Depth-Camera Based Visual Inspection System

Team: Dec1704

Team Members:

Evan Woodring

Joseph Elliott

Cory Itzen

Nicholas Gerleman

Client: Danfoss - Radoslaw Kornicki

Advisor: Aleksandar Dogandzic

1. Project Statement and Purpose

Using a depth camera and a computer-aided design (CAD) model as a reference, we developed a proof-of-concept system for detecting errors in products on an assembly line, such as improperly placed parts and incorrect configurations. Our solution visually displays where error is occurring, if any exists, so that the error can be addressed before the product is shipped out.

This brings us to the primary purpose of the project; eliminating waste. An incorrectly configured product that is shipped out to a client will always be shipped back. This is wasteful for the sellers, consumers, and shipping companies. Reducing the number of times this occurs through the error detection system will be beneficial to all involved parties and will save money and time.

2. Project Requirements

For requirements, we focus on ease of use, speed, and accuracy.

2.1 Functional

- A CAD model shall be used as the reference
- The system should detect what area the error occurs in
- A 3D representation of the scan should be created and displayed to the user with the error region highlighted

2.2 Non-Functional

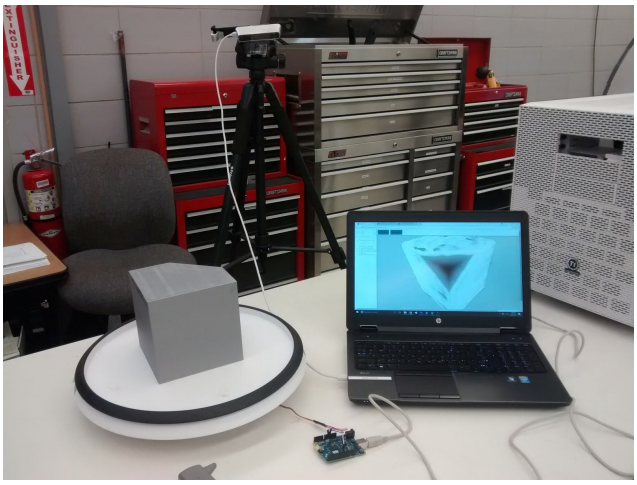
- The scanning process should take less than 30 seconds
- A user with limited technical knowledge should be able to operate the system
- The error should be reliably detected, with limited intervention

2.3 Other

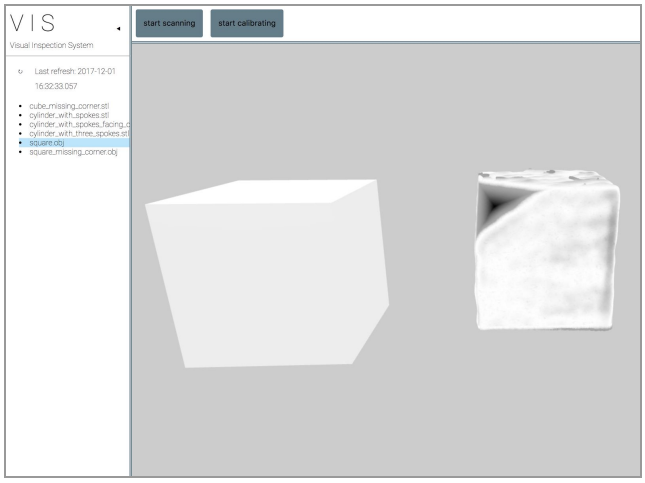
The system should be able to be integrated with an existing assembly line

3. Project Design

The system we developed allows a user without technical experience to easily visualize error between a physical object and a CAD model. The user must first calibrate the system to their environment. This is done by showing the user a 3D capture of the environment the camera sees, then asking them to move a box over the area to capture. After this initial calibration, the user is able to simply place the object on a platform, click the desired CAD model, then click a scan button. After a short period, the user is presented with an interactive view showing potential error.



The server processes data from the 3D camera and displays it on the client.



A comparison between the CAD model and a 3D scan of the object.

We divided the project into two distinct applications, a client and server. A headless server, responsible for scanning and error detection, is connected to a camera and servo. A separate web-based client is present to allow a user to control the scanning process and visualize errors. This approach brings several advantages. Because the server is headless, it is able to be put in places close to manufacturing equipment where it would be difficult or hazardous to place users. The client/server distinction reduces coupling between the two components, allowing each to be written in different languages.

3.1 Client Design

The client has responsibilities of displaying CAD models, their respective scanned point clouds, and calibrating the server's camera. Communication with the server is done using the server's Representational State Transfer (REST) endpoints. Model and point cloud viewing are done using ThreeJS, an open-source 3D rendering library.

