Assignment Assumptions:

We worked collaboratively on the beginning sections and under each section labelled by our names we worked independently to expand on the interfaces and functionality of each section that we are responsible for. This was vague on the description but have decided to keep it in one document so that it is easier to review.

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.

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 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
    - VTK (Visualization Toolkit)
        - Used to build a graphical user interface that displays different perspectives of the object, and defines interactions to physically draw onto these images
        - vtkPNGReader
            - Source object used to read PNG files, the image format passed from the optical arm
        - vtkActor
            - Object used to represent an object, its geometry, and other properties in a rendering scene
        - vtkRenderer
            - Object that controls the rendering process for objects, which includes converting the geometry of the object, and specifying the lighting and camera view of the scene
        - vtkRenderWindow
            - Object that specifies the behavior of a rendering window, used for the renderers to draw into
        - vtkRenderInteractor
            - Provides an interaction mechanism for mouse, key, and time events, used to define the interaction of the window
        - vtkInteractorStyle

            - Provides an event-driven interface to the rendering window, used to process the user interaction and update the window as needed
        - vtkPolyData
            - Data object used to represent the geometric structure of an object, consisting of vertices, lines, and polygons, used to draw points onto the image
        - vtkLineSource
            - Object used to represent a line segment in the rendering window, used to draw lines onto the image
        - vtkCubeSource
            - Object used to represent a cube in the rendering window, used to draw and size cubes onto the image
        - vtkTransform
            - Object that describes the linear transformations of an entity in a 4x4 matrix, or a homogenous transformation matrix

- Current Functions Explained:
    - void DrawPointOntoImage(double p[3])
        - This function is called by the interactor style object (vtkInteractorStyle) when the user clicks onto the image. The click event will return the position of the click in 3D coordinates, and this position will be passed to this function to draw the point (hence an array of three doubles). The point's position is stored in internal storage.
    - void DrawLine()
        - This function is called by the interactor style object when two new points are drawn and the user enables drawing lines. Since points are stored in internal storage, the function takes the two most recent points drawn and draws a line in between them, and displays it onto the image.
    - void SetLineWidth(double width)
        - Set the width of a line
    - void DrawCube(double p[3])
        - This function is called by the interactor style object when the user clicks onto the image. The user will enable an option to draw cubes, and instead of drawing a point onto the mouse click, a cube is drawn instead.
    - void SetWidthOfCube(double width)
        - Set the width of a cube
    - void SetHeightOfCube(double height)
        - Set the height of a cube
    - void RotateCube(double angle, int axis)
        - Rotate a cube by an angle (in degrees) in the specified axis (e.g. 1=x, 2=y, 3=z)
    - void ScaleCube(int scaleFactor)
        - Scale a cube by the factor specified
- 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.

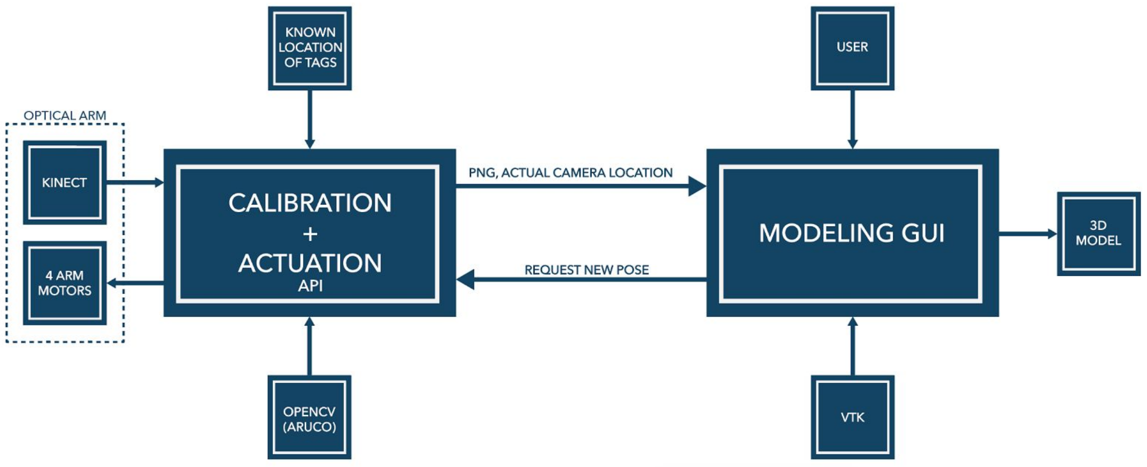
Calibration & Actuation API (Tom):

- 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 & Arucois
        - 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:
    - Pose getNewPose(Point3f desiredPoint)
        - This API is called by the modeling functionality to move the robot to the desired position. The returned Pose is an object containing the new Point3f location of the camera as well as a String holding the name to the new pose saved.
        - 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++.
- 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)
        - This function takes in two points and invokes Karl's API to move the motors. It then calls calibrate()
    - 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.
    - 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 size

- 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 to txt file so it can be imported upon restart of system.
- bool loadCameraCalibration(string name, Mat& cameraMatrix, Mat& distanceCoefficients)
    - Imports calibration numbers previously calculated into matricies from txt file
- 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.

Overall Architecture Diagram:


KNOWN LOCATION OF TAGS
USER
OPTICAL ARM
KINECT
4 ARM MOTORS
CALIBRATION + ACTUATION API
PNG, ACTUAL CAMERA LOCATION
REQUEST NEW POSE
MODELING GUI
3D MODEL
OPENCV (ARUCO)
VTK