2D23D

Authors: Alex Patel, Chakara Owarang, Jeremy Leung: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*— It cannot be overstated how much traditional 2D camera technology has shaped our progress and culture. However, 2D images do not effectively capture the details of an object when it is rotated around. This information is potentially crucial to archaeological archivists that want to preserve the object in its entirety, with the potential to 3D print and recreate these objects. Our goal is to be able to accurately map and scan the physical object into a digital three-dimensional representation for the purpose of archaeological documentation. Our design goals are to design this device to be able to be used by a non-technical audience: easy to use, time-efficient, and most of all, accurate in scanning. Currently, manual 3D modeling by users is time-ineffective, tedious, and costly, as well as prone to errors. Our project will uniquely address these issues in being user-friendly and accurate.**

***Index Terms*— 3D scanning, laser depth triangulation, point cloud generation, pairwise registration, Iterative Closest Points, point cloud mesh triangulation, stepper motor, motor controller, motor driver, GPU-accelerated computation**

1 INTRODUCTION

The main application use case of our project is to be able to scan archaeological objects for preservation in a 3D format. The goal is to be able to accurately map these objects into a widely usable 3D format for documentation and reproduction via 3D printing. Thus, our requirements will be focused around the accuracy of the scan. However, the device should also have a reasonably fast scanning time. We thus aim to have these two accuracy requirements: 90% of non-occluded points must be within 2% of the longest axis to the ground truth model, and 100% of non-occluded points must be within 5% of the longest axis to the ground truth model. Other requirements will be covered in the Design Requirements section (3). Our basic approach was be to use a projected laser stripe along with a high resolution digital camera to scan the object atop a rotating platform. Our approach also allows for combinations of multiple scans in case some angles of the object are hidden from the scan.

Because of the unfortunate current situation of the Coronavirus Outbreak, our team members are scattered in different places with different time zones. This means that our project must be able to be completed remotely, which would void any hardware or physical requirement for our project. Since we are unable to create a physical device to carry out 3D scanning, we have changed our project to instead utilize a virtual simulation of a 3D scan. This simulation is done by taking renders of a 3D scene using Blender, a 3D animation software, which act as the camera inputs from our original design. The remainder of our software pipeline remains unchanged. This decision to simulate data capture allows us to stay true to our original design requirements, while adapting to a changing environment. Our initial design which contains hardware and physical components are still included in the Initial Design section (2) and Appendix A.

There are currently several other competing technologies which mostly involve either digital cameras or depth cameras. However, scans that combine multiple views from a digital camera (multi view stereo reconstruction) tend to have lower accuracy and are unable to detect concavities in the object accurately, and also requires very strict lighting setups to ensure the best scans. We also considered using depth cameras to give a 2D depth map which would tell us depth of each pixel scanned, but most depth cameras do not give very precise accuracy information, and depth cameras tend to be much less precise than using laser based approaches. We will discuss more details of the many different approaches we considered in depth in the Design Trade Studies section (5).

2 INITIAL DESIGN

This section contains the general idea of our initial design. More information can be found in Appendix A.

2.1 Architecture Overview

Allow us to start from the user story to link into our initial architecture design.

The user first presses the start button, and if the device is not calibrated, the program will prompt the user to insert the laser plane calibration object such as a checkerboard pattern and perform the calibration. Then, the user will start the scan and a point cloud will be generated. If additional scans are required, the user will rotate the object in a way that provides more information about the bottom or the concavities of the object, then these point clouds will be coalesced with pairwise registration. After a final point cloud is obtained, we will perform triangulation to output the final mesh.

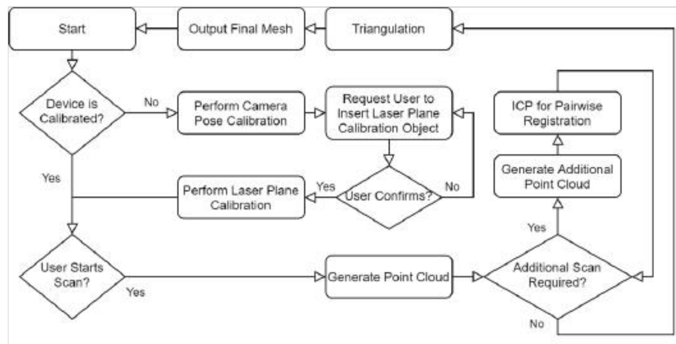


Figure 1: User Story Flowchart

This process leads into our software pipeline design, which directly corresponds with the user story. Note that the pipeline diagram above assumes the camera is already calibrated. We will use the camera image to perform laser detection, and for each image with the laser in it, we will generate Cartesian coordinate values based on how the laser warps around the object. We will also remove the background points and the points from the turntable. Combining this with rotational information we have from the stepper motor driver, we will be able to generate a point cloud, which will be passed through some noise reduction and outlier removal filtering. If multiple scans are required, then we will use pairwise registration to combine the two point clouds and output a final point cloud. Then, we will use triangulation on the point cloud to produce an output mesh. Triangulation basically involves forming triangles on the surface using close neighboring points, which produces a triangle mesh that can be easily 3D printed.

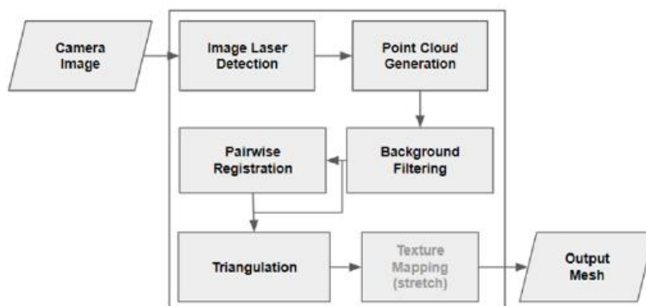


Figure 2: Software Pipeline

Based on this software pipeline, we can have a more comprehensive system specification diagram now, which includes components for the stepper motor, GPGPU, controllers, and more. Please refer to Figure 37: Initial System Specification Diagram in Appendix B for our initial system specification diagram.

The proposed integrated hardware platform includes a 4-core embedded system with programmable GPGPU, interfaces between the system and other components of our design, as well as logical connections between software elements. From the camera, data comes into the embedded system by USB. This data is processed by our core system controller process, which will either initiate the execution of GPU kernels and subroutines for 3D scanning using the image, or tell the motor controller to rotate a certain about to capture additional images, depending on the state of the controller within the process it is trying to accomplish. A software state machine within the core system controller process will match the behavior that can be found in Figure 1: User Story Flowchart based on user input and the current executing operation.

A motor controller process will be responsible for setting the GPIO pins to interface with the stepper motor through the motor driver, setting the rotation to a specific angle. A breadboard and wires will be used to interface between the motor driver and the stepper motor. The stepper motor angle will be communicated to the motor controller from the core system controller based on the stage of the scan. An additional process will be used to handle interrupts an processing related directly to user input. Having this additional indirection for user I/O helps enforce protection boundaries between our system and the outside world (only very specific types of input can make it through). Since we have 3 processes and 4 cores on the embedded system, each process can have a core dedicated to it to maximize performance. For the motor controller, we can change the kernel scheduling algorithm used to allow it to be prioritized for real-time deadlines.

Besides the physical components and the software processes, various GPGPU kernels and subroutines will exist within the system to carry out optimized execution of our required algorithms. A GPGPU kernel describes the execution of a single core in a many-core programming platform. Launching a GPGPU kernel first copies data from DRAM into the GPU's memory, sets up the scheduling and synchronization structures of the GPGPU to support the provided kernel, and begins execution of each execution core "simultaneously". The result is then copied back into DRAM. We will implement a majority of our optimized algorithms with compute kernels, including calibration, ray-plane intersection, iterative closest points for pairwise registration (single object multiple scans), and other geometric operations.

2.2 System Description

Our initial system consists of an integrated hardware platform, a laser line diode, a digital camera, the rotational platform, and the physical structure to hold up each component. The integrated hardware platform can be further broken down into a motor component (consisting of the software motor controller, the motor driver board, and the stepper motor itself), the algorithmic components (consisting of the GPGPU kernel routines on the embedded system), and the state control component (consisting of processes executing on the 4 core embedded system, determining the control flow of the system). Figure 37: Initial System Specification Diagram in Appendix B to see connections between components.

2.2.1 Sensor Setup

The sensor setup consists of a statically arranged laser line diode and digital camera. The laser line diode provides a red light at 650nm wavelength with ideally a consistent Gaussian intensity distribution across the line for the entire length of the line. The digital camera must have sufficient resolution to meet our accuracy requirement. We chose an 8 megapixel (the highest commercially available for reasonable price) camera to minimize the effective distance between each pixel and maximize our accuracy.

2.2.2 Mechanical Setup

The mechanical setup consists of all the components of the system, and determines how they are arranged in physical space with respect to each-other. Figure 3: Mechanical Setup (Front View) and Figure 4: Mechanical Setup (Side View) show how the components would be set up.

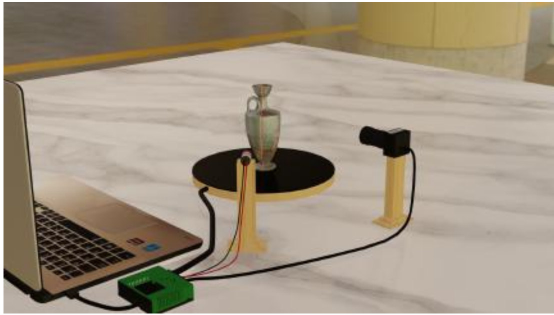


Figure 3: Mechanical Setup (Front View)

Figure 4: Mechanical Setup (Side View)

The rotational platform forms the physical basis of the device, with the camera and laser line diode positioned at a short distance (approximately 20cm) away pointing towards the center of the rotational platform. The platform would be mainly composed of a base, a motor, a gear on the motor's shaft, a lazy susan bearing to reduce friction, an internal gear, the top platform, and a high-friction surface. The high-friction surface here is to simply help reduce the chance of the object slipping off-center while the platform is rotating. The base here is to give the platform itself enough height so that the motor can be put under. The motor with a gear attached to the shaft will be inside the platform. The lazy susan bearing will be on top of the base, and the internal gear will be attached on top of the lazy susan bearing, and the top platform will be attached on the internal gear. The gear on the motor's shaft will be connected to the internal gear, and when the motor rotates, the platform will rotate with it. Figure 5: Platform Component Breakdown below shows a breakdown of how the platform itself would look like.

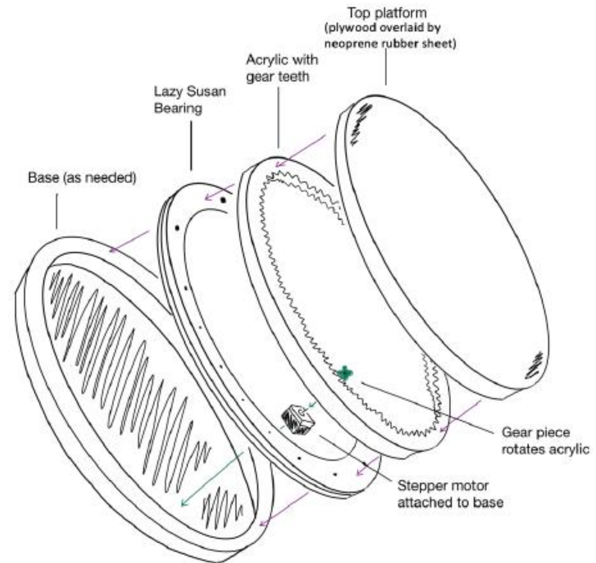


Figure 5: Platform Component Breakdown

From the design trade studies, we are using the following items for the components mentioned above [16]:

- Motor: NEMA 23 Stepper Motor
- Gear on motor's shaft: 3D-printed to control the gear ratio and dimensions
- Lazy Susan Bearing: 10" Swivel Lazy Susan Bearing
- Internal Gear: Laser-cut Acrylic Plexiglass Sheet to control the gear ratio and dimensions
- Top Platform: Plywood 15" Circular Disc
- High-friction Surface: Neoprene Rubber Sheet

2.2.3 Hardware Setup

The integrated hardware platform consists of the NVIDIA Jetson Nano embedded platform, the DM542T motor driver, and USB/wires/breadboards to provide the interface to our external sensor, motor, and user. The NVIDIA Jetson Nano has the following notable specifications relating to our project:

- 128-core Maxwell GPU
- Quad-core ARM A57 @ 1.43 GHz CPU
- 4 GB 64-bit LPDDR4 25.6 GB/s Memory

- Video Encode/Decode at 30fps for 2x 4k (H.264/H.265)
- 4x USB 3.0 external ports
- GPIO pins for motor control

2.2.4 State Control Subsystem

A 4-core embedded system with programmable GPGPU will be the main computational platform for our device. Specifically, the NVIDIA Jetson Nano embedded system. This system will have 3 of our dedicated processes running on it while the device in in action:

- User I/O Controller: Enforces protection boundary between user behavior and device logic. Only reasonable commands will be passed from this controller to the other processes, and only intended output will pass from other processes to the user. This indirection layer helps prevent the user from accidentally (or intentionally) tampering with the device logic.
- Core System Controller: This process orchestrates the order of total functionality of the system. A software state machine is implemented to match the behavior of our user story flow chart (See Figure 1: User Story Flowchart). The Core System Controller interfaces with the user via the User I/O Controller (and indirectly USB), with the rotational platform via the Motor Subsystem, with the GPGPU via kernel launches (Algorithmic Subsystem), and with the camera via a USB interface on the NVIDIA Jetson Nano.
- Motor Controller: This process rotates the stepper motor by angles determined and communicated by the core system controller. The motor subsystem description will go deeper into its behavior.

We chose to divide the computation work across three independent cores to maximize performance. The User I/O Controller rarely runs, but it should respond quickly to user inputs, and therefore it is left idle so that we can maximize the response time between an I/O device interrupt (new data from USB) and processing the I/O. The motor controller process on the other hand is real-time deadline driven, as it must continuously step the motor at specific angles to coordinate with the camera via the core system controller. Because of this, we plan on using a soft real-time scheduling algorithm for this process. We will also ask the kernel to separate these processes across the cores. The remaining core can be used for background processes for the OS.

2.2.5 Motor Subsystem

The motor subsystem is responsible for providing accurate control of the stepper motor turning the rotating platform during 3D scanning. This subsystem consists of both hardware and software components. The stepper motor itself accepts PWM wave modulation to control how many steps to take in which direction (or partial steps). Since such signals are difficult to generate manually, we are using a motor driver board DM542T to convert GPIO signals from the Nvidia Jetson Nano into waveform commands for the stepper motor. A breadboard with wires attached to the pins on both the driver and the stepper motor will transfer this signal to the motor. As noted before, the motor subsystem will have heavy communication with the state control subsystem to coordinate camera image captures. Both the camera latency and throughput, as well as the step amount and speed, have an effect on the details of this coordination.

2.2.6 Algorithmic Subsystem

The algorithmic subsystem is completely in software, and is mostly data-driven. Wheras the state control subsystem determines what happens across the system at any point in time by using the CPU of the embedded system with I/O and interrupts, the algorithmic subsystem utilizes the GPGPU to massively parallelize data-driven algorithmic code, such as image processing and geometric transformations. The algorithmic subsystem generally consists of GPGPU kernels which implement the required computations of our device, including calibration, computer vision, point-cloud generation, pairwise registration, and mesh triangulation, as well as accuracy computation for testing and validation. Algorithms and descriptions of their implementations are described in detail in the remaining subsections of the system description.

2.2.7 Camera and Scene Calibration

There are several components to calibration which must be performed prior to any 3D scanning [12]. The general method to convert between a pixel and a world space coordinate is, for pixel u and coordinate p:

$$\lambda u = K(Rp + T)$$

where :

- λ is a scalar intrinsic to the camera and screen size in pixels
- K is a distortion matrix intrinsic to the camera lens
- R is the rotation matrix between world space and camera space coordinates.
- T is the translation matrix between world space and camera space coordinates.

Then to calibrate each of these parameters:

- Intrinsic Camera Calibration: this calibration resolves constants related to the camera lens and polynomial distortion that may have on the image verses real world space. This can be done by the mapping

between known world-space points and known pixels. This calibration determines the K and λ parameters above.