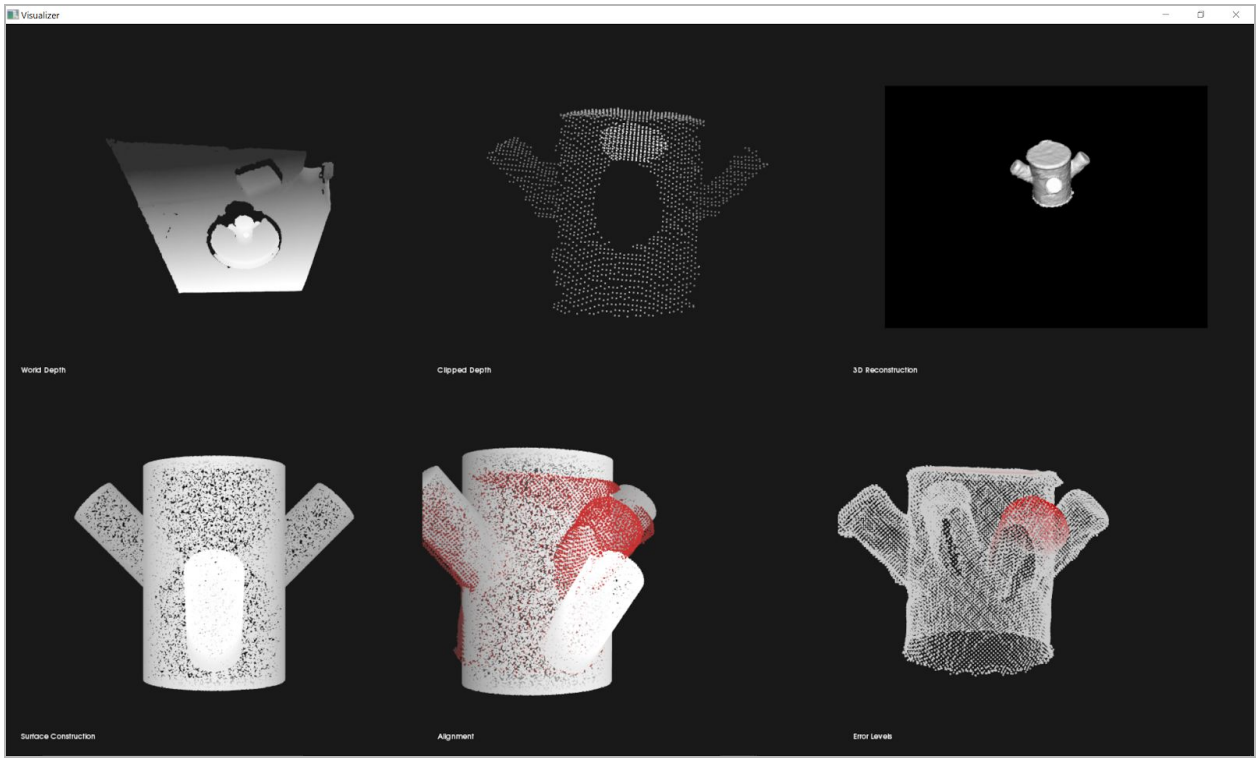
The web client itself is designed around Redux, a React framework. The idea is that there is one state for the entire client. Individual UI (user interface) components redraw with new information when the store updates. Because there can be multiple updates, and not every component cares about each update, components can listen on "substreams". These are basically updates focused on certain chunks of data.

The user is first greeted with the list of CAD models provided by the server. These are displayed on the left-hand side of the client. Clicking on one of these models fetches the respective CAD model and displays it in the ThreeJS scene. The user can request a scan based off their selected CAD model. When the request has finished, the point cloud is displayed next to the CAD model in the ThreeJS scene. If the user wishes to calibrate the camera, the current state of the ThreeJS scene is saved and replaced by the environment scan.

3.2 Server Design

The server has responsibilities of file-storage and retrieval, camera capture, and error detection. REST endpoints are exposed to allow the client access to these functionalities. The majority of the design work went into the error detection process. This process can be thought of as a pipeline of transformations, from the original mesh and camera capture, to the final output representing error locations on a reconstructed model of a physical object.

When the user begins scanning an object, the server sends a command to an attached Arduino to start a servo, rotating the platform the object is placed on. Frames from the depth camera are continually captured while the object is rotating. Each frame is transformed into world-space coordinates, then clipped to the volume the user determined during calibration. Each of these clipped viewpoints are fed into a 3D reconstruction algorithm to estimate the geometry of the physical object. Once this is complete, the CAD model is transformed into a point cloud representing the surface of the ideal object. The reconstructed physical object, and this ideal object are then aligned. Localized error values between the two point clouds are then determined.



4. Implementation Details

We will now focus on a detailed discussion of the implementation of our system. There are two distinct parts: the client and the server.

4.1 Client

We wanted our users to not be locked to the same computer that is running the camera. For this, we needed to separate the inspection system into two parts. Having a separate web client for interacting with the camera lets the user move freely, while still being able to view 3D models is a massive benefit. Furthermore, there are many open source libraries that help smooth the development process.

