Eren Çağlaroğlu 11500168978 Deniz Caner Akdeniz 14021504096

PROJECT 8

Overview

The JavaScript code block represents a key part of a robot arm simulation using WebGL. The simulation aims to render a 3D model of a robot arm and enables user interaction through key presses for manipulating its joints.

Code Analysis:

The render function in the provided JavaScript code is a critical component responsible for rendering the robot arm's components based on their transformations. It utilizes WebGL to clear buffers and draw the arm's various segments.

Clearing Buffers:

The function starts by clearing the depth and color buffers using WebGL's clear method.

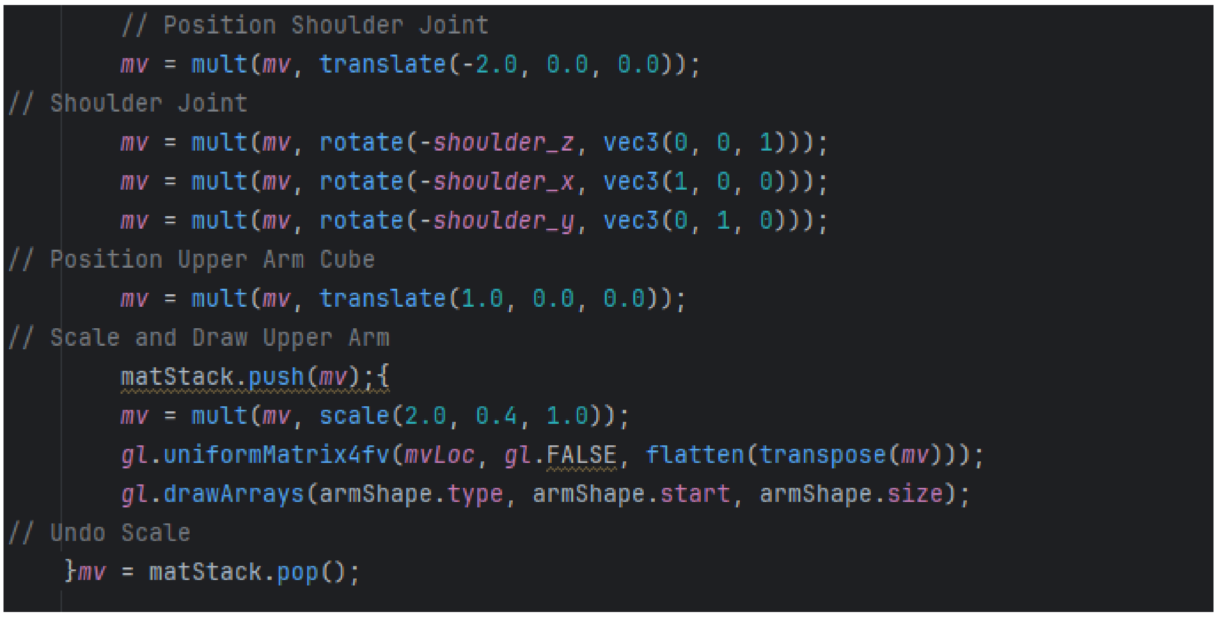
Arm Shape and Matrix Stack:

The variable armShape is assigned the shape data for the wireframe cube, and matStack is initialized as an empty array to store transformation matrices.

The current model-view matrix (mv) is pushed onto the matrix stack to save the current view.

Shoulder and Upper Arm:

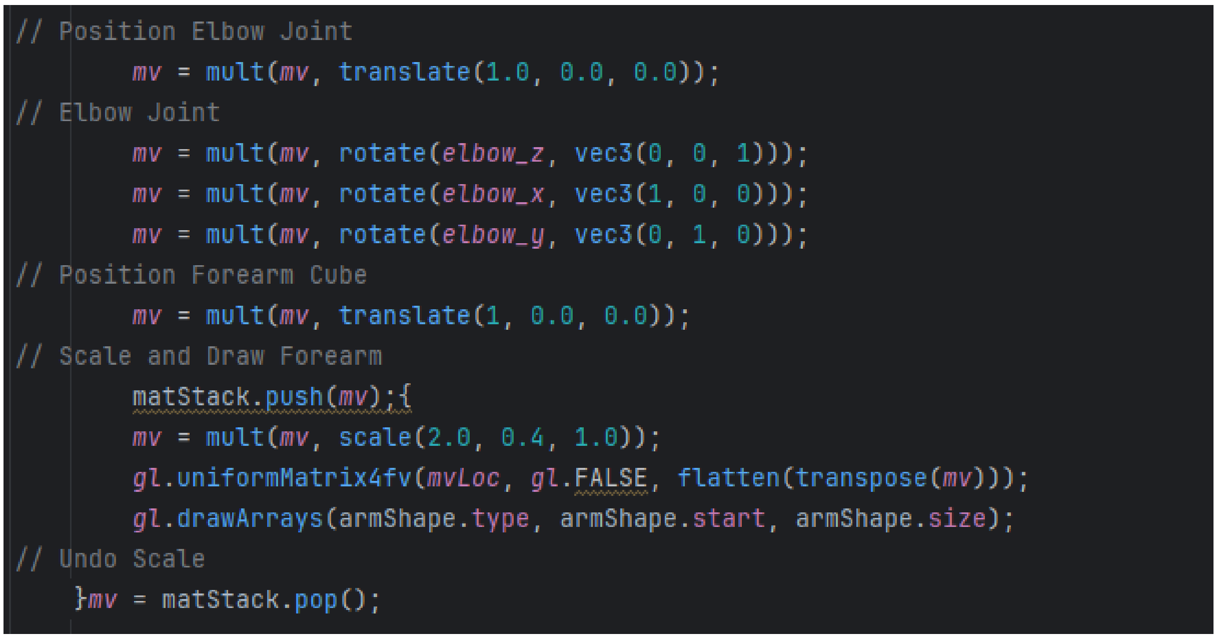
The code then positions the shoulder joint, rotates the upper arm based on the shoulder angle, and translates to place the upper arm cube. It scales and draws the upper arm using the specified shape.



```
        // Position Shoulder Joint
        mv = mult(mv, translate(-2.0, 0.0, 0.0));
// Shoulder Joint
        mv = mult(mv, rotate(-shoulder_z, vec3(0, 0, 1)));
        mv = mult(mv, rotate(-shoulder_x, vec3(1, 0, 0)));
        mv = mult(mv, rotate(-shoulder_y, vec3(0, 1, 0)));
// Position Upper Arm Cube
        mv = mult(mv, translate(1.0, 0.0, 0.0));
// Scale and Draw Upper Arm
        matStack.push(mv);{
        mv = mult(mv, scale(2.0, 0.4, 1.0));
        gl.uniformMatrix4fv(mvLoc, gl.FALSE, flatten(transpose(mv)));
        gl.drawArrays(armShape.type, armShape.start, armShape.size);
// Undo Scale
    }mv = matStack.pop();
```

