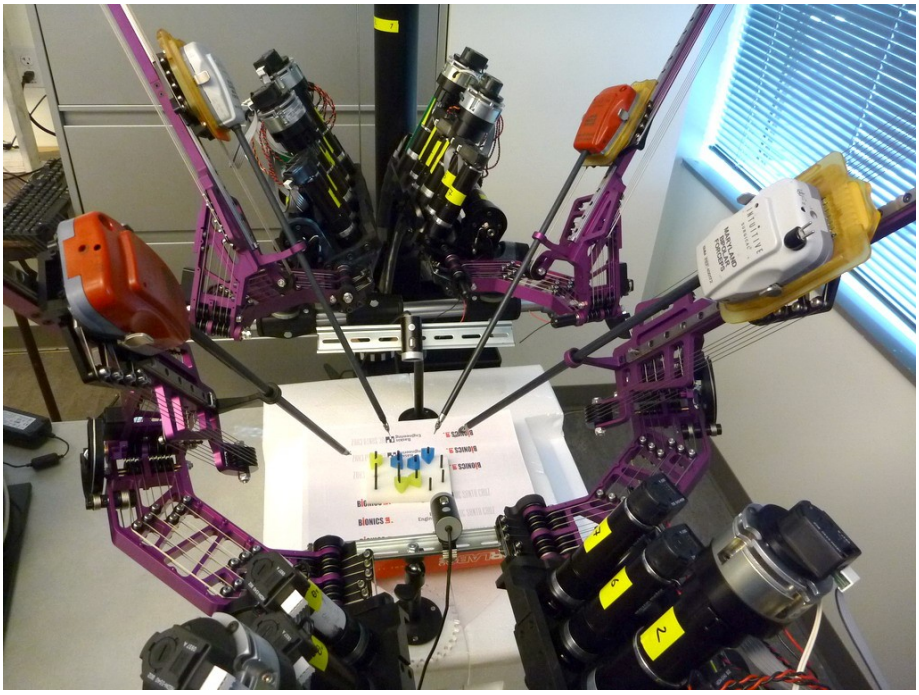
Extending the Raven Surgical Robot System

Amanda Gentzel
Advisor: Jacob Rosen

The Raven Surgical Robot System

The Bionics Lab at the University of California Santa Cruz contains the Raven Surgical Robot System, a system consisting of four robotic arms with surgical tools. The arms can be controlled by a surgeon using haptic devices, allowing him to operate the arms remotely. This means that the surgeon does not need to be anywhere near the patient to operate, allowing surgery to be performed in remote or hard-to-reach locations, such as the battlefield. The Raven system also allows for two surgeons to operate collaboratively, each controlling two arms and each potentially in completely separate locations.

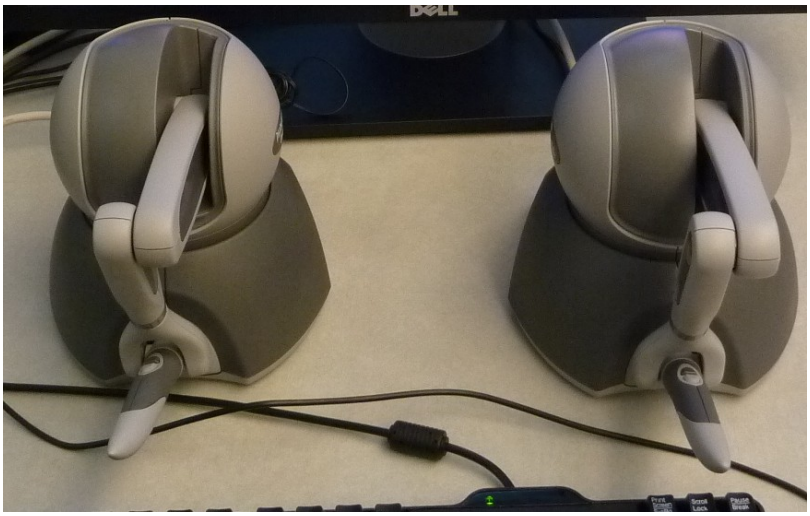


The Raven Surgical Robot System

Simulation

In order to effectively operate the robotic arms, training is required. However, surgeons would not necessarily always have access to the actual arms. To aid in training, then, a simulation can be used. The simulation, written in C++ using OpenGL, contains simulated arms that are controlled using SensAble Phantom Omni haptic devices in the same manner as the physical arms. Each haptic device keeps track of its position, and the simulation constantly records the change in that position. This change is used to update the location of the tool on the end of each simulated arm. By knowing the location of the tool, inverse kinematics can be used to

calculate the joint angles of the arm which are used to correctly orient the different sections of the arm in the simulation.



The SensAble Phantom Omni haptic devices

Extending the Simulation

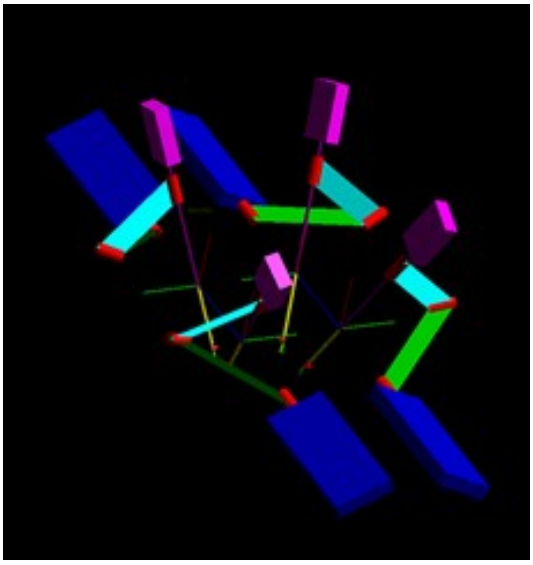
In order to improve the simulation, I worked on three main things. First, I increased the number of arms present in the simulation from two to four. The kinematics calculations were already in place for the additional arms, so all that was required was some additional checks to position the arms in the correct places as well as a couple simple variable changes. My second goal was to increase the realism of the simulation by replacing the original, simple models of the arms with more detailed models from SolidWorks. Finally, I worked on adding a surgical training task to the simulation.