4.1.1 Language Choice

While many languages work for client-side development, we chose to use Dart. It provides access to a performant React framework, Redux, for managing client state. Furthermore, Dart can be compiled to Javascript. That, along with the JS Interop package, allows use to communicate with raw javascript code. This is important due to our choice of a 3d object rendering library, written in Javascript.

For testing the client, we chose to use Ruby. It provides us with Sinatra, a Ruby Gem used to spin up simple RESTful servers. This proved crucial to the development of the client. The C++ server, seen below, is fairly heavy in terms of libraries. Spinning that up on every machine that wants to test the client's RESTful API requests would have been costly.

4.1.2 ThreeJS

A crucial part of our system is displaying 3d models. ThreeJS is a lightweight Javascript library for rendering multiple 3d models on a web client. It is used for displaying the user's chosen CAD model, scanned models, and provide a tool for calibrating the camera. Because this is in Javascript and our client is written in Dart, we use the JS interop package provided by Dart to interact with the raw Javascript code.

Our ThreeJS code is fairly simple. It provides an API to the Dart code that lets the Dart code send temporary file locations (downloaded CAD models) and lists of points (point clouds sent from the camera) to the ThreeJS library for rendering.

4.1.3 Visualizing Error

When generating point clouds using ThreeJS, we provide a way to physically see where the error points are. These points are colored on a gradient, where points with a higher error value have a more contrasting color compared to the normal points. The error value for each point is based on the squared distance to its ideal nearest neighbor. The details for error visualization are discussed in Appendix 4.1.

4.2 Server

The server is responsible for controlling hardware and performing the heavy-lifting required to do error detection. Special consideration was done to ensure the server was capable of meeting our 30 second performance requirement.

4.2.1 Language Choice

It was decided early on to write the server using C++ for a variety of reasons. Most of the available libraries for point cloud processing and geometry manipulation were all written for C++. Custom code that had to be written had to deal with large sets of data, with strict performance requirements. C++ was additionally the most commonly used language to interface with the depth cameras tested.

4.2.2 Networking and IO

An http library, cpprestsdk, was used at a low-level to create a simple HTTP server. Abstractions such as request routing and error handling were built on top of this. Due to the potentially large size of point clouds transferred between client and server, a custom binary format was developed for interchange. It was chosen to use STL mesh files as a format for CAD models due to their simple binary structure which is fast to parse and small on disk. Due to the simplicity of the format, and lack of high quality libraries, custom code was built to parse these files. Mesh files and calibration data are persisted onto the local filesystem of the server. This is sufficient for the project requirements, where scalability is not a concern.

4.2.3 Camera Capture

Access to the Occipital Structure Sensor is provided through the open source OpenNI library. Frames are captured at VGA resolution, which is the maximum allowable by the camera. In order to convert depth pixels to world-space coordinates, an inverse perspective projection based on camera geometry is used. This functionality is provided by OpenNI.

4.2.4 Calibration Volume Clipping

The calibration volume is specified as a 4x4 affine transformation matrix on a 1x1x1 cube centered at the origin. This simplifies implementation on both the client and server. The client will already maintain this matrix internally when displaying the calibration volume. The server can use the inverse of this matrix to transform world-space points to be in a space relative to this 1x1x1 volume. This allows simple rejection of points outside of the volume, as well as reorientation to match the volume.

4.2.5 3D Reconstruction

3D reconstruction is achieved using the KinectFusion method [1]. Rather than implementing this ourselves, a Graphics Processing Unit (GPU) accelerated implementation provided by the Kinect Software Development Kit (SDK) is used. Even with GPU acceleration, the process is not guaranteed to run in real-time. To alleviate this, a seperate thread maintaining a work queue of depth frames is maintained. Performing reconstruction in this separate thread allows the main thread to do only small amounts of work during camera capture, ensuring no frames are missed.

4.2.6 Ideal Surface Construction

It is assumed the input CAD model represented as a mesh with no non-manifold geometry. Random points of a uniform density are sprayed onto the mesh's triangles to construct a point cloud representing the surface of the ideal object. Triangle area is calculated in 3D space using Heron's formula. An efficient algorithm to fill points in 3D space was created. While not the exact implementation, the algorithm can be thought of as containing two steps. For each point, two random variables are chosen between 0 and 1. Combinations whose sum is greater than 1 are rejected and a new combination is picked. When interpreted as x and y coordinates, this points fill a right triangle with a width and height of 1. These points are projected onto the 3D face by doing a basis transform, where the new basis for x and y are two adjacent edges picked from the face.

4.2.7 Mean Squared Error Estimation

Our alignment process depends on being able to quickly determine the mean-squared-error between two point clouds. Profiling determined this process to take longer than expected on large point clouds. This led to the development of an iterative algorithm to estimate the error without calculating it exactly. Points from our ideal surface are first put into a K-D tree. Batches of randomly chosen points are then chosen

from the capture cloud. The square distance to nearest neighbor is used to contribute to a running estimation of mean-squared error. If the delta between previous and current error after incorporating the new batch is less than a certain amount (eg 2%), the convergence criteria is considered to have been met. In empirical tests, this process reduced time to determine mean-squared-error by two orders of magnitude while maintaining high accuracy.

