Project Title:

BRIEF Visualization

Authors:

A. Tarek Hatata and Thomas Magnan

Purpose & Objective:

This proposed project is an extension of a pre-existing "human-in-the-loop" application utilizing two robots and a turntable to image and grasp objects within a predetermined region.

This project focuses on advancement of the applications of the optical robotic arm. The end goal is to add functionality of the controlling GUI to easily move the arm and obtain images from other perspectives to build a refined 3D model of the object being imaged. Ideally this process can be automated through machine learning and analyzing an image stream to determine where another image is required.

The next stage of the project includes taking a series of images of this object over time and connecting the 3D models to create a 4D model where one can see the changes over time (the 4th dimension). An ideal object to model in three dimensional space over time is a plant, specifically a flower.

Users & Client Interfaces:

Our product is a semi-automatic process where the user can choose how much they interact with the modeling process. Clients will interact with a centralized GUI allowing them to (1) calibrate the camera and retrieve saved calibrations (2) request additional poses (3) build a model atop those poses from a preselected list of frames (4) manipulate the model (move, rotate, resize, etc) and (5) output the model.
The user also will eventually have the option to have most of the modeling process done for them for final refinement and will output a 3D model.

Use Cases:

The first significant impact that this project relates to the 3D modeling technology alone. Through this project one will be able to obtain an accurate 3D model of an object and model over time. One potential application of this technology includes monitoring plant growth in a lab setting. This technology will minimize the monitoring and meticulous notes required by lab personnel when observing growth or changes on a daily, or even hourly, basis.

The second significant impact that this project will provide incorporates the grasping arm technology in a few ways. The 3D models created by this project can be further analyzed to grasp the object and even move it (e.g. helping a disabled person drink a glass of water). Additionally the algorithms created to control the movements of the camera arm can be modified to enhance the functionality of the grasping arm.

Architecture Diagrams:

