Interactive Graphics

Project Name: "Manipulator Simulator"

Lorenzo Vianello

1 Project Overview

I created a simulator for a robot manipulator. For this task it has been used mainly the three.js library. For better represent the environment I imported some object, like a table and a shelf, in format .json. Then I created the robot, based on a hierarchical model.

The simulator uses some concept of robotics like workspace, direct kinematics, inverse kinematics, path planning: for this theoretical part I made a preliminary work in which I applied these concepts to my robot extracting all the useful equations.

In the simulator I built there are two way to move the robot, the first one is selecting one of the balls present in the scene and the robot will reach that with its end-effector, the other way is changing directly the joint angles using some controllers.

2 Initialization

2.1 Environment

I have created a scene and added a perspective camera to this scene; the camera is initially settled with an angle of around 45 degrees. In order to control the camera orbit around the robot I have used the class OrbitControls; I have also set a distance range from the center of the scene, this distance can be changed by using the mouse wheel.

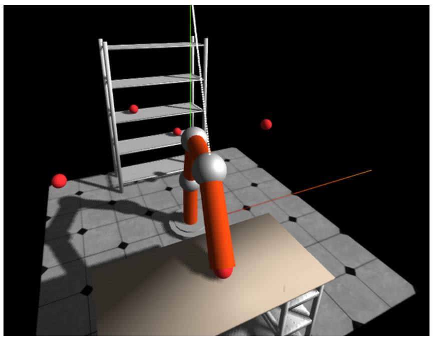
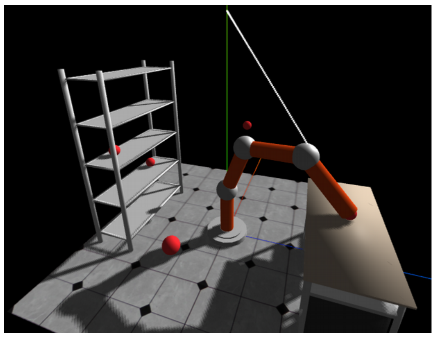
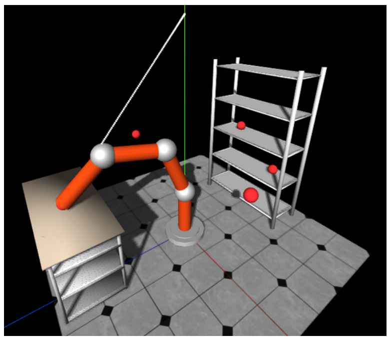
For the environment's lighting has been used a white directional light, and a gray ambient light. Only directional light give shadows to the objects inside the scene.

The plane is built using *PlaneBufferGeometry*, which is a class representing a 3D plane. A texture that remembers a tiled floor is associated to the plane.

Over the plane are posed some objects; to load the .json files I used the function *ObjectLoader*; the object is made up by one or more material and each of these include the color, the texture and the reflection map of individual materials.

2.2 Enviroment's Element

In this section I will describe the various objects included in the scene.

2.2.1 Fixed Objects

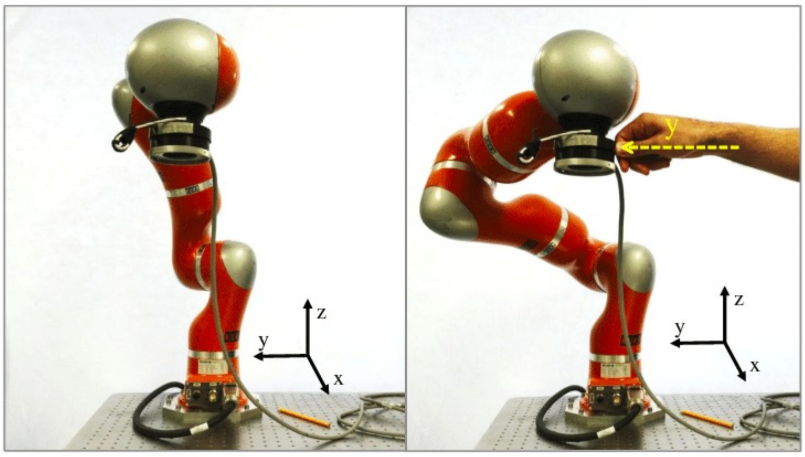
Let’s start by analyzing fixed objects like the table and the shelf. Both have been loaded with the function *ObjectLoad*, then I have rotated, translated and scaled them in order to fit all these objects to my simulation. The result is intended to recall a normal robotics laboratory or a simple room in which the robot has to perform some tasks.

In the scene are also present some spheres that are used to tell the robot how to move.

To select which sphere is the next target of the robot I used *THREE.Raycaster()* that helps to figure out where a mouse click occurs, in my case over which one of the spheres it is.

2.2.2 Robot

The robot is a hierarchic model composed by 4 elements; each element is a cylindric link of unitary length and we can rotate each link from its base with respect to all axes, but, how we’ll see later, I choose to consider only some axes for the rotations. At the base of each link is collocated a sphere, the color and the structure want to remember KUKA Robots.