Elbow and Forearm:

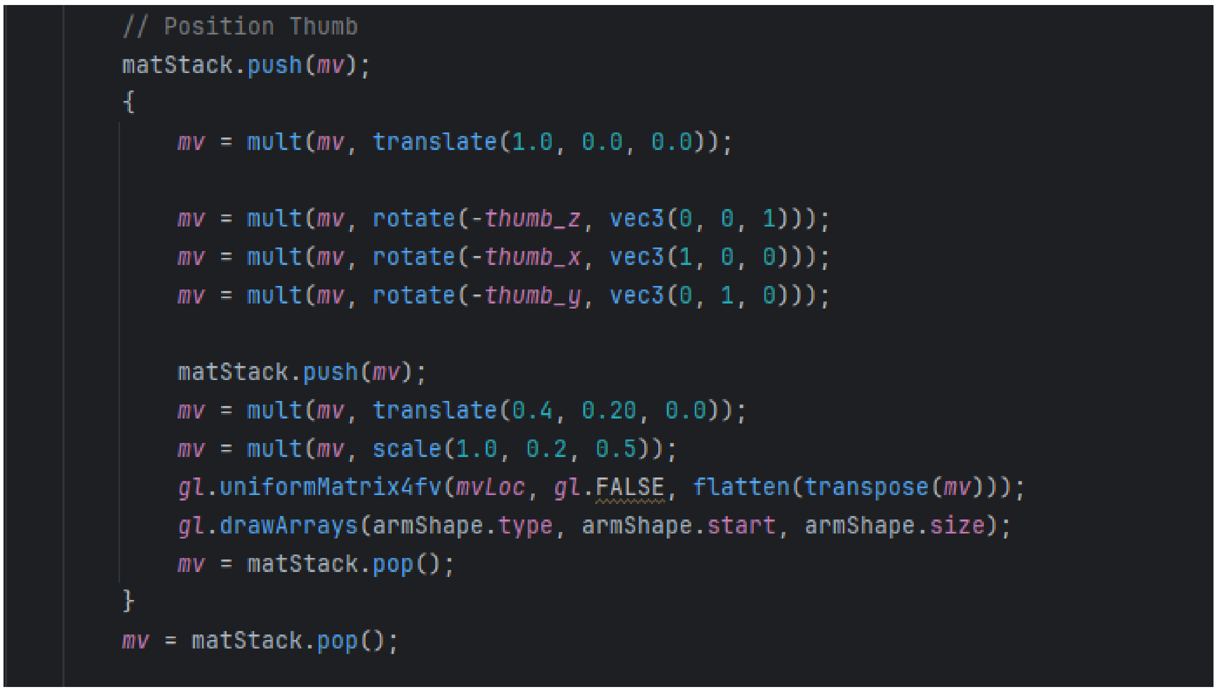
Similar transformations are applied for the elbow joint and forearm. The forearm is scaled and drawn using the specified shape.



```
// Position Elbow Joint
        mv = mult(mv, translate(1.0, 0.0, 0.0));
// Elbow Joint
        mv = mult(mv, rotate(elbow_z, vec3(0, 0, 1)));
        mv = mult(mv, rotate(elbow_x, vec3(1, 0, 0)));
        mv = mult(mv, rotate(elbow_y, vec3(0, 1, 0)));
// Position Forearm Cube
        mv = mult(mv, translate(1, 0.0, 0.0));
// Scale and Draw Forearm
        matStack.push(mv);{
        mv = mult(mv, scale(2.0, 0.4, 1.0));
        gl.uniformMatrix4fv(mvLoc, gl.FALSE, flatten(transpose(mv)));
        gl.drawArrays(armShape.type, armShape.start, armShape.size);
// Undo Scale
    }mv = matStack.pop();
```

Thumb Joint and Segment:

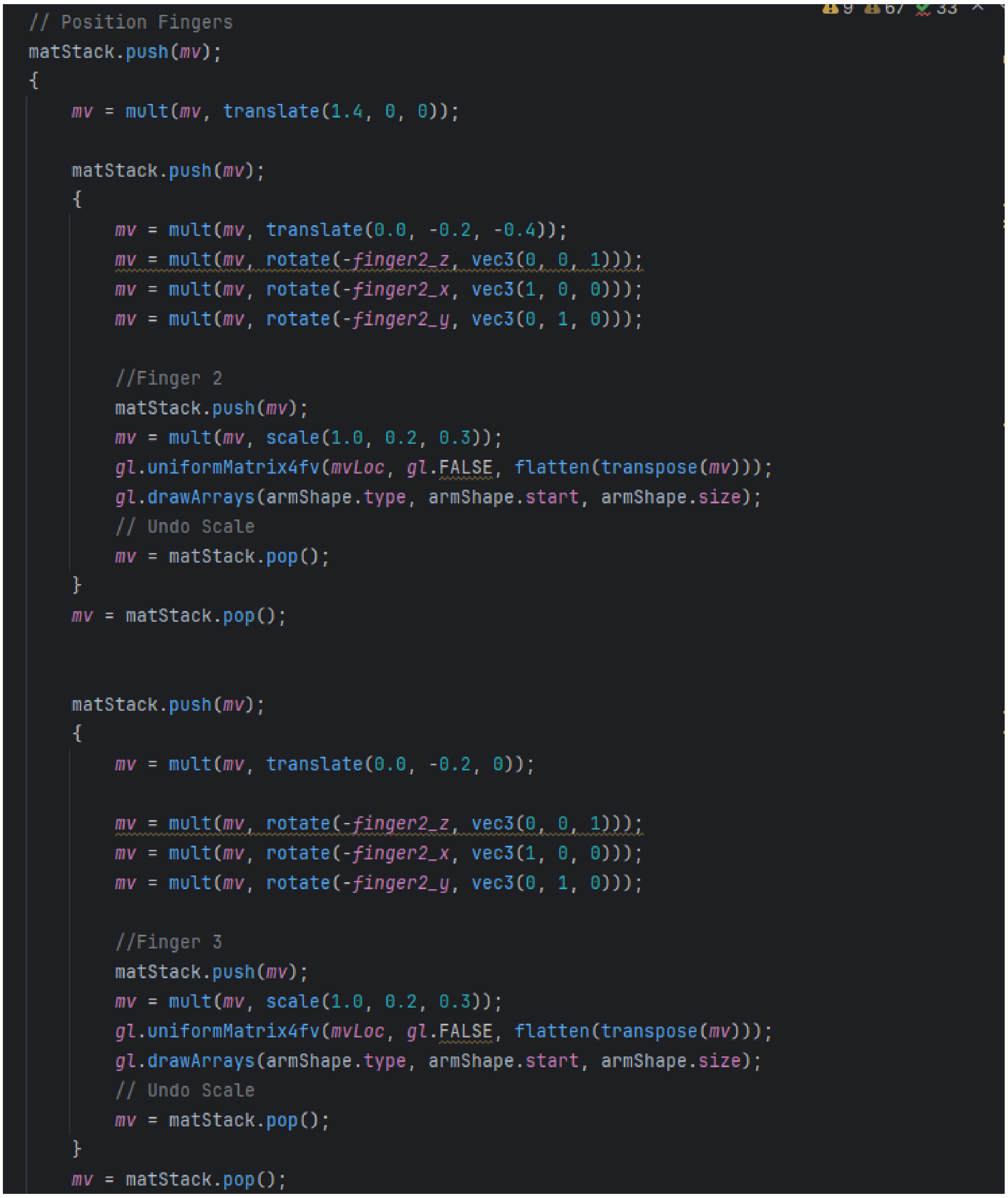
The code proceeds to position the thumb joint, rotate the thumb relative to the elbow, and place the thumb segment. It is then scaled and drawn using the specified shape.



```
        // Position Thumb
        matStack.push(mv);
        {
            mv = mult(mv, translate(1.0, 0.0, 0.0));

            mv = mult(mv, rotate(-thumb_z, vec3(0, 0, 1)));
            mv = mult(mv, rotate(-thumb_x, vec3(1, 0, 0)));
            mv = mult(mv, rotate(-thumb_y, vec3(0, 1, 0)));

            matStack.push(mv);
            mv = mult(mv, translate(0.4, 0.20, 0.0));
            mv = mult(mv, scale(1.0, 0.2, 0.5));
            gl.uniformMatrix4fv(mvLoc, gl.FALSE, flatten(transpose(mv)));
            gl.drawArrays(armShape.type, armShape.start, armShape.size);
            mv = matStack.pop();
        }
        mv = matStack.pop();
```

Finger Joints and Segments:

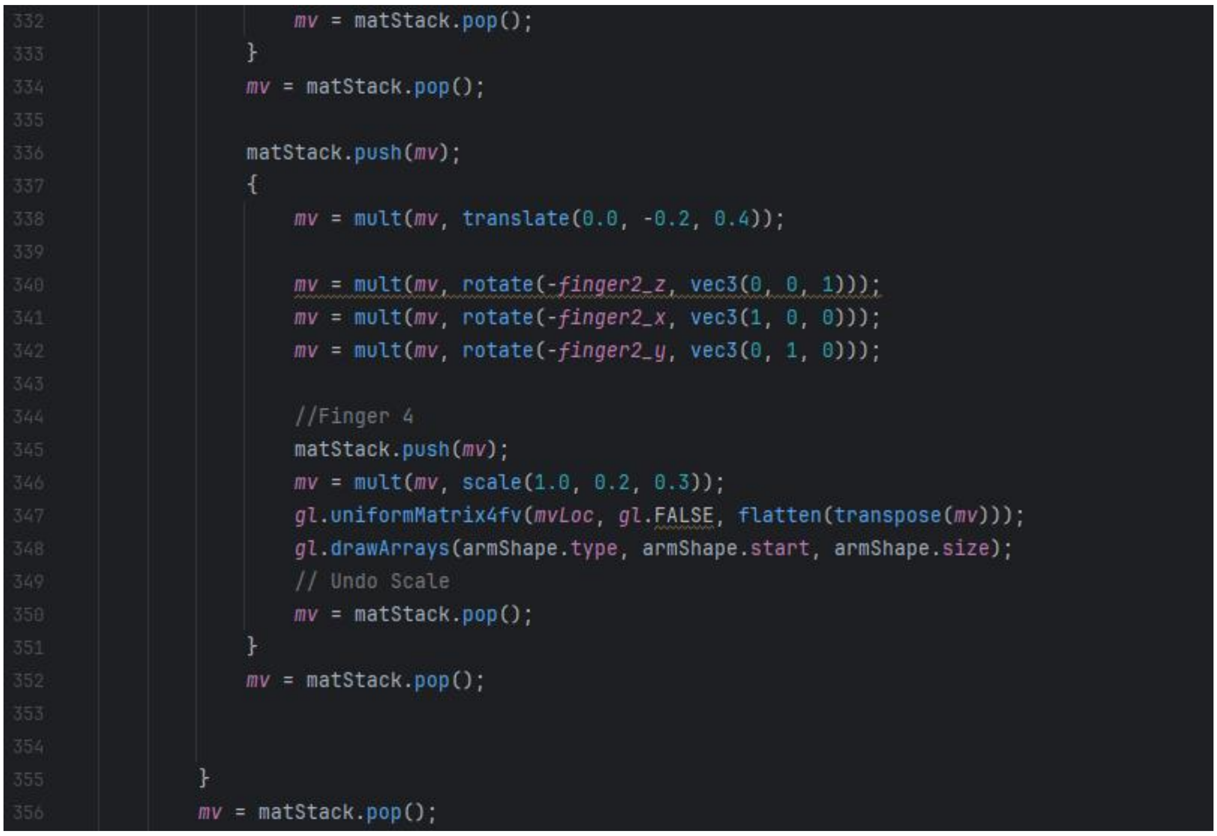
Similar transformations are applied for finger joints and segments. The code handles rotation, translation, scaling, and drawing for each finger.



```
// Position Fingers
matStack.push(mv);
{
    mv = mult(mv, translate(1.4, 0, 0));

    matStack.push(mv);
    {
        mv = mult(mv, translate(0.0, -0.2, -0.4));
        mv = mult(mv, rotate(-finger2_z, vec3(0, 0, 1)));
        mv = mult(mv, rotate(-finger2_x, vec3(1, 0, 0)));
        mv = mult(mv, rotate(-finger2_y, vec3(0, 1, 0)));

        //Finger 2
        matStack.push(mv);
        mv = mult(mv, scale(1.0, 0.2, 0.3));
        gl.uniformMatrix4fv(mvLoc, gl.FALSE, flatten(transpose(mv)));
        gl.drawArrays(armShape.type, armShape.start, armShape.size);
        // Undo Scale
        mv = matStack.pop();
    }
    mv = matStack.pop();


    matStack.push(mv);
    {
        mv = mult(mv, translate(0.0, -0.2, 0));

        mv = mult(mv, rotate(-finger2_z, vec3(0, 0, 1)));
        mv = mult(mv, rotate(-finger2_x, vec3(1, 0, 0)));
        mv = mult(mv, rotate(-finger2_y, vec3(0, 1, 0)));

        //Finger 3
        matStack.push(mv);
        mv = mult(mv, scale(1.0, 0.2, 0.3));
        gl.uniformMatrix4fv(mvLoc, gl.FALSE, flatten(transpose(mv)));
        gl.drawArrays(armShape.type, armShape.start, armShape.size);
        // Undo Scale
        mv = matStack.pop();
    }
    mv = matStack.pop();
```