Figure 1: BRIEF Visualization

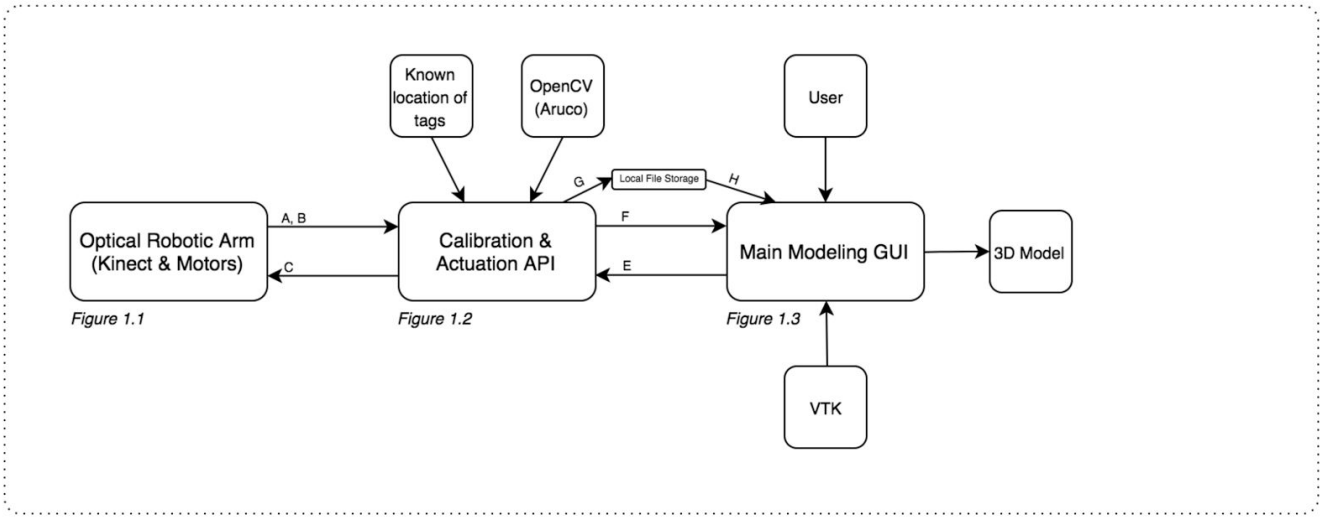

Figure 1.1: Optical Arm

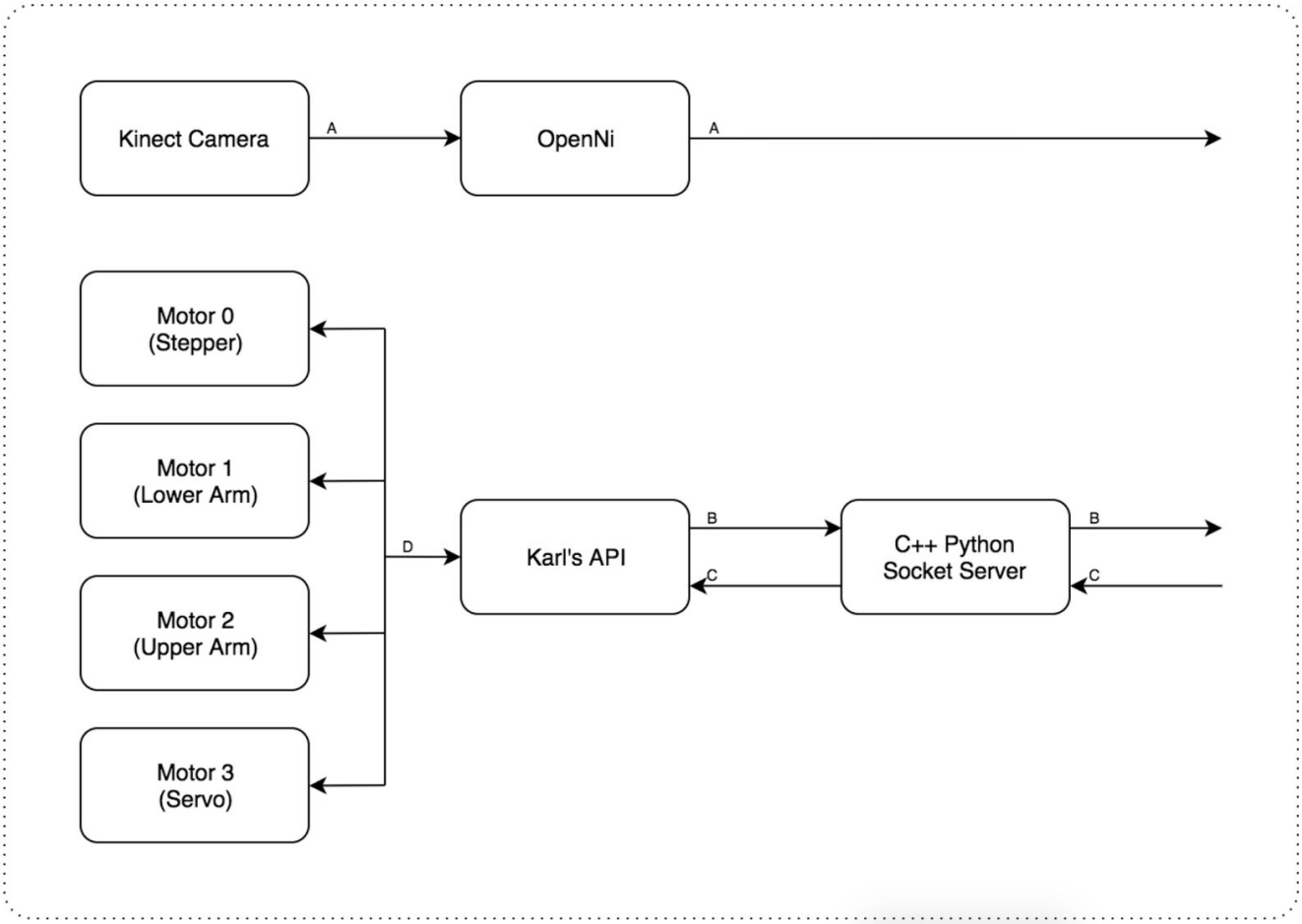

Figure 1.2: Calibration & Actuation API

