts() {
		_stepper.close();
		_camera.stop();
		_camera.release();
	}

	public BufferedImage grabFrame() {
		return _camera.getFrame();
	}

	// start streaming the captured images onto the display panel
```

```
// in the interface
private void _startPreview() {
        _isStreaming = true;
        _frame = 0;
        _previewTimer = new Timer(10, new PreviewListener());
        _previewTimer.setActionCommand("update");
        _previewTimer.start();
}

private void _stopPreview() {
        _isStreaming = false;
        _previewTimer.stop();
}

public Dimension getImageSize() {
        return new Dimension(_width, _height);
}

// Prepare for the scan and start the scan-governing timer.
private void _runScan() {

        // Make sure the translation bounds for the active lighting system
        // are correctly set for the scan.
        if (_stepperHigh < 0 || _stepperLow < 0) {
                System.out.println("Must set first and last frame for image sequence
                    capturing.");
                return;
        } else if (_stepperHigh <= _stepperLow) {
                System.out.println("First frame must be set lower than last frame.");
                return;
        }

        // prepare for the scan and go to the starting position
        _prepareForScan();
        _stepper.goTo(_stepperLow);

        // wait half a second, then set the stepper's direction to OUT, and start
        // the timer that governs the incremental stepping of the translation stage.
        try {
                Thread.sleep(500);
                _stepper.setDirection(Stepper.DIR_OUT);
                _scanTimer = new Timer(_scanTick, new RunScannerListener(_slowSteps));
                _scanTimer.setActionCommand("scan");
                _frame = 0;
                _scanTimer.start();
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
}

// Stop the scanner and reset the scanning flags. Calculate the total stage
// translation based on the number of steps taken.
private void _stopScan() {
        _scanTimer.stop();
        _saveToDisk = false;
        float steps = (float)(_stepperHigh - _stepperLow);
        float stepSize = 0.001f/8.0f;
        float stepInch = stepSize*steps;
        float stepMm   = stepInch*25.4f;
        System.out.println("Scan finished: " + String.valueOf(stepInch) + " inches ("
            + String.valueOf(stepMm) + " mm)  for " + _frame + " frames.");
        _calibrateProjector();
}

// Interactively calibrate the projector once the scan has finished, based on the
    corners
// of the pattern projected onto the stage.
private void _calibrateProjector() {

        // Get an array of all the captured frame filenames.
        File files[] = _imageDir.listFiles(new FilenameFilter() {
                public boolean accept(File path, String name) {
                        return (name.toLowerCase().startsWith("frame"));
                }
        });

        // Waits for user input. See ProjectorCalibrationListener() below
```

```
        try {
                BufferedImage first = ImageIO.read(files[0]);
                _imagePanel.setImage(first, "Click the right-angle corner of the
                     quarter circle");
                _imagePanel.addMouseListener(new ProjectorCalibrationListener());
        } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

}

// Update the camera gain based on user input
private void _setCameraGain(float newGain) {
        float checkedGain = _checkGain(newGain);
        _gainField.setText(String.valueOf(checkedGain));
        _gain = checkedGain;
        _camera.setGain(checkedGain);
}

// Check the user-input gain value against the known
// hardware limits.
private float _checkGain(float newGain) {
        if (newGain < _gainBnds[0]) newGain = _gainBnds[0];
        else if (newGain > _gainBnds[1]) newGain = _gainBnds[1];
        return newGain;
}

// Update the camera shutter based on user input
private void _setCameraShutter(float newShutter) {
        float checkedShutter = _checkShutter(newShutter);
        _shutterField.setText(String.valueOf(checkedShutter));
        _shutter = checkedShutter;
        _camera.setShutter(checkedShutter);
}

// Check the user-input shutter value against the known
// hardware limits.
private float _checkShutter(float newShutter) {
        if (newShutter < _shutterBnds[0]) newShutter = _shutterBnds[0];
        else if (newShutter > _shutterBnds[1]) newShutter = _shutterBnds[1];
        return newShutter;
}

// Update the camera brightness based on user input.
private void _setCameraBrightness(float newBrightness) {
        float checkedBrightness = _checkBrightness(newBrightness);
        _brightnessField.setText(String.valueOf(checkedBrightness));
        _brightness = checkedBrightness;
        _camera.setBrightness(checkedBrightness);
}

// Check the user-input brightness value against the
// known hardware limits.
private float _checkBrightness(float newBrightness) {
        if (newBrightness < _brightnessBnds[0]) newBrightness = _brightnessBnds[0];
        else if (newBrightness > _brightnessBnds[1]) newBrightness = _brightnessBnds
             [1];
        return newBrightness;
}

/*
 *
 * javax.swing setup & layout functions elided
 *
 */

// Simple ActionListener to update the scanning parameters
// when the GUI text field values are changed
private class TextFieldListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                String command = e.getActionCommand();

                // The gain has been changed
                if (command.equalsIgnoreCase("gain")) {
                        String gainStr = _gainField.getText();
                        try {
```

```
                                float newGain = Float.parseFloat(gainStr);
                                _setCameraGain(newGain);
                        } catch (NumberFormatException formatE) {
                                _gainField.setText(String.valueOf(_gain));
                        }
                }

                // The shutter has been changed
                else if (command.equalsIgnoreCase("shutter")) {
                        String shutterStr = _shutterField.getText();
                        try {
                                float newShutter = Float.parseFloat(shutterStr);
                                _setCameraShutter(newShutter);
                        } catch (NumberFormatException formatE) {
                                _shutterField.setText(String.valueOf(_shutter));
                        }
                }