Matrix Stack Restoration:

Finally, the matrix stack is used to restore the model-view matrix to its initial state.



```
                mv = matStack.pop();
            }
            mv = matStack.pop();

            matStack.push(mv);
            {
                mv = mult(mv, translate(0.0, -0.2, 0.4));

                mv = mult(mv, rotate(-finger2_z, vec3(0, 0, 1)));
                mv = mult(mv, rotate(-finger2_x, vec3(1, 0, 0)));
                mv = mult(mv, rotate(-finger2_y, vec3(0, 1, 0)));

                //Finger 4
                matStack.push(mv);
                mv = mult(mv, scale(1.0, 0.2, 0.3));
                gl.uniformMatrix4fv(mvLoc, gl.FALSE, flatten(transpose(mv)));
                gl.drawArrays(armShape.type, armShape.start, armShape.size);
                // Undo Scale
                mv = matStack.pop();
            }
            mv = matStack.pop();


        }
        mv = matStack.pop();
```

Functions

1. loadShape(myShape, type): This function takes a shape object (myShape) and a primitive type (type) as parameters. It concatenates the shape's points and colors to global arrays (points and colors) and records the start, size, and type of the shape.
2. init(): The initialization function is executed when the window loads. It sets up the WebGL rendering context, loads shaders, initializes attribute buffers, and loads shape data. Additionally, it configures WebGL settings such as the clear color, depth testing, and viewport.
3. animate(): This function handles the animation loop using requestAnimationFrame. It calculates the elapsed time and calls handleKeys for continuous key actions. It then invokes the render function.
4. render(): Responsible for rendering the 3D model, this function clears the canvas, updates the projection matrix, and iterates through the arm's components to apply transformations and draw each part.
5. handleKeyDown(event) and handleKeyUp(event): These functions are event handlers for keydown and keyup events. They update an array (currentlyPressedKeys) to keep track of the pressed state of keys and store the current state of the shift key.
6. isPressed(c): A utility function that checks if a specific key is pressed, using the character code of the key.

7. handleKeys(timePassed): This function is called continuously during animation to handle key inputs. It calculates how much to move based on the elapsed time and updates the model accordingly. Key presses are mapped to specific actions like rotating joints, toggling rendering modes, and switching projections.
8. These functions collectively provide the structure and behavior for the interactive WebGL application, allowing users to manipulate and observe the 3D robotic arm through keyboard inputs.

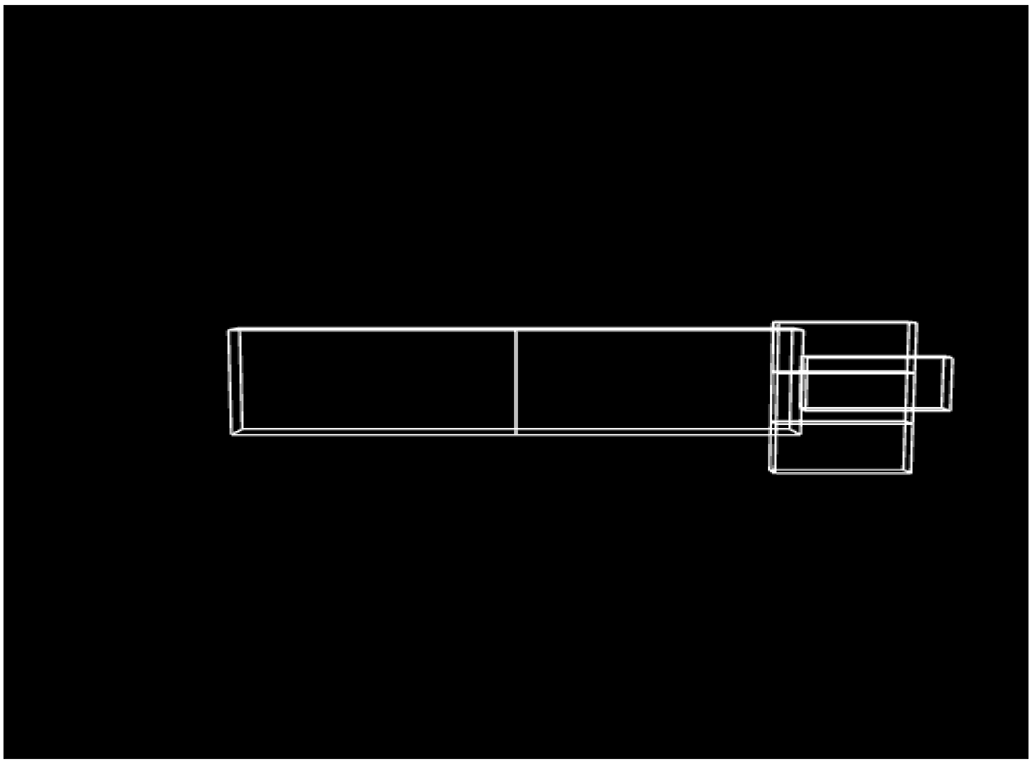


Figure 1. x/X: to rotate the arm on the X axis so you can see it from different angles