- Extrinsic Camera Calibration: this calibration solves a system of linear equations to find the translation and rotation matrices for the transformation between camera space and world space. This system is made non-singular by having sufficiently many known mappings between camera space and world space (specific identifiable points on the turntable). Therefore, this requires an object on the turntable to act as these known points. We will use a CALTag checkerboard pattern pasted on the turntable surface to help perform this calibration. The CALTag pattern is especially useful since position can be determined even if some of the image is occluded. This calibration determines the R and T parameters above.

We setup the linear equation as follows. Suppose

$$u = \begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix}, \qquad p = \begin{bmatrix} p_x \\ p_y \\ 0 \end{bmatrix}$$

Where u is in homogeneous pixel coordinates and p is on the turntable plane ($z = 0$), and

$$\tilde{u} = K^{-1}u, \qquad R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, \qquad T = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix}.$$

From this we get that

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \tilde{u} \times (\lambda \tilde{u}) = \tilde{u} \times (Rp + T).$$

Then we solve for R and T from the following system:

$$\tilde{u} \times \begin{bmatrix} r_{11}p_x + r_{12}p_y + T_x \\ r_{21}p_x + r_{22}p_y + T_y \\ r_{31}p_x + r_{32}p_y + T_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

From this we group the unknowns in a single vector

$$X = \begin{bmatrix} r_{11} & r_{21} & r_{31} & r_{12} & r_{22} & r_{32} & T_x & T_y & T_z \end{bmatrix}^T$$

from which we can derive that

$$\begin{bmatrix} 0 & -p_x & \tilde(u)_y p_x & 0 & -p_y & \tilde(u)_y p_y & 0 & -1 & \tilde(u)_y \\ p_x & 0 & -\tilde(u)_x p_x & p_y & 0 & -\tilde(u)_x p_y & 1 & 0 & -\tilde(u)_x \end{bmatrix} X = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Here we are only considering a single point p. During calibration, we take multiple images, and so we actually have a set of points $\{p_i | i : 1...n\}$. We can use all such points to form a single system of the form $AX = 0$ where $A \in \mathbb{R}^{2n \times 9}$. Since vector X has 8 degrees of freedom, matrix A must have rank 8, meaning we need at least 4 non-collinear points. Finally we are left to solve the following optimization problem to compute R and T:

$$\hat{X} = \underset{X}{\operatorname{argmin}} ||AX||, \qquad s.t. ||X|| = 1$$

- Axis of Rotation Calibration: this is computed in a similar manner to extrinsic camera calibration. The center position of the turntable is discovered, and the axis of rotation is assumed to be in the positive z direction. Note that this assumes that the 'up' direction of our camera image is the same angle as the up direction of the turntable.

- Plane of Laser Line Calibration: With laser line detection, points along the laser in the digital image can be identified. From this, a linear equation must be solved to determine the A, B, C, and D parameters of the plane of the laser line in world space, where the plane is $Ax + By + Cz + D = 0$. An additional calibration object is required here to completely define the laser plane, instead of simply defining a pencil of planes (all planes rotated around the axis of the turntable). The known object, which will be a checkerboard pattern, allows us to know where on the object the laser points are detected, and from that collect a set of non-collinear points, which allow us to solve the linear system.

The Camera and Scene Calibration subsystem is replaced by simulation in our final design and will be discussed in the System Description section (6)

2.2.8 Outlier removal/noise reduction

We will first do outlier removal, and may or may not do noise reduction depending how bad the point cloud obtained is - we do not want to over-smooth the object since many archaeological objects tend to have weird shapes and contours and jagged edges. Point clouds obtained from 3D scanners with any methods, including laser projection, would regularly be contaminated with some level of noise and outliers. The first step for dealing with raw point cloud data obtained after conversion from laser projection to depth would be to discard outlier samples. Note that this step would come after removing the points in the background and the foreground points on the turntable. There are generally three types of outliers: sparse, isolated, and nonisolated. Sparse outliers have low local point density that are obvious erroneous measurements i.e. points that float outside of the rest of the data. Isolated outliers have high local point density but are separated from the rest of the point cloud, i.e. outlier clumps. Nonisolated outliers are the most tricky – they are outliers that are very close to the main point cloud and cannot be easily separated, akin to noise. We will focus on sparse and isolated outliers, and we can use a method that looks at average distance to k-nearest neighbors, then removes that point based on a threshold defined in practice. Let the average distance of point p_i be defined as:

$$d_i = 1/k \cdot \sum_{j=1}^{k} \text{dist}(p_i, q_j) \tag{1}$$

where q_j is the jth nearest neighbor to point p_i. Then, the local density function of p_i is defined as follows:

$$LD(p_i) = \frac{1}{k} \cdot \sum_{q_j \in kNN(p_i)} \exp\left(\frac{-\text{dist}(p_i, q_j)}{d_i}\right) \quad (2)$$

with d_i defined earlier in equation (1). Now we can define the probability that a point belongs to an outlier as:

$$P_{\text{outlier}}(p_i) = 1 – LD(p_i)$$

We can then take this probability and if it is above a certain threshold, that point will be removed from the point cloud data (PCD). One reference paper uses $P_{\text{outlier}}(p_i) > 0.1 \cdot d_i$ as their threshold in practice which is dynamic based on d_i, we can determine this threshold through empirical testing by seeing accuracy numbers based on threshold values on multiple test samples.

2.2.9 Global Parameter Optimization

Calibration parameters will be solved for in the least-squared error sense, to match our recorded points. Global optimization will then be applied to all the parameters to reduce reconstruction error. Our implementation of optimization will consist of linear regression. Global Parameter Optimization is not required in our final design due to the simulation calibration procedure.

2.2.10 Subsystems that Remain in our Final Design

The following subsystems are still in our final design and will be discussed in the the System Description section (6).

- Image Laser Detection
- Point Cloud Generation
- Point Cloud Processing

3 DESIGN REQUIREMENTS

Below are our final design requirements. These requirements are updated from our initial requirements which can be found under Appendix A.

a. Accuracy

Our first requirement is that of accuracy. We aim to satisfy this requirement: 90% of non-occluded points must be within 2% of the longest axis to the ground truth model, and 100% of non-occluded points must be within 5% of the longest axis to the ground truth model. Previously, to be able to test this, we would have had to find ground truth 3D models and 3D print them to input into our scanning pipeline, then compare with the original mesh to get the accuracy score. However, since we are simulating the scans with Blender, we can directly compare the ground truth mesh with the one we generate without having to 3D print anything - this allows for a more precise accuracy computation, since previously 3D printing would have caused a level of error unless we perfectly smooth out all the objects we test. We will compute the accuracy number by computing distances from each vertex in our constructed mesh to the surface of the ground truth mesh. The distance of a vertex to a mesh is the minimum distance between that vertex's position and the closest triangle on the other mesh. These results must match our accuracy requirement as stated above.

Our accuracy metric that we use in our report is thus driven by this requirement. We take the percentage of vertex distances from our generated mesh to the ground truth mesh that is within 2% of the longest axis, as well as the percentage going the other way around, from the vertices of the ground truth mesh to the surface of our generated mesh, and average out these two scores for a final accuracy score. Additionally, if any of the points are outside of 5% of the longest axis, the generated mesh immediately fails our accuracy requirements.

b. Usability

Our next requirements involves usability. We have removed the input object size and maximum weight requirements since our input data is now simulated data and we can adjust these parameters on Blender. However, we are still requiring our program to output a common 3D format such as OBJ or STL for ease of 3D printing and storage. This requirement can be easily evaluated and doesn't require any special quantitative tests. Since the program is now a script, we are not running any user experience surveys anymore.

c. Efficiency

Since we are now simulating sensor data and we would not be using NVIDIA Jetson GPU as our processor anymore (which means we cannot write GPU optimized code which would run a lot faster than Python, which we are currently using) and due to uncertainties, we are relaxing our efficiency requirement to 10 minutes which only includes the time from inputting the scan images to getting the output mesh, which does not include time to render images from Blender or time taken by our verification pipeline. We would be using an i7-2600K processor with clock rate 3.40 GHz, and had access to 8.00 GB DDR3 RAM to test our benchmarks.

4 ARCHITECTURE OVERVIEW

Allow us to start from the user story to link into our final architecture design.

Users first run our main script `driver.py` with an input directory containing one or more directories of the scanned images. Users can choose other parameters to adjust our

script to their specific use case or to analyze different parameters or algorithms. This will be discussed in more detail in the System Description section (6) under Algorithmic Subsystem. The program then goes through all the directories within the main directory and leads into our software pipeline design. We use the camera image to perform laser detection, and for each image with the laser in it, we generate Cartesian coordinate values based on how the laser warps around the object. We will also remove the background points and the points from the turntable. From this and the angle from each simulated scanned image, we can then generate a point cloud, which is passed through some noise reduction and outlier removal filtering. If multiple scans are required, then we use pairwise registration through Iterative Closest Point algorithm (ICP) to combine these point clouds and output a final point cloud. Then, we use triangulation on the final point cloud to produce an output mesh. Triangulation basically involves forming triangles on the surface using close neighboring points, which produces a triangle mesh that can be easily 3D printed.

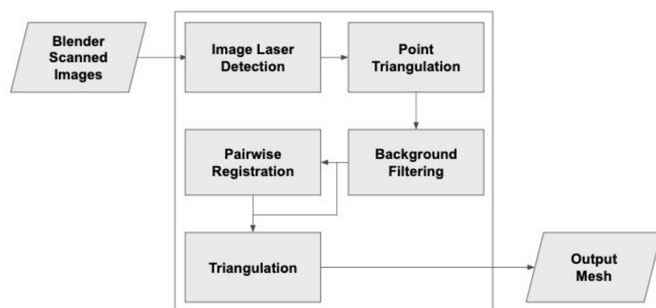


Figure 6: Final Software Pipeline

Based on this software pipeline, we can have a more comprehensive system specification diagram now, which includes components for blender, software subroutines, and verification subsystem. Please refer to Figure 38: Final System Specification Diagram in Appendix B for our full system specification diagram.

All code is now run on the CPU of the computational platform that executes the python code. Our benchmark system utilized an i7-2600K processor with clock rate 3.40 GHz, and had access to 8.00 GB DDR3 RAM.

Users would only need a computer with Python 3 and required python libraries installed to be able to run our program. The required libraries are

- `scikit-image`: our system uses scikit-image to efficiently load in scanned images from blender simulation to our program
- `NumPy`: our system uses NumPy as an efficient multi-dimensional container of our input image data and to perform matrix operations to for point cloud generation
- `Open3D`: our system uses Open3D to read and write both point clouds and triangular meshes. We also use Open3D for some triangulation methods to perform tradeoff analysis. Open3D library provides many useful functions such as estimating normals and performing triangulation (constructing a triangle mesh from XYZ points and normals)
- `Pyvista`: our system uses pyvista to run our main triangulation algorithm, Delaunay triangulation.

We used Blender to simulate the scanned images. Blender allows us to control many different parameters. The main parameter with accuracy-time trade-offs is number of frames, which we will discuss in Design Trade Studies section (5) under Number of Frames subsection. We also set our render settings to be at 60% quality of 1080p which imitates our original USB camera as closely as possible, which had 8 megapixels but not the clearest quality. The 60% number was determined by rendering at various render qualities then comparing them with an image output of our original camera. We also use the Eevee render engine in Blender, which runs much faster than the Cycles engine since it does not perform raytracing, and is realistic enough to be sufficient for our purposes. To animate a mesh for scanning custom meshes in Blender, the user would set up a light to project a laser line in Blender at the center of the object, as well as set the camera to point at the mesh, and set the animation keyframes to be 0 degrees of Z-rotation at the 0th frame, and 360 degrees of Z-rotation at the nth frame, with n being the number of frames the user wants to render. The user would then need to run our Blender script to obtain the camera and laser intrinsic parameters. We have a Blender file in our repository that already sets all this up and all the user needs to do is input a mesh and set up the animation keyframes properly. It is recommended that the user nests the mesh under an "Empty" object in Blender and rotate the "Empty" instead in case the user wants to move the mesh around but still have it rotate around the center.

5 DESIGN TRADE STUDIES

5.1 Scanning Sensor

Although we do not have any physical sensors in our final design, the blender simulation setup is still based off of our design trade studies on the scanning sensor used. This subsection discusses how we came up with this setup.

The specific sensor setup we will use for our project required the most research and evaluation to compare many different possible methods. We started from our requirements to choose this sensor. In the most extreme case, to accurately capture an object whose longest axis is 5cm, in order to meet the requirement that 90% of reconstructed points are within 2% of that 5cm, we must have points within 1mm to the ground truth model. From this, if we consider the number of samples we need across the surface,

we must have accurate samples within 1mm for each direction (X, Y along the surface). Since the surface itself is a continuous signal we are sampling from, we can compute the Nyquist sampling rate as being every 0.5mm along each direction of the surface.

Now considering the rotational mechanism, the largest radius of the object from the center of the rotating platform will be within 15cm. We need then a single rotation per data capture to be such that the amount of the surface rotated passed the sensor is less than or equal to 0.5mm. This gives us: $\frac{0.5mm}{150mm} = 0.0033$ radians of rotation per sample.

There are five main types of sensors for 3D scanning that we have extensively considered:

1. **Contact sensors:** These sensors are widely used in manufacturing. A Coordinate Measuring Machine (CMM) or similar may be used, which generally utilizes a probing arm to touch the sensor, and through angular rotations of the joints the coordinates of each probed area can be computed. This is a non-option for our application, both for price and the fact that we should not allow a large machine to touch timeless archeological artifacts.

2. **Time-of-flight sensors:** By recording the time between sending a beam of light and receiving a reflected signal, distance can be computed to a single point. The disadvantage of this approach is that we can only measure times so precisely, and the speed of light is very fast. With a timer that has 3.3 picosecond resolution, we are still not within sub-millimeter depth resolution, which is not reasonable for this project. Time-of-flight sensors in the domain of 3D scanning are more applicable to scanning large outdoor environments [2].

3. **Digital camera:** By taking multiple digital photographs from many perspectives around the object, computer vision techniques can be used to match features between pairs of images, and linear transforms can be computed to align such features. After feature alignment, and depth calculation, point clouds can be generated. Computer vision techniques to accomplish the above include Structure from Motion (SFM) and Multi-View Stereo reconstruction (MVS). This approach has a fundamental flaw: concavities in the object to be scanned cannot be resolved, since cameras do not capture raw depth data. Surface points within the convex hull of an object cannot be easily distinguished from points on the convex hull. The digital camera methods also have very strict lighting requirements to produce accurate scans and will definitely suffer from accuracy compared to more precise approaches like the laser-based ones. This is an immediate elimination for our project, since archeological objects may have concavities, and we do not want to limit the scope of what type of objects can be captured, along with accuracy being our primary focus of the requirements [10].

4. **Structured/coded light depth sensor (RGB-D Camera):** The idea of such a sensor is to project light in specific patterns on the object, and compute depth by the distortion of the patterns. Such sensing devices have become incredibly popular in the 3D reconstruction research community with the consumer availability of the Microsoft Kinect. The original Microsoft Kinect only has 320 pixels wide of depth information for a single depth image. With the upper bound of 30cm across the surface of the object, this results in 30cm/320px = 0.094cm between each pixel, which does not meet our sensor requirement of being able to detect differences within 0.5mm (0.05 cm). The newer Microsoft Kinect v2 actually uses a time-of-flight sensor, and thus does not get measurements more accurate than 1mm depth resolution. Intel RealSense has recently released new product lines for consumer and developer structured light depth sensors that are very affordable. Most notably, for short range coded light depth sensing, the Intel RealSense SR305 offers 640480 pixel depth maps, which correlates to 30cm/640px = 0.047cm, which is within our requirements. However, Intel does not advertise any specific depth resolution for the device, and we cannot guarantee sub-millimeter depth accuracy - these depth cameras are more commonly used to scan a whole room or scan objects from a meter distance. Perhaps we can obtain a better accuracy figure after some extensive testing but it may not be valuable considering we can just build our own laser stripe triangulation sensor. Since this method only relies on the camera, lighting environment and material of the object can have much greater influence compared with laser triangulation. It also requires a significant algorithmic effort after data collection to reduce noise and correlate the views [17].