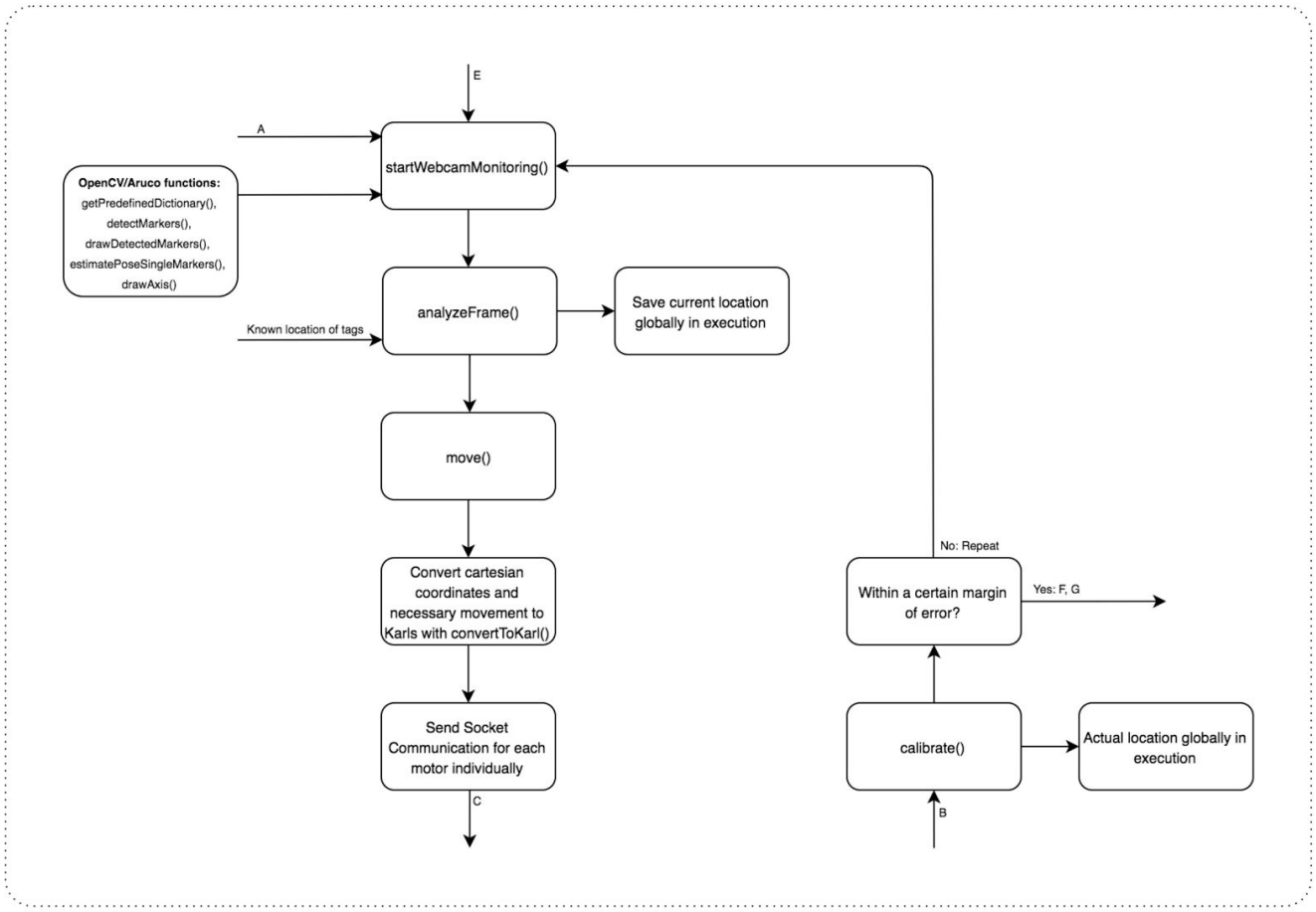

Figure 1.3.1: Calibration Process

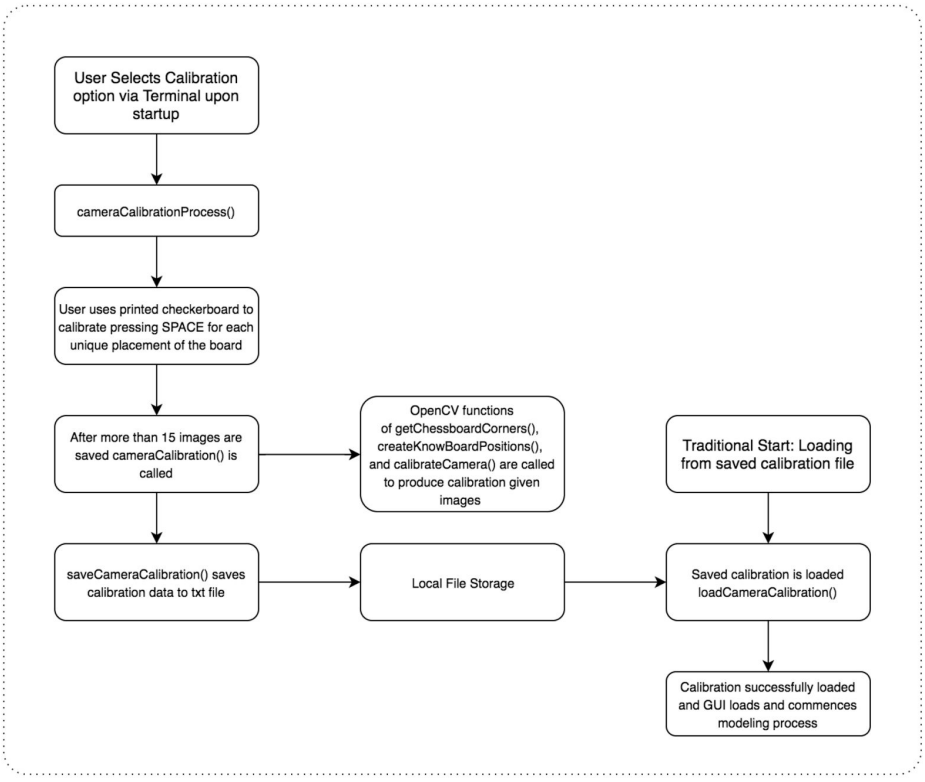

Figure 1.4: Modeling Process

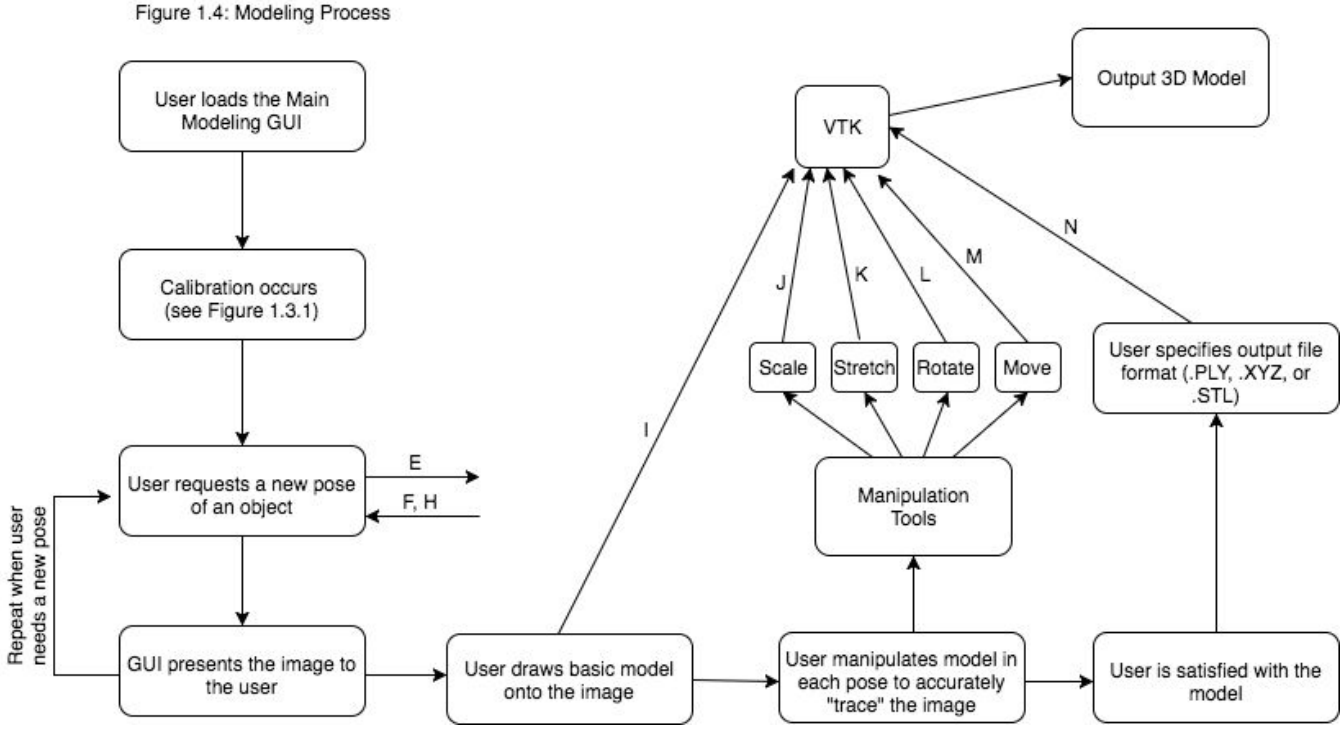

Diagram Key:

A. RGB and Depth Data data from the Kinect is interpreted through OpenNi, allowing the data to be treated much like any other camera in OpenCV.

