Final Project

Gioele Migno - 1795826

01 September 2021

Contents

1 Introduction **2**
1.1 Topic . . . 2
1.2 Project Requirements . . . 2
1.3 Hierarchical Model . . . 3

2 Libraries Used **3**

3 Libraries Developed **3**
3.1 X_Bot.js . . . 3
3.2 Tween_spline.js . . . 4
3.3 Linear_animation.js . . . 5
3.4 X_Bot_Animation.js . . . 5
3.5 X_bot_Interpolation_Configurations.js . . . 6

4 Environment **6**
4.1 3D Model . . . 6
4.2 Lights . . . 7
4.3 Materials . . . 7
4.4 Textures . . . 7
4.4.1 Company Poster . . . 7
4.4.2 Red Carpet . . . 7
4.5 Walkable Space . . . 8
4.6 Environment.js . . . 9

5 Animations **9**
5.1 Red Ball Animation . . . 9
5.1.1 X_Bot_Sphere_Animation.js . . . 11
5.2 Walk . . . 12
5.2.1 X_bot_Walk_Animation.js . . . 12
5.2.2 X_bot_Walk.js . . . 12
5.3 Blender Pose Library Exporter . . . 16

6 User Interaction **18**
6.1 Walk . . . 18
6.2 Change Color . . . 18
6.3 Change View . . . 18

7 Browser Compatibility **18**

1 Introduction

1.1 Topic

The topic of the project is a 3D representation of a robotics company showroom in which several robots are shown. In the scene there are four stationary robots in T configuration and two animated ones. One of them can be moved by the user (customer) using the keyboard directional arrows, other one instead performs a fixed animation in which the robot moves a red ball between two fixed bases placed on the table.

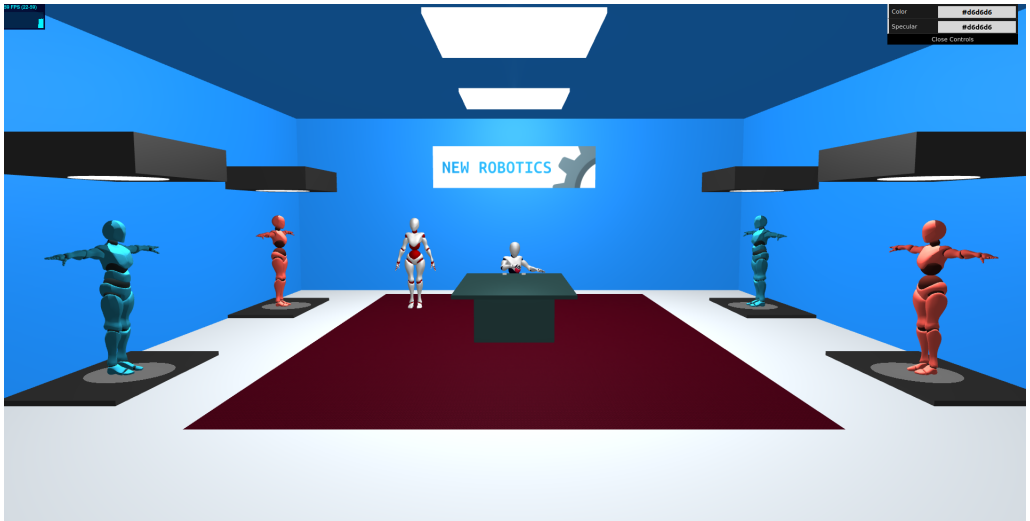


Figure 1: Initial screen view

1.2 Project Requirements

All project requirements has been satisfied:

- **Complex Hierarchical Model:** The two animated robots use the same hierarchical model (X Bot by Mixamo), composed by 53 parts.
- **Lights and Texture:** In the scene there are three different types of lights: one AmbientLight, two PointLight and four SpotLight. Moreover, as required, there are two types of texture: a simply color texture used to display the company poster and a more complex texture that uses color, normal, displacement, roughness and ambient occlusion maps to represent the red carpet.
- **User Interaction:** User can interact with the scene changing the view using mouse, moving the white standing robot using keyboard, and changing the color of the walking robot using the GUI at top right corner of the screen.
- **Animations:** As required, the two animations are implemented in JavaScript and are not imported, moreover, they exploit the robot structure, walk animation exploits all the body expect the hands that are excited by the second one.