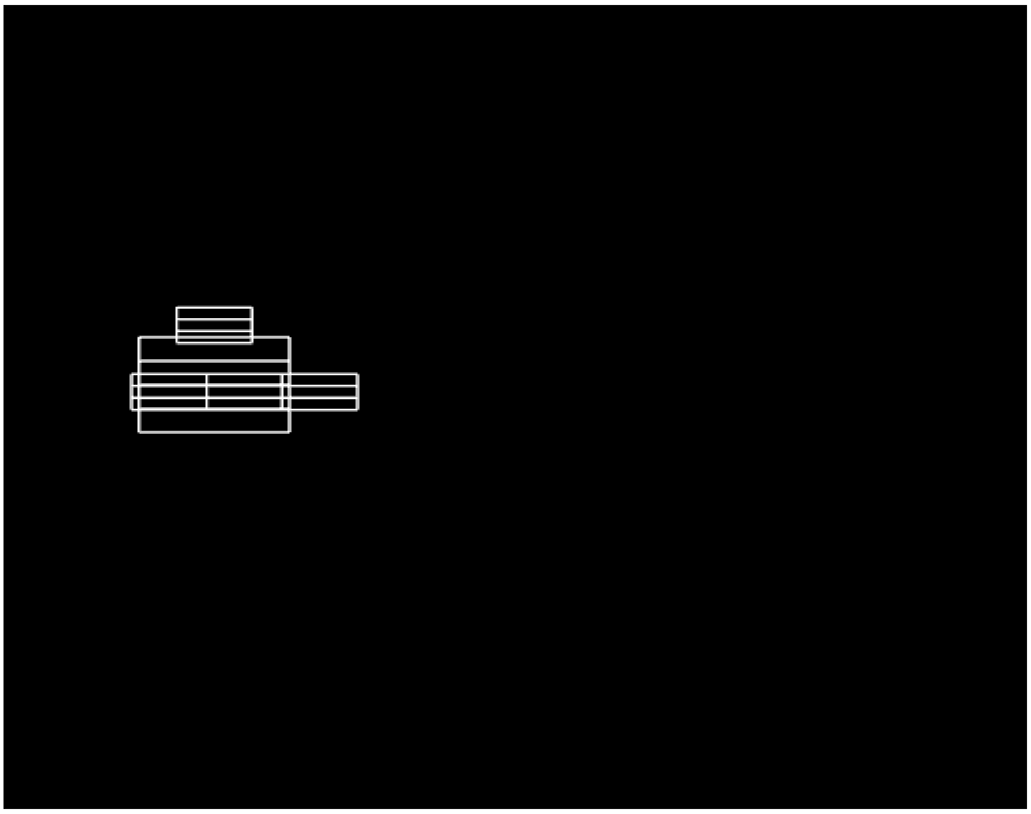


Figure 2. y/Y: to rotate the arm on the Y axis so you can see it from different angles