5. **Laser point triangulation:** The principle of the single point laser sensor is that an emitting diode and corresponding CMOS sensor are located at slightly different angles of the device in comparison to the object, so depth can be computed by the location on the sensor the laser reflects to [14]. Generally the position of the laser on the surface is controlled by a rotating (or pair of rotating) mirrors. We can assume that such a sensor is affordable, can easily measure with resolution of less than 0.5mm, and that we will not likely encounter any mechanical issues. However, the total number of distance measurements we are required to record is:

$$\frac{2\pi}{0.0033\text{rad}} \cdot \frac{300mm}{0.5mm} = 1142398 \text{ points}$$

Assuming the sensor has a sampling rate of about 10kHz (common for such a sensor), 1142398 points divided by 10000 points per second gives us 114.23 seconds theoretical minimum capture time with one sensor. From our timing requirement, assume that half of our time can be attributed to data collection

(30 seconds). Then, with perfect parallelization, we could achieve our goals with $114.23s/30s = 3.80 \approx 4$ sensors collecting concurrently. However, with our budget, this is not achievable. Even if we had the budget, it is possible for systematic errors in, for example, the mounted angle of a sensor, to propagate throughout our data with no course for resolution. To add another set of sensors to mitigate this error, we would be even farther out of our budget. 8 sensors x $300 per sensor (low-end price) gives us $2400 for this sort of setup. Note these calculations are unrelated to any mechanical components, but are directly derived from required data points.

To make budget not an issue for the single point laser triangulation method, we could choose to adopt cheaper sensors, such as those with under 1kHz sampling rate. Performing the same calculations as above with 1kHz sampling rate shows us that we would require 39 sensors to meet our timing requirement, which is well out of the realm of possibility (and this is not accounting for error-reduction, which may require 78 sensors). If we did not purchase this amount of sensors, we would drastically under-perform for our timing requirement.

6. **Laser stripe triangulation:** Fortunately, there is an alternative to single-point laser triangulation. We may use a laser stripe depth sensor, which gets the depth for points along a fixed width stripe. This would improve our ability to meet our timing requirements significantly. Such devices are not easily available with high accuracy to consumers, but are usually intended for industry and manufacturing. Because of this, we would have the responsibility of constructing such a device ourselves. We have considered the risk of building our own sensor since none of us are experts in sensors and electronics. However, as long as we can find an affordable laser stripe with sufficient brightness, our laser stripe sensor should not suffer in accuracy. A laser stripe sensor consists of a projected linear laser source and a digital camera. Many laser stripe sensors use a CCD camera instead to avoid the projection brightness issue completely, but these CCD cameras tend to be too expensive and would put an unnecessary strain on our budget, and we can do just as well with a digital camera.

   After a calibration process to determine the intrinsic camera parameters as well as the exact angle and distance between the camera and the laser projector, linear transformations may be applied to map each point from screen space to world space coordinates. Because of the ease of achieving sub-millimeter accuracy, and the relative independence on lighting conditions and materials that photogrammetry is harmed by, we plan on constructing a laser stripe depth sensor with a digital camera.

After extensively considering the many available sensor options, we can see that contact, time-of-flight, and digital cameras are clearly not options for us to explore, and depth cameras cannot guarantee as much accuracy as laser stripe triangulation. Given that the main focus of our project is the accuracy of the scans, our sensor setup should be able to provide as much accuracy as possible so as to not cascade errors down the pipeline.

5.2 Number of Frames

The greatest limiting factor Blender has on our project is the number of frames. The number of frames directly controls the granularity of the rotation in our scans. Clearly, more frames means more minute rotations of the object, which leads to a higher accuracy number. However, this accuracy comes at a massive time cost, which multiplies at about 2.2x when the number of frames are doubled. We have ran tests with the monkey mesh to determine the optimal number of frames to scan at.

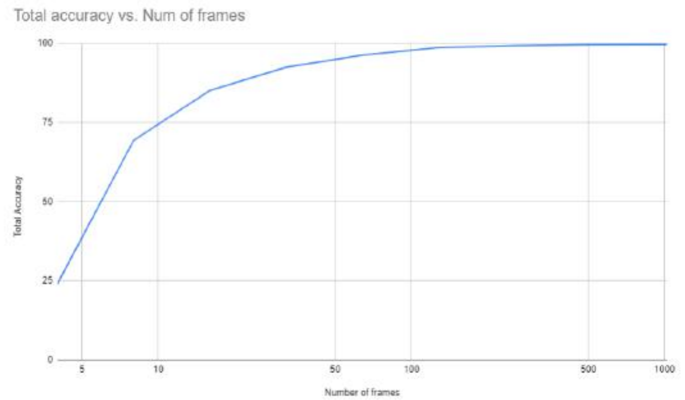


Figure 7: Graph of accuracy vs. number of frames

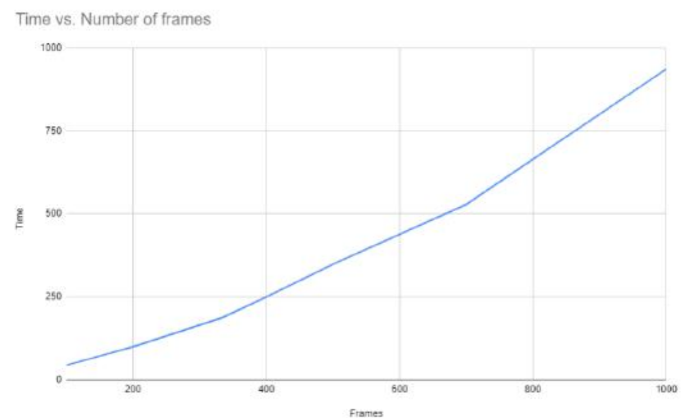


Figure 8: Graph of time vs. number of frames

Looking at the accuracy and time graphs in Figure 7: Graph of accuracy vs. number of frames and Figure 8: Graph of time vs. number of frames, we can see that after 400 frames the accuracy gains are very marginal, while the

time cost still increases at a quadratic rate. Thus, we determined the optimal number of frames to be 400. This can be adjusted in Blender by changing the animation keyframes and the total number of frames rendered.

5.3 Pixel Skip

Depending on the number of frames generated by the simulation, it may not be required to analyze all of the pixels in each of the scan images. This is because with a smaller number of frames, the angle is larger between two scan images, but if the spacing of samples does not change along the laser line, the point cloud distribution becomes uneven. This topology issue should not directly harm the correctness of our reconstruction, but it results in significant computation which may not be necessary. To vary the behavior of how many samples are taken for a single image along the laser line, the pixel skip parameter is introduced. A pixel skip of n means only every n'th row of each image is sampled for laser pixels.

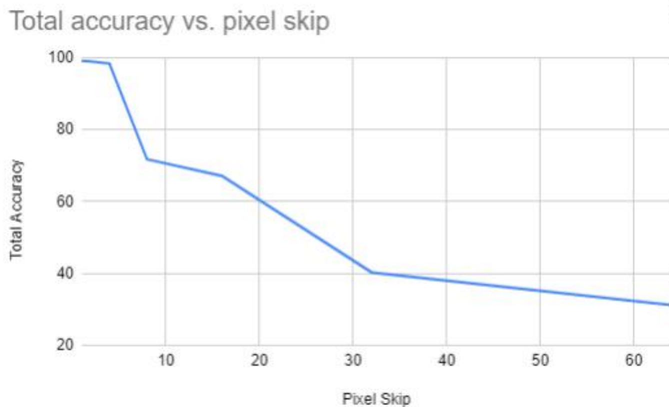


Figure 9: Graph of accuracy vs. pixel skip

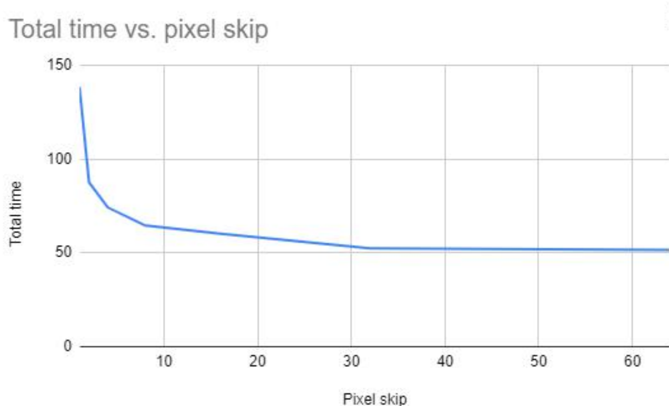


Figure 10: Graph of time vs. pixel skip

Analyzing the tradeoff between accuracy and time in Figure 9: Graph of accuracy vs. pixel skip and Figure 10: Graph of time vs. pixel skip, after a pixel skip of 4, we begin to introduce a significant drop in accuracy. However, after this point the time efficiency does not improve significantly. Thus, we have determined that a pixel skip of 4 is optimal as a default parameter for our project.

5.4 Laser Threshold

To determine if a pixel is part of the laser line, we ensure its intensity in the 650nm spectrum meets a certain threshold and is a local maximum of its row of pixels. Specifically, the intensity of an 8-bit channel RGB(A) pixel is determined by:

$$intensity = max(red - 0.5 * green - 0.5 * blue, 0) \quad (3)$$

Note that red,green,blue are the 3 8-bit channels of the RGB color, and that 650nm corresponds well to the color RGB(255,0,0).

The threshold value that is used to determine if a pixel is part of the laser line changes overall accuracy: if it is too low, reddish points on the object not related to the laser may be seen as being part of the laser line, and if it is too high, some laser points may not be included in the point cloud.

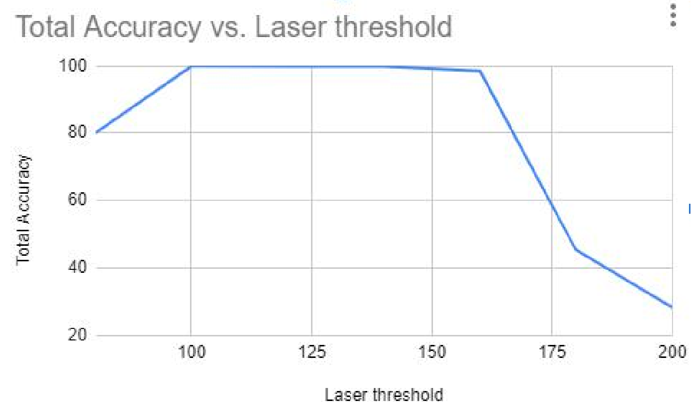


Figure 11: Graph of accuracy vs. laser threshold

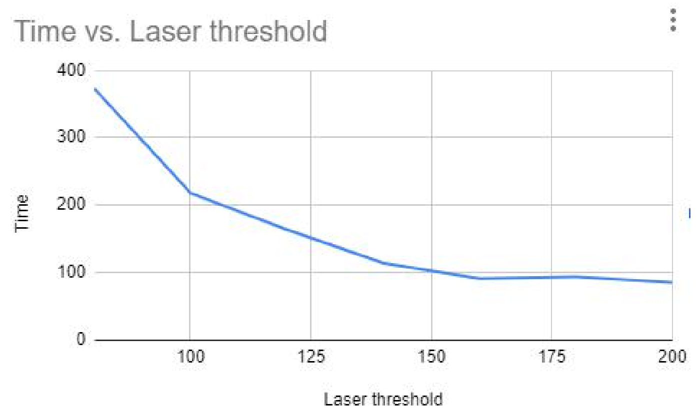


Figure 12: Graph of time vs. laser threshold

Analyzing the tradeoff between accuracy and time in Figure 11: Graph of accuracy vs. laser threshold and Figure 12: Graph of time vs. laser threshold, after laser threshold of 140 we begin to introduce a significant drop in accuracy. However, after this point the time efficiency does not improve significantly. Thus, we have determined that a laser threshold of 140 is optimal as a default parameter for our project. Note that all these parameters may be modified by the user as desired.

5.5 Iterative Closest Point Algorithm (ICP)

As mentioned, ICP is the algorithm that allows us to combine multiple scan angles of an object. Specifically, it takes two similar point clouds aligned differently and finds an alignment that reduces the Root Mean Squared Error between the point clouds the most. This RMSE number is given by Open3D and is related to the average distance between corresponding points on the two point clouds. We ran tests on the monkey mesh to determine the effectiveness of ICP.

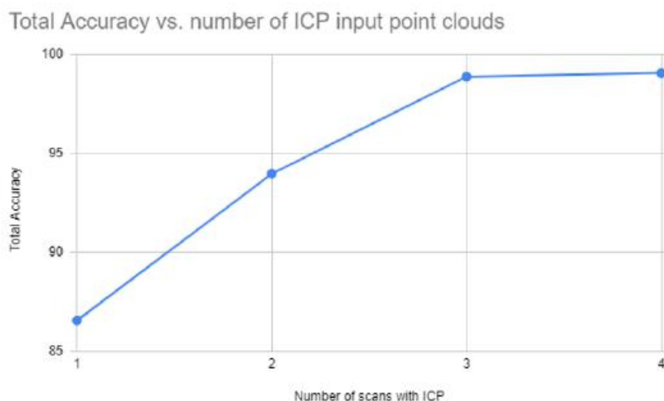


Figure 13: Graph of accuracy vs. number of ICP scans

Looking at the graph in Figure 13: Graph of accuracy vs. number of ICP scans, we can see that accuracy drastically increases up to 3 point clouds or angles used, after which accuracy gains are marginal. This is because for the monkey mesh, the bottom and ears of the monkey were occluded in the original scan, thus adding an angle which allowed the laser and camera to both see the bottom of the monkey, as well as another angle for the ears, improved our accuracy by quite a bit. However, the main tradeoff with adding scan angles is time - both time to convert the scanned images into point clouds and time to run ICP. The number of point clouds needed to achieve good accuracy with ICP also depends on the object. With many of the objects we tested, they had few concavities and two scan angles was enough to achieve reasonable accuracy. For some other objects, perhaps 4 or 5 scan angles were required since those objects had many obscured parts or weird shapes. Thus, the graph shown is a specific case meant to prove the effectiveness of combining multiple scan angles with ICP. In all cases, however, the minimum of scan angles required would be 2 because the bottom of the object is always obscured in the first scan.

We capped our maximum number of ICP iterations at 2000. This is because a large majority of our scans run at 150 to 400 iterations, but 2000 for max iterations means that the ICP part of the scanning process will never run more than 2 minutes (400 iterations takes less than 30 seconds). This number allows us to not obstruct the accuracy of a majority of our scans, while capping the total maximum time at a reasonable amount. Note that the fewer iterations that ICP has, the more randomness occurs as well. Initially, one challenge we faced was where we ran ICP at 30 maximum iterations, not realizing that most objects take more than a hundred to reach convergence for the algorithm, which caused our verification pipeline to spew out accuracy numbers that differed drastically between each run. Thus, our main objective is to cap the maximum ICP iterations such that it does not reduce the number of iterations *most* objects take to reach convergence.

5.6 Triangulation Algorithms

We have tested out multiple triangulation algorithms to see which one meets our requirements best. However, the general flow of all the triangulation is roughly the same.

Algorithm 1 Generic Triangulation Flow

Input: $in_filename$ - point cloud as PCD format
Output: $out_filename$ - output mesh as STL or OBJ format

pcd ← Open3D reads $in_filename$

cloud ← process *pcd* to necessary format

Estimate normals from *cloud* if necessary

volume ← perform triangulation from *cloud*

mesh ← process *volume* to necessary format and remove irrelevant data;

Store *mesh* as STL or OBJ file format as specified by user

5.6.1 Screened Poisson Reconstruction

The Screened Poisson reconstruction algorithm takes in input data which consists of a set of points and inward-facing normals of those points. The goal of this reconstruction algorithm is to reconstruct a watertight triangulated mesh by approximating the indicator function and extracting the isosurface This algorithm derives a relationship between the gradient of the indicator function and an integral of the surface normal field. It then approximates this surface integral by a summation over the given oriented point samples. Finally, it reconstruct the indicator

function from this gradient field as a Poisson problem. The algorithm then solves the poisson problem from the vector field (Kazhdan, Bolitho, Hoppe, 2013) [8].