                // The brightness has been changed
                else if (command.equalsIgnoreCase("brightness")) {
                        String brightnessStr = _brightnessField.getText();
                        try {
                                float newBrightness = Float.parseFloat(brightnessStr);
                                _setCameraBrightness(newBrightness);
                        } catch (NumberFormatException formatE) {
                                _brightnessField.setText(String.valueOf(_brightness));
                        }
                }
        }
}

// Simple ActionListener to start and stop the live streaming
// when the PREVIEW button is activated and deactivated.
private class PreviewListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                String command = e.getActionCommand();
                if (command.equalsIgnoreCase("startPreview")) {
                        _startPreview();
                        _previewButton.setActionCommand("stopPreview");
                        _previewButton.setText("STOP");
                } else if (command.equalsIgnoreCase("stopPreview")) {
                        _stopPreview();
                        _previewButton.setActionCommand("startPreview");
                        _previewButton.setText("PREVIEW");
                } else if (command.equalsIgnoreCase("update")) {
                        _frame++;
                        _updateDisplay(_camera.getFrame(), _frame);
                }
        }
}

// MouseListener to react to the Move Stage buttons and zero the
// stage once the button has been released.
private class ZeroListener implements MouseListener {

        private boolean _beforeState = false;

        public void mouseClicked(MouseEvent e) {}

        public void mousePressed(MouseEvent e) {
                _beforeState = _isStreaming;
                if (!_beforeState) {
                        _startPreview();
                }
                _stepper.setDirection(Stepper.DIR_IN);
                _stepperTimer = new Timer(_scanTick, new StepListener());
                _stepperTimer.setActionCommand("zero");
                _stepperTimer.start();
        }

        public void mouseReleased(MouseEvent e) {
                if (!_beforeState) {
                        _stopPreview();
                }
                _stepperTimer.stop();
                _stepper.zeroLocation();
        }
```

```
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}
}


// MouseListener to continuously manually move the stage in or out
// based on user input while the button is pressed.
private class MoveListener implements MouseListener {

        private boolean _beforeState = false;

        public void mouseClicked(MouseEvent e) {}

        public void mousePressed(MouseEvent e) {
                _beforeState = _isStreaming;
                if (!_beforeState) {
                        _startPreview();
                }
                String command = ((JButton) e.getSource()).getText();
                if (command.equalsIgnoreCase("in")) {
                        _stepper.setDirection(Stepper.DIR_IN);
                } else if (command.equalsIgnoreCase("out")) {
                        _stepper.setDirection(Stepper.DIR_OUT);
                }
                _stepperTimer = new Timer(_scanTick, new StepListener());
                _stepperTimer.setActionCommand("move");
                _stepperTimer.start();
        }

        public void mouseReleased(MouseEvent e) {
                if (!_beforeState) {
                        _stopPreview();
                }
                _stepperTimer.stop();
        }

        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}

}

// Used to quickly move the active lighting system
private class StepListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                String command = e.getActionCommand();
                if (command.equalsIgnoreCase("zero")) {
                        for (int i = 0; i < _fastSteps; i++) {
                                _stepper.stepUnsafe();
                        }
                } else if (command.equalsIgnoreCase("move")) {
                        for (int i = 0; i < _fastSteps; i++) {
                                _stepper.stepSafe();
                        }
                }
        }
}

// Set the scan limits on the active lighting system
private class FrameBoundsListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                String command = e.getActionCommand();
                if (command.equalsIgnoreCase("setLow")) {
                        _stepperLow = _stepper.getLocation();
                        System.out.println("First frame set to location " +
                             _stepperLow);
                } else if (command.equalsIgnoreCase("setHigh")) {
                        _stepperHigh = _stepper.getLocation();
                        System.out.println("Last frame set to location " +
                             _stepperHigh);
                }
        }
}

// Used to start the scanning process when the user clicks the
// RUN SCANNER button.
private class RunScannerListener implements ActionListener {
```

```
        private int _steps;

        public RunScannerListener() {
                _steps = 0;
        }

        public RunScannerListener(int steps) {
                _steps = steps;
        }

        public void actionPerformed(ActionEvent e) {
                String command = e.getActionCommand();
                if (command.equalsIgnoreCase("initiate")) {
                        if (_isStreaming) {
                                _stopPreview();
                                _previewButton.setActionCommand("startPreview");
                                _previewButton.setText("PREVIEW");
                        }
                        _runScan();
                } else if (command.equalsIgnoreCase("scan")) {
                        if (_stepper.getLocation() < _stepperHigh) {
                                _frame++;
                                _updateDisplay(_camera.getFrame(), _frame);
                                for (int i = 0; i < _steps; i++) {
                                        _stepper.stepSafe();
                                }
                        } else {
                                _stopScan();
                        }
                }
        }
}

// Switch to the reconstruction panel if the user hits the
// RECONSTRUCT button.
private class ReconstructListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                String command = e.getActionCommand();
                if (command.equalsIgnoreCase("reconstruct")) {
                        _application.reconstruct();
                }
        }
}