B. Response from motors from Karl's API verifying the successful actuation of motors
   1 = success | !1 = failure

C. Call to Karl's API to move the motors. Description for how this process can be found accordingly.

D. Motor actuation occurs

E. struct pose requestNewPose(point3f desired)
   This serves as the main C+A API call. Given the current position of the robotic arm the API moves the arm to the *desired* position

F. API returns a new struct pose containing a point3f actualPosition (where the robot ended up) and String imageName (name of the image of new pose taken)

G. Saves image file out to local file storage

H. GUI retrieves image from file storage given name in E

I. **void DrawCube(double p[3])**

J. void ScaleCube(double scaleX, double scaleY, double scaleZ)

K. void StretchCube(int axis, double length)

L. void RotateCube(int axis, double angle)

M. void MoveCube()

N. void OutputFile(int option)

- Optical Robotic Arm

    - Current working arm that we are working with to perform our research. Consists of 4 motors (motor 0-3) to provide movement around the turntable for modeling.
- OpenNI
    - Open-source software that allows us to interact with the Kinect camera with supporting drivers.
- Karl's API
    - It is a python based API that controls all direct interaction with the motors and thus making our software modular with additional setups
    - int moveMotor(int MotorNum, int degree)
    - *MotorNum* is depicted in Figure 1.1
    - *Degree* is a unique number >0 which represents the fraction of movement out of the maximum movement of that particular motor. Those exact values are given to us in the API but actual measured angles will have to be converted to this unique measurement to provide movement.
- Known location of tags
    - Aruco virtual reality markers will be placed on all four axises of the turntable. We save the known transformation matrixes for each marker with respect to the origin (center) so we can find location of Kinect.
- OpenCV & Aruco
    - Open-source libraries that assist with camera calibration and getting the pose of the camera with respect to the printed tags for analysis for calibration.
- Local File Storage
    - Standard file system storage. No need for a database at this time. Used to save aruco tags for printing, calibration files, and requested images for modeling.
- 3D Model
    - User specifies an output file format for the 3D model, either as a .PLY file (polygon file format), a .XYZ file, or a .STL file (standard triangle language).
- VTK
    - Virtual ToolKit, used to build GUI, create and model objects, and output those 3D models.

Core Algorithms Described & Breakdown:

- **Main Modeling GUI**
    - **The first iteration of the main modeling GUI allows the user to interact with different perspectives of an object to create an accurate 3D model. Starting with two camera poses of an object, the user will be able to build a model on top of these images and request new perspectives. If the object to model is a common object, such as a cube, coffee cup, or leaf, the user can also apply a mesh of the object to the image, and adjust the mesh if needed.**
    - **The second iteration of the main modeling GUI aims to eliminate the need for user interaction, and hence any GUI. The modeling process will be automated, so that the user just needs to identify the object to model, and our system will apply that mesh to the given camera poses, determine any additional perspectives needed, generate the 3D model, and output it.**

    - **Breakdown: Tarek is currently working on a beginning GUI with his work for the Modeling API. Tom will work with Tarek to expand functionality once the Calibration and Actuation API is complete.**
- **Calibration & Actuation API**
    - **This API focuses on the hardware interaction (Kinect and robotic motors). We started by creating a 3D world coordinates (center of the world is the base of the Shunk arm, x-axis is the starting position of the stepper motor, z-axis is directly up, and the y-axis is perpendicular to both. This API must answer three core functionalities: (1) Where the camera is with respect to the object being modeled, (2) manipulating the four arm motors to move the optical arm to a desired position, and (3) calibrating the camera to account for motor mechanical failures.**
    - **Breakdown: Tom**

Main Modeling GUI: (Currently Tarek, eventually combined)

- **Functional Requirements:**
    - **Request camera poses from the optical arm, and display the images to the user**
    - **Model an object in 3D space using poses provided by the optical arm**
    - **Allow user to manipulate the object on top of the image to more accurately model the object**
    - **Allow user to either (1) draw onto the image and convert to 3D object, or (2) apply a pre-existing mesh of a common object to output an accurate 3D model**
    - **Output a data representation of the model for further processing**
- **Non-Functional Requirements:**
    - **Quickly receive and display camera poses from the optical arm**
    - **Use a limited number of camera poses to fully model the object**
    - **Data representation of the 3D model must be portable for other applications**
- **Components & Technologies:**
    - **C++**
        - **Language used to build the graphical user interface, process the data received from the optical arm, and output an accurate data representation of the 3D model**
    - **VTK (Visualization Toolkit)**
        - **Used to (1) build a graphical user interface that displays different poses of the object, (2) draw simple 3D objects onto the image, and (3) output the model created in the desired file format**
        - **vtkPNGReader**
            - **Source object used to read PNG files. This object takes the filename of the PNG file and holds it for processing and eventually displaying the image to the user.**
        - **vtkSimplePointsWriter**
            - **Source object used to write .XYZ files. This object outputs the modeled object to a .XYZ file based on the filename specified by the user.**
        - **vtkPLYWriter**
            - **Source object used to write .PLY files. This object outputs the modeled object to a .PLY file based on the filename specified by the user.**