In our program, we used `Open3D` library function to generate the mesh. This function computes a triangle mesh from an oriented point cloud and it implements the Screened Poisson Reconstruction proposed in Kazhdan and Hoppe [4].

`Open3D` allows for multiple paramaters for us to test and analyze:

- `depth`: maximum depth of the tree that will be used for surface reconstruction
- `scale`: specifies the ratio between the diameter of the cube used for reconstruction and the diameter of the samples' bounding cube.
- `linear_fit`: whether or not to use linear interpolation to estimate the positions of the iso-vertices.

Screened Poisson is one of the faster algorithms and works really well for watertight meshes. However, since our user story is mainly for archaeological documentation, a lot of archaeological objects are not in their full shapes and have cracks. Additionally, Screened Poisson requires an accurate estimation of normals of the points in the point cloud, which is difficult to do efficiently when the mesh object has holes or cracks.

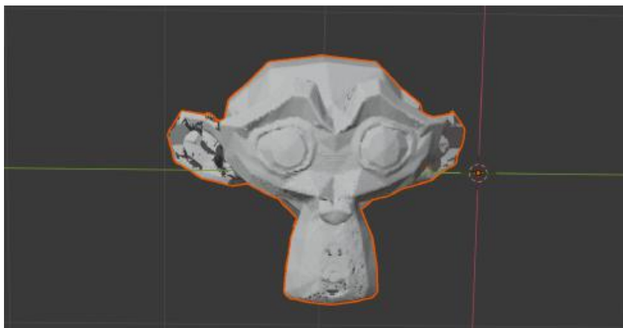


Figure 14: Monkey Mesh Triangulation using Screened Poisson Reconstruction

We have tried to optimize the algorithm by better approximating the normals. We used `Open3D` library to estimate the normals. The program uses a `KDTree` and search the neighbors around each point and tries to estimate the normal for each point. We tried to increase the number of neighbors looked at each point and avoid using non-interative method to extract the eigenvector from the covariance matrix to get a more stable numerical values, but the normals were still not well estimated for objects with cracks. We also can't just simply point the normals inward. Thus, our program couldn't really use this algorithm as it only works well for watertight meshes.

5.6.2 Ball Pivoting Reconstruction

The Ball Pivoting reconstruction algorithm computes a triangle mesh interpolating a given point cloud by a simple concept, three points form a triangle if a ball of a specified radius touches them without containing any other point. Starting with a seed triangle, the ball pivots around an edge until it touches another point, forming another triangle. The process continues until all reachable edges have been tried, and then starts from another seed triangle, until all points have been considered. The process can then be repeated with a ball of larger radius to handle uneven sampling densities (Bernardini, Mittleman, Rushmeier, Silva, Taubin, 1999) [1]. This algorithm is very accurate and can satisfy our accuracy requirements if we specify the ball with an appropriate radius and allows the algorithm enough time to consider all the points and repeats with a ball of larger radius.

In our program, we used `Open3D` library function to implement this algorithm. This function computes a triangle mesh from an oriented point cloud. This implements the Ball Pivoting algorithm proposed in F. Bernardini et al., "The ball-pivoting algorithm for surface reconstruction", 1999. The implementation is also based on the algorithms outlined in Digne, "An Analysis and Implementation of a Parallel Ball Pivoting Algorithm", 2014 [4].

`Open3D` allows for the radii of the ball that are used for surface reconstruction.

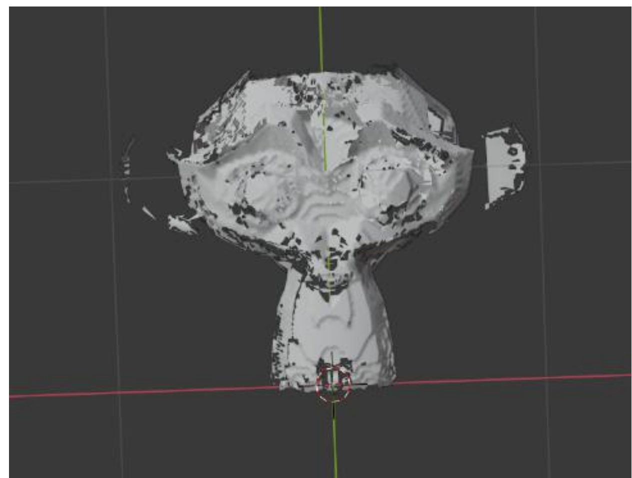


Figure 15: Monkey Mesh Triangulation using Ball Pivoting Reconstruction

In our program, we computed the base radii by computing the mean of the distances from the nearest neighbors for each point. We then multiply this base radii by different constants to achieve the accuracy we require. However, as the accuracy increases, the time it takes to triangulate the mesh also increases. We are unable to achieve the accuracy required within the time constraint.

5.6.3 Delaunay Triangulation Reconstruction

Delaunay triangulation is a triangulation of the convex hull of the points in the diagram in which every circumcircle of a triangle is an empty circle [3]. Note that a convex hull of a set of points is defined as the smallest convex polygon, that encloses all of the points in the set. In our case, since we are performing 3D triangulation, the 3-D set of points is composed of tetrahedra instead of just triangles. A 3-D Delaunay triangulation produces tetrahedra that satisfy the empty circumsphere criterion instead of the 2D empty circle circumcircle criterion, ensuring that the circumsphere associated with each tetrahedron contains no other point in its interior [15]. Delaunay triangulation tries to maximize the minimum angle within the triangles generated while connecting points to their nearest neighbors.

In our program, we used `Pyvista` library function to implement this algorithm. `Pyvista` implementation of Delaunay 3D triangulation also helps smooth out a rugged mesh. In this function, we can customize multiple parameters [5].

- `alpha`: distance value where for a non-zero alpha value, only verts, edges, faces, or tetra contained within the circumsphere (of radius alpha) will be output.
- `tol`: tolerance to control discarding of closely spaced points
- `offset`: multiplier to control the size of the initial, bounding Delaunay triangulation

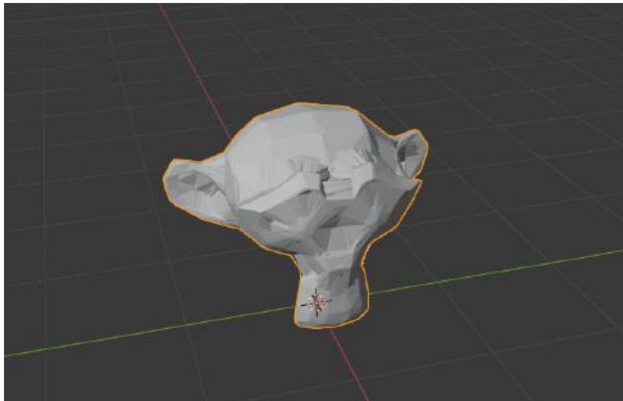


Figure 16: Monkey Mesh Triangulation using Delaunay Triangulation

For our input data, the tolerance doesn't really matter since all the points in the point cloud are relevant points on the laser line created from our point cloud generation subsystem. The only main parameter to consider is the alpha value. By decreasing the alpha value, the tetrahedron sizes get smaller and thus more representative of the ground truth mesh. However, if the alpha value gets too small, the algorithm fails to connect some points which are further apart to each other.

Analysis

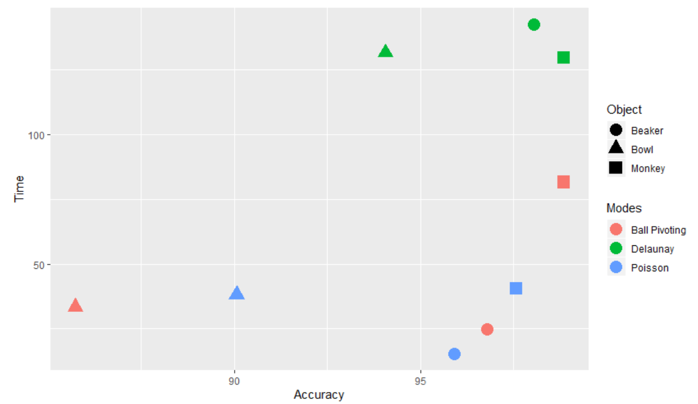


Figure 17: Comparison between triangulation algorithms

Note that the time in this graph is only triangulation time ran on a MacOS and not the benchmark computer.

Algorithm	Object	Result
Delaunay	Monkey	Pass
Delaunay	Bowl	Pass
Delaunay	Beaker	Pass
Screened Poisson	Monkey	Pass
Screened Poisson	Bowl	Fail
Screened Poisson	Beaker	Fail
Ball Pivoting	Monkey	Fail
Ball Pivoting	Bowl	Fail
Ball Pivoting	Beaker	Pass

Table 1: Accuracy Results Triangulation Algorithms

Note that we only look at comparing the output mesh to the ground truth mesh and not from ground truth mesh to output mesh here in Table 1: Accuracy Results Triangulation Algorithms above.

From the graph above in Figure 17: Comparison between triangulation algorithms, we can see that Screened Poisson is generally faster than the other two algorithms but has lower accuracy values. The overall hurdle of the Screened Poisson algorithm is that the algorithm generates additional points to match the vector field generated from the point cloud. Assuming our point cloud has high accuracy but our normal estimation is inaccurate, the Screened Poisson mesh will generate additional geometry where there are no points in the point cloud. This additional generation of vertices is the main cause of error for Screened Poisson, and its main source is the difficulty to accurately estimate point normals. The good time efficiency of Screened Poisson does not make up for its significantly poorer reconstruction accuracy.

The ball pivoting algorithm is slightly more time-efficient than Delaunay triangulation and has a slightly lower accuracy results. This may seem sufficient for our

project requirements; however, we also have another accuracy that requires all the distances between the ground truth mesh and the output mesh to be within 5% of the longest axis to take care of any outliers. The fundamental technique of the ball pivoting algorithm is to iteratively build the mesh outward from a starting set of vertices, much like a graph traversal algorithm. This procedure is carried out multiple times for multiple radii of ball size, to be able to generate triangles for both fine and course grain geometry. Our experiments only utilized a single radius to perform ball pivoting, since adding additional radii no longer meets our timing requirements. However, we suspect that with sufficient experimentation with the parameters of ball pivoting radii, we may meet the accuracy requirements, however the mesh triangulation would take significantly longer that our time efficiency requirement. With two radii, the algorithm takes upwards of 15 minutes to execute.

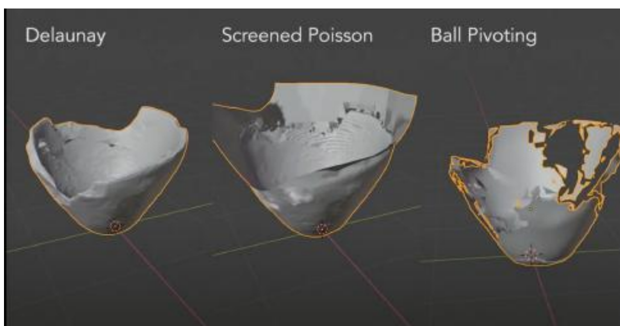


Figure 18: Mesh Outputs from Different Triangulation Algorithms

Although Delaunay triangulation takes a decent amount of time, it still passes our timing requirement and passes both of our accuracy requirements. Thus, Delaunay triangulation is the final algorithm we are using in our default pipeline. The main reason Delaunay is a great choice for our requirement is that it both uses the original points of the point cloud as vertices, and executes within our time efficiency requirement. Methods such as Screened Poisson generate new points as vertices based on a computed vector field from the point normals. Even if our point cloud is extremely accurate, if our normals are not, these generated vertices will harm the accuracy significantly. Ball pivoting, however, does use the original point cloud points as vertices, but in order for the triangles to cover the entire surface of the mesh, multiple ball radii must be used, which does not meet our timing requirement. Delaunay is a much faster algorithm to execute, and simply creates triangles between point cloud points greedily where there are no intersections. Because of this, Delaunay both meets our time efficiency requirement, and is accurate to the point cloud geometry.

6 SYSTEM DESCRIPTION

The final system is now solely software. As shown in Figure 38: Final System Specification Diagram in Appendix B, the system contains multiple subsystems with the algorithmic and control subsystem integrating other subsystems together. Other subsystems include Image Laser Detection, Point Cloud Generation, Background and Turntable Point Filtering, Iterative Closest Point Algorithm, Triangulation, Visualization and Debug System, and Verification.

6.1 Algorithmic and Control Subsystem

The algorithmic and control subsystem consists of our driver script which integrates and runs other subsystems including image-laser detection, point-cloud generation, background and turntable point filtering, iterative closest point algorithm, and mesh triangulation, as well as accuracy computation for testing and validation and visualization and debug system. Algorithms and descriptions of their implementations are described in detail in the remaining subsections of the system description.

```
usage: driver.py [-h] -m MODE [-v] [-d] -s MAIN_DIRECTORY [-o OUT_FILENAME]
                 [-n NUM_ITERS] [-vf VERIFY_FILENAME] [-l LASER_THRESHOLD]
                 [-w WINDOW_LEN] [-p PIXEL_SKIP] [-f IMAGE_SKIP]
                 [-df | -ds | -af | -as | -sp | -bp] [-t TRUTH_FILENAME]
                 [-k NUM_KD_TREE_NEIGHBORS] [-i NUM_IPC_POINTS]

optional arguments:
  -h, --help            show this help message and exit
  -m MODE, --mode MODE  <mode (f: full pipeline, p: generate pcds in temp
                        folder, t: start with pcds in directory)>
  -v, --verbose         verbose mode
  -d, --display         display mode
  -s MAIN_DIRECTORY, --input_directory MAIN_DIRECTORY
                        <input_directory>
  -o OUT_FILENAME, --output_filename OUT_FILENAME
                        [output_filename (otherwise <scan_directory>.obj)]
  -n NUM_ITERS, --num_iters NUM_ITERS
                        <num_iters>
  -vf VERIFY_FILENAME, --verify_filename VERIFY_FILENAME
                        [verify_filename (process result into csv)]
  -l LASER_THRESHOLD, --laser_threshold LASER_THRESHOLD
                        [laser_threshold]
  -w WINDOW_LEN, --window_len WINDOW_LEN
                        [window_len]
  -p PIXEL_SKIP, --pixel_skip PIXEL_SKIP
                        [pixel_skip]
  -f IMAGE_SKIP, --image_skip IMAGE_SKIP
                        [image_skip]
  -df, --delaunay_fast  fast delaunay triangulation
  -ds, --delaunay_slow  slow delaunay triangulation
  -af, --alpha_fast     fast alpha shape convex hull triangulation
  -as, --alpha_slow     slow alpha shape convex hull triangulation
  -sp, --poisson        screened poisson triangulation
  -bp, --ball_pivoting  ball pivoting triangulation
  -t TRUTH_FILENAME, --ground_truth_filename TRUTH_FILENAME
                        [ground_truth_filename (perform verification)]
  -k NUM_KD_TREE_NEIGHBORS, --num_kd_tree_neighbors NUM_KD_TREE_NEIGHBORS
                        [num_kd_tree_neighbors]
  -i NUM_IPC_POINTS, --num_ipc_points NUM_IPC_POINTS
                        [num_ipc_points]
```

Figure 19: Driver Usage

This driver script that takes in a directory containing directories of the scanned images and produces an output mesh with either an OBJ or an STL format. The script allows for multiple optional parameters to allow users to customize their program and to allow us to be able to test and perform a thorough tradeoff analysis.

Users can choose between 3 different modes: full pipeline, just generating point clouds as pcd files, or generating a triangular mesh from input pcd files. Users can

also choose to display point clouds and meshes when the program runs or to print out debugging messages.

For the point cloud generation of our pipeline, users can specify the laser threshold, window length, how many pixels to skip per scan, and how many images to skip per directory. If not specified, the program would run with our default parameters (laser threshold = 140, window length = 5, pixel skip = 4, and image skip = 1).

For the triangulation argument of our pipeline, users can choose between multiple triangulation algorithms to test out such as Delaunay (both faster and slower version), Screened Poisson, and Ball Pivoting algorithm. However, our program is default to Delaunay triangulation which works best for our requirements.