1.3 Hierarchical Model

The hierarchical model animated is taken from Mixamo website and it is called X Bot. It is made of 53 parts, the hands are composed of all human joints: pinky, ring, middle, index and thumb, each of them is composed by four parts. (e.g. Thumb_1, Thumb_2, Thumb_3, Thumb_4). Fig. 2 shows the parts of the model.

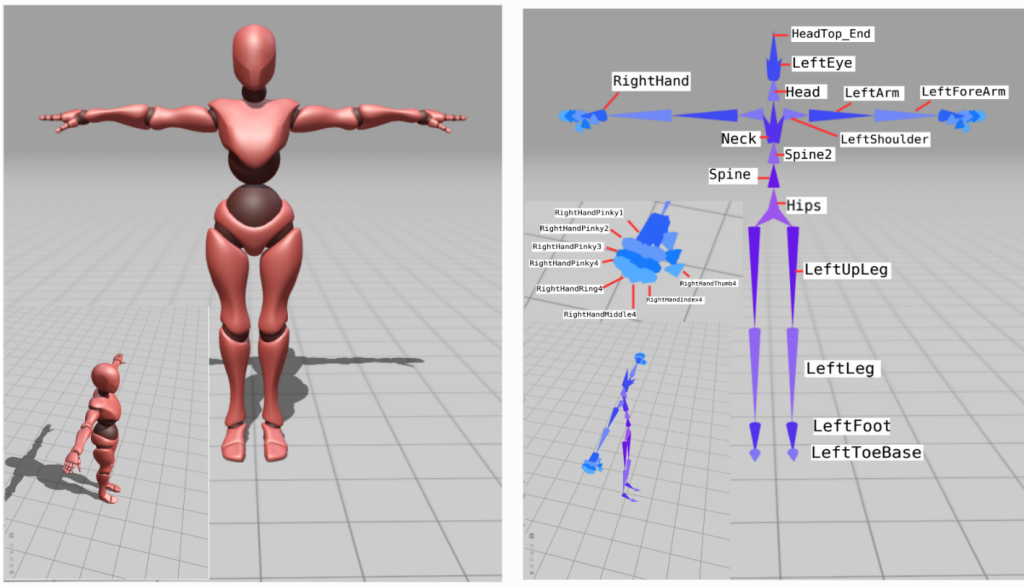


Figure 2: X Bot

2 Libraries Used

The project has been implemented using Three.js 2021 library, alongside, the following modules/libraries have been used:

- **dat.gui.module.js:** Module used to build the GUI that allows to change color of the robot.
- **OrbitControls.js:** Module used to move the camera using mouse.
- **stats.module.js:** Module used to show current frames per second (FPS).
- **GLTFLoader.js:** Module used to load X Bot model that has been converted from .fbx to .gltf format using Blender.
- **Tween.js:** Library used to implement smooth animation in JavaScript.

3 Libraries Developed

3.1 X_Bot.js

In order to easily manage X Bot model, X_Bot.js has been developed, it allows to access to each parts, set a configuration and get the current configuration. Let's see the methods available for a X_Bot object:

```
async load(model_path); // Load model
set_color(surface_color, joints_color); //set colors
set_configuration(config); // set configuration config
get_current_configuration(); // get current configuration
set_rest_configuration(); // set rest configuration
```

load() function is *async* because uses loadAsync() of GLTF loader, in this way the model is immediately loaded and it is ready to be initialized by _init() function called in load(). _init() is a private method used to associate proprieties of the object to the real parts of the model, for example:

```
this.parts.hips = this.model.getObjectByName('mixamorigHips');
```

set_color() function allows to set materials of joints and surface of the robot, it is uses a MeshPhongMaterial type (based on Phong model) and set diffuse and specular components:

```
this.materials.surface.material = new THREE.MeshPhongMaterial({color: surface_color,
                                                               specular:surface_color,
                                                               shininess:5});
this.materials.joints.material = new THREE.MeshPhongMaterial({color: joints_color,
                                                               specular:joints_color,
                                                               shininess:5});
```

