# Projection Mapping in Surgery

## Technical Summary

The goal of this project was to develop a projection mapping prototype for surgery (particularly for cranioplasty) that can project important patient data, such as CT scans, onto patient bodies in real time. For the purpose of cranioplasty, the projection mapping prototype ideally will be able to project a STL model of the skull along with the appropriate implant onto the patient's actual skull. And through this system, it will be possible to obtain the appropriate position and orientation of the implant to best fit the patient's skull defect.

The significance of this development is the impact it will have in the implant design process. Currently, in cranioplasty, the surgeon designs the implant for a patient's skull based solely on what he or she is able to see. This means that the design process is highly dependent on trial-and-error and can therefore be strenuous and tedious for the surgeon. This technology will streamline the design process such that it is automated and precise.

Our prototype consists of the Realsense D415 camera and a portable projector, and the main problem that I tackled is how to solve for that appropriate position and orientation of the implant for the patient's defect and how to obtain the necessary data to be able to solve that problem. The crux of the solution I have designed is a marker-based registration procedure, and the solution can be broken down into the following parts: camera calibration, AR marker pose estimation, pivot calibration, and registration.

For calibration, there are existing dynamic calibration tools for the Realsense camera. These tools will perform a typical checkerboard calibration procedure and save the intrinsic parameters into the camera's hardware. I also performed my own camera calibration procedure using OpenCV just in case the camera's default intrinsic parameters were not accurate anymore.

The most time-consuming portion of the project was the AR marker pose estimation. The first draft of my software development utilized OpenCV to perform the pose estimation. Unfortunately, the results obtained using OpenCV's built-in functions produce inaccurate and unstable results due to a phenomenon called pose ambiguity, which occurs most prominently when the camera optical plane is parallel to the plane of the AR marker. In order to perform pose estimation, OpenCV first locates the four corners of the AR marker and then uses these four corners as inputs to a method that solves a perspective-n-point problem. Four points is the minimum number of points necessary to obtain a unique solution for the marker's pose. However, because of the noise that is present in the readings of the marker's four corners, even slight instability in the corner locations can produce drastically different poses. And when the camera frame is approximately parallel to the plane of the marker, it is highly likely that a marker could project at the same pixels from two different camera locations. Having the camera look at the AR marker from an oblique angle is best in obtaining the correct pose of the marker. Ultimately, this means that simply relying on the four corners of one marker to estimate its pose may produce very erroneous results, especially in orientation. And this would affect all parts of the system's pipeline because it relies on marker pose estimation. In order to mitigate this error, I implemented a geometric filter based on the known geometric orientation of the markers on the marker tool. For example, one of the marker tools used had a hexagonal head. There is one marker placed on the top face, and there is a marker placed on each side of the hexagon, which means six markers on the side of the tool. By forcing the system to ignore frames where only one

side marker is detected, we can verify that the orientations between pairs of consecutive side markers are approximately 60 degrees. If a pair of markers from any two consecutive sides of the hexagon has a difference in orientation of approximately 60 degrees, it is reasonable to believe that the orientations that were calculated using OpenCV were in fact relatively accurate and not subject to pose ambiguity. But in cases, where the orientation drastically differs by 60 degrees, the system can ignore those frames and not use the poses estimated using that frame. Unfortunately, although this filtering did improve the accuracy of both the position and orientation of the pose estimation, the results were not as stable as I desired. After conducting an accuracy evaluation with the markers at a distance of 0.6 m away from the camera, the orientation in marker pose had an error of approximately 5 degs with a standard deviation of 0.5 deg while there was approximately an error of 4 mm and standard deviation of 0.75 mm in position. An alternative to writing software based off the barebones methods provided by OpenCV that I then pursued was utilizing an existing ROS package called ar-track-alvar. Although I was unable to perform an in-depth accuracy evaluation as I had done with the OpenCV software I wrote, basic preliminary tests showed that the readings from the ROS package had a negligible standard deviation and produced very stable results at a wide range of distances. The error in position is approximately 6-8 mm.

After achieving stable and relatively accurate results for the pose estimation, the next step is to perform a pivot calibration. This is necessary because without it, the dimensions of the marker tool are unknown, leaving the user unable to record points touched using the tool. This portion of the project was simply a rehash of what I had learned in CIS I. Using the ar-track-alvar package available on ROS to obtain transformations to the tool coordinate frame as defined by the tool's markers, we achieved an error of approximately 8-10 mm, which unfortunately is larger than we had hoped but is reasonable given that the pose estimation error is approximately the same.