- **vtkSTLWriter**
  - **Source object used to write .STL files. This object outputs the modeled object to a .STL file based on the filename specified by the user.**
- **vtkCubeSource**
  - **Object used to represent a cube in the rendering window. When a user clicks to draw a cube into the scene, a vtkCubeSource object is created to represent a generic cube in 3D, using the coordinates of the mouse "click" event.**
- **vtkPolyData**
  - **Object used to represent the geometric structure of an object consisting of vertices, lines, polygons, and/or triangle strips. This object is held by the vtkCubeSource and contains all the information about the geometry of that cube.**
- **vtkPolyDataMapper**
  - **Object used to map vtkPolyData to graphics primitives. This object is responsible for mapping the geometric data held by the vtkPolyData to proper coordinates so it can be rendered correctly, and serves as a middleman between the actual object representation and the vtkActor (i.e. vtkActor holds the reference to the vtkPolyDataMapper, which holds the reference to the vtkCubeSource).**
- **vtkActor**
  - **Object used to represent an object, its geometry, and other properties in a rendering scene. The vtkActor holds any entities that are present in the scene, such as a drawn cube or even the image to display.**
- **vtkRenderer**
  - **Object that controls the rendering process of the scene, which includes converting the geometry of every entity in the scene and specifying the lighting and camera view. References to any entity in the scene is held by a vtkActor, and all the actors in a scene are rendered and held by a vtkRenderer.**
- **vtkRenderWindow**
  - **Object that specifies the behavior of a rendering window. Each vtkRenderer object can specify a "viewport" that represents a section of the rendering window. The vtkRenderWindow object holds references to any renderer in the scene, and it's responsible for calling the Render() function for each vtkRenderer object.**
- **vtkRenderInteractor**
  - **Object that provides an interaction mechanism for mouse, key, and time events. The vtkRenderInteractor object is responsible for defining the interaction of the window, based on a vtkInteractorStyle.**
- **vtkInteractorStyle**
  - **Object that provides an event-driven interface to the rendering window. The vtkInteractorStyle allows the programmer to define their own implementation for any mouse, key, or time event that occurs in the**

window, and in our work is used to call functions that draw and manipulate objects in the scene.

- **vtkTransform**
  - **Object that describes the linear transformations of an entity in a 4x4 matrix, or a homogenous transformation matrix. The vtkTransform object makes it simple to transform objects without deriving the 4x4 matrices themselves, as it provides functions to rotate, translate, and scale an object in the scene.**
- **vtkTransformPolyDataFilter**
  - **Object that represents a filter to transform point coordinates, associated point and cell normals, and vectors. This object takes in an input connection, which is the entity to transform (e.g. vtkCubeSource) and the vtkTransform object that defines the transformation to perform. In this context, when an object is transformed, the vtkPolyDataMapper now holds the vtkTransformPolyDataFilter instead of the entity itself.**

- **Current Functions Explained:**
  - **void DrawCube(double p[3])**
    - **This function is called by the vtkInteractorStyle object when the user clicks onto the image. The user will enable an option to draw cubes, and a vtkCubeSource object will be created and centered at the coordinates of the mouse "click" event. The vtkCubeSource will be mapped to a vtkPolyDataMapper, which will be held by a vtkActor object that is added to the scene renderer, or vtkRenderer.**
  - **void StretchCube(int axis, double length)**
    - **This function is called by the vtkInteractorStyle object when the user selects the "Stretch" tool and attempts to stretch or shrink the cube. The user must specify an axis to manipulate the object (i.e. x = 1, y = 2, z = 3) and the stretch length in pixels. Say the user specifies the X axis to stretch, the vtkCubeSource will call SetXLength() with the stretch length provided, and the cube will be updated on the scene to reflect this change.**
  - **void RotateCube(int axis, double angle)**
    - **This function is called by the vtkInteractorStyle object when the user selects the "Rotate" tool and attempts to rotate. The user must specify an axis to manipulate the object and the angle to rotate the object (in degrees). Say the user specifies the X axis to rotate, a vtkTransform object is created and calls the RotateX() with the angle as a parameter. A vtkTransformPolyDataFilter object is created to hold the vtkTransform and vtkCubeSource objects, and is mapped to a new vtkPolyDataMapper object. The original vtkActor object will set the mapper to this new vtkPolyDataMapper to update the scene.**
  - **void ScaleCube(double scaleX, double scaleY, double scaleZ)**
    - **This function is called by the vtkInteractorStyle object when the user selects the "Scale" tool and attempts to scale the object. The user must specify the scale factor for each axis. The vtkTransform object is created and calls the Scale() function with scaleX, scaleY, and scaleZ as parameters. See RotateCube() for how the object is updated in the scene.**
  - **void MoveCube()**