// Interactively calibrate the projector system.
private class ProjectorCalibrationListener implements MouseListener {

        private int click = 0;
        int[] points = new int[10];
        File[] files;

        public ProjectorCalibrationListener() {
                files = _imageDir.listFiles(new FilenameFilter() {
                        public boolean accept(File path, String name) {
                                return (name.toLowerCase().startsWith("frame"));
                        }
                });
        }

        public void mouseClicked(MouseEvent e) {}

        // Get the location of the click, and store it appropriately
        public void mousePressed(MouseEvent e) {
                if (e.getButton() == 1) {
                        Point p = e.getPoint();
                        int x = (int)(p.x*_width/800.0);
                        int y = (int)(p.y*_height/600.0);
                        points[2*click] = x;
                        points[2*click+1] = y;
                        switch(click) {
                        case 0:
                                BufferedImage last;
                                try {
                                        last = ImageIO.read(files[files.length-1]);
                                        _imagePanel.setImage(last, "Click the right-
                                            angle corner of the quarter circle");
```

```
                                } catch (IOException e1) {
                                        e1.printStackTrace();
                                }
                                break;
                        case 1:
                                IplImage middle = cvLoadImage(files[2*files.length/3].
                                     getPath());
                                IplImage gray = cvCreateImage(middle.cvSize(),
                                     IPL_DEPTH_8U, 1);
                                cvCvtColor(middle, gray, CV_RGB2GRAY);
                                cvEqualizeHist(gray, gray);
                                IplImage gray2 = cvCreateImage(middle.cvSize(),
                                     IPL_DEPTH_8U, 1);
                                cvSmooth(gray, gray2, CV_BILATERAL, 15, 0, 100.0, 5.0)
                                     ;
                                cvEqualizeHist(gray2, gray2);
                                _imagePanel.setImage(gray2.getBufferedImage(), "CLICK
                                     THE THREE CORNERS OF THE QUARTER CIRCLE");
                                break;
                        case 4:
                                _imagePanel.removeMouseListener(this);
                                _savePoints();
                                _startPreview();
                        }
                        click++;
                } else {

                }
        }

        // Save the user input to the calibration file
        private void _savePoints() {
                File fileName = new File(_imageDir.getPath() + File.separator + "
                     projector.txt");
                try {
                        PrintWriter out = new PrintWriter(new FileWriter(fileName));
                        out.print(String.format("Frame %d \t%d %d\n"+
                                                                "Frame %d \t%d %d\n"+
                                                                "Calibration Corners: \t%d %d
                                                                     \t %d %d \t %d %d", 1,
                                                                points[0], points[1], files.
                                                                     length, points[2], points
                                                                     [3], points[4],
                                                                points[5], points[6], points
                                                                     [7], points[8], points
                                                                     [9]));
                        out.close();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }

        public void mouseReleased(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}

    }

}
```

B.5 Calibration Control Panel

Listing 5: `CalibrationPanel.java` — The `CalibrationPanel` class controls the user interface for the calibration control panel. For brevity, code involving component layout is elided.

```
/*
 * import packages
 */

public class CalibrationPanel extends JPanel {

        private         App                     _application                    = null;

        private         int                     _imageWidth                     = 1024;
        private         int                     _imageHeight                    = 768;
        private         int                     _vSpace                         = 16;

        private         ImagePanel              _imagePanel                     = null;
        private         Timer                   _streamTimer                    = null;
        private         JPanel                  _controlPanel                   = null;

        private         JPanel                  _cameraPanel                    = null;
        private         JComboBox               _dirField                       = null;
        private         JPanel                  _projectorPanel                 = null;
        private         int                     _cHeight                        = 250;

        private         Flea2                   _camera                         = null;

        private         File                    _calibDir                       = null;
        private         int                     _checkerImages                  = 0;
        private         int                     _successes                      = 0;
        private         Checkerboard[]          _camIntrRects                   = null;
        private         File[]                  _camIntrImages                  = null;
        private         Dimension               _patternSize                    = new
             Dimension(20, 20);

        private         Calibration             _calibration                    = null;
        private         double[]                _imagePoints                    = null;
        private         double[]                _objectPoints                   = null;
        private         int[]                   _pointCounts                    = null;


        public CalibrationPanel(App app, Flea2 camera) {
                super();
                _application = app;
                _camera = camera;
                _calibration = new Calibration();
                this.setPreferredSize(new Dimension(_imageWidth, _imageHeight));
                this.setMaximumSize(new Dimension(_imageWidth, _imageHeight));
                this.setLayout(new BoxLayout(this, BoxLayout.LINE_AXIS));
                _addPanels();
        }

        public void setCalibration(Calibration calibration) {
                _calibration = calibration;
        }

        // Prepare for the calibration process by finding all
        // the appropriately named calibration images in the
        // calibration directory. Allocate the arrays for the
        // various points, and present the checkerboard images
        // for user input.
        private void _calibrateCamera() {
                _checkerImages = 0;
                _successes = 0;
                _camIntrImages = _calibDir.listFiles(new FilenameFilter() {
                        public boolean accept(File path, String name) {
                                return (name.toLowerCase().startsWith("cam_intrinsic"));
                        }
                });
                int imageN = _camIntrImages.length;
                _camIntrRects = new Checkerboard[imageN];
                _imagePoints = new double[imageN*_patternSize.width*_patternSize.height*2];
                _objectPoints = new double[imageN*_patternSize.width*_patternSize.height*3];
                _pointCounts = new int[imageN];
```

```
        _presentCheckerboardImage(0, false);
}

// Present a checkerboard calibration image on the display
// panel, and wait for user input.
private void _presentCheckerboardImage(int i, boolean repeat) {