4.2.8 Point Cloud Alignment

Coarse alignment is first done by matching scale and translation of the ideal and captured objects. This is done by calculated a bounding sphere around each object, translating the captured object based on bounding sphere centers, and scaling the captured object based on bounding sphere radius. Rough orientation can be obtained by first transforming captured points to match the orientation of the clipping volume. While the object is then level, orientation may be incorrect in the Y axis. To guess rough orientation here, multiple orientations are tested and the one with the least mean-squared-error is chosen. At this point, the two point clouds roughly match in scale, translation, and orientation. At this point, it is sufficient to use an implementation of the Iterative Closest Point algorithm to achieve fine alignment.

4.2.9 Localized Error Detection

Localized error is reported to the client by determining square distance from each point in the capture point cloud to the nearest point in the ideal surface point cloud. This method is able to avoid false-positives from holes in reconstruction do to camera position. Actual holes in the object will still be detected as error points will be present as geometry inside the hole in the captured point cloud. These distances are determined using simple nearest-neighbor searches in a K-D tree.

4.2.10 Debug Visualization

In order to aid debugging of the error detection process, each process completed by the server may be visualized in real-time. This visualizer is built on top of the Point Cloud Library (PCL). For performance reasons, the visualizer is run on its own thread, with hooks present in server code to submit new data to it to be shown. Because point clouds may have different locations, scales, and orientations, a heuristic is used to dynamically adjust camera parameters to ensure the entirety each point cloud remains visible.

4.2.12 Servo Control

Creating a complete scan of an object involves rotating an object so all sides are viewable to a camera. To accomplish this, the object is placed on a rotating platform that is controlled by an Arduino, which is then wired to a continuously rotating servo under the platform. The server establishes a connection to the Arduino via Serial Port. When a scan is started, the server communicates with the Arduino to rotate the platform. Once the rotation is complete, the Arduino sends a message back to the server, indicating that it may conclude its camera scanning.

4.2.13 Mocking

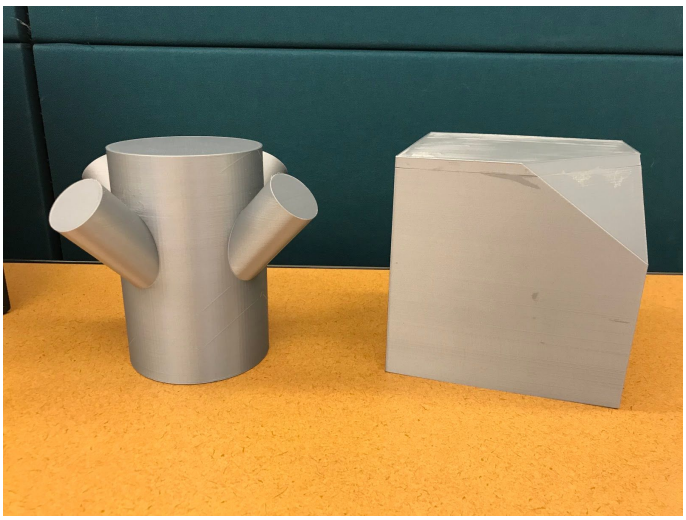
It is not feasible to always have a platform and camera connected while developing the server. Support is present to mock both of these using pre-recorded data. Multiple videos using the OpenNI .oni format were taken of an object moving on a platform. One of these files may be passed to the server as a command line argument, causing a mock camera to loop through the video and report servo information based on its length.

5. Testing Process and Testing Results

Given the research-centric nature of this project, a lot of work had done be done as far as testing goes. There were three major components that needed testing: the camera and 3D scanning, the alignment, and the error detection.

Our client initially provided us with the Intel RealSense camera, and we were tasked with evaluating if this camera would suit our needs. Our initial testing led us to believe that the Realsense would be sufficient for our project. We eventually did some additional testing with another camera, the Occipital Structure sensor. While the RealSense provided built-in support for 3D reconstruction, the Occipital structure sensor provided better quality data for individual frames. Both cameras were tested by initially scanning an assortment of different household objects.

After initially deciding on the RealSense, we continued testing the scanning process by 3D printing objects and scanning those. Our first model was a cube with a missing corner. The RealSense SDK was generally able to produce reconstructions of this, but occasionally missed detail or warped sections of the object. We initially attributed this to the inconsistent testing environment. We later printed a cylinder with four spokes, a much more complicated object. The RealSense camera was unable to pick up large sections of the object, and reconstruction consistently failed. At this point, we decided to reevaluate the Occipital. The better raw data from the camera enabled us to capture information about these objects, but its lack of built-in support for 3D reconstruction required us to use a separate implementation.