The final step of the pipeline is registration, which consists of the user recording points representing distinct anatomical landmarks on the patient's skull and then aligning that point cloud with the skull STL model obtained through CT scans. Using the ROS node I created, the user can launch the node and begin to hold and move the tool to anatomical landmarks on the skull, and the system will constantly be recording points as long as at least one of the markers on the tool is visible to the camera. Within a few seconds, the system is able to pick up as many as 1000 points, and then perform registration using a Python package called PyCPD, which defines the point cloud alignment problem as a probability density function and uses an algorithm called Coherent Point Drift. The registration outputs the transformation between the frame of the marker fixed rigidly on the patient's skull and the coordinate frame of the STL skull model. The system only needs to perform this registration one time, and using this initial transformation, the transformation from the projector frame to the STL skull model frame can be calculated as long as there is also information about the transformation from the marker frame to the camera frame. Because the accuracy of the points recorded by the user depends on the accuracy of the pivot calibration, I suspect that the points recorded also have a similar amount of error. But ideally, in future work, once I am able to greatly reduce the error in marker pose estimation, the recorded points will become highly accurate. If we ignore the error stemming from the marker pose estimation in the pivot calibration, PyCPD is able to perform registration with a well-recorded point cloud of anatomical landmarks and achieve an error of approximately 3 mm, which is promising.

For future work, the main focus will be greatly reducing the error in the marker pose estimation or circumventing the need for any markers. To address the former, I plan on incorporating depth data that is available through the Realsense camera. I also think it may be beneficial to do some sensor fusion with the IMU data and the depth data available in the Realsense camera. To address the latter, I believe there is a great viability in developing a markerless version of registration. This registration pipeline would directly use the point cloud data captured by the Realsense camera. Currently, it is unknown how accurate and how high-resolution the point cloud data is, but if the raw point cloud data could be registered to the skull STL model, then there would be no need for markers. The point cloud data would have relatively accurate depth data as it uses two IR cameras as opposed to the current marker pose estimation, which only uses RGB images. One potential issue is how accurate the registration will be given the number of points in the point cloud that are not a part of the skull. Unfortunately, there is no good way to solely segment the skull out of the point cloud. This could be done through potentially two ways. The first is that a distance filter can be implemented such that any point that is farther than some threshold is discarded, and the skull should be placed at a distance less than that threshold from the camera frame. The other way that does not rely as much on user behavior is developing a network that can identify regions of a point cloud that resemble a skull. If such a network is developed, then the point cloud can be automatically segmented, and the segmented region of the point cloud can be registered against the skull CT model.

## Management Summary

As the sole member of this team, I worked on all the technical parts as mentioned in the technical summary. For the purpose of achieving a demo and some sort of visualization, all the projector-related work was done by my mentor Joshua Liu and a fellow labmate Weilun Huang.

What my mentors and I originally planned to do in terms of our maximum deliverables is to create the projection mapping prototype that will be able to project a skull CT model and the appropriate implant onto the actual skull model. After the projection, we had hoped to extract point cloud data of the skull defect using the Realsense RGBD camera. The registration used for the projection would provide the appropriate transformation to position and orient the implant while the point cloud data of the defect will provide information on the shape that the implant should be. What we accomplished is a projection of the skull CT model onto the actual skull model. Unfortunately, no CT models of the appropriate implants were available to us, and texture wrapping needs to be implemented such that the projection can conform nicely to the actual skull model. In addition, the marker-based registration is not as accurate as we had hoped for due to the inaccuracies of the pose estimation in the ar-track-alvar ROS package. We plan on pursuing two paths to improve this: incorporate depth data from the Realsense camera to the marker-based registration procedure and create a markerless registration pipeline.

Throughout my work on this project, I have learned a great deal about computer vision and have increased my proficiency in using open-source software, such as OpenCV and ROS. Working with the Intel Realsense camera helped me understand what a pinhole camera model is, how to calibrate RGB and IR cameras, and how to perform stereo calibration. Performing ArUco marker detection and tracking taught me what sort of conditions affect detection and pose estimation accuracy and how to mitigate these circumstances. Other parts, such as pivot calibration, point recording, and registration, acted as a refresher for many concepts that I had learned in CIS I, and I was glad to implement them in real-world applications. And throughout every step of this project, I have learned the importance of performing accuracy evaluations. These evaluations helped inform me which part of the pipeline requires the most improvement and focus instead of blindly moving forward to other parts that depend on previous parts' accuracy.

**Technical Appendices**

Software - https://git.lcsr.jhu.edu/camp/ar-2019/ppmd (also listed as a link on wiki)

Other non-critical code is stored in private Github repository.