        // The method is called iteratively from i=0 to i=N
        if (i < _camIntrImages.length) {
                try {

                        BufferedImage im = ImageIO.read(_camIntrImages[i]);
                        String mesg = "Click the four BLACK corners of a 13 x 13
                             checkerboard.";
                        if (repeat) mesg += " ~~~ TRY AGAIN ~~~ ";
                        _imagePanel.setImage(im, mesg);
                        Checkerboard corners = new Checkerboard();

                        // Add a CornerListener (see below) to the panel
                        // to wait for user input.
                        _imagePanel.addMouseListener(new CornerListener(corners));

                } catch (IOException e) {
                        e.printStackTrace();
                }
        }

        // if i==N, then all the images have been processed, and the clicked points
        // are sent to the DataAnalysis library for calibration.
        else {
                DataAnalysis.calibrateCamera(_calibration, _imagePoints, _objectPoints
                     , _pointCounts, new Dimension(_imageWidth, _imageHeight));
                _calibration.saveToFile(_calibDir);
                _streamTimer.start();
        }
}

// Function for analysis of calibration image after the four corners of the
// 13 x 13 grid have been selected.
private void _analyzeIntrinsicImage() {

        // Get the image and the checkerboard points.
        BufferedImage image = _imagePanel.getImage();
        ArrayList<Point> points = _imagePanel.getPoints();
        Checkerboard pattern = _camIntrRects[_checkerImages];
        _imagePanel.setPoints(points);

        // Use the DataAnalysis library to map the checkerboard grid
        // up to the size of the whole image.
        IplImage fullColor = IplImage.createFrom(image);
        CvMat trans = cvCreateMat(3, 3, CV_32FC1);
        IplImage zoomGray = DataAnalysis.zoomToRect(pattern.getCorners(), fullColor,
             trans);

        CvSize pSize = cvSize(_patternSize.width, _patternSize.height);
        int N = pSize.width()*pSize.height();
        CvPoint2D32f corners = new CvPoint2D32f(N);
        int[] cornerCount = new int[1];

        // Use the OpenCV library to find the chessboard corners
        // in the transformed checkerboard patter.
        int result = cvFindChessboardCorners(zoomGray, pSize, corners, cornerCount,
             CV_CALIB_CB_ADAPTIVE_THRESH);

        // Invert the corners so they can be redrawn on the display.
        DataAnalysis.invertCorners(corners, trans);
        cvDrawChessboardCorners(fullColor, pSize, corners, cornerCount[0], result);
        cvReleaseImage(zoomGray);

        // Continue to next image and wait for user input.
        if (cornerCount[0] == N && result != 0) {
                _imagePanel.setImage(fullColor.getBufferedImage(), cornerCount[0] + "
                     Corners Extracted. CLICK to continue.");

                double[] cornerValues = corners.get();
                int x, y, i;
                int iOffset = _successes*_patternSize.width*_patternSize.height*2;
```

```java
				int oOffset = _successes*_patternSize.width*_patternSize.height*3;
				for (i = 0, y = 0; y < _patternSize.height; y++) {
					for (x = 0; x < _patternSize.width; x++, i++) {
						_imagePoints[iOffset+2*i] = cornerValues[2*i];
						_imagePoints[iOffset+2*i+1] = cornerValues[2*i+1];
						_objectPoints[oOffset+3*i] = (double)x*.1;
						_objectPoints[oOffset+3*i+1] = (double)y*.1;
						_objectPoints[oOffset+3*i+2] = 0.0;
						_pointCounts[_successes] = cornerCount[0];
					}
				}
				_checkerImages++;
				_successes++;
				_imagePanel.addMouseListener(new ContinueListener());
			} else {
				_presentCheckerboardImage(_checkerImages, true);
			}
	}

	private void _calibrateProjector() {
		IplImage ipl = IplImage.createFrom(_imagePanel.getImage());
		IplImage gray = cvCreateImage(ipl.cvSize(), IPL_DEPTH_8U, 1);
		cvCvtColor(ipl, gray, CV_RGB2GRAY);
		cvEqualizeHist(gray, gray);
		IplImage gray2 = cvCreateImage(ipl.cvSize(), IPL_DEPTH_8U, 1);
		cvSmooth(gray, gray2, CV_BILATERAL, 15, 0, 100.0, 5.0);
		cvEqualizeHist(gray2, gray2);

		_imagePanel.setImage(gray2.getBufferedImage(), "CLICK THE THREE CORNERS OF THE
		     QUARTER CIRCLE");

		_imagePanel.addMouseListener(new MoonCornerListener());

	}

	/*
	 *
	 * Component layout functions elided
	 *
	 */

	// ButtonListener to handle user input to the calibration
	// interface.
	private class ButtonListener implements ActionListener {
		public void actionPerformed(ActionEvent e) {
			String command = e.getActionCommand();

			// return to the reconstruction control panel
			if (command.equalsIgnoreCase("reconstruct")) {
				if (_calibration.isCameraCalibrated() && _calibration.
				     isProjectorCalibrated()) {
					_application.useCalibration(_calibration);
					_calibration.saveToFile(new File(_calibDir.getPath()+
					     File.separator+"calibration.txt"));
				}
				_application.reconstruct();
			}

			// show the camera calibration components
			else if (command.equalsIgnoreCase("camIntrinsic")) {
				_cameraPanel.setVisible(true);
				_projectorPanel.setVisible(false);
			}