(Left) The original CAD model matches the 3D printed model.
(Right) The original CAD model is a cube, but the 3D printed model used for testing is missing a corner.

For testing alignment, we used some of the better scans we had generated while still working on the scanning process. As mentioned before, we were scanning 3D objects that we had printed ourselves. This meant that we had not only the source model to test alignment when the object should align fully, but could also easily modify these models and compare in the case where the 3D printing did not match the scan, and thus should not fully align.

Finally, for testing the error detection. Initially, we tested error detection using manually aligned data. We would have two point clouds aligned to one another, one with an error one without, and validated that we could correctly pick up this error; we also guaranteed that if there was not an error, this was properly conveyed.

As pieces started falling into place, we could then begin testing things end to end. Once we had working alignment and error detection, we were able to perform the alignment on our scans, and then error detect with real alignment data. Then, we were eventually perform a scan, immediately use that scan and perform alignment, and then perform error detection.

Performance testing was done during implementation of each step in our process. Profiling helped us determine individual areas that consumed significant portions of our performance budget and make changes accordingly. This performance testing was done on several computers to determine the effect of GPU and CPU used on overall performance.

6. Context: Related Products and Literature

Research in developing the Visual Inspection System has encompassed a wide variety of topics, including research into fields such as computer vision, computational geometry, 3D cameras, and other related areas. Some of this research is documented here.

6.1 Digital Face Inspection System

The Digital Face Inspection (DFI) System, developed in 2009, aimed to assist dentists in detecting deviation of a patient's face before and after an orthodontic treatment. We found similarities in the DFI System with our Visual Inspection System in both the problem being solved, as well as the solutions that were implemented.

Deviation analysis compares a scan of the patient's face before and after a scan. The Point Cloud Library's Iterative Closest Point (ICP) algorithm is utilized to align these two data sets. To increase the likelihood of ICP producing a convergence, DFI performs preprocessing and coarse registration on the raw data of the scans. Preprocessing removes noisy data, random artifacts, or other statistical anomalies in the raw data, as well as reducing the data set to keep the system performing efficiently. Coarse registration estimates surface normals, calculating features, identifying strong correspondences and rejecting weak correspondences, and then calculating an initial transformation matrix. The modified data set is then inputted into the ICP algorithm for alignment [2].

The Visual Inspection System and the Digital Face Inspection System both perform preprocessing before inputting raw data into the Point Cloud Library's Iterative Closest Point algorithm. The goal is the same: to increase the ICP's likelihood of finding a fine convergence between two data sets.

6.2 Kd-trees

The Visual Inspection System uses the Point Cloud Library to construct kd-trees, and then performs a k-nearest neighbor search operation using the Fast Library for Approximate Nearest Neighbors (FLANN).

A kd-tree is a spatial partitioning data structure storing points in a k-dimensional space, allowing for fast nearest neighbor searches. The kd-tree is a binary tree in which every

node represents a k-dimensional point in a k-dimensional space. Non-leaf nodes are then thought of as splitting the k-dimensional space into two half spaces. Each node's left subtree is contained entirely in the left half-space, and right subtree is contained in the right half-space. Moving down the kd-tree, each level partitions a different dimension. For example, the first level might partition the x dimension, the second partitions the y dimension, the third partitions the z dimension, and so on. This is repeated until all points all exhausted [3].

The nearest neighbor search algorithm allows finding a point in the kd-tree nearest to a given input point. Starting from the root node, the query moves down the tree recursively, saving the current best nearest neighbor. At each node, the algorithm updates the current best nearest neighbor if necessary, and then moves to either the left or right subtree depending on the position of the given query point relative to the position of the current node. Given that a kd-tree is a binary tree is a balanced binary tree, the time it takes to move from the root to a leaf node is proportional to the set of n elements. Performing a nearest neighbor query for a spatial point existing in a set of n elements takes $O(log\ n)$ time to perform [4].

Appendix I: Operation Manual

A1.1 Setup

The Visual Inspection System consists of a Server and Client, both of which need to be initially setup. The installation for each of these are detailed below

A1.1.1 Server

The project must be compiled and run on a Windows machine, and is compatible with a Occipital Structure Sensor. The project should be used with Visual Studio 2015 Update 3.

Install vcpkg. Follow the instructions on the vcpkg ReadMe.

Once installed, use vcpkg to install the CppRestSdk and PCL:

```
vcpkg install cpprestsdk cpprestsdk:x64-windows
vcpkg install pcl:x64-windows
```

Finally, install the Kinect SDK V2.

A1.1.2 Client

On a windows machine, install Chocolatey.
On a Mac, install Homebrew.