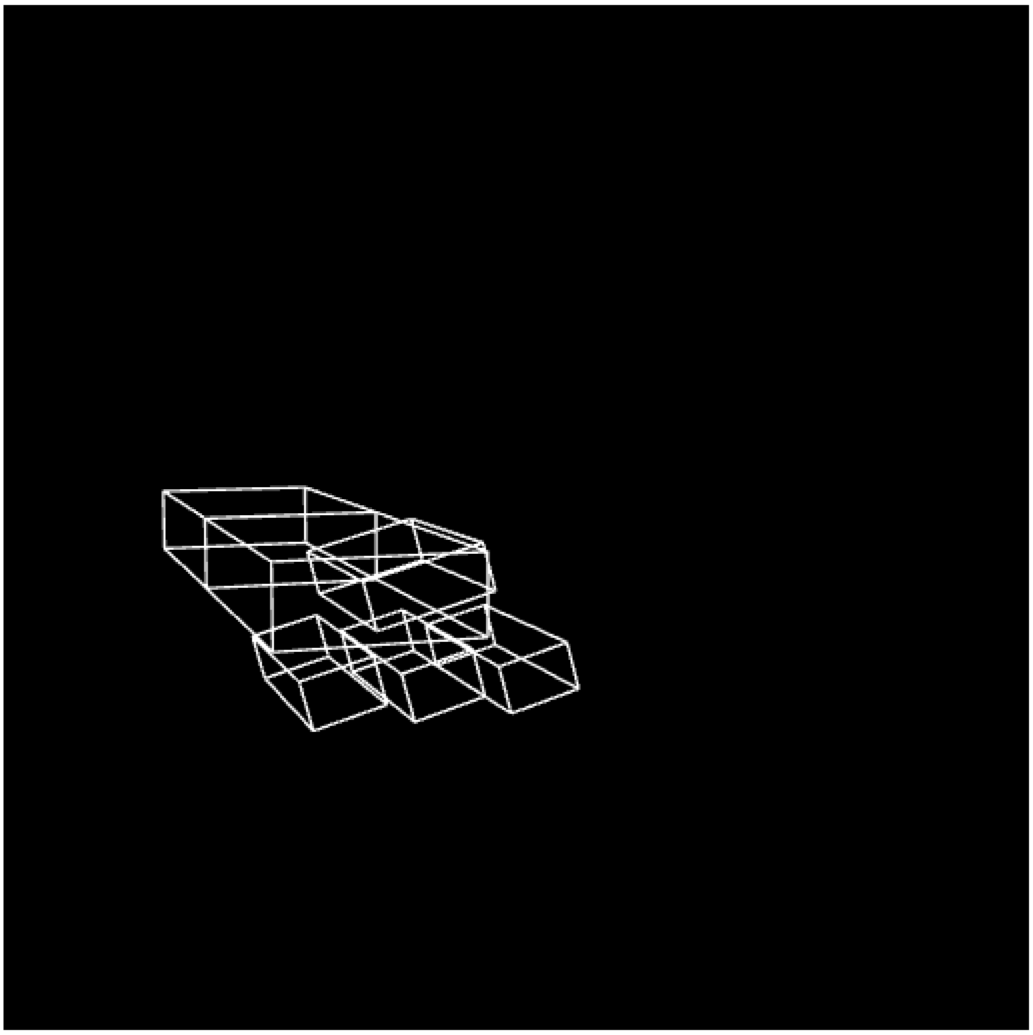


Figure 3 a/A: to rotate the fingers on the x-axis with positive direction

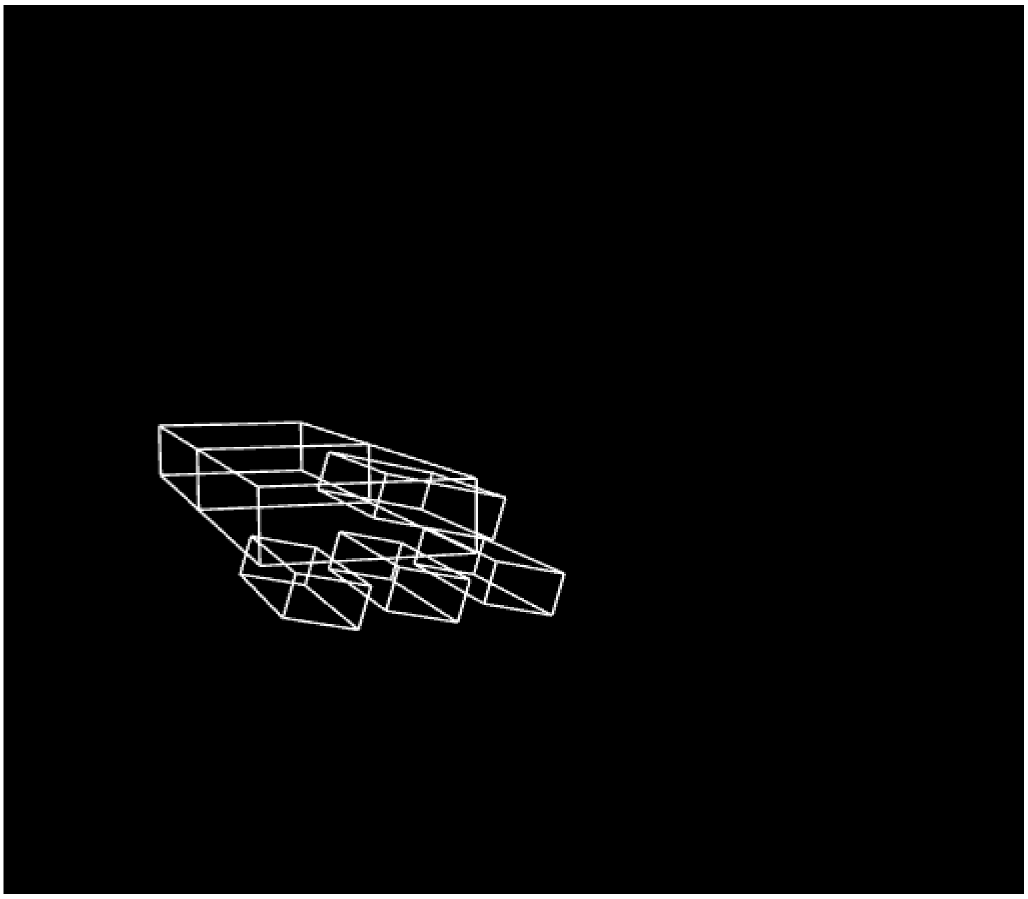


Figure 4 b/B: to rotate the fingers on the x-axis with negative direction

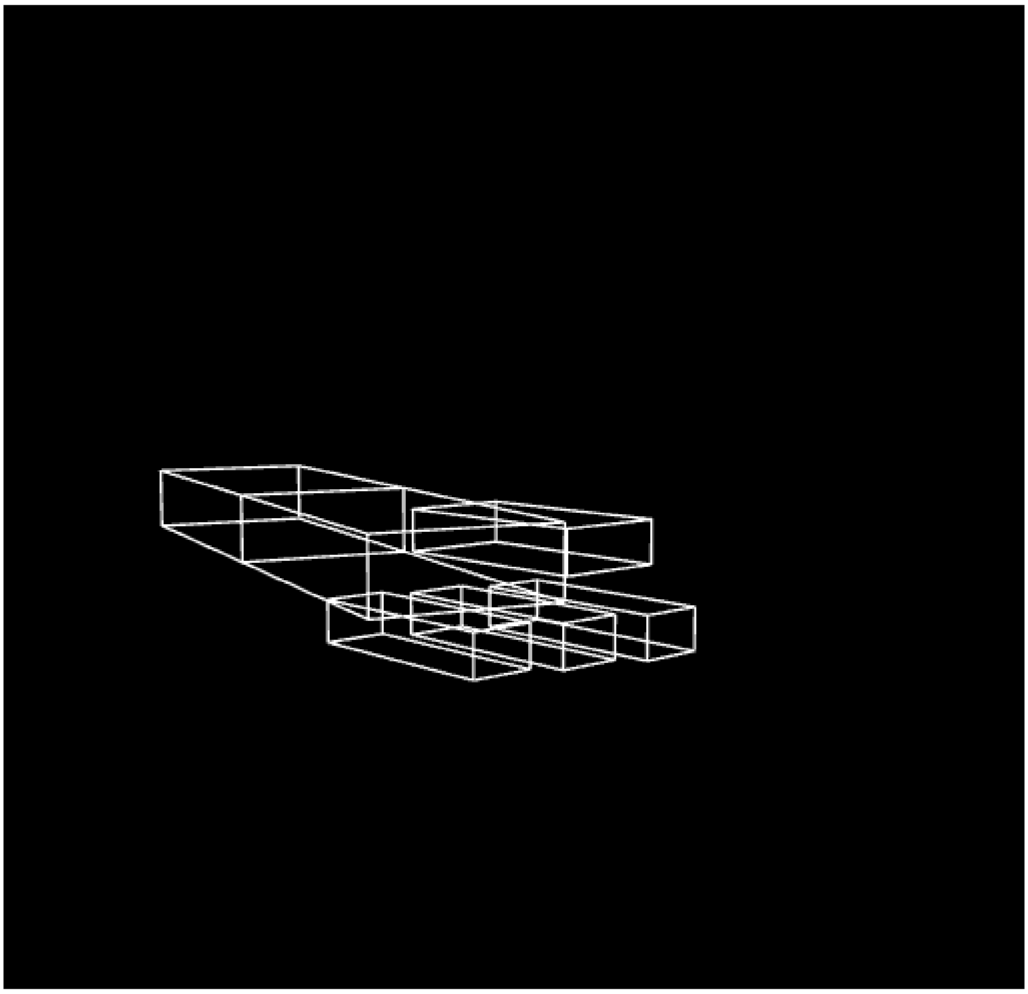


Figure 5 m/M: to rotate the fingers on the y-axis with positive direction

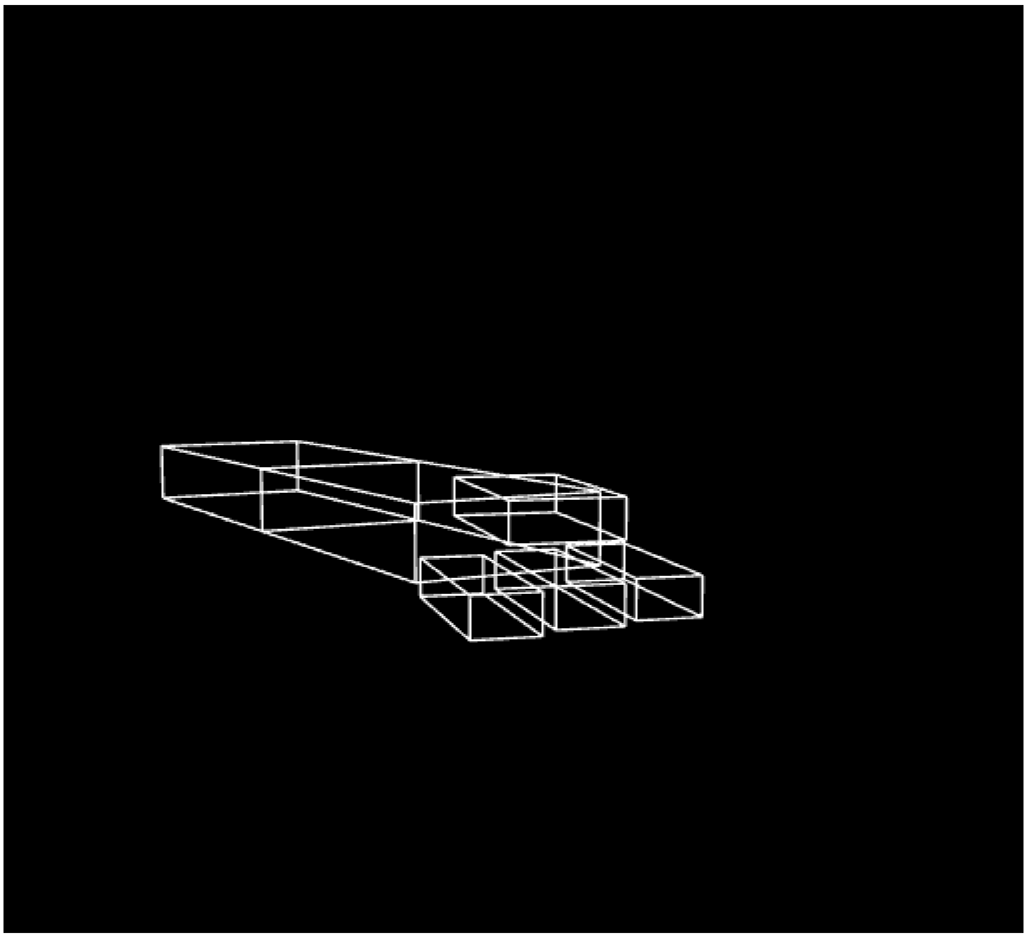


Figure 6 n/N: to rotate the fingers on the y-axis with negative direction

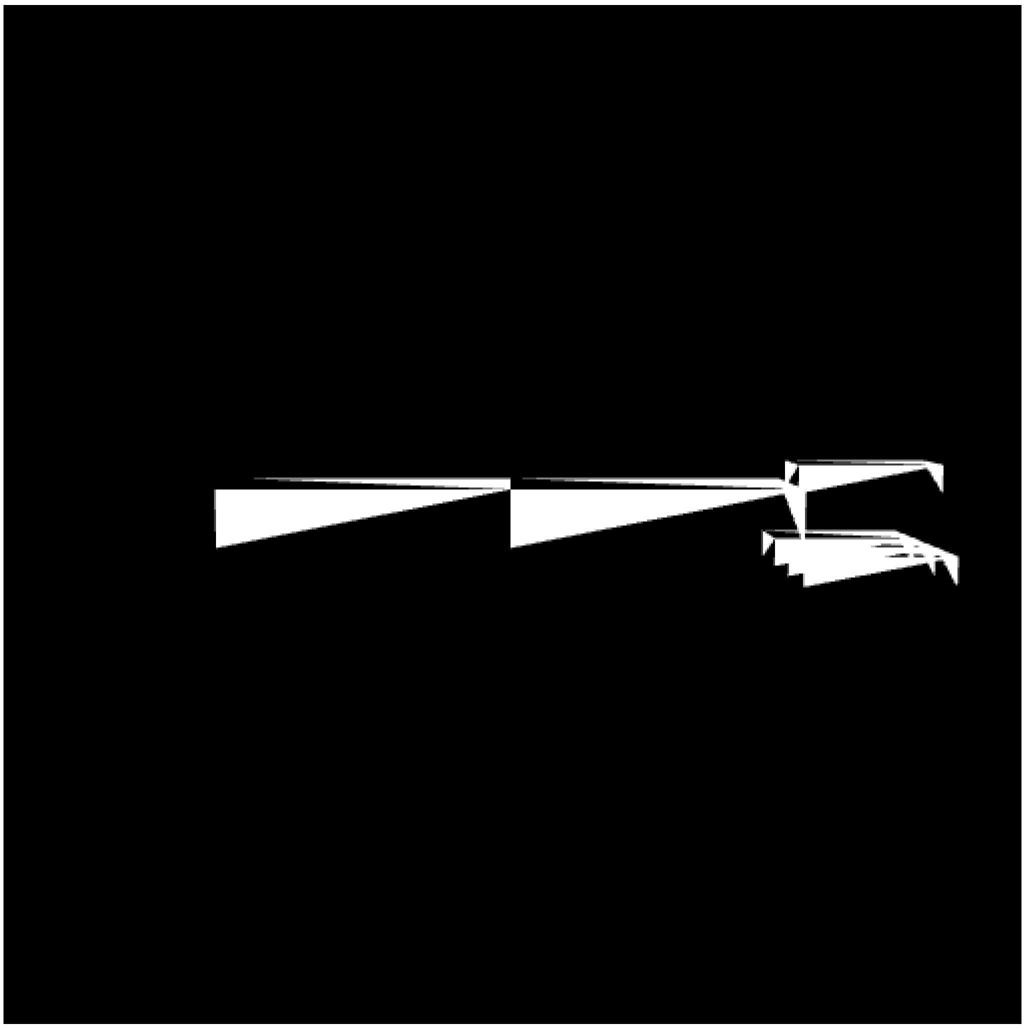