			// show the projector calibration components
			else if (command.equalsIgnoreCase("projExtrinsic")) {
				_cameraPanel.setVisible(false);
				_projectorPanel.setVisible(true);
			}

			// capture a calibration image
			else if (command.equalsIgnoreCase("intrinsicCapture")) {
				BufferedImage image = _imagePanel.getImage();
				File file = getFileName("cam_intrinsic");
				try {
					ImageIO.write(image, "bmp", file);
				} catch (IOException e1) {
```

```
                                e1.printStackTrace();
                        }
                }

                // calibrate the intrinsic camera parameters
                else if (command.equalsIgnoreCase("intrinsicCalibrate")) {
                        _streamTimer.stop();
                        _calibrateCamera();
                }

                // calibrate the extrinsic projector parameters
                else if (command.equalsIgnoreCase("projectorCapture")) {
                        _streamTimer.stop();
                        if (!_calibration.isCameraCalibrated()) {
                                _calibration = Calibration.buildFromFile(new File(
                                    _calibDir.getPath() + File.separator + "
                                    calibration.txt"));
                        }
                        _calibrateProjector();
                }

                // Update file I/O directories
                else if (command.equalsIgnoreCase("directory")) {
                        String dir = (String)(_dirField.getSelectedItem());
                        _calibDir = new File(dir);
                } else if (command.equalsIgnoreCase("makeDirectory")) {
                        _addDirectory();
                }

        }
        private File getFileName(final String mask) {
                File files[] = _calibDir.listFiles(new FilenameFilter() {
                        public boolean accept(File path, String name) {
                                return (name.toLowerCase().startsWith(mask));
                        }
                });
                String fileName = _calibDir.getPath() + File.separator + mask + String
                    .format("%02d.bmp", files.length+1);
                return new File(fileName);
        }
}

// Update the live camera stream
private class StreamListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                String command = e.getActionCommand();
                if (command.equalsIgnoreCase("update")) {
                        _imagePanel.setImage(_camera.getFrame(), "LIVE STREAM");
                }
        }
}

// MouseListener to handle interactive calibration
private class CornerListener implements MouseListener {

        private Checkerboard _corners;
        private int _count = 0;

        public CornerListener(Checkerboard corners) {
                _corners = corners;
                _count = 0;
        }

        public void mouseClicked(MouseEvent e) {}

        // When the mouse is pressed, store the click position
        // as one of the checkerboard corners. If all four of the
        // corners have been selected, process the calibration
        // image.
        public void mousePressed(MouseEvent e) {
                if (e.getButton() == 1) {
                        _count++;
                        if (_count <= 4) {
                                Point click = e.getPoint();
                                float x = click.x*1024.0f/800.0f;
                                float y = click.y*768.0f/600.0f;
                                _imagePanel.addPoint(click);
```

```
                                        _corners.setCorner(_count, x, y);
                                        if (_corners.isDefined()) {
                                                _camIntrRects[_checkerImages] = _corners;
                                                _imagePanel.removeMouseListener(this);
                                                _analyzeIntrinsicImage();
                                        }
                                }
                        } else {
                                _imagePanel.removeMouseListener(this);
                                _checkerImages++;
                                _presentCheckerboardImage(_checkerImages, false);
                        }
                }

                public void mouseReleased(MouseEvent e) {}
                public void mouseEntered(MouseEvent e) {}
                public void mouseExited(MouseEvent e) {}

        }

        private class ContinueListener implements MouseListener {

                public void mouseClicked(MouseEvent e) {}

                public void mousePressed(MouseEvent e) {
                        _presentCheckerboardImage(_checkerImages, false);
                        _imagePanel.removeMouseListener(this);
                }

                public void mouseReleased(MouseEvent e) {}
                public void mouseEntered(MouseEvent e) {}
                public void mouseExited(MouseEvent e) {}

        }

        // MouseListener to perform interactive projector calibration
        private class MoonCornerListener implements MouseListener {

                int clicks = 0;
                int[] corners = new int[6];

                public void mouseClicked(MouseEvent e) {

                }

                // Handle the three clicks of the projector pattern corners.
                public void mousePressed(MouseEvent e) {
                        if (e.getButton() == 1) {
                                clicks++;
                                Point p = e.getPoint();
                                corners[(clicks-1)*2] = p.x;
                                corners[(clicks-1)*2+1] = p.y;
                                _imagePanel.addPoint(e.getPoint());
                                if (clicks==3) {
                                        double[] points = DataAnalysis.pointsOnWorldHorizontal
                                             (corners, _calibration);
                                        DataAnalysis.calibrateProjector(_calibration, points);
                                        _imagePanel.removeMouseListener(this);
                                        _streamTimer.start();
                                }
                        } else {
                                clicks--;
                                _imagePanel.removeLastPoint();
                        }
                }

                public void mouseReleased(MouseEvent e) {}
                public void mouseEntered(MouseEvent e) {}
                public void mouseExited(MouseEvent e) {}

        }

        // Simple class to contain the checkerboard information.
        private class Checkerboard {
                private float[] c1 = null;
                private float[] c2 = null;
                private float[] c3 = null;
```