        - **This function is called by the vtkInteractorStyle object when the user selects the "Move" tool and attempts to move the object. When the "Move" option is selected, the mouse "down" event will check if the vtkCubeSource object is under the mouse cursor. If so, the vtkCubeSource object will follow the cursor as long as the mouse is down, and when the mouse "up" event is triggered, the object's location is updated to reflect its new position and the cube will stop following the mouse.**
    - **void OutputFile(int option)**
        - **This function is called by the vtkInteractorStyle object when the user selects the "Output File" button. The user will specify a file type (i.e. .PLY = 1, .STL = 2, .XYZ = 3), and will create either a vtkSimplePointsWriter, vtkPLYWriter, or vtkSTLWriter object depending on the user's selection. This object is responsible for outputting the modeled object to the desired file format.**
- **Shared Data Storage:**
    - **Local File System Storage**
        - **Since the system is local to the machine running the two robotic arms and the Kinect, only the names of the image files will need to be shared and accessed between parts of the product. There is no need for a more formal data structure.**
        - **See Figure 1**

Calibration & Actuation API (Tom):

- **Functional Requirements:**
    - **Where the camera is with respect to the object being modeled**
    - **Manipulating the four arm motors to move the optical arm to a desired position**
    - **Calibrating the camera to account for motor mechanical failures**
- **Non-Functional Requirements:**
    - **Security is not a concern of this project. No form of authentication is required to utilize this API**
    - **Must be fast enough to updated poses in 'real-time' so the functionality can be extended to model changing objects (i.e. growing plants).**
    - **Algorithm is modular enough to extend to different robotic setups.**
- **Components & Technologies:**
    - **C++**
        - **Primary language used for building of the API, it's external interfaces, and interactions with core libraries.**
    - **OpenCV & Aruco:**
        - **getCameraCalibration()**
            - **Uses checkerboard from various angle to get camera matrix and distance coefficients associated with the camera itself**
            - **Current calibration for Kinect has been measured**
        - **getEstimatedPose()**
            - **Returns matrices with translation vectors (location of marker with respect to the Kinect) and rotation vectors (pose) for each marker found. This data is then combined with our known location of the markers to find the best average of where the Kinect is wrt the world origin.**

- **This function is called before and after each movement to get current location and make sure that the Kinect ended up where it needed to go. If not, the process repeats until it's within a margin of error.**

- **Aruco Markers**
  - **Printed out virtual reality tags interpreted by the Aruco module**
  - **We are going to start by placing four tags onto the turntable, one along each of the positive and negative x and y axes.**
- **Karl's API & Python Interface**
  - **We are currently building off of the work of a graduate student named Karl Preisner. He currently has an API with its own documentation that handles the movement of the motors. Rather than operate the motors directly, we will interact with the motors through this API.**
  - **API takes in a motor number and degree of movement.**
- **Optical Arm**
  - **Microsoft Kinect**
    - **Main Camera - RGB image stream as well as point cloud data which provides depth associated with each pixel relative to the camera reference frame.**
  - **Stepper Motor**
    - **Moves arm around the turntable**
    - **Most inaccurate motor movements that will be needed to be calibrated. Easily can be moved by being knocked by outside force.**
  - **2 hydraulic motors**
    - **Very accurate, but slow. Stays in place even when removed from power.**
    - **Used to move camera up and out**
  - **Servo Motor**
    - **Very accurate, but slow. Stays in place even when removed from power.**
    - **Rotates the Kinect on the end of the arm to change pose.**

- **Interfaces & Communication:**
  - struct pose requestNewPose(point3f desired)
    - This serves as the main C+A API call. Given the current position of the robotic arm the API moves the arm to the *desired* position
    - **Since this functionality is written in C++, it can be called from the main.**
  - **Python-C++ Socket connection:**
    - **This interface is required to move the various motors since Karl's API described above is written in Python and the rest of the application is written in C++.**
    - **Only two integers will needed to be sent via the socket communication (motor number, and degree of movement in Karl's system of measurement**
- **Current Functions Explained:**
  - **Point3f getCurrentPosition()**
    - **This function calls the startWebcamMonitoring() function that invokes getEstPose(). Given the returned transformation matrixes, this function does the matrix multiplication and averaging to return the best position for the Kinect.**
  - **Point3f move(Point3f current, Point3f desired)**