set_confguration() and get_configuration() work both with the following type of structure. Hips is the only part that needs also position, other ones are specified only by the orientation defined using quaternion (XYZW) representation.

```
let configuration = {
    // HIPS -------------------------------------------------------
    hips_position: {x: 0.33, y: 105, z:0},
    hips_quaternion: {x: 0.0, y: 0.0, z:0.0, w: 1.0},
    //-----------------------------------------------------------

    // SPINES -----------------------------------------------------
    spine_quaternion: {x: 0.0, y: 0.0, z:0.0, w: 1.0},
    spine1_quaternion: {x: 0.0, y: 0.0, z:0.0, w: 1.0},
    spine2_quaternion: {x: 0.0, y: 0.0, z:0.0, w: 1.0},
    // ----------------------------------------------------------
    [...]
```

3.2 Tween_spline.js

Tween.js library does not allow to easily make an animation based on a spline, namely define an animation using two array of times and values, indeed Tween.js supports only the following definition method:

```
let position = {x: 100, y: 0}
// Create a tween for position first
let tween = new TWEEN.Tween(position)

// Then tell the tween we want to animate the x and y properties over 1000 milliseconds
tween.to({x: 200, y:100}, 1000)
```

For this reason, Tween_spline.js library has been developed. It is allow to easily define an animation as a spline, and to repeat or not it in loop. It implements all controlling methods of Tween.js (start(), upload(), stop()) and in addition reset() that is not available.
An example of Tween_spline object definition is the following:

```
let position = {x: 100, y: 0};

// Function used to upload position
let upload_func = (object) => {
    position.x = object.x;
    position.y = object.y;
}

let values = [{x: 100, y: 0}, {x: 200, y: 10}, {x: 400, y: 30}];
let times = [0, 1000, 2000];
```

```
let loop = false;
let tween = new Tween_spline(times, values, upload_func,
                                TWEEN.Easing.Quadratic.None,
                                TWEEN.Interpolation.CatmullRom);
tween.init(loop);
```

Tween_spline.js defines a tween for each couple of values, (in the previous example it defines two tweens). In order to make all animations independent from each others, a tween_spline object collects its tweens as a tween group, in this way all controlling methods do not affect other tween objects.

3.3 Linear_animation.js

During walk animation is necessary to move the X_Bot model, to do that cannot be used Tween.js library since the next position depends by the user, so Linear_animation.js module has been developed, it allows to animate an object by moving it linearly respect to time. Let's see its interface:

```
// Necessary to define angular coefficient of the linear function and the object to control
init(angular_coefficient, object_to_control, name_component_to_control);

// Controlling functions
    start();    stop();     restart();      reset();        update();
```

This library updates the value of the variable to control using incremental steps, in this way is not necessary to specify the initial position of the object (i.e. y=f(t) + q with q=0). The code of update() function is the following:

```
update(){
    this.elapsed = this.clock.getElapsedTime();
    const y = this.angular_coefficient * this.elapsed - this.distance_traveled;
    this.distance_traveled += y;
    this.object_to_control[this.name_component_to_control] += y;
}
```

3.4 X_Bot_Animation.js

X_Bot_Animation.js is a module developed to easily implement an animation for a X_Bot model, walk animation and the animation of the robot sitting at the table, have been built using this module. This is its interface:

```
constructor X_Bot_Animation(_x_bot);
init(array_animation, loop); // init requires animation and a boolean flag 'loop'
start();    update();   reset(); // controlling functions
```

init() function requires as input the following structure:

```
    let array_animation = {
        hips_position: {
            enable: false,
            func: null,
            tween: null,
            animation: {
                times: [0, 500.0, 1000, 1500.0, 2000],
                values: [{x: 2.38, y: 99.44},
                    {x: 1.28, y: 95.57},
                    {x: 0.33, y: 99.54},
                    {x: 1.72, y: 96.49},
                    {x: 2.38, y: 99.44}]
            }
        },
```

```
        hips_quaternion: {
            enable: false,
            func: null,
            tween: null,
            animation: {
                times: [0, 500.0, 1000, 1500.0, 2000],
                values: [{x: -0.02, y: -0.03, z:0.04, w: 0.99},
                    {x: -0.01, y: 0.05, z:-0.01, w: 0.99},
                    {x: -0.01, y: 0.01, z:-0.02, w: 0.99},
                    {x: -0.01, y: -0.04, z:0.1, w: 0.99},
                    {x: -0.02, y: -0.03, z:0.03, w: 0.99}]
            }
        },
        [...]
```

enable, *func* and *tween* will be initialized by init(). *enable* field will be set to true if *times* and *values* arrays are not empty, *tween* field will contain the Tween_spline object used and *func*, its update function. The default tween configuration uses quadratic easing function and Catmull–Rom interpolation function.