```
		private float[] c4 = null;
		private boolean _defined;

		public Checkerboard() {
			_defined = false;
		}

		public void setCorner(int c, float x, float y) {
			switch (c) {
			case 1:
				c1 = new float[2];
				c1[0] = x;
				c1[1] = y;
				break;
			case 2:
				c2 = new float[2];
				c2[0] = x;
				c2[1] = y;
				break;
			case 3:
				c3 = new float[2];
				c3[0] = x;
				c3[1] = y;
				break;
			case 4:
				c4 = new float[2];
				c4[0] = x;
				c4[1] = y;
				break;
			}
			_checkForDefined();
		}

		private void _checkForDefined() {
			if (c1 == null || c2 == null || c3 == null || c4 == null) return;
			_defined = true;
		}

		public boolean isDefined() {
			return _defined;
		}

		public double[] getCorners() {
			double[] corners =  {c1[0], c1[1], c2[0], c2[1], c3[0], c3[1], c4[0],
			    c4[1]};
			return corners;
		}

		public String toString() {
			String result = "";
			result += c1[0] + " " + c1[1] + "\n";
			result += c2[0] + " " + c2[1] + "\n";
			result += c3[0] + " " + c3[1] + "\n";
			result += c4[0] + " " + c4[1] + "\n";
			return result;
		}

	}


}
```

Listing 6: `Calibration.java` — The `Calibration` class holds all the camera and projector calibration parameters and handles calibration file I/O.

```
/*
 * import packages
 */

public class Calibration {

	private double[]			_cameraK			= null;
	private double[]			_cameraInvK			= null;
	private double[]			_cameraDistortion		= null;
```

```
private double[]                        _cameraR                        = null;
private double[]                        _cameraT                        = null;

private double[]                        _projFocalT                     = null;
private double[]                        _projCornerT                    = null;


public Calibration() {}

public boolean isCameraCalibrated() {
        return (_cameraK != null && _cameraDistortion != null && _cameraR != null &&
             _cameraT != null);
}

public boolean isProjectorCalibrated() {
        return (_projFocalT != null && _projCornerT != null);
}

public void setCameraK(double[] k) {
        _cameraK = new double[k.length];
        _cameraInvK = new double[k.length];

        CvMat K = cvCreateMat(3, 3, CV_32FC1);
        K.put(k);
        cvInvert(K, K, CV_LU);
        double[] kinv = K.get();

        for (int i = 0; i < k.length; i++) {
                _cameraK[i] = k[i];
                _cameraInvK[i] = kinv[i];
        }
}

public double[] getCameraK() {
        return _cameraK;
}

public double[] getCameraInvK() {
        return _cameraInvK;
}

public void setCameraDistortion(double[] r) {
        _cameraDistortion = new double[r.length];
        for (int i = 0; i < r.length; i++) {
                _cameraDistortion[i] = r[i];
        }
}

public double[] getCameraDistortion() {
        return _cameraDistortion;
}

public void setCameraR(double[] rot) {
        _cameraR = new double[rot.length];
        for (int i = 0; i < rot.length; i++) {
                _cameraR[i] = rot[i];
        }
}

public double[] getCameraR() {
        return _cameraR;
}

public void setCameraT(double[] trans) {
        _cameraT = new double[trans.length];
        for (int i = 0; i < trans.length; i++) {
                _cameraT[i] = trans[i];
        }
}

public double[] getCameraT() {
        return _cameraT;
}

public void printCameraCalibration() {
        printCameraK();
        printCameraDistortion();
```

```
                printCameraR();
                printCameraT();
        }

        public void printCameraK() {
                if (_cameraK != null) {
                        String out = "  Camera K Matrix:\n";
                        for (int i = 0; i < 3; i ++) {
                                for (int j = 0; j < 3; j++) {
                                        out += String.format("%.4f\t", _cameraK[i*3+j]);
                                }
                                out += "\n";
                        }
                        System.out.println(out);
                } else {
                        System.out.println("Camera K matrix undefined.");
                }
        }

        public void printCameraDistortion() {
                if (_cameraDistortion != null) {
                        String out = "  Distortion Coefficients:\n";
                        for (int i = 0; i < 4; i++) {
                                out += String.format("%.3f\t", _cameraDistortion[i]);
                        }
                        System.out.println(out + "\n");
                } else {
                        System.out.println("Camera distortion coefficients undefined.");
                }
        }

        public void printCameraR() {
                if (_cameraR != null) {
                        String out = "  Camera Rotation Matrix:\n";
                        for (int i = 0; i < 3; i ++) {
                                for (int j = 0; j < 3; j++) {
                                        out += String.format("%.4f\t", _cameraR[i*3+j]);
                                }
                                out += "\n";
                        }
                        System.out.println(out);
                } else {
                        System.out.println("Camera rotation matrix undefined.");
                }
        }

        public void printCameraT() {
                if (_cameraT != null) {
                        String out = "  Camera Translation Vector:\n";
                        for (int i = 0; i < 3; i ++) {
                                out += String.format("%.4f\n", _cameraT[i]);
                        }
                        System.out.println(out + "\n");
                } else {
                        System.out.println("Camera translation vector undefined.");
                }
        }

        public int saveToFile(File directory) {
                try {
                        File fileName = new File(directory.getPath() + File.separator + "
                            calibration.txt");
                        PrintWriter out = new PrintWriter(new FileWriter(fileName));
                        out.print("Camera Calibration:\n"+
                                         "\tIntrinsic Parameters:\n"+
                                         "\t\tK:\n");
                        for (int i = 0; i < 3; i++) {
                                out.print("\t\t\t");
                                for (int j = 0; j < 3; j++) {
                                        out.print(String.format("%f\t", _cameraK[3*i+j]));
                                }
                                out.print("\n");
                        }
                        out.print("\t\tDistortion Coefficients:\n");
                        for (int i = 0; i < _cameraDistortion.length; i++) {
                                out.print(String.format("\t\t\t%f\n", _cameraDistortion[i]));
                        }
```