Install the Dart sdk

```
Windows: choco install dart-sdk
Mac OS: brew tap dart-lang/dart && brew install dart
```

In a terminal, navigate to the VIS_Client root directory and run:

```
pub get
```

Once the packages are installed, serve the client with:

```
pub serve
```

The client will be available at:

```
http://localhost:8080/
```

A1.2 Controls

A1.2.1 Navigating to the webpage

The client only works locally, so to navigate to the web page, you must go to

```
http://localhost:8080/
```

There are two query parameters for the client, normal and spc.

You can access the normal mode by either specifying not query parameter (like above) or like:

```
http://localhost:8080/useSpc=false
```

Either will configure the client to use the *json* format for retrieving 3d models.

You can access the spc mode by specifying the *useSpc* query parameter like:

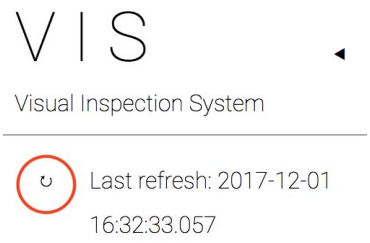
```
http://localhost:8080/useSpc=true
```

This will configure the client to use the *spc* format for retrieving 3d models.

The *spc* format is preferred for performance reasons. Regardless of the format you choose, the client's functionality will remain the same.

A1.2.2 Selecting a CAD model

To select a CAD model, you must first populate the list of available CAD models. This list is found in the left hand panel. The list is initially empty. To populate the list, click on the refresh button.

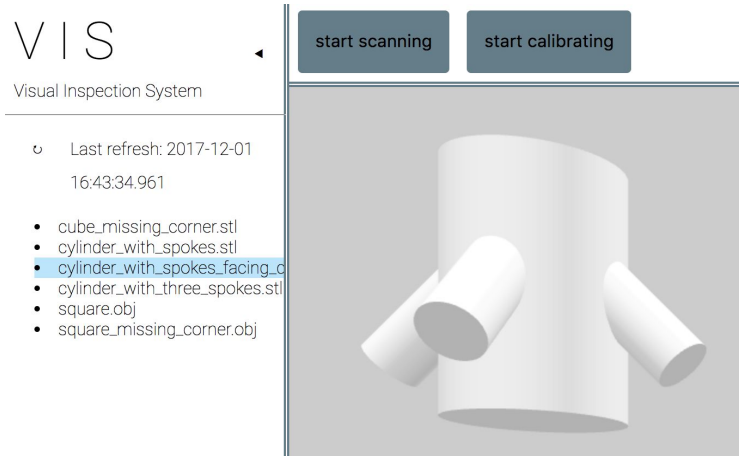


The top left of the client. The refresh button is circled in red.

Once the information has been retrieved, the list will show all available CAD models. They are named after their file names (including the file extension).

The top left of the client. The known CAD models are listed.

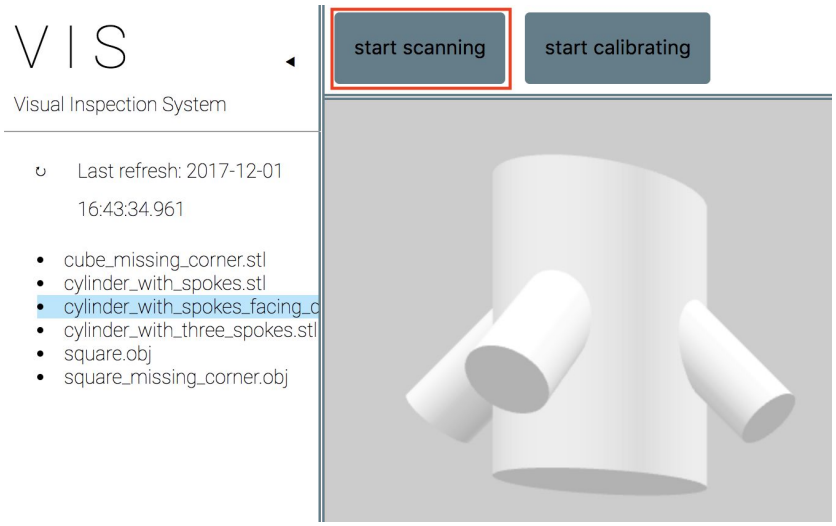
To request and render a 3d model, click on a file in the list. This will highlight the file to show that it has been chosen. Once the server has sent the file to the client, the file will be rendered on the left side of the viewing scene. Only one file may be rendered at a time. If a file is chosen while one is already being rendered, the current file will be removed.



The selected model is highlighted in blue. The respective CAD model is being rendered on the right.

A1.2.3 Requesting a scan

To request a scan, select a CAD model (as seen above). The currently selected 3d model is what will be used for processing error points in the object-to-be-scanned. Start the scan by clicking on the “start scanning” button in the toolbar (above the viewing scene). This button is on the far left.