To build this hierarchic model I called the function *createArms()* that creates one by one all the components of the Robot and links them each other with the function *.add()*.

To create all the objects in the scene I used the Three.js library that helps a lot in building complex objects. Each one is built using the function *mesh()* that want the geometry and the material of the object. Then we can transform the object as we want, translating, scaling or rotating it.

The robotics behind my project:

Direct Kinematics:

Forward kinematics refers to the use of the kinematic equations of a robot to compute the position of the end-effector from specified values for the joint parameters.

Since in my project the robot has 7 degree of freedom my equations are very complicated, so I decided to simplify them adding some approximations.

First of all I assumed that only the base can rotate around the y-axis while the others joint can execute only rotations around x-axis, in this way I can cover all the 3D positions in the map reducing a lot the complexity of my system.

So my robot now is a 4 DoF one. The equations I obtained are the following:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} sen(beta)*(sen(q1)+sen(q1+q2)+sen(q1+q2+q3)) \\ 1+\cos(q1)+\cos(q1+q2)+\cos(q1+q2+q3) \\ \cos(beta)*(sen(q1)+sen(q1+q2)+sen(q1+q2+q3)) \end{pmatrix}$$

The robot can execute the given positions in different configuration of the joints, so this can bring to some problems in my implementation; So I decided to introduce also the final orientation of the end effector, in this way we can cover all the Dof eliminating the redundancy .

$$\begin{pmatrix} \partial 1 \\ \partial 2 \\ \partial 3 \end{pmatrix} = \begin{pmatrix} sen(beta)sen(q1+q2+q3) \\ \cos(q1+q2+q3) \\ \cos(beta)sen(q1+q2+q3) \end{pmatrix}$$

Inverse Kinematics:

In robotics, inverse kinematics makes use of the kinematics equations to determine the joint parameters that provide a desired position for each of the robot's end-effectors.

Specification of the movement of a robot so that its end-effectors achieve the desired tasks is known as motion planning.

Inverting the equations calculated before we can reach functions that map position and orientation of the robot in joint configurations.

$$\begin{gathered} \mathbf{beta} = Atan2(x,z) \\ sen(q1+q2+q3) = \sqrt{(\partial 1)^2+(\partial 3)^2} \\ [q1+q2+q2] = Atan2(sen(q1+q2+q3), \partial 2) \\ x' = \frac{x}{sen(beta)} - sen(q1+q2+q3) \\ y' = y-1-\partial 2 \end{gathered} \quad > \quad \begin{gathered} y' = \cos(q1)+\cos(q1+q2) \\ x' = sen(q1)+sen(q1+q2) \\ solve\, the\, system, get\, (\mathbf{q1}), (\mathbf{q2}) \\ (\mathbf{q3}) = [q1+q2+q3]-q1-q2 \end{gathered}$$

Once we have these equations we can associate to a pose of the end-effector a set of angles of the joints. This will be useful for the path planning;

Path Planning:

For my simulation I need to move the Robot’s end effector from an initial pose (position and orientation) to a final one keeping an harmonic movement as much as possible.

In my configuration I chose to implement linear path planning, this is the shortest way to go from a point to another but can create lot of problems, for example:

- If we need to go from a point to another specular with respect to the robot, the configuration of the robot encounters some singularities;
- The path doesn't consider the presence of obstacles in the scene, so to solve this problem I sometimes introduced manually some middle points that help to get close to the target points.

To do that we have to build the equations that represent the position of the moving end-effector during motion from initial to finale pose.

$$x(t)=x0+3\left(\frac{xf-x0}{T^2}\right)t^2-2\left(\frac{xf-x0}{T^3}\right)t^3$$

$$y=\left(\frac{yf-y0}{xf-x0}\right)(x-x0)+y0$$

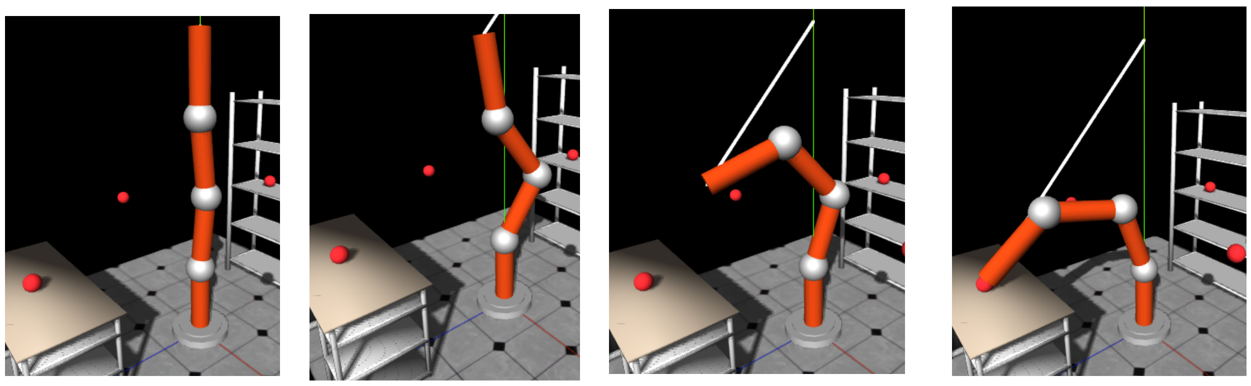
$$z=\left(\frac{zf-z0}{xf-x0}\right)(x-x0)+z0$$

Then knowing where the end-effector is at time T we can reconstruct the configuration of the robot at same time using the inverse kinematics;

3. Render function:

This is the main loop of my program, it calls itself so to be repeated infinite times.