For verification purposes, users can specify the ground truth mesh filename and can provide the number of iterations and csv filename to output verification results. Users can also adjust other parameters such as number of kd tree neighbors and number of ipc points to speed up the verification process.

6.2 Image Laser Detection

This component of the algorithmic subsystem is responsible for detecting points on the laser line in a camera image. The basic idea is assume a model of the distribution of the laser line across its width as being a Gaussian distribution, such as in Figure 20: Laser Line Light Intensity Distribution.

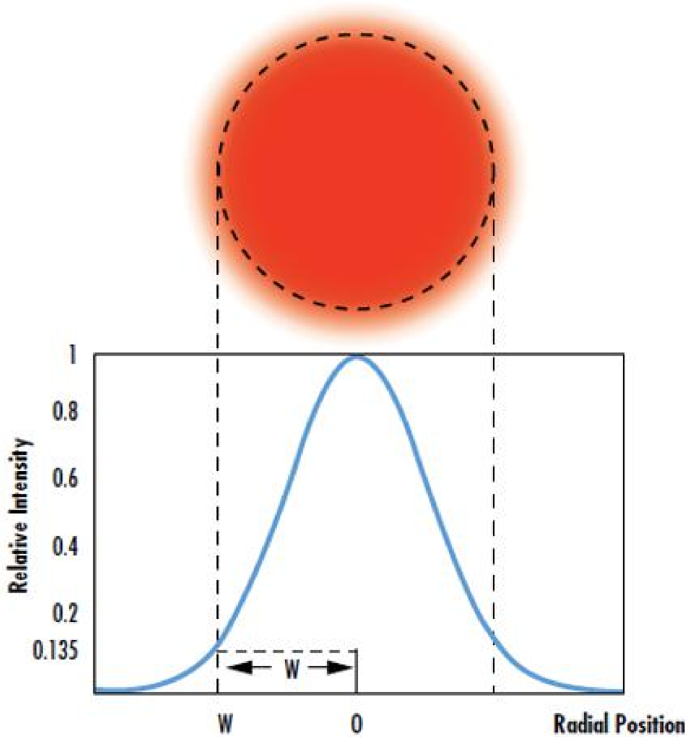


Figure 20: Laser Line Light Intensity Distribution

First we apply an intensity filter to extract higher values at pixels with color close to the wavelength of the laser light (650nm). Then we apply a horizontal Gaussian filter to enhance the Gaussian distribution of the laser intensity (since the laser line will be close to vertical in the camera image). For each row, the center of this Gaussian distribution is the horizontal location of the laser. Note that this suffers from the problem that only one point can be found for each row of the image. However, the case that a single row of the image has multiple parts of the laser line is a case of occlusion already, and these holes in the scan can be resolved by merging the results with an additional scan (single object multiple scans).

6.3 Camera and Scene Calibration

Blender provides a python interface to control and access its internal data. This functionality can be used to write a script which extracts properties of objects in the Blender scene. Specifically, we are able to search the Blender internal data for the Camera, Laser, and Turntable objects, and extract their parameters. Because we are able to extract this information, no calibration procedure is necessary, since the true values are given to us by the simulation.

6.4 Point Cloud Generation

A single Blender simulated scan consists of two independent components:

1. A directory of N scan images, ordered in the directory by alphabetical/numeric order. This ordering is done automatically by default blender output render animation settings.

2. A collection of scene parameters, including the transformation matrices of the camera, laser source, and turntable. These transformation matrices are vital to performing the many basis change computations required to generate the point cloud. The scene parameters also include the aspect ratio of the camera sensor (cw, ch), the aspect ratio of the rendered images (pw, ph), and the focal depth of the camera sensor (f).

From these components, we can derive the following scan constants for point cloud generation:

1. The angle between two subsequent scan images is $\frac{2\pi}{N}$. From this we can tag the i'th scan image as having been rotated about the turntable by $\frac{i \cdot 2\pi}{N}$, assuming linear interpolation was chosen to generate the scan animation. Linear interpolation is a good choice, since it does not prioritize any angle of the scan.

2. The origin points of the camera, laser source, and turntable as follows:

   - $c_0 = C \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

- $l_0 = L\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

- $t_0 = T\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

Where C, L, T is the camera, laser source, turntable transformation matrices appropriately, and c_0, l_0, t_0 are the origin coordinates of the camera, laser source, and turntable. Note that the transformation matrices represent affine functions in the 3-dimensional space, and thus must be 4-dimensional to offer both rotation and translation.

3. The axis of rotation, which is defined as:
   $a = T\begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}$
   We enforce that -Z is always the upward direction in the local coordinate space of the turntable.

4. The laser plane of rotation, whose normal is defined as:
   $l_n = L\begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \end{bmatrix} - l_0$
   We enforce that the Y and Z axes of the laser source's coordinate space correspond to points on the laser plane, so points only on -X must directly perpendicular. Similarly, we get the point on the laser plane closest to the origin by:
   $l_p = l_n * (l_n \cdot l_0)$

Once all of these parameters is determined, we can begin performing point cloud generation. The first step is to identify pixels in each image corresponding to the laser line, utilizing the formulation discussed in the previous section.

Below is a current diagram of all the components involved in point cloud generation:

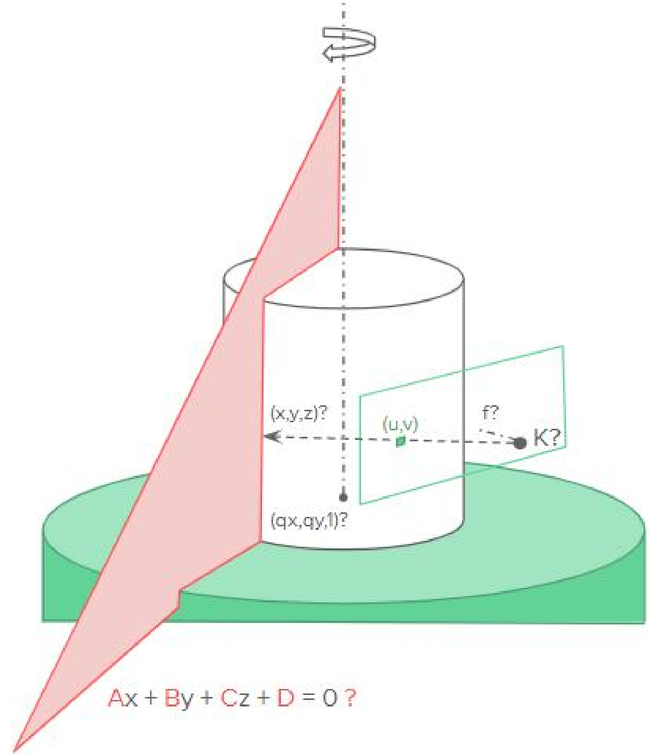


Figure 21: Ray-Plane Intersection

Here the point (u, v) is the pixel coordinate of a point in the on the laser line. A ray is cast from c_0 in the direction of normalized [9]:

$$dir = C\begin{bmatrix} (\frac{u}{pw} - 0.5) * cw \\ -(\frac{v}{ph} - 0.5) * ch \\ -f \\ 1 \end{bmatrix} - c_0 \tag{4}$$

This direction corresponds to the direction from the origin of the camera through the center of the pixel's position in the global coordinate space. Note the importance of the focal distance to computing this ray's direction.

The ray is then cast until it collides with the laser plane. The point of intersection is a point on the surface of the object, where the laser line is in the image. The "time" the ray needs to travel in the direction is given by the following formulation:

$$time = \frac{(l_p - c_0) \cdot l_n}{l_n \cdot dir} \tag{5}$$

And simply, the collision point is:

$$intersection = c_0 + time * dir \tag{6}$$

This intersection point is in world space coordinates, so it must be rotated around the rotational axis by the reverse angle of the turntable rotation about the turntable axis a to get the corresponding point in object space. All such points are computed and aggregated together for each image to form the point cloud for a scan.

6.5 Background and Turntable Point Filtering

All points on or below the turntable must be removed from being considered part of the point cloud. We know the axis of rotation and a position a the center of the turntable form our blender scene parameters. From this, we remove all points p where:

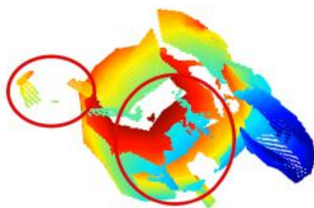
$$(p - t_0) \cdot a <= \epsilon \tag{7}$$

With a small ϵ to correctly capture points on the actual surface of turntable.

Then, points in background geometry are removed by limiting the *time* during ray-plane intersection to be less than the distance from the camera to the edge of the bounding box of the reasonable capture scene. This is limited as a bounding cylinder around the turntable by our maximum object size.

6.6 Iterative Closest Point Algorithm (ICP)

We will use an ICP (Iterative Closest Point) algorithm to combine different scans. This is needed in the use case where the user wants to combine a scan from another angle since some part of the object was occluded in the original scan. See Figure 22 for an example of the monkey mesh which had holes on the bottom and ears since it was occluded from the camera and laser in the original scan angle. The ICP algorithm determines the transformation between two point clouds from different angles of the object by matching similar or duplicate points. Similar to gradient descent, ICP works best when starting at a good starting point to avoid being stuck at local minima and also save computation time. This leads the ICP algorithm to have two steps - global and local registration. We can see the initial misaligned point clouds in Figure 23.



Holes in bottom and ear of the monkey

Figure 22: Original monkey point cloud with holes since only one scan angle is used

Figure 23: Original alignment before ICP is run

6.6.1 ICP - Global Registration

In both registration steps (registration referring to aligning the two point clouds), ICP tries to minimize the root mean squared error (provided by Open3D) between the two point clouds. However, initially, the meshes are extremely misaligned, which is why global registration is necessary. Global registration first downsamples the point clouds with a specified voxel size - we use 0.05 in `Open3D` which corresponds to 5mm (our input objects are generally 1 to 2m in Blender, but all the numbers can easily be scaled down by a factor of 10, it is just that we set up our render parameters and everything with that initial size). We also estimate normals for the downsampled point cloud and compute the FPFH feature for each point, which describes local geometry of a point. The downsampled point cloud and FPFH features are then used in RANSAC registration using Open3D. This gives us a rough alignment on the downsampled point cloud, the results of which can be seen in Figure 24.

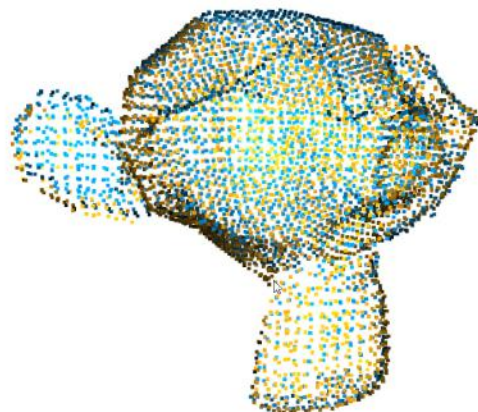


Figure 24: Result of global registration step in ICP

6.6.2 ICP - Local Registration

Now that we have a rough alignment of the point clouds, we can run local registration, which finds a much tighter alignment with the assumption of having a reasonable starting point. Again, this step works similar to gradient descent, finding a local minimum by minimizing the root mean squared error between the two point clouds. However, the distance threshold allowed between the two point clouds is much tighter for this step. The result can be seen in Figure 25.

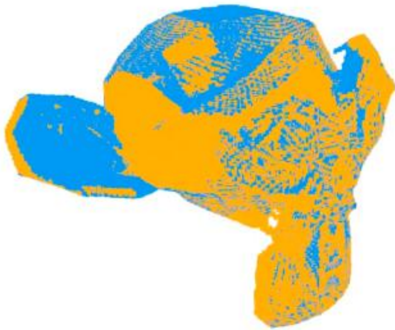


Figure 25: Result of local registration step in ICP

6.6.3 Results and caveats

The final result after ICP can be seen in Figure 26. We can see that the bottom and ears of the monkey are filled in after adding scan angles that specifically target these areas. Refer to the previous section of design tradeoffs to see our graph showing the effectiveness of adding additional ICP scans up to a certain degree.

The caveat of ICP is that it tries to match as many points as possible between the two point clouds. This means that if we are scanning something like a perfect sphere where the bottom is slightly obscured, since the sphere is uniform on all sides, a second scan angle would output the exact same point cloud as the first, and it would align the holes on these two point clouds. However, most objects, especially archaeological ones, have minimally distinct shapes which works with ICP almost all the time. In the case that this specific edge case comes up, a workaround would be for the user to know roughly the alignment between the two different scans, then input that was the initialization for local registration and *only run the local registration step.* For example, I can scan my sphere at 0 degrees of x-rotation, then rotate it by roughly 90 degrees on the x-axis to get the bottom of the sphere, then input that 90 degrees as initialization for local registration which would find the more precise alignment matrix.

Figure 26: Result of running ICP on the monkey model with 3 scan angles

6.7 Triangulation from Point Cloud to Mesh

The final point cloud is stored in a PCD (Point Cloud Data) file format. PCD files provide more flexibility and speed than other formats like STL/OBJ/PLY, and we use `Open3D` [4] and `Pyvista` [5] libraries to process this point cloud. `Open3D` library provides many useful functions such as estimating normals and performing triangulation (constructing a triangle mesh from XYZ points and normals). In our final design, we also use `Pyvista` library to perform triangulation. We think that implementing triangulation ourselves completely from scratch will be out of scope for this project and also unnecessary, since `Pyvista` library is efficient enough and results meet our project requirements. Since our data is just be a list of XYZ coordinates, we convert this to the PCD format in our point cloud generation subroutine to be used with the `Open3D` and `Pyvista` libraries (the PCD format is a list of XYZ coordinates with a few header lines in the beginning).

The triangulation algorithm works by maintaining a fringe list of points from which the mesh can be grown and slowly extending the mesh out until it covers all the points. More information was discussed in the Design Trade Studies section under Triangulation subsection. Our final triangulation algorithm has many parameters such as distance value to control output of the filter, tolerance to control discarding of closely spaced points, and multiplier to control the size of the initial, bounding Delaunay triangulation. We then store the output mesh as either an OBJ or STL file format as specified by the user. The flow of our final triangulation algorithm is as follows:

Algorithm 2 Delaunay Triangulation Flow

Input: *in_filename* - point cloud as PCD format
Output: *out_filename* - output mesh as STL or OBJ format

pcd ← Open3D reads *in_filename*

cloud ← Convert *pcd* to Pyvista Polydata for processing

volume ← Construct Delaunay Triangulation from *cloud*

mesh ← Extract outer surface of *volume*

Store *mesh* as STL or OBJ file format as specified by user

6.8 Visualization and Debug System

Our visualization, and debug system help us to easily pinpoint the cause of any errors our program has and help us understand which part of the process the program is at. Users can choose to visualize each step of the program: from point cloud generation, to ICP, to mesh triangulation. In addition to our own logging procedure, our system utilizes `Open3D` visualization modules and its Debug mode to output relevant information.

6.9 Verification

The verification engine takes as input two meshes, the original ground truth mesh and the reconstructed mesh from the simulated scan. Accuracy metrics are broken down into two numbers: forward accuracy and backward accuracy. Forward accuracy is how close the vertices of our reconstructed mesh are to the surface of the ground truth mesh. Backward accuracy is how close the vertices of the ground truth mesh are to our reconstructed mesh. Each type of accuracy is computed in the following way where src_mesh is being compared to target_mesh.