Modeling the Arms

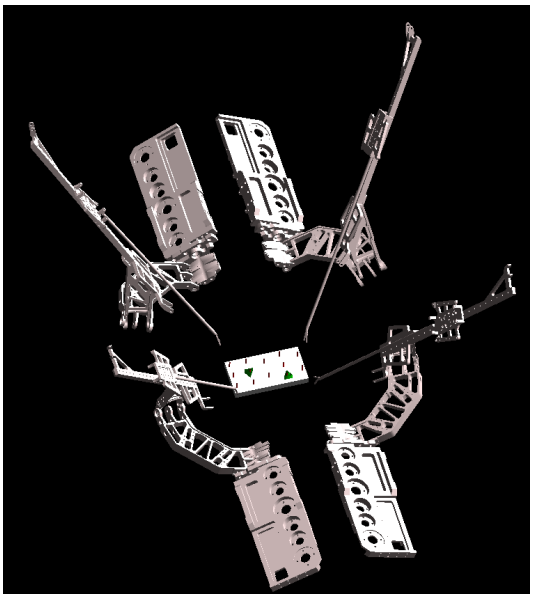
There are currently two options for the graphics of the simulation. If the public variable *useObjFiles* in Main.cpp is set to false, the arms are simply represented using OpenGL primitives. If *useObjFiles* is set to true, the graphics are instead rendered using more detailed representations from SolidWorks. The detailed models are exported from SolidWorks as .obj files. This is a very transparent file format that simply lists all the vertices in a three-dimensional model and then specifies which vertices make up faces. In order to make the arm move realistically, the detailed representation had to be stored in four different obj files per arm, one for each independently moving part. Since there are two possible orientations for an arm (the first and second arms are mirror images of each other, as are the third and fourth), eight total obj files are needed.

When the simulation is run and *useObjFiles* is true, the obj files are loaded. An open source obj loader was found online and used to actually read in the vertices, vertex normal and faces from

the obj files. Each element is stored as a struct, and these structs are stored in three arrays for each model (for the vertices, normal, and faces). Each model is defined by a struct containing these three arrays, so each arm is described by four of these model structs, one for each moving part. Each time the scene is drawn, the program loops over all of the faces and, using OpenGL triangles, draws them using the coordinates in the vertex and vertex normal arrays. Each model is then translated to its origin, rotated by the joint angles retrieved from the inverse kinematics, and translated into position. In this way, the detailed models move in the same way as the simple ones. This can be seen by commenting out the four *else* statements in the DrawRobot function in Main.cpp.



Simple graphics in the simulation (when useObjFiles is set to false)



SolidWorks graphics from obj files (when useObjFiles is set to true)

Creating a Task

When the variable *drawTheTask* in Main.cpp is set to true, a surgical training task is drawn in the simulation. The base and pegs are drawn using OpenGL primitives. The triangular objects that can rest on the pegs are a bit more complicated. Their basic shape is a triangular prism with a cylinder cut through it. However, OpenGL does not provide any functions for creating such a shape. The shape, then, is created in multiple parts. To define the basic shape, a hollow cylinder is drawn, followed by a hollow triangular prism. (created as a cylinder with three slices) The space between them is filled using an OpenGL quadrilateral strip. By calculating the equation of the circle and the three lines comprising the triangle, thin quadrilaterals can be drawn between the two, filling the space.

With the task drawn, the next goal was to allow the user to manipulate it with the arms. This consists of three steps: picking up an object, moving it around with the arm, and dropping it. To pick up an object, two conditions must be met. First, the tool must be within a small radius of the center of the object, and second, the tool must be closed. Once these conditions are met, the object is considered grabbed. Once grabbed, the object needs to move around with the tool. This is accomplished by drawing the object using the same model view matrix that was used to draw the tool. This means that, when the tool is moved or rotated, the object moves in the exact same way. At any point, the user can press the button to open the tool, leading to the third component of manipulating the task: dropping an object. This action is triggered when the tool holding the object is opened. The object should then fall back to the board from its final held location. At this point, this location is acquired by using the existing *posMatrix* of the arm. This method, however, does not quite work at this point. This is because *posMatrix* defines the location where the tool connects to the arm, not the tip of the tool. In order for the dropping to function properly, the angle of the tool needs to be taken into account to calculate the location of the tip of the tool.

Future Work

There is still a lot of work that can be done with this simulation. One major improvement would be to add haptic feedback, which would allow the user to actually feel the simulated task and interact with it more realistically. In its current state, with the simple graphics view of the simulation, the grabbing of objects is simplistic, and dropping them is not quite functional. With the more realistic graphics, an issue with the kinematics needs to be addressed before it can interact with the task. For both the simple and realistic models, the position of the haptic device is tracked, and from that, the angles of the joints of the arm are calculated. However, when the realistic arms are drawn using those joint angles, the tool does not move in the same way as the tool on the simple arms, despite the fact that the joint angles seem to correspond. Once this discrepancy is addressed, the grabbing functionality will function in the same way, regardless of the chosen graphical style. Other future work could involve incorporating more training tasks and allowing for collaborative control of the simulation.