        - **Invokes Karl's API to move the motors from the *current* point to the *desired* point**
    - **calibrate(Point3f desired)**
        - **Invokes getCurrentPosition() to find out where the Kinect ended up and measures to determine the margin of error of returned point with *desired* point passed into the function. If within, simply return, else continue. If not within the error call move() function with returned point from first function call to desired point.**
    - **void createArucoMarkers()**
        - **Invokes Aruco function to create markers (JPG's) for print**
    - **void getChessboardCorners(vector<Mat> images, vector< vector<Point2f> >& allFoundCorners, bool showResults = false)**
        - **Invoked in calibration process to get corners of chessboard for further analysis.**
        - **Passes in found *images* and saves *allFoundCorners* for further analysis**
    - **void createKnowBoardPositions(Size boardSize, float squareEdgeLength, vector<Point3f> corners)**
        - **An additional supporting function in calibrating the camera determining where we know the board corners are given known *boardSize* and *squareEdgeLength* and *corners***
    - **void cameraCalibration(vector<Mat> calibrationImages, Size boardSize, float squareEdgeLength, Mat& cameraMatrix, Mat& distanceCoefficients)**
        - **Overall camera calibration process. Invokes OpenCV function and supporting functions to get necessary inputs listed above. Fills *cameraMatrix* and *distanceCoefficients* with necessary values**
    - **bool saveCameraCalibration(string name, Mat cameraMatrix, Mat distanceCoefficients)**
        - **Outputs calibration numbers in *cameraMatrix* and *distanceCoefficients* to txt file of a given *name* so it can be imported upon restart of system.**
    - **bool loadCameraCalibration(string name, Mat& cameraMatrix, Mat& distanceCoefficients)**
        - **Imports calibration numbers previously calculated into matrices (*cameraMatrix* and *distanceCoefficients)* from txt file of given *name***
    - **int analyzeFrame(vector< vector<Point2f> > markerCorners, const Mat& cameraMatrix, const Mat& distanceCoefficients, vector<Vec3d> rotationVectors, vector<Vec3d> translationVectors, vector<int> markerIds)**
        - **Testing file that is used to print out known location of markers to users to compare against physical measurements for margin of error.**
    - **int startWebcamMonitoring(const Mat& cameraMatrix, const Mat& distanceCoefficients, bool showMarkers)**
        - **Obtains image frame from Kinect and calls Aruco function getEstPose() to fill transformation and rotation vectors.**
    - **void cameraCalibrationProcess(Mat& cameraMatrix, Mat& distanceCoefficients)**
        - **Wrapping function that calls cameraCalibration() after obtaining a minimum of 15 images from the user.**

Development Timeline:

- **August/September:**
  - **Brainstorming, defining team roles between who works on calibration & actuation or modeling GUI, set initial tasks for the team, and brush up on basic kinematics**
- **October:**
  - **Learn and explore the libraries we'll be using (e.g. VTK, OpenCV)**
  - **Modeling GUI: present a PNG image to the user, draw points and lines onto the image**
  - **API: Camera Calibration functionality (Figure 1.3.1) development and unit testing with standard webcam. Began developing calibrate() process of API.**
- **November:**
  - **Modeling GUI: draw a cube onto the image, allow user to cycle through multiple PNG images to simulate multiple poses, adapt to a real scan of an object (specifically a cube)**
  - **API: Unit testing calibrate() with Kinect**
- **December/January:**
  - **Modeling GUI: finish basic user interface and every manipulation tool, and define the mathematical process for mapping multiple poses of a cube into one model**
  - **API: Complete Unit testing on calibrate(); Development of and Unit testing for socket communication; Begin integration testing with Karl's API; Begin Development in reverse kinematics for actuation**
- **February:**
  - **Modeling GUI: create a cube preset that allows a user to drag and drop a generic cube onto the image and automatically "snap" the cube into the correct position, begin integration with Calibration & Actuation API to request and receive poses of the object**
  - **API: Integration testing with GUI; Finish development and Begin Unit testing for actuation**
- **March:**
  - **Modeling GUI: continue integration testing, finish cube preset (if haven't already), and implement a more complex object (e.g. sphere, cylinder, leaf)**
- **April:**
  - **Final integration testing**
  - **Clean up GitHub**
  - **Finalize presentation tests**
  - **Present final demo**