Algorithm 3 Accuracy Computation

Input: *src_mesh*, *target_mesh*
Input: *two_percent_dist*, *five_percent_dist*

centroid_tree = construct kd-tree of triangles of target_mesh by centroid ;
p0_tree = construct kd-tree of triangles of target_mesh by p0 of triangle ;
p1_tree = construct kd-tree of triangles of target_mesh by p1 of triangle ;
p2_tree = construct kd-tree of triangles of target_mesh by p2 of triangle ;

closest_triangles is an array of lists of the closest triangles from all kd-trees for each vertex id of the src_mesh.

src_vertices = get_vertices(src_mesh) ;
closest_triangles = get_nearest(centroid_tree, src_vertices) @ get_nearest(p0_tree, src_vertices) @ get_nearest(p1_tree, src_vertices) @ get_nearest(p2_tree, src_vertices) ;

num_two_percent = 0 ;
num_five_percent = 0 ;
for v in src_vertices **do**
  min_dist = inf
  for t in closest_triangles[v] **do**
    dist = point_triangle_distance(v, t) ;
    if dist < min_dist **then**
      min_dist = dist ;
    end if
  end for
  if min_dist > two_percent_dist **then**
    num_two_percent ++ ;
  end if
  if min_dist > five_percent_dist **then**
    num_five_percent ++ ;
  end if
end for

Now will all the distances computed, we determine the final accuracy result.

src_num_vertices = len(src_vertices) ;
accuracy = 100 * (1 - 0.9 * num_two_percent/src_num_vertices - num_five_percent/src_num_vertices) ;

Point to triangle distance is a vital component of determining the distance from a vertex to a mesh [11]. The procedure to do this is to project the point onto the plane of the triangle, then determine which region of the triangle the point is in on the plane, and finally based on that region perform the correct computation. The regions are illustrated below for the 2D case [6] in Figure 27.

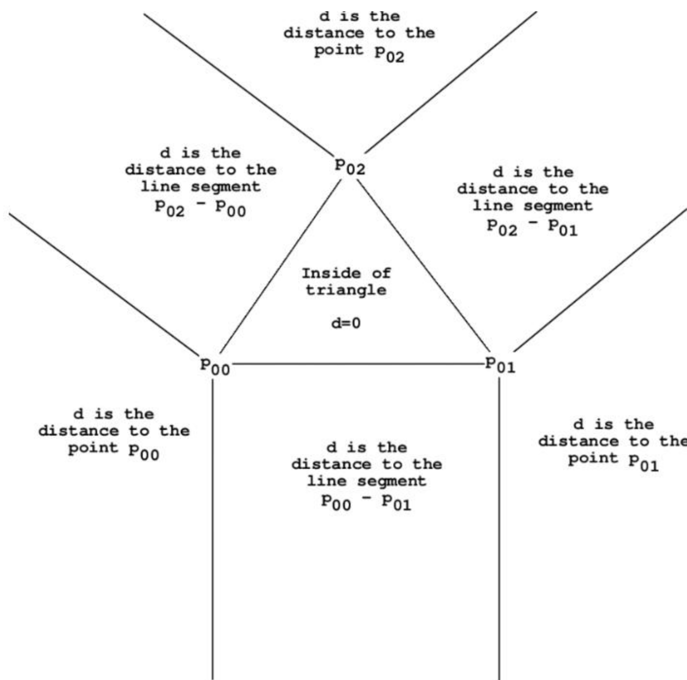


Figure 27: Regions of the triangle a point can be, 2D case

The computation is as follows for each of these cases in the 3D problem:

- Inside the triangle, the point distance is the distance from the point to the plane.
- To the line segment, the distance is the hypotenuse of the distance to the plane and the distance from the point on the plane to the line segment. This distance is computed by subtracting the portion of the vector from the projected point to a vertex that is parallel to the segment.
- To a point of the triangle, the distance is the hypotenuse of the distance to the plane and the distance from the point on the plane to the point of the triangle.

Finally, the accuracy for each direction, forward and backward, is averaged to obtain our final accuracy result for a mesh reconstruction. Note that the use of kd-trees is to reduce the overall computation necessary to perform the verification. Although this is not part of our timing requirement, it is very difficult to verify without this due to the time it takes (around 15 mins without the kd-tree optimization to 1 min with the kd-tree optimization).

Analysis of our accuracy and results is performed later later under the System Validation and Results section.

7 SYSTEM VALIDATION AND RESULTS

7.1 Validation

The final validation of metrics for our requirements was conducted as system-wide tests of ground truth archaeological 3D meshes. Free meshes were taken from The Royal Museums of Art and History [13] and Global Digital Heritage [7], whose 3D recordings are publicly hosted on Sketchfab.

Meshes from each of these sources were aggregated and we have narrowed our focus in a specifically few set of meshes:

1. **beaker.obj -** an ancient ceramic beaker
2. **bowl.obj -** an ancient ceramic bowl
3. **broken.obj -** a broken ceramic object
4. **carving.obj -** a carving on a tablet
5. **greek.obj -** an ancient greek vase
6. **mould.obj -** a stone mould for metal working
7. **pendant.obj -** an engraved pendant
8. **plaque.obj -** a metal plaque
9. **monkey.obj -** a monkey head provided by blender

For all of the tests, we used our full pipeline (simulated scans to meshes), with default parameters (frame skip, pixel skip, laser threshold, delaunay triangulation). To measure timing efficiency, the clock is started as soon as point cloud generation begins, and is stopped as soon as the mesh is ready in file format. To measure accuracy, the reconstructed mesh is compared against the ground truth mesh and two metrics are analysed: does the accuracy meet the requirement, and what is the accuracy numeric metric.

The dataset we have chosen includes challenging objects, such as broken, which has significant holes in it, and pendant, which is flat and does not have much surface for the laser and camera to be aligned in the scan. Delaunay triangulation aims to solve the hole issue, and ICP aims to solve the occluded points issue.

7.2 Results

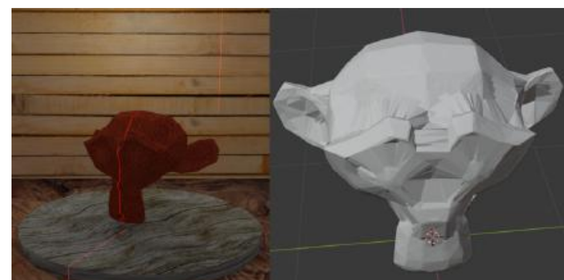


Figure 28: Monkey Mesh Result

Input Object	Forward Accuracy	Backward Accuracy	Total Accuracy	Accuracy Result	Total Time (s)	Time Result
beaker.obj	99.93%	85.28%	92.65%	Pass	87.00	Pass
bowl.obj	100.00%	89.94%	94.97%	Pass	76.14	Pass
broken.obj	100.00%	69.17%	75.88%	Pass	66.54	Pass
carving.obj	100.00%	65.85%	81.35%	Pass	36.54	Pass
greek.obj	99.87%	96.95%	98.52%	Pass	89.68	Pass
mould.obj	99.45%	82.43%	90.94%	Pass	54.10	Pass
pendant.obj	99.16%	73.81%	86.49%	Pass	31.33	Pass
plaque.obj	97.13%	71.54%	84.34%	Pass	29.13	Pass
monkey.obj	100.00%	97.74%	98.87%	Pass	101.02	Pass

Table 2: Accuracy Results

Figure 29: Bowl Mesh Result

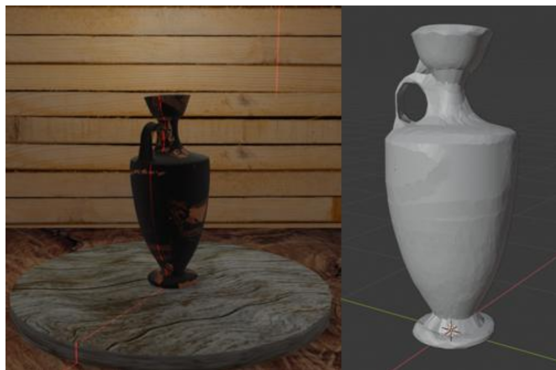


Figure 30: Greek Mesh Result

As mentioned in the Design Requirements section (3) under Accuracy subsection, the accuracy number that we use is computed by looking at the distance between the vertices of the mesh we generate and the surface of the ground truth mesh, getting a percentage that is within 2% of the longest axis of the ground truth mesh. This process is repeated the other way around, comparing vertices of the ground truth mesh to the surface of our generated mesh, and the average of these two numbers is our final accuracy number. Note that the accuracy number is a measure of how well the reconstructed mesh is both accurate to and completely covers the ground truth mesh. Our requirement only requires it to be accurate to the ground truth mesh. Because of this, forward accuracy is the relevant metric to our accuracy requirement, and total accuracy is a more general measurement that would be nice to achieve. (Backward accuracy identifies how well the reconstruction covers the entire ground truth mesh, and is not a measure of how well the points of the reconstruction are accurate to the ground truth mesh.)

See Table 2: Accuracy Results for data.

8 PROJECT MANAGEMENT

8.1 Schedule

Our schedule has slightly changed since we decided to remove the hardware components due to the Coronavirus situation. Please refer to Figure 39: Gantt Chart in Appendix B for the full Gantt Chart. Our original schedule was built around the platform and sensor assembly. We know converted the time allocated for that portion to accurately simulating scanned images instead and still continued to work on the software pipeline as planned. However, we also allocated more time to simulate noises and edge cases to test our project as closely to the real world as possible.

8.2 Team Member Responsibilities

Our team divides work among each team member equally. The team deals with logistics, integration, and decision-making together but each member still has his main tasks assigned as follows:

Jeremy:

- Blender simulation
- ICP for combining multiple scans
- Demo video animation and editing

Chakara:

- Original Hardware Design
- Triangulation
- Driver script

- Output validation code

Alex:

- Laser stripe detection and mapping to world coordinates
- Point cloud generation from sensor data
- Driver script
- Testing benchmark code and verification script

8.3 Software Interfaces

The interfaces between the various software components of our pipeline is an important aspect to the overall functionality of the system. Generally, since all of the software components are running with the CPU on a single machine, the stages of the pipeline are executed as in-order code. This means the output from one state is passed in as the input to the next stage. However, some of the interfaces require the prior generation of files in storage to use for the next stage. Below is an overview of the various interfaces between pipeline stages:

1. Blender to Point Cloud Generation: For this interface, the Point Cloud Generation script takes as input a directory of scan images from blender, and a file constants.py containing the Blender scene parameters. Thus, this interface utilizes file storage to communicate information.
2. Point Cloud Generation to ICP: For this interface, ICP takes in a directory of point cloud files (PCD) to merge. Thus, this interface utilizes file storage to communicate information.
3. ICP to Mesh Triangulation: This interface also utilizes file storage for communication, via a single stored merged point cloud file.
4. Mesh Triangulation to Verification Engine: This interface also utilizes file storage for communication, via a single stored reconstructed mesh file (either OBJ or STL).
5. Driver Application to each pipeline stage: This interface consists of function calls to activate each driver stage with specific file arguments. The data flow between pipeline stages, even when orchestrated by the driver application, is via files in storage.
6. Pipeline Stages to `Open3D/PyVista` Libraries: Matrix data in `Numpy` format is passed in to library calls when needed throughout the stages to perform computation. Numpy is a popular library in python to perform data computation, and is supported by all of the utilized libraries in our project.

As a summary, data within a pipeline stage is a `Numpy` object, data passed between stages is in stored files, and control of the pipeline operation is done by function calls. This describes all of the interfaces in our software system and we do not feel that these details are pertinent to the general block diagram of our system, so they are not included there in the logical connections.

All code not in the libraries we are using (`Open3D/PyVista`) is hand-written by us. All code on the other side of that interface is open-source python code which implements some of the functionality that we require, so we use it instead of reinventing the wheel. As a summary of which code was developed and which code was borrowed/modified, please see Table 3: Software Breakdown.

8.4 Challenges

Throughout our design and implementation process, we have faced multiple challenges.

Part of the input object is obscured and potential holes in the point cloud

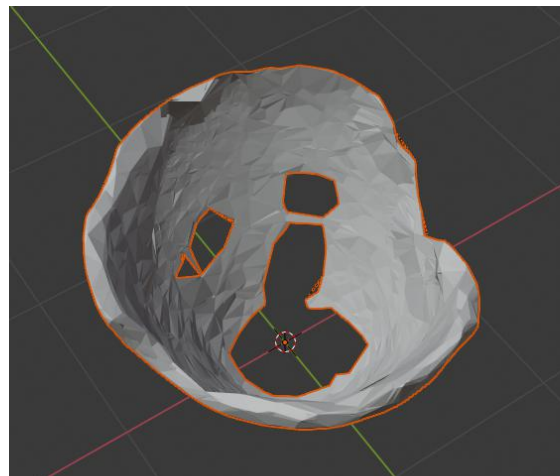


Figure 31: Bowl Mesh Output without ICP

Subroutine	Development	Description
Blender Simulation	Self-developed	-
Driver Module	Self-developed	-
Image Laser Detection	Self-developed	-
Ray-Plane Intersection	Self-developed	-
Iterative Closest Point (ICP)	Modified	Combines many Open3D functions for ICP and downsampling and FPFH generation
ICP Scaling Between Point Clouds	Self-developed	Handles edge case of point clouds scaled slightly off
Image Gaussian Filtering	Self-developed	-
Mesh Triangulation	Modified	Combines multiple Open3D and Pyvista functions
Visualization, and Debug System	Self-developed	Combines multiple Open3D features
Verification - KD-Tree Construction	Self-developed	-
Verification - Closest Points Querying	Self-developed	-
Verification - Data Processing	Self-developed	-

Table 3: Software Breakdown

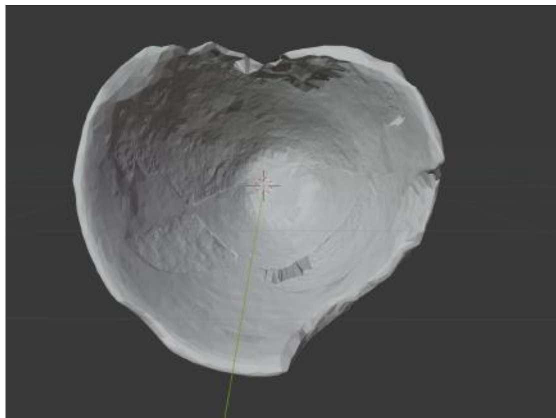


Figure 32: Bowl Mesh Output with ICP

In some cases, a part of the object is obscured, such as the bottom of a bowl or a vase. Thus, we have implemented ICP and created our scanning our pipeline to allow users to place objects at different angles and combine the generated point clouds.

Our algorithm does still struggle with certain objects that have many obscured parts or very deep concavities, since the camera cannot see the laser line in those parts of the object.

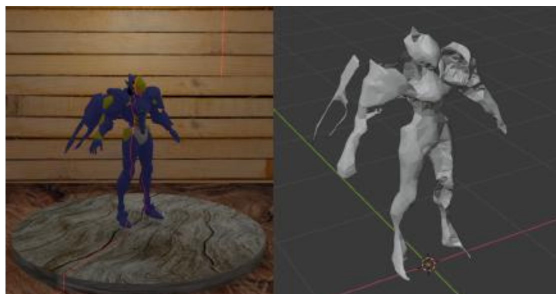


Figure 33: Obscured Object with Poor Result

Consider this example of a video game figure in Figure 33: Obscured Object with Poor Result, the camera cannot see the laser lines behind the wings and can't capture that data. We could potentially combine 10 different angles with ICP to get every single concavity of the object, but that would also increase our time cost tenfold. Our team comes different approaches that could potentially solve this issue which will be discussed later in the Summary section under Future Work subsection.

ICP Ordering

The order the point clouds are merged with ICP affects the final accuracy of the reconstructed mesh. It is difficult to know which ordering will result in the best accuracy, so as of now, our process merges in alphabetical order of point cloud file name. This is an area where we could improve our overall system slightly if we used a heuristic to decide which point cloud we should merge with next. However, the improvements to be gained are marginal and we did not want to waste resources dedicated to the small problem.

General Implementation Challenges

We have also encountered many challenges that we have resolved. For example, code never works the first time, and we have learned to develop logging and debugging mechanisms to fix issues early on. Next, we have had to iteratively redesign some of our software block diagram as we learned that some of the components were unnecessary. Finally, there was the logistical challenge of merging all the components into a single package via the driver application (integration), which was only able to be done with all of us working together using our expertise in each individual part we worked on.

