Bryant Pong

ECSE-4750

11/29/14

**Computer Graphics Final Project – Robotic Arm Simulator**

**Project Overview**

Robotics is a relatively new field that has gained traction and interest quickly in recent years. However, there are two problems that budding roboticists face when learning about this exciting field. Robotics has a high initial barrier to entry; even the cheapest industrial robot is at least $10,000! Secondly, robotics requires a vast amount of theoretical knowledge before one can even implement basic functionality. These two barriers can easily be overcome with the creation of a robot simulation.

Unfortunately, many robot simulators today are currently either extremely expensive (to the tune of several hundred dollars) or are platform-specific. For example, a popular robot simulator is the Gazebo project (<http://www.gazebosim.com>). However, this software is currently only supported on Linux platforms and has a large set of software requirements, forcing interested roboticists into learning how to use Linux. Additionally, current simulators are difficult to use and have unintuitive controls. I decided to create a WebGL robotic arm simulator to allow users to simulate robotic arms on their browsers to avoid the problems of high-cost and steep learning curves.

**Third-Party Libraries Used**

To avoid reinventing the wheel, I am using Angel’s files “MV.js”, “init-shaders.js”, and “webgl-utils.js”. The other files comprising my project are my own work.

**Supported Platforms**

My simulator was tested in Ubuntu Linux 14.04 64-bit as well as Windows 7 64-bit using the Google Chrome browser.

**Key Design Requirements and Design Decisions Made**

**Positioning and Orienting the Camera**

I first looked at the dimensions of industrial robotic arms to determine the scaling of the simulator’s canvas. From my research, I determined that the majority of robotic arms at zero-configuration (this is the position that the manufactures define as all joints in the arm are at 0 degrees; usually zero-configuration is when the arm is pointing straight out) are within a 10-foot workspace. This means that my canvas scaling was from -10 to 10 in all directions. As a result, my projection matrix was set using this constraint.

Next, I had to determine what the best possible viewing angle was. Originally, I wanted a wide and sweeping view of the robot from Cartesian location (1, 1, 1). However, after initial setting my model-view matrix to look at (1, 1, 1), I realized that the camera was zoomed out too far; the robot arm was rendering too small! To compensate, I set the camera at Cartesian (0.1, 0.1, 0.1); this gives the same effect as setting the camera at (1, 1, 1) except the field of view is narrower and thus the robot is more clearly seen. Thus, my model-view matrix is initially set to (0.1, 0.1, 0.1) while treating the positive Z-Axis as the “up” direction.

**Forward Kinematics Calculations**

With the viewing out of the way, I now needed to determine the functions needed to calculate the robot’s forward kinematics. The forward kinematics is the procedure in which the orientation and location of each joint in the robot is determined. The forward kinematics is comprised of two parts: a rotational and a translational calculation. The rotational portion determines the angle that a joint is oriented; it is simply the sum of all the prior joint angles together. The translational forward kinematics determines the current Cartesian location of a joint. For a reference frame and a frame attached to the ith joint, the Cartesian location of the ith joint can be found using the following recursive formula:

Where denotes a position vector between two consecutive frames and denotes a 3x3 Rotation Matrix between two frames.

I looked into Angel’s “MV.js” to determine the matrix and vector functions that already existed. I noticed that while the “mult()” function existed, it only works on multiplying matrices together; this function does not work for multiplying a matrix and a vector together. Therefore, I created my own function “multMatVec()” that accomplishes this task. I also created functions to calculate the 3-Dimensional Rotation Matrix for a joint via the Euler-Rodrigues formula.

With these functions in place, I was able to easily calculate the forward kinematics for any robot arm by using dynamic programming: my program simply iterates through the list of joints. For each joint, it calculates the forward kinematics for that joint, adds the forward kinematics of the previous joint, and finally pushes the result into an array for user displaying. In this way, my program calculates the forward kinematics efficiently; it can read the previous kinematic value from the kinematics data array (an O(1) operation) rather than having to redundantly calculate the previous kinematics values.

**Dynamically Adding Joints**

The next challenge was to determine the best user interface to easily add new joints to a robot arm. A robot joint in the simulator needs to store its Cartesian Location (X, Y, Z) coordinates, its rotation angle and axis, and its joint color. Thus, my user interface provides text boxes for the user to enter in the joint location and rotation axis.

Providing the interface to set the joint color was slightly tricky. I created an HTML5 “color” element to easily select a color. Unfortunately, a color element returns the specified color in a 6-digit hexadecimal number (from #000000 (black) to #FFFFFF (white)). WebGL requires a vec4() object where the red, green, and blue values are from 0 – 255. Therefore, I created the functions “colorHexToDec()” and “convertColor()” to help me solve this problem.

Dynamically adding joints posed an interesting problem. Originally, I wanted to limit the number of joints that could be created to 10; however I felt that such a restriction was too limiting for a user. I quickly learned that I could dynamically add any number of joints and control them independently by using Javascript’s “document.createElement()” function. By using this function, I was able to create an HTML slider element for each joint that allows each joint to be independently controlled from -360 to +360 degrees. Please refer to the function “addJointCallback()” for more details.

Finally, I stored all the joint data into a series of 3 arrays. One array (“joints”) holds the Cartesian location of each joint as well as its color. A second array (“links”) holds the lengths of the links connecting joints. Finally, the third array (“jointAngles”) contains the joint angles for each joint. By segregating the joint data in this fashion, I was easily able to access only what I needed when I needed the data.

**Rendering Joints and Links – Original Method Attempted**

My greatest challenge came to me when it was time to have all the joints and links rotate with respect to the previous joints. Originally, when a joint or link was created, I had a function that would push another set of 36 vertices into the points vector to be sent to the GPU. I had originally created a matrix that would store all the joint/link positions, angles, rotation axes/matrices, and colors. This matrix would be passed to the vertex shader – which would then manually rotate

**Rendering Joints and Links – Using Hierarchical Modeling Approach**

By using the hierarchical modeling approach and creating a single instance of a cube and then performing transformations on the cube, I was able to scrap 400 lines of code!

**Displaying and Formatting Kinematics Data**

**Conclusion**

After this class has concluded, I will be handing my codebase off to Professor Wu, who may be using my simulator to help teach her Robotics 1 course in Fall 2015.