```
                        out.print("\tExtrinsic Parameters:\n"+
                                        "\t\tRotation Matrix:\n");
                        for (int i = 0; i < 3; i++) {
                                out.print("\t\t\t");
                                for (int j = 0; j < 3; j++) {
                                        out.print(String.format("%f\t", _cameraR[3*i+j]));
                                }
                                out.print("\n");
                        }
                        out.print("\t\tTranslation Vector:\n");
                        for (int i = 0; i < 3; i++) {
                                out.print(String.format("\t\t\t%f\n", _cameraT[i]));
                        }
                        if (_projCornerT != null && _projFocalT != null) {
                                out.print("Projector Calibration:\n"+
                                                "\tCorner Translation:\n");
                                for (int i = 0; i < 3; i++) {
                                        out.print(String.format("\t\t%f\n", _projCornerT[i]));
                                }
                                out.print("\tFocal Point Translation:\n");
                                for (int i = 0; i < 3; i++) {
                                        out.print(String.format("\t\t%f\n", _projFocalT[i]));
                                }
                        }
                        out.close();
                        return 1;
                } catch (IOException e) {
                        return 0;
                }
        }

        public static Calibration buildFromFile(File file) {
                Calibration calib = new Calibration();
                double[] k, dist, rot, trans, projF, projC;
                try {
                        Scanner scanner = new Scanner(file);
                        int i;
                        while (!scanner.hasNextFloat()) {try {scanner.nextLine();} catch(
                            Exception e) {return calib;}}
                        k = new double[9];
                        for (i = 0; i < 9; i++) {
                                if (scanner.hasNextFloat()) {
                                        k[i] = scanner.nextFloat();
                                } else return calib;
                        }
                        calib.setCameraK(k);
                        while (!scanner.hasNextFloat()) {try {scanner.nextLine();} catch(
                            Exception e) {return calib;}}
                        dist = new double[4];
                        for (i = 0; i < 4; i++) {
                                if (scanner.hasNextFloat()) {
                                        dist[i] = scanner.nextFloat();
                                } else return calib;
                        }
                        calib.setCameraDistortion(dist);
                        while (!scanner.hasNextFloat()) {try {scanner.nextLine();} catch(
                            Exception e) {return calib;}}
                        rot = new double[9];
                        for (i = 0; i < 9; i++) {
                                if (scanner.hasNextFloat()) {
                                        rot[i] = scanner.nextFloat();
                                } else return calib;
                        }
                        calib.setCameraR(rot);
                        while (!scanner.hasNextFloat()) {try {scanner.nextLine();} catch(
                            Exception e) {return calib;}}
                        trans = new double[3];
                        for (i = 0; i < 3; i++) {
                                if (scanner.hasNextFloat()) {
                                        trans[i] = scanner.nextFloat();
                                } else return calib;
                        }
                        calib.setCameraT(trans);
                        while (!scanner.hasNextFloat()) {try {scanner.nextLine();} catch(
                            Exception e) {return calib;}}
                        projC = new double[3];
                        for (i = 0; i < 3; i++) {
```

```
                                if (scanner.hasNextFloat()) {
                                        projC[i] = scanner.nextFloat();
                                } else return calib;
                        }
                        calib.setProjCornerT(projC);
                        while (!scanner.hasNextFloat()) {try {scanner.nextLine();} catch(
                             Exception e) {return calib;}}
                        projF = new double[3];
                        for (i = 0; i < 3; i++) {
                                if (scanner.hasNextFloat()) {
                                        projF[i] = scanner.nextFloat();
                                } else return calib;
                        }
                        calib.setProjFocalT(projF);
                        return calib;
                } catch (FileNotFoundException e) {
                        return null;
                }
        }

        public void setProjFocalT(double[] t) {
                _projFocalT = new double[t.length];
                for (int i = 0; i < t.length; i++) {
                        _projFocalT[i] = t[i];
                }
        }

        public double[] getProjFocalT() {
                return _projFocalT;
        }

        public void setProjCornerT(double[] t) {
                _projCornerT = new double[t.length];
                for (int i = 0; i < t.length; i++) {
                        _projCornerT[i] = t[i];
                }
        }

        public double[] getProjCornerT() {
                return _projCornerT;
        }

}
```

B.6 Reconstruction Control Panel

Listing 7: `ReconstructionPanel.java` — the `ReconstructionPanel` class controls the user interface for the reconstruction control panel. For brevity, code involving component layout is elided.