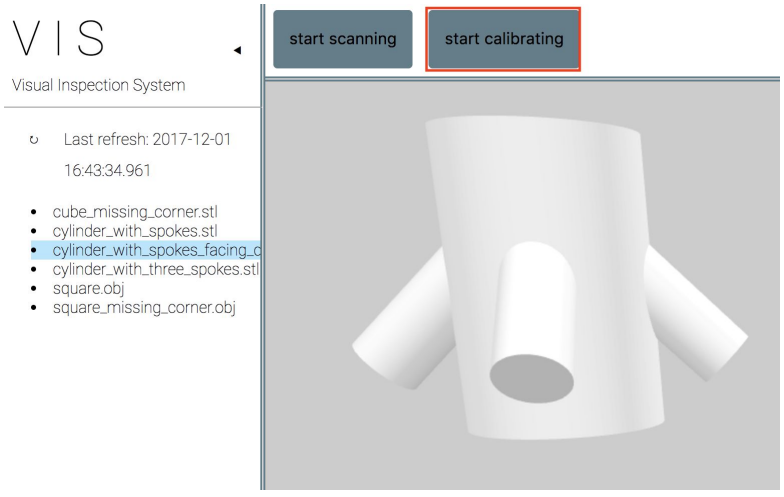


The top of the client. The start scan button is circled in red.

Once the scan has completed, the point cloud will be rendered on the right side of the viewing scene. Only one scan may be rendered at a time. If a new scan is performed while one is already being rendered, the current scan will be removed.

A1.2.4 Calibrating the scanning environment

To calibrate the scanning environment, click on the “start calibrating” button in the toolbar (above the viewing scene). This button is on the left, just after the “start scanning” button.

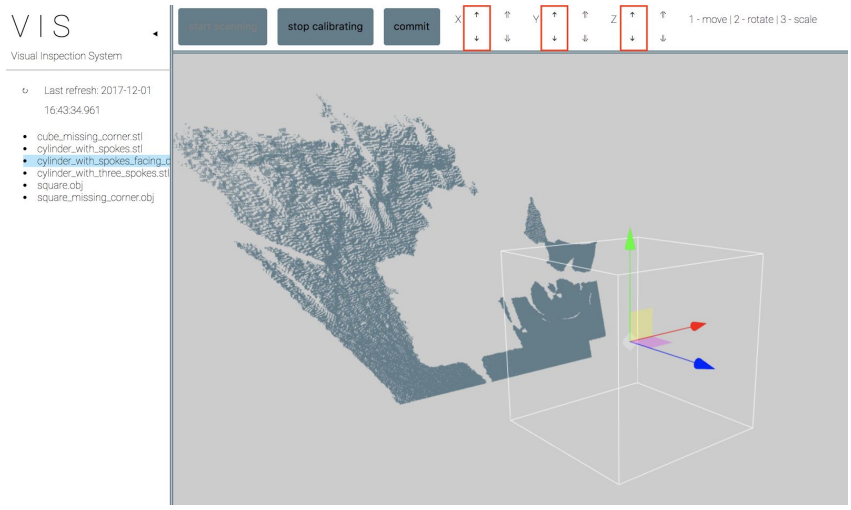


The top of the client. The calibration button is circled in red.

Once clicked, the scanning environment will be sent to the client. All currently rendered objects will be temporarily replaced with the scanned environment (a point cloud). A

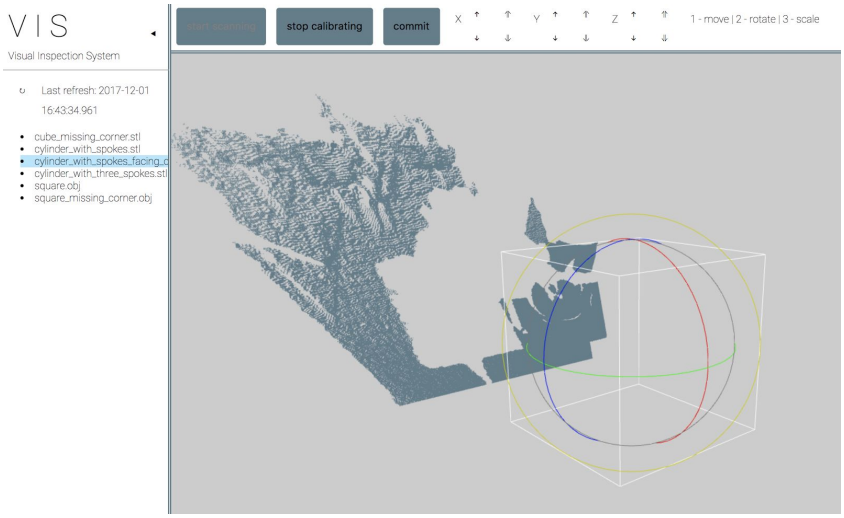
wireframe box will also be added to the viewing scene. The *inside* of this box is what the server will use for scanning. There are three ways to configure the box.

Move controls - Press 1 to enable move controls. This enables x, y, and z movement. You may drag the arrows to move the cube. For fine tuning, use the small arrows on the toolbar.