Figure7. t/T: toggle between solid and wire cubes

p/P: toggle between perspective and ortho projections

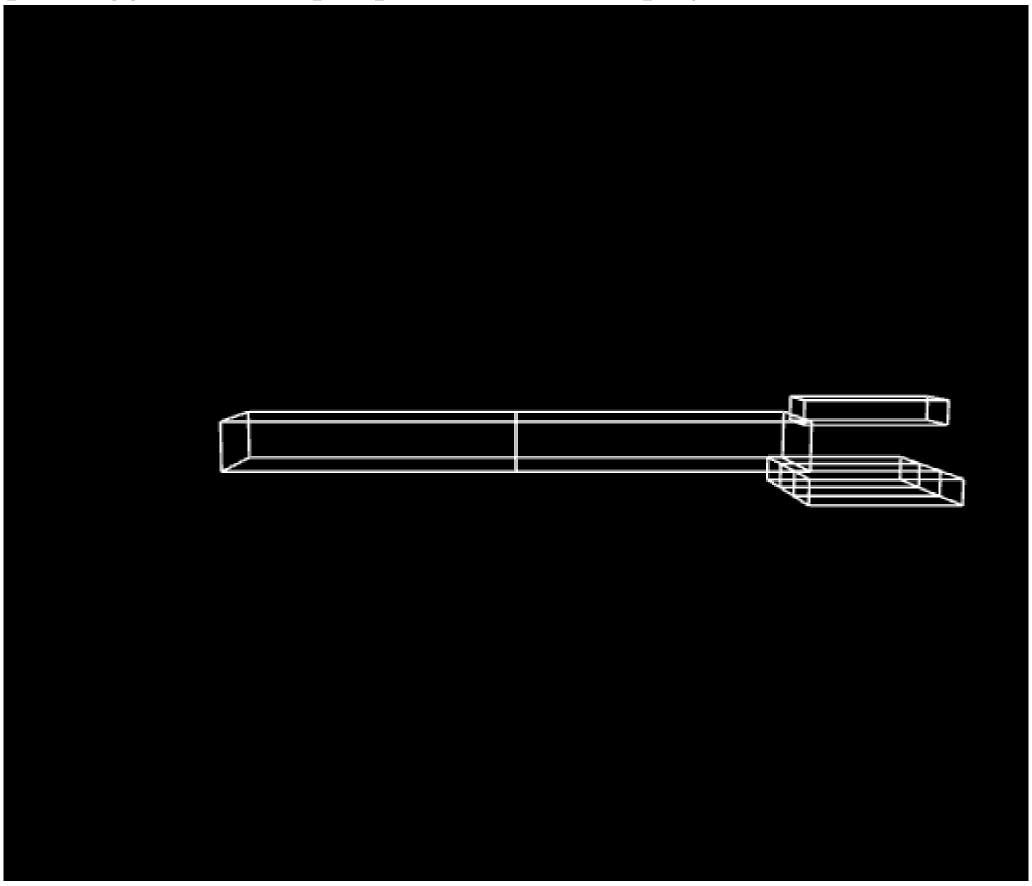


Figure8. Perspective

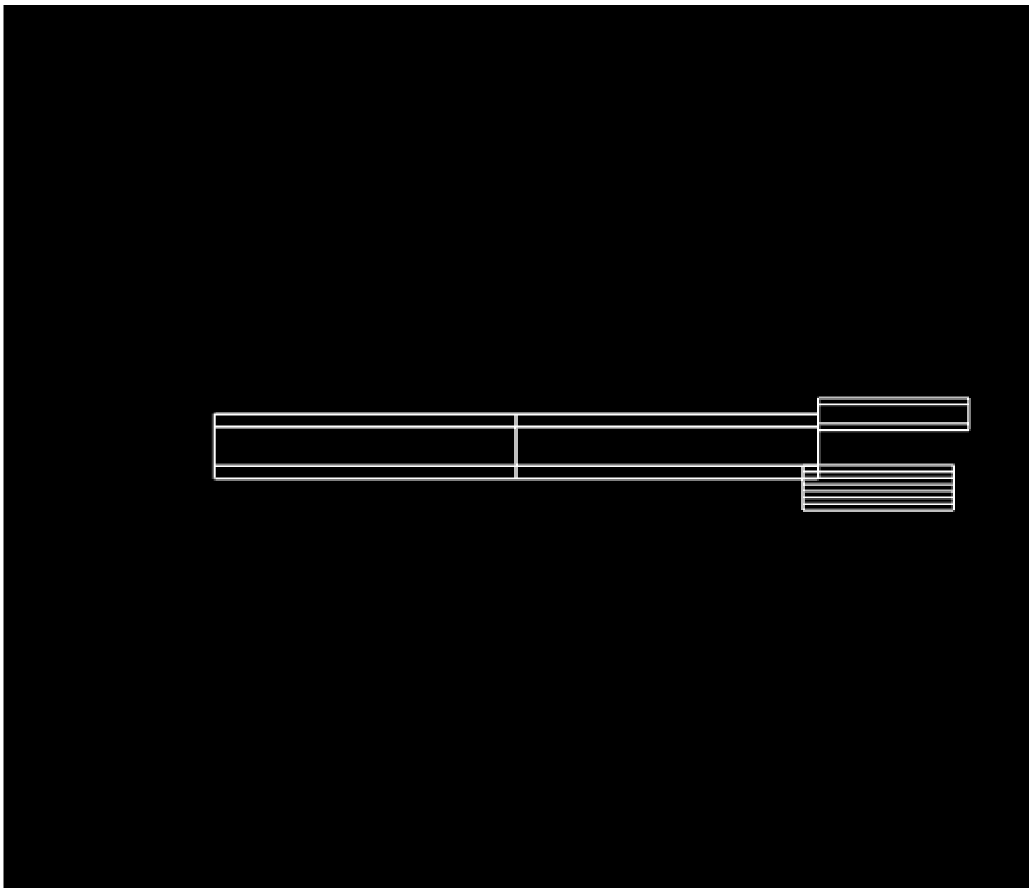


Figure9. Orthogonel

Conclusion:

The fundamental functionality of a WebGL-based robot arm simulation is provided by the JavaScript code block. It renders the robot arm components, manages user input for joint movement, specifies forms, and sets up the environment. An interactive and dynamic simulation is made possible by the main event handling and hierarchical modifications.

The render function considers the component hierarchy when coordinating the rendering of the robot arm. It draws every segment according to the given geometry, efficiently applies transformations, and maintains the matrix stack. This function is essential for the movement and structure of the robot arm to be visually represented in the WebGL environment.

Drive link: https://drive.google.com/file/d/1w7adXX6gh_4oV_Tb_aaxi2-LYOYE5MvY/view?usp=sharing