Item	Price ($)	Total Price With Shipping ($)	Description
Nema 23 Stepper Motor	23.99	23.99	Motor for rotational mechanism
15 Inch Wooden Circle by Woodpeckers	14.99	14.99	Plywood for platform
Lazy Susan Bearing	27.19	27.19	Reduce friction in rotational mechanism
Acrylic Plexiglass Sheet, Clear	9.98	9.98	To create internal gear
DM542T Step Driver	38.99	38.99	Step Driver for NEMA 23
Neoprene Rubber Sheet Rolls	14.80	14.80	To add friction to the platform
Adafruit Line Laser Diode (1057)	8.95	17.94	Projected Laser Stripe
Webcamera usb 8MP 5-50mm Varifocal Lens	76.00	76.00	Camera
Nvidia Jetson Nano Developer Kit	99.00	117.00	Embedded Systems
256GB EVO Select Memory Card	42.96	42.96	MicroSD card for NVIDIA Jetson
MicroUSB Cable (1995)	9.00	17.99	MicroUSB cable for NVIDIA Jetson

Table 4: Project Components (Purchase)

8.5 Budget

Table 4: Project Components (Purchase) shows the project components that would be purchased. Note that some of the item names are shortened for readability.

This comes up with a total price of **$401.83**. *However, we no longer use any of the components except for us to model them in our Blender setup.* If we were to actually build the physical components, we would have $198.17 left in our budget which would still need to be used to purchase wood for supporting our platform and for risk management.

9 RELATED WORK

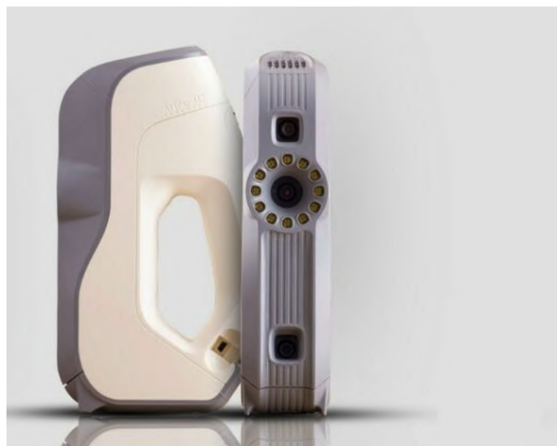


Figure 34: Eva Lite from Artec 3D

We are aware of other commercial 3D scanners that exist in the market. These scanners take in relatively similar input sizes as our initial design. However, most of the 3D scanners have much higher prices than our initial target cost of producing our 3D scanner with the hardware component (which is $600). Moreover, a lot of these models do not provide accuracy and timing data. Below are some examples in Table 5: Related Work Comparison.

There are a few reasons why our approach is more suitable for specific demographics than many of the 3D scanners on the market:

- Many commercial 3D scanners are much more expensive than our product.
- Many commercial 3D scanners do not advertise their accuracy guarantees and thus are not especially useful in high-accuracy requirement scenarios.
- Many commercial 3D scanners require a lot more from the user, such as setting up the scene with specific lighting conditions or the use of a hand-held object to perform the scan carefully. Our approach is completely hands-free and user-friendly.

For these reasons we see the potential of a strong demographic for our specific solution to the problem at hand, since it meets certain requirements that no commercial 3D scanners do (namely: affordability, accuracy guarantees, and ease of use).

10 SUMMARY

10.1 Future Work

If we have the opportunity to continue working on this project. There are a few things we would like to try.

Building the Hardware

We would like build the initial design including assembling all the hardware components to test if our software pipeline actually works with real camera, laser, rotational mechanism, and real-world environment settings.

Improving on Obscured Parts

We would like to try allowing position of the camera and the laser to be changing with the time. This should allow us to get inside objects such as vases and bowls. We might

Item	Price ($)	Approx. Max Input Size (cm)	Average Time (s)	Accuracy (mm)
V2 from Matter & Form	750	18 x 18 x 25	N/A	0.1
EINSCAN-SE from Shining 3D	1,399	20 x 20 x 20	120	N/A
RangeVision Smart	1,399	100 x 100 x 100	12	0.1
SLS Pro S3 from HP	3,995	50 x 50 x 50	N/A	0.05
Eva Lite from Artec 3D	9,800	N/A	N/A	0.5

Table 5: Related Work Comparison

also try using goose-neck cameras instead of our static camera that just stays outside the surface of the input object. The issue with the goose-neck camera approach is proximity of the mechanism to the ancient object, which should not be touched or damaged in any way. This is one of the reasons we did not go with a contact-based approach to reconstruction. Another potential idea is to add a reflector to be able to capture other obscured parts. Both these approaches would require us to adjust our laser detection and ray-plane intersection subroutines.

10.2 Lessons Learned

From working on this project, we learn that we should make clear and specific requirements that can be validated. We also learn that our designs, requirements, and testing must always be driven by the user-story, as exemplified in the selection of models we used during development and testing. This helps us make decisions that would potentially lead us to satisfying our requirements and thus reach our project goals and solve the problem we aim to. Moreover, we also learn that everything is unpredictable and our plans need to be adjustable. Because of the coronavirus sitaution, we had to adapt our project to still be able to achieve most of our goals. We adapted our plan as best as we could and since our requirements and user story were well-designed, we were able to still complete our project.

References

[1] F. Bernardini et al. "The ball-pivoting algorithm for surface reconstruction". In: *IEEE Transactions on Visualization and Computer Graphics* 5.4 (1999), pp. 349–359.

[2] Yan Cui et al. "3D shape scanning with a time-of-flight camera". In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE. 2010, pp. 1173–1180.

[3] *Delaunay triangulation*. 2020. URL: https://en.wikipedia.org/wiki/Delaunay_triangulation.

[4] Open3D Developers. *Open3D*. URL: http://www.open3d.org/.

[5] PyVista Developers. *PyVista*. URL: https://docs.pyvista.org/.

[6] *Distance Point Triangle*. 2020. URL: https://www.geometrictools.com/Documentation/DistancePoint3Triangle3.pdf.

[7] *Global Digital Heritage (@GlobalDigitalHeritage)*. URL: https://sketchfab.com/GlobalDigitalHeritage.

[8] Hugues Hoppe2 Michael Kazhdan Matthew Bolitho. *Poisson Surface Reconstruction*. http://sites.fas.harvard.edu/~cs277/papers/poissonrecon.pdf, year = 2013,

[9] Scratchapixel. 2014. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-plane-and-ray-disk-intersection.

[10] Steven M Seitz et al. "A comparison and evaluation of multi-view stereo reconstruction algorithms". In: *2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06)*. Vol. 1. IEEE. 2006, pp. 519–528.

[11] Joshua Shaffer. *Computes the distance between a point an triangle. Python*. URL: https://gist.github.com/joshuashaffer/99d58e4ccbd37ca5d96e.

[12] Gabriel Taubin, Daniel Moreno, and Douglas Lanman. "3d scanning for personal 3d printing: build your own desktop 3d scanner". In: *ACM SIGGRAPH 2014 Studio*. 2014, pp. 1–66.

[13] *The Royal Museums of Art and History (@kmkg-mrah)*. URL: https://sketchfab.com/kmkg-mrah.

[14] Klaus Thoeni et al. "A Comparison of Multi-view 3D Reconstruction of a Rock Wall using Several Cameras and a Laser Scanner". In: *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XL-5 (June 2014), pp. 573–580. DOI: 10.5194/isprsarchives-XL-5-573-2014.

[15] *Working with Delaunay Triangulations*. URL: https://www.mathworks.com/help/matlab/math/delaunay-triangulation.html.

[16] DroneBot Workshop. *Stepper Motors with Arduino – Bipolar & Unipolar*. https://dronebotworkshop.com/stepper-motors-with-arduino/. Feb. 2018.

[17] Michael Zollhöfer et al. "State of the Art on 3D Reconstruction with RGB-D Cameras". In: *Computer graphics forum*. Vol. 37. 2. Wiley Online Library. 2018, pp. 625–652.

Acknowledgements

We would like to thank the ECE department for providing the opportunity for us to work on this project. We would like to thank Professor Tamal Mukherjee, Professor Bill Nace, and Joe Zhao for giving us great feedback and advice and helping us mitigate our risks in our design decisions.

Appendix A

This section includes additional information regarding our initial design. The general idea of our initial design was discussed in the Initial Design section (2). However, our testing methods differ from the current ones. To be able to test this, we will have two possible methods where we would have tried both. The first method involves finding ground truth 3D models of common objects such as a solo cup or a coke bottle - however we may want to consider spray painting the coke bottle so that the transparency of the sides don't mess up the laser. The second method will be 3D printing 3D models that we find on the internet, then scanning these 3D printed models and comparing with the ground truth model. For the 3D printed models we will allow for an extra 1mm buffer since there is a fair bit of inaccuracy induced from 3D printing. We will compute the accuracy number by computing mesh vertex distances from each point in our constructed mesh to the surface of the ground truth mesh

Initial Design Requirements

a. Accuracy

Our accuracy requirement remains unchanged and was be discussed in the Design Requirements section (3).

b. Usability and Portability

Our next requirements involves usability and portability. The input object must be 5cm to 30cm along every axis, and have a maximum weight of 7kg. Thus, our platform will be tested with a 7kg load and we will see if the platform is able to withstand that weight without warping, as well as rotate the object without hindrance to rotational velocity. We also have other usability requirements such as the device being easy to setup and outputs a common 3D format that we will be able to input to a 3D printer. These requirements are easily evaluated and do not require any special quantitative tests. We will also test usability by doing user testing and evaluating survey responses.

c. Efficiency

We also have a requirement for efficiency. We will allow one minute total time for the scan including the rotation and the processing time. The rotation time should be well under 30 seconds, which gives more than 30 seconds for processing time, which will involve point cloud construction, filtering, and triangulation into a mesh. This will simply be measured by timing the process from pressing start to obtaining the 3D mesh. Note that the time for calibration will not be included since this is a one-time operation which will have amortized time cost across multiple successive scans.

Related to efficiency, we will also test software components we create against well-known open source library functions. This is due to our goal of implementing several components of the software pipeline to be optimized for the Nvidia Jetson GPU. We will first start off with open source code, but slowly replace components one by one with our optimized code. Thus, we must unit test all software components extensively, on real data and manufactured test cases. This includes filter routines, computer vision routines, mesh generation, and CUDA kernels. We will be testing for processing speed and correctness of these few components.

d. Affordability

Our final requirement is affordability. The whole system should cost less than \$600. This is also to ensure that we are bridging the gap since commercial 3D scanners cost much more than \$600.

Initial Design Trade Studies

In this section, we will discuss some design trade-offs and evaluation of different options for the design of several components of the project.

Scanning Sensor

Our final design for blender simulation setup still uses the scanning lasor setup we designed. The design tradeoff was discussed in the Design Trade Studies section (5).

Platform Material

To fit our *Usability* requirement, the platform must be able to withstand an object with dimensions from 5cm to 30cm and weighing up to 7kg. The platform itself will be a circular disc with a diameter of approximately 38.1cm. We performed a rough estimation to compute the stress that the platform needs to be able to handle. From our maximum input object's mass of 7kg, the maximum gravitational force that it can exert on the platform is around $68.67N$. The lazy susan bearing our team might end up using has an inner diameter of around 19.5cm (or 0.0975m radius). This would give us an area of around $0.03m^2$ that

would not have any support. We simplified the stress analysis of this design down but it should still give a good estimation of the stress the object would apply.

$$\text{Stress} = \frac{F}{A} = \frac{68.67N}{0.03m^2}$$

which is around $2300\frac{N}{m^2}$.

After getting the stress, we did more research on the material that would be able to handle this much stress, be cost-efficient, easily accessible, and easy to use (cut, coat, etc). We did some optimization based on cost and mass-to-stiffness ratio to narrow down the number of materials we had to do research on. Below is an image of the optimization graph. Note that we only looked into plastic and natural materials as they are easier to use and more easily accessible. The line in the second image is the optimization line.

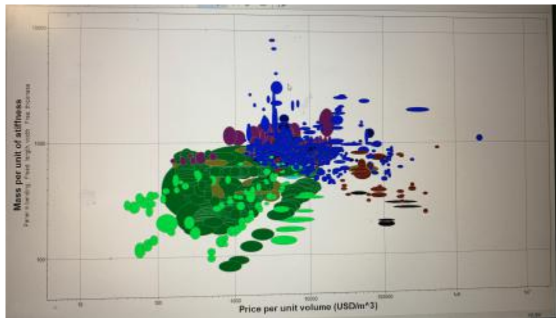


Figure 35: Material Optimization Graph

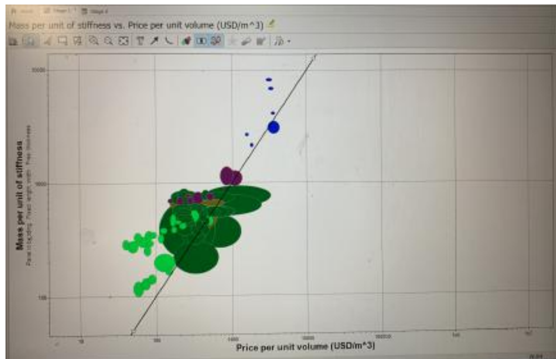


Figure 36: Optimized Material

After that, we narrowed the materials down more to 3 materials: plywood, Epoxy/hs carbon fiber, and balsa. Table 6: Material Tradeoffs Analysis in Appendix A shows the tradeoffs between different main properties that would affect our decision. Young's modulus, specific stiffness, and yield strength are mainly to see if the material would be able to handle the amount of stress the object would exert on it or not. The price per unit volume is to keep this within our project's constraint. The density is used to compute the mass of the platform (for computing the torque required and to stay within our Portability requirement).

From the table, we can see that carbon fiber is the strongest but is relatively expensive and might not suit our project well. Balsa is very light but is not as strong (even if the values here are still higher than the stress we computed, it might be because of the simplified stress analysis we did). Thus, our group decided to use plywood which is strong, inexpensive, easy-to-cut, and not too heavy. With plywood, the maximum mass our of platform would just be around $0.6kg$ (computed using density and dimensions of the platform).

Motor

The final part of the main platform design is to choose the right motor for the project. The main 2 motors we looked into to rotate the platform are the servo motor and the stepper motor. A servo motor is a motor coupled with a feedback sensor to facilitate with positioning for precise velocity and acceleration. A stepper motor, on the other hand, is a motor that divides a full rotation into a number of equal steps.

To figure out the motor used, we computed the torque required to rotate the platform with the maximum object size. From the density and dimensions of the platform, we computed that the plywood platform would weight around 0.64kg and carbon fiber would be around 1.2kg (We still accounted for the heaviest material and strongest material in case of a change in the future). From that we computed the moments of inertia which is around $0.024kgm^2$. For the input object, we used maximum size and different dimensions and shapes to cover most cases, the maximum moments of inertia computed is around $0.1575kgm^2$. Thus, the total maximum moments of inertia is less than $0.2kgm^2$. Refer to Table 7: Input Object Moments of Inertia in Appendix A for the full calculations. To get the torque, We also estimated the angular acceleration needed. We need at least a rotation of 0.0033 rad per step to capture enough data to meet our accuracy requirement. Assuming that 10% of the time requirement, which is 6s, can be used for data capturing (so that we have more buffer for the algorithmic part even if we anticipate 30s for data capturing), we would get that the angular velocity is around $3\frac{rad}{s}$. Assuming we want our motor to be able to reach that velocity fast enough (0.5s), we have an estimated acceleration of $2.094\frac{rad}{s^2}$. From here, the estimated torque needed to rotate the platform is around

$$\tau = I * \alpha = \mathbf{0.4188Nm.}$$

Since we need a high torque and from our algorithm we would need an accurate step, the stepper motor is preferred. The two stepper motor we looked into are the NEMA 17 and NEMA 23. NEMA 17 has a holding torque of 0.45Nm, and NEMA 23 has a holding torque of 1.26Nm. Even though NEMA 17 seems like it might be enough, in the computation, I neglected the friction which would drastically affect the torque the motor has to supply. Moreover, I also neglected the energy that would be lost through using the internal gear to rotate the platform. Since NEMA 23

is not that much more expensive, we believed NEMA 23 would fit our project best.

For the step driver, we just need one that can provide required current for NEMA 23. We decided to go with DM542T step driver since it could go up to 4.2A which is enough for NEMA 23. Moreover, it is relatively inexpensive and has 1/128 Micro-step Resolutions. This means that we can have more rotations per revolution and could achieve a smaller step angle (much smaller than the required 0.0033 rad if needed).