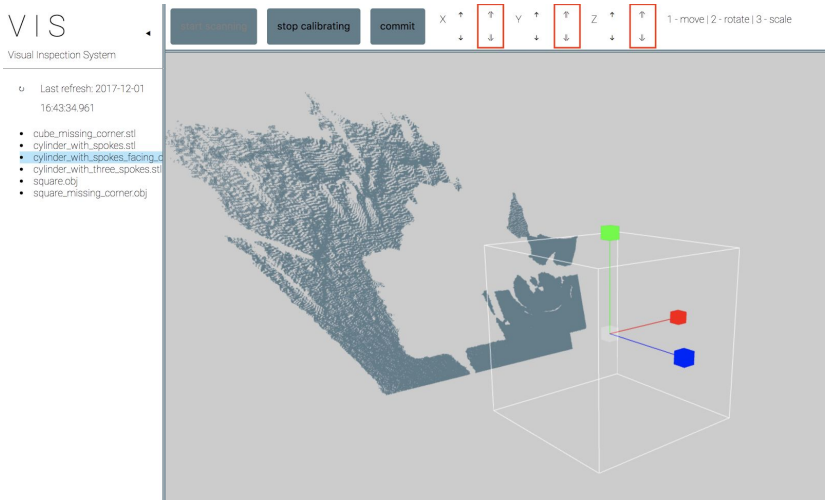
The calibration controls for movement. The fine tuning controls are circled in red.

Rotate controls - Press 2 to enable rotate controls. This enables complete rotation. You may drag the circles to rotate the cube.



The calibration controls for rotation.

Size controls - Press 3 to enable size controls. This enables x, y, and z sizing. You may drag the blocks to update the size of the cube. For fine tuning, use the small arrows on the toolbar.



The calibration controls for updating the size. The fine tuning controls are circled in red.

Appendix II: Alternative Versions

As mentioned before, we initially evaluated another camera, the Intel RealSense, and intended to use this camera throughout the project. This was because the RealSense provided a lot of built in tools that we needed, such as 3D reconstruction. After more extensive testing, we realized the limitations the RealSense had when it came to scan quality, and the complexity of objects that could be accurately scanned.

Appendix III: Other Considerations

Early on in the project, we received a lot of context from our client on the importance of our project. We were given a tour of the facility, and were told what the current error detection process was. At the present moment, there is an employee at the end of the assembly line with a couple of photographs of the correct version of the product, and they are asked to validate that the assembled version is correct. This is very error prone, as it is prone to both human error, as well as the photos not showing all possible error points.

Throughout the work on this project, many of us stepped outside of our comfort zones. Our team consisted of mostly Software Engineers, so hardware work was outside of the the comfort zone for many of us. Most of us had little to no experience in 3D modeling, 3D reconstruction and how 3D cameras work, and even many of the programming languages that we ended up using. There was a lot of research done, and in general we were exposed to a lot of things we did not have a lot of knowledge on, which allowed us to learn a lot during our work on the project.

Appendix IV: Additional Details

A4.1 Gradient Coloring

Visualizing error in an object is accomplished by coloring each point on a gradient, where points with a higher error value have a more contrasting color compared to the points with with a lower error value. The error value for each point is based on the squared distance to its ideal nearest neighbor.

A threshold for all points is computed by finding statistical outliers in the squared distances of the set of points. Points that have a squared distance that exceed this threshold are categorized as error points. The threshold for outlier points is calculated using the Interquartile Range, which is defined using the following equation

$$IQR = Q1 + Q3 * 3$$

Q1 and Q3 are the first and third quartile of the squared distance of all points. Points that exceed this IQR threshold are colored on a gradient; points that have a larger squared distance are given a darker color than those with smaller squared distance. Users will easily be able to identify errors in a scanned object by finding clusters or regions of darkly colored points.

A4.2 Structured Light

The Occipital Structure Sensor uses structured light in order to generate 3D scans. This is a process by which a known pattern of infrared light is sent out from the camera. The camera's infrared sensor records the distortion in this pattern to get a picture of how the object looks. As this process is repeated during the rotation of the object, many separate images are captured, and then stitched together in order to create one 3D representation of the object.

Citation

[1] Izadi, S., Davison, A., Fitzgibbon, A., Kim, D., Hilliges, O., Molyneaux, D., . . . Freeman, D. (2011). KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera. Proceedings of the 24th annual ACM symposium on User interface software and technology - UIST 11, 559-568. doi:10.1145/2047196.2047270

[2] Hsieh, Cheng-Tiao. “An efficient development of 3D surface registration by Point Cloud Library (PCL).” IEEE Conference Publication.

[3] Moore, Andrew D. “An Introductory Tutorial on Kd-Trees.” CiteSeerX.

[4] Friedman, Jerome H, et al. “An Algorithm for Finding Best Matches in Logarithmic Expected Time.” ACM Transactions on Mathematical Software (TOMS), ACM.