```java
/*
 * import packages
 */

public class ReconstructionPanel extends JPanel {

        private App                       _application            = null;
        private File                      _imageDir               = null;

        private int                       _vSpace                 = 16;

        private JPanel                    _controlPanel           = null;

        private int                       _imageWidth             = 800;
        private int                       _imageHeight            = 600;

        private CvMat                     _minI                   = null;
        private CvMat                     _maxI                   = null;
        private CvMat                     _crossover              = null;
        private CvMat                     _shadThresh             = null;
        private CvMat                     _mask                   = null;

        private boolean                   _ready                  = false;

        private ImagePanel                _iP1                    = null;
        private ImagePanel                _iP2                    = null;
        private ImagePanel                _iP3                    = null;
        private ImagePanel                _iP4                    = null;

        private JTextField                _threshField            = null;

        private Calibration               _calibration            = null;

        // Initialize the intermediate matrices involved in the
        // reconstruction process.
        public ReconstructionPanel(App app) {
                super();
                this.setPreferredSize(new Dimension(_imageWidth, _imageHeight));
                this.setMaximumSize(new Dimension(_imageWidth, _imageHeight));
                this.setLayout(new BoxLayout(this, BoxLayout.LINE_AXIS));
                _minI = cvCreateMat(768, 1024, CV_8UC1);
                _maxI = cvCreateMat(768, 1024, CV_8UC1);
                _shadThresh = cvCreateMat(768, 1024, CV_32FC1);
                _crossover = cvCreateMat(768, 1024, CV_32FC1);;
                _mask = cvCreateMat(768, 1024, CV_8UC1);
                _application = app;

                _calibration = Calibration.buildFromFile(new File("data" + File.separator + "
                    calib_default" + File.separator + "calibration.txt"));
                _addPanels();
        }

        public void setCalibration(Calibration calibration) {
                _calibration = calibration;
        }

        // Display the four images in their respective panels
        private void _updateImagePanels() {
                IplImage minI = cvCreateImage(cvSize(_minI.cols(), _minI.rows()), IPL_DEPTH_8U
                    , 1);
                minI = cvGetImage(_minI, minI);
                IplImage maxI = cvCreateImage(minI.cvSize(), IPL_DEPTH_8U, 1);
                maxI = cvGetImage(_maxI, maxI);
                IplImage mask = cvCreateImage(minI.cvSize(), IPL_DEPTH_8U, 1);
                mask = cvGetImage(_mask, mask);
                IplImage crossover = cvCreateImage(minI.cvSize(), IPL_DEPTH_8U, 1);
                CvMat crossoverScaled = cvCreateMat(_crossover.rows(), _crossover.cols(),
                    CV_8UC1);
                if (_imageDir != null) {
                        File files[] = _imageDir.listFiles(new FilenameFilter() {
```

```java
                                public boolean accept(File path, String name) {
                                        return (name.toLowerCase().startsWith("frame"));
                                }
                        });
                        double N = (double)files.length;
                        cvConvertScale(_crossover, crossoverScaled, 255.0/N, 0);
                }
                crossover = cvGetImage(crossoverScaled, crossover);

                _iP1.setImage(minI.getBufferedImage(), "Minimum Intensity");
                _iP2.setImage(maxI.getBufferedImage(), "Maximum Intensity");
                _iP3.setImage(mask.getBufferedImage(), "Contrast Mask");
                _iP4.setImage(crossover.getBufferedImage(), "Shadow Edge Tracking");
        }

        /*
         *
         * Component layout functions elided.
         *
         */

        // Perform the various image processing functions based
        // on the user input.
        private class ComputeListener implements ActionListener {
                public void actionPerformed(ActionEvent e) {
                        String command = e.getActionCommand();

                        // Compute the minimum and maximum intensity images
                        // for the scan sequence.
                        if (command.equalsIgnoreCase("minmax")) {
                                DataAnalysis.computeMinMax(_imageDir, _minI, _maxI);
                                _updateImagePanels();
                                _ready = false;
                        }

                        // Compute the contrast mask based on the user defined
                        // threshold
                        else if (command.equalsIgnoreCase("threshold")) {
                                cvSub(_maxI, _minI, _mask, null);
                                float threshold = Float.valueOf(_threshField.getText());
                                cvCmpS(_mask, threshold, _mask, CV_CMP_GT);
                                cvSmooth(_mask, _mask, CV_MEDIAN, 5);
                                _updateImagePanels();
                                _ready = false;
                        }

                        // Compute the temporal edge values for the scan sequence
                        else if (command.equalsIgnoreCase("edge")) {
                                cvAdd(_maxI, _minI, _shadThresh, null);
                                cvConvertScale(_shadThresh, _shadThresh, 0.5, 0.0);
                                DataAnalysis.computeTemporalEdge(_imageDir, _shadThresh, _mask
                                        , _crossover);
                                _updateImagePanels();
                                _ready = true;
                        }

                        // Perform the reconstruction
                        else if (command.equalsIgnoreCase("reconstruct")) {
                                if (_ready) {
                                        DataAnalysis.computeReconstruction(_crossover,
                                                _calibration, _imageDir);
                                } else {
                                        System.err.println("Must load parameters or calibrate
                                                system.");
                                }
                        }

                        // Switch to the calibration panel
                        else if (command.equalsIgnoreCase("calibrate")) {
                                _application.calibrate(_calibration);
                        }

                        // Switch to the scanning panel
                        else if (command.equalsIgnoreCase("scan")) {
                                _application.scan();
                        }
```

```
                // Import a calibration file
                else if (command.equalsIgnoreCase("loadCalibration")) {
                        JFileChooser fc = new JFileChooser("data" + File.separator);
                        fc.setFileFilter(new FileNameExtensionFilter("Text Files", "
                            txt"));
                        int returnVal = fc.showOpenDialog(new JFrame());
                        if (returnVal == JFileChooser.APPROVE_OPTION) {
                                File file = fc.getSelectedFile();
                                setCalibration(Calibration.buildFromFile(file));
                                if (_calibration != null) {
                                } else {
                                        System.out.println("Error loading calibration
                                            parameters.");
                                }
                        }
                }
        }
    }
}
```