Initial Project Management

Initial Team Member Responsibilities

Our team divides work among each team member equally. The team deals with logistics, integration, and decision-making together. Initially, each member has his main tasks as follows

Jeremy:

- Testing database construction
- Outlier removal and noise reduction
- Point cloud triangulation
- ICP for combining multiple scans

Chakara:

- Rotational mechanism
- Motor controller/driver
- Platform construction
- Optimization of software components for GPU

Alex:

- Laser stripe detection and mapping to world coordinates
- Point cloud generation from sensor data
- Camera calibration code
- Testing benchmark code

Initial Risk Management

Our design still consists of some potential risks. We have several plans to mitigate these risks, which would require more implementation complexity on our part.

- **Part of the input object is obscured:**
  We plan on dealing with this issue by merging multiple scans of multiple angles of the objects to get more perspectives. We will be using pairwise registration to merge the point clouds generated from these scans.
- **Vibrational noise:**
  We can add an additional laser stripe for triangulation to correct for the noise. We can also modify the sensor setup to be disjoint from the platform if there is too much vibration induced from the stepper motor mechanism.
- **The stepper motor angle data is inaccurate:**
  There is a very low possibility this would happen but if it does, we can combine the computed angle from the motor with computer vision to get a more accurate angle. However, this may require placing some sort of calibration mat underneath the object to be able to track the angle rotated throughout the process.
- **The laser stripe doesn't have enough intensity:**
  We can buy a stronger laser since we have a buffer in our budget that amounts to around $200.
- **Potential holes in the point cloud:**
  We can merge multiple scans of the object, again using pairwise registration. We can also perform hole-filling mesh triangulation.
- **NVIDIA Jetson Nano and motor driver integration:**
  Although using NVIDIA Jetson Nano to control the motor step driver is possible, there is a possibility that it would not work since we have not found an article supporting the use of NVIDIA Jetson to control the DM542T motor driver model. If they can't integrate, then we plan on borrowing an Arduino from the Capstone course to control the motor driver instead.

Appendix B

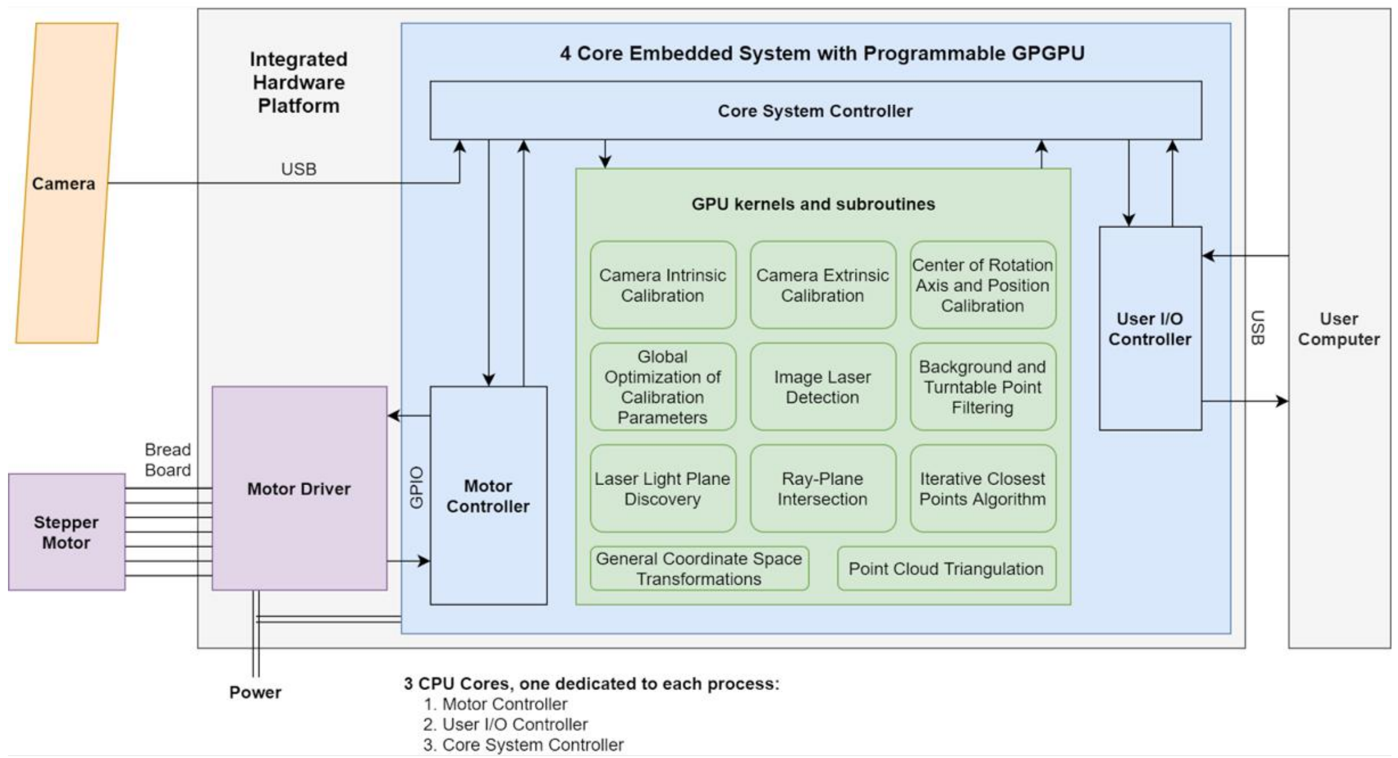


Figure 37: Initial System Specification Diagram

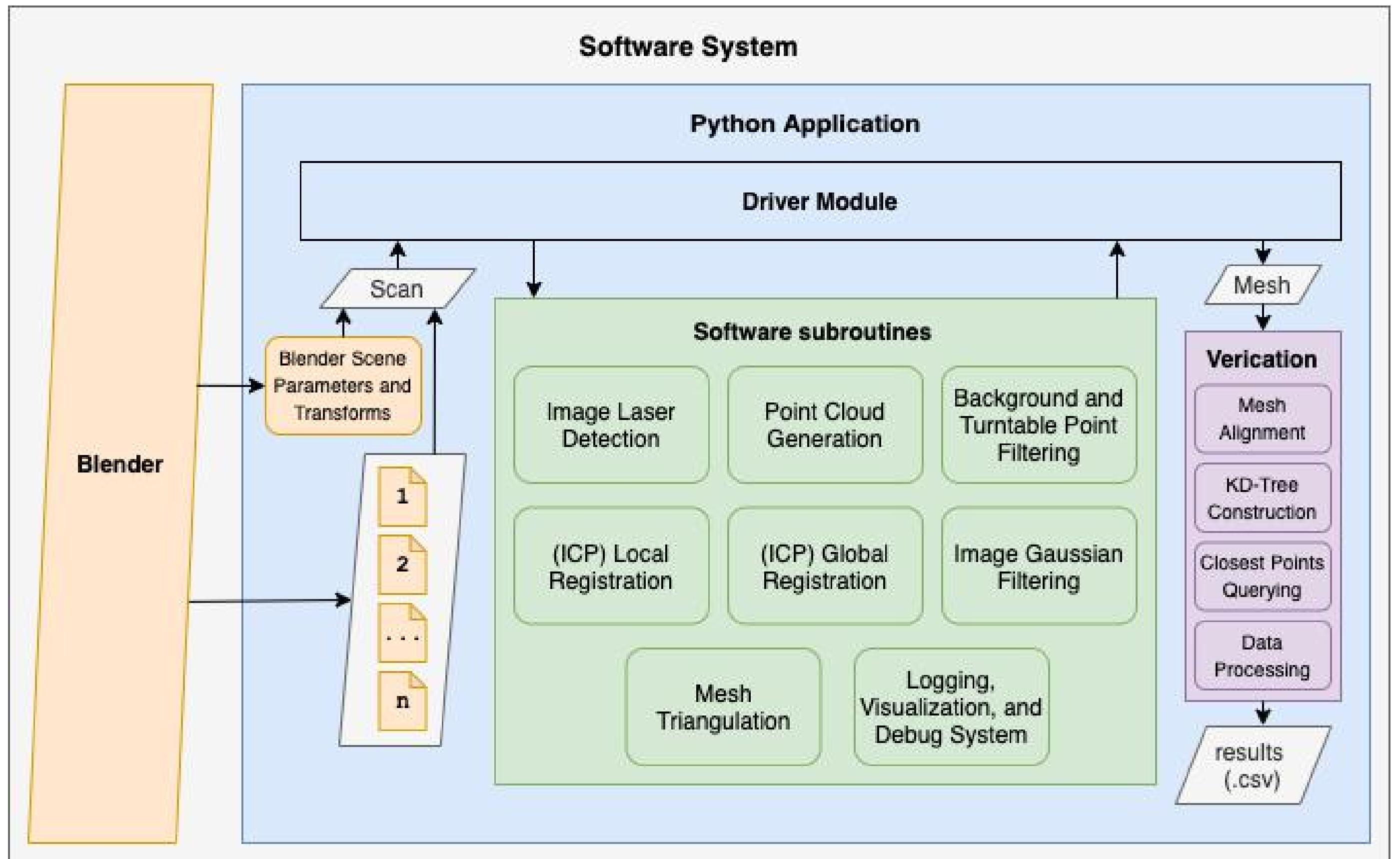
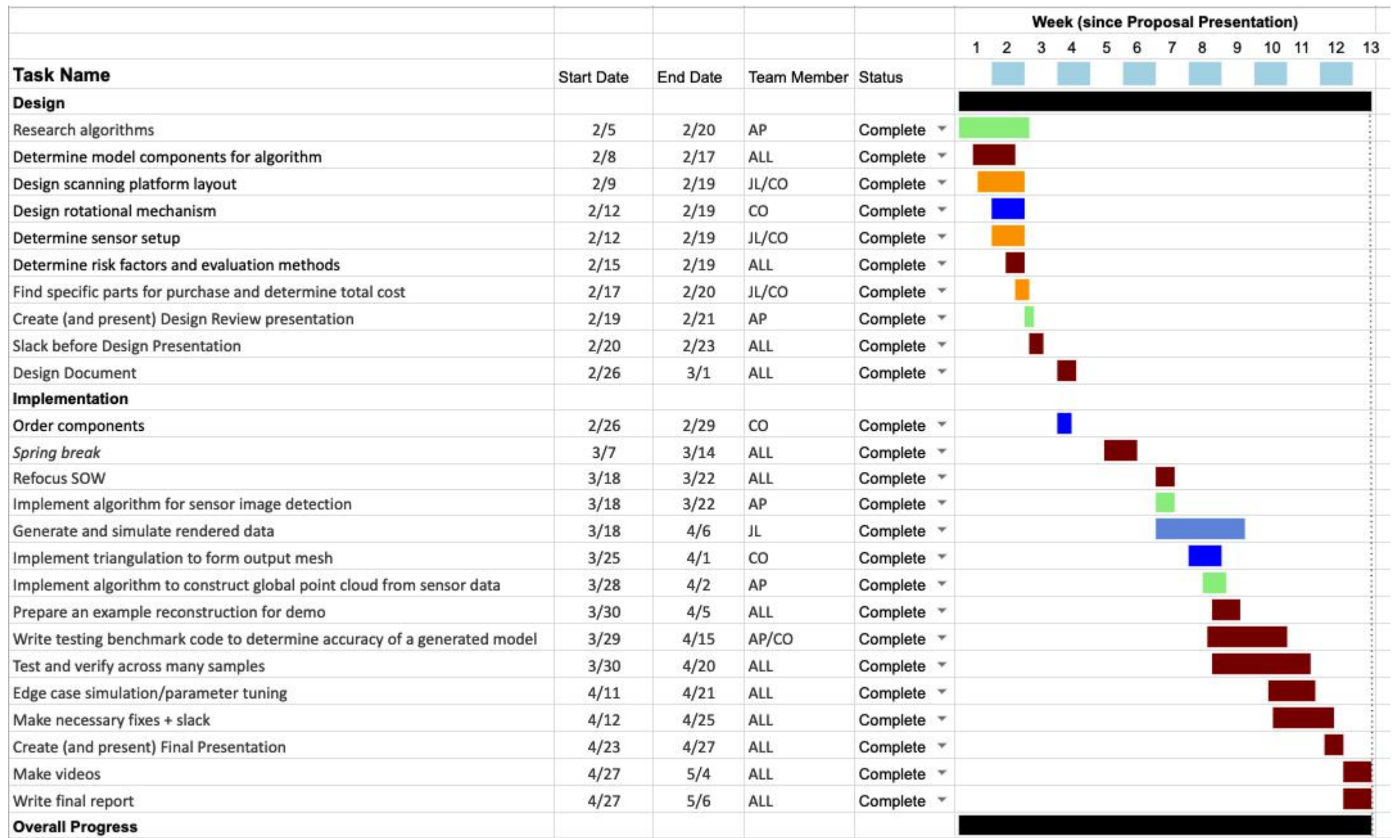


Figure 38: Final System Specification Diagram



Task Name	Start Date	End Date	Team Member	Status
Design				
Research algorithms	2/5	2/20	AP	Complete
Determine model components for algorithm	2/8	2/17	ALL	Complete
Design scanning platform layout	2/9	2/19	JL/CO	Complete
Design rotational mechanism	2/12	2/19	CO	Complete
Determine sensor setup	2/12	2/19	JL/CO	Complete
Determine risk factors and evaluation methods	2/15	2/19	ALL	Complete
Find specific parts for purchase and determine total cost	2/17	2/20	JL/CO	Complete
Create (and present) Design Review presentation	2/19	2/21	AP	Complete
Slack before Design Presentation	2/20	2/23	ALL	Complete
Design Document	2/26	3/1	ALL	Complete
Implementation				
Order components	2/26	2/29	CO	Complete
Spring break	3/7	3/14	ALL	Complete
Refocus SOW	3/18	3/22	ALL	Complete
Implement algorithm for sensor image detection	3/18	3/22	AP	Complete
Generate and simulate rendered data	3/18	4/6	JL	Complete
Implement triangulation to form output mesh	3/25	4/1	CO	Complete
Implement algorithm to construct global point cloud from sensor data	3/28	4/2	AP	Complete
Prepare an example reconstruction for demo	3/30	4/5	ALL	Complete
Write testing benchmark code to determine accuracy of a generated model	3/29	4/15	AP/CO	Complete
Test and verify across many samples	3/30	4/20	ALL	Complete
Edge case simulation/parameter tuning	4/11	4/21	ALL	Complete
Make necessary fixes + slack	4/12	4/25	ALL	Complete
Create (and present) Final Presentation	4/23	4/27	ALL	Complete
Make videos	4/27	5/4	ALL	Complete
Write final report	4/27	5/6	ALL	Complete
Overall Progress				

Figure 39: Gantt Chart

Material	Young's Modulus (GPa)	Specific Stiffness $(\frac{MN.m}{kg})$	Yield Strength (MPa)	Density $(\frac{kg}{m^3})$	Price per Unit Volume $(\frac{\$}{m^3})$
Plywood	3-4.5	3.98-6.06	8.1-9.9	700-800	385-488
Carbon Fiber	58-64	38.2-42.4	533-774	1490-1540	26200-31400
Balsa	0.23-0.28	0.817-1.09	0.8-1.5	240-300	1610-3230

Table 6: Material Tradeoffs Analysis

Shape	Dimensions (m)	Formula	Moments of Inertia (kgm^2)
Cube	0.3*0.3*0.3	$\frac{1}{12}m(w^2+d^2)$	0.105
Solid Sphere	r = 0.3	$\frac{2}{5}mr^2$	0.063
Hollow Sphere	r = 0.3	$\frac{2}{3}mr^2$	0.105
Solid Cone	r = 0.3	$\frac{3}{10}mr^2$	0.04725
Hollow Cone	r = 0.3	$\frac{1}{2}mr^2$	0.07875
Solid Cylinder	r = 0.3	$\frac{1}{2}mr^2$	0.07875
Hollow Cylinder	r = 0.3	mr^2	0.105
Rod	L = 0.3	$\frac{1}{2}mL^2$	0.0525

Table 7: Input Object Moments of Inertia