Each cycle of the loop I update a counter T that help me to face the time passing from initial time, and to control the path planning of my robot. Every time that a new linear path start, T is put to zero. This is an example of a path planning execution:



The white line shows the executed path.
Inside the render function are called 2 functions used to update the scene in execution:

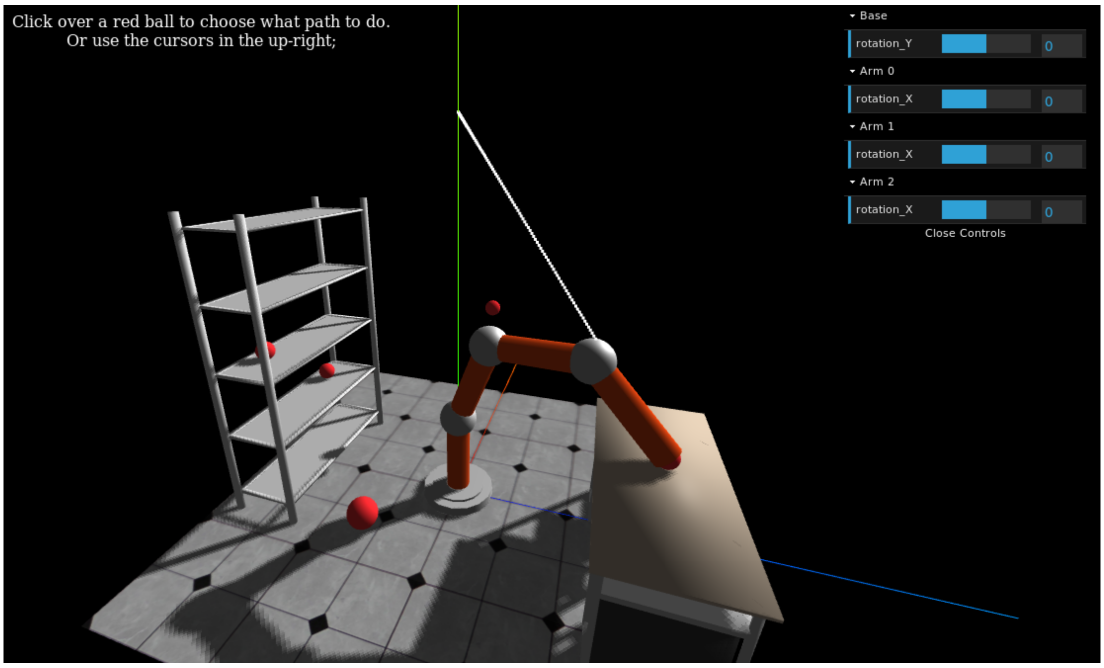
-updateDatGui()

-updateMovement()

3.1 UpdateDatGui:

In this function we use the library dat.guy to read informations passed through the switches present in the scene; the switches are created from the same library and a folder corresponds to each one, the switch have a min and a max value that represent how much we can rotate a link of the robot.

Every time that a movement is imposed we recall the function *directKinematics()* to update the position of the end-effector.


Click over a red ball to choose what path to do.
Or use the cursors in the up-right;
Base
rotation_Y
0
Arm 0
rotation_X
0
Arm 1
rotation_X
0
Arm 2
rotation_X
0
Close Controls

3.2 UpdateMovement:

Inside this function, using the current time T and the current path, I calculate where the end-effector should be using the function

positionT, orientationT =calculatepathPoseAtTime(position0, orientation0, position1,orientation1, t)

that executes the path planning. Once we have these value I use the function *inverseKinematics(position, orientation)* that calculates the joint angles using the inverse kinematics. So I update the position of the robot. To update the orientation of the joint is sufficient to cycle over all arms and change their orientation:

```
baseArm.rotation.y=ang[0];
        for ( var i = 0; i < arms.length; i +
              + ) { var arm = arms[i];
              arm.rotation.x=ang[i+1];}
```

4. Conclusions:

The simulator I’ve created works and is efficient, it implements several features of robotics. Moreover, as it is implemented, it leaves a large space for possible modifications both in the structure of the robot and in its movements, giving the possibility to those who need to reuse it as they want.

List of Libraries used:

- three.js;
- three.min.js;
- OrbitControl.js;
- Detector.js;
- stats.min.js;
- dat.gui.min.js;