3.5 X_bot_Interpolation_Configurations.js

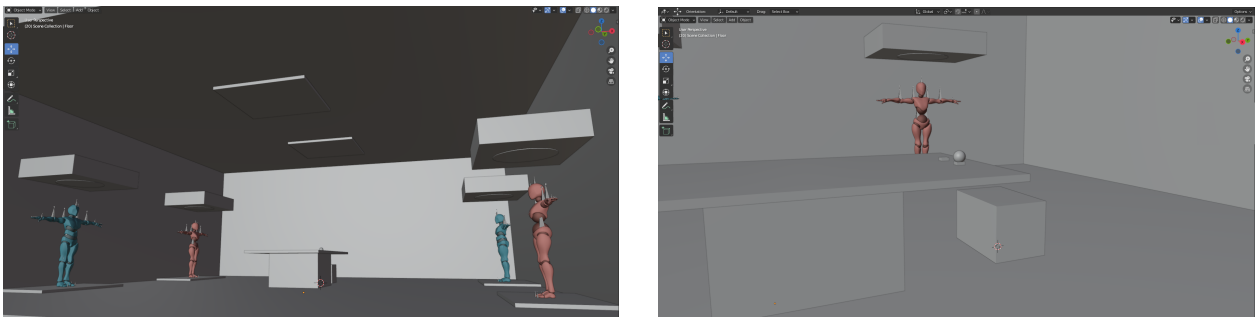
During walk animation, when user stops robot walk it is necessary to go to rest configuration starting from the current one, to do that X_bot_Interpolation_Configurations.js module has been developed. It allows to build an animation starting from two X_Bot configurations (start_config and end_config) or using an array of configurations and its relative array of times (array_config, array_times). This module is naturally based on X_Bot_Animation.js module.

```
constructor X_bot_Interpolation_Configurations(_x_bot);

// build animation starting from two configurations
init(start_config, end_config, time_interval);

// build animation using array of configuration (Tween_spline module style)
init_v2(array_config, array_time, loop);

// wrappers of X_Bot_Animation.js controlling functions
start();    update();   reset();
```

4 Environment

4.1 3D Model

The 3D model of the showroom has been designed using Blender 2.93.



(a) (b)

Figure 3: 3D Modelling in Blender

4.2 Lights

In the room there are six lights, on the ceiling two lights are represented as squares, then there is a circular light on each robot platform. In Three.js to represent these lights a MeshPhongMaterial material type with emissive component has been used. Lights on the ceiling use PointLight, those on platform instead SpotLight, in the scene there is also an AmbientLight light. In Fig.4 lights are clearly visible.

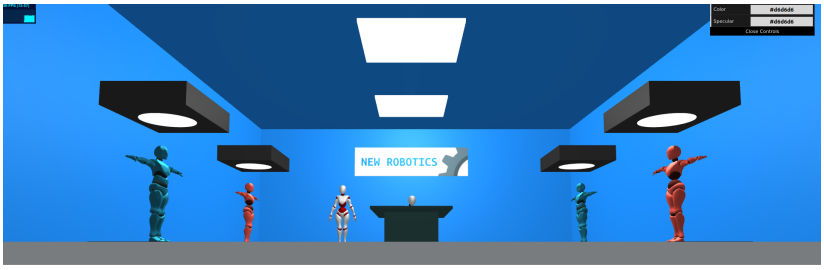


Figure 4: Lights in the showroom

4.3 Materials

All objects in the scene, expect the red carpet, use MeshPhongMaterial material with diffuse and specular components. Red carpet uses MeshStandardMaterial in order to use a PBR texture.

4.4 Textures

4.4.1 Company Poster

The poster on the front wall contains the name of the fictitious robotics company "New Robotics", it is built using a BoxGeometry geometry type and applying a color texture to the visible face.

```
const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);
const materials = [
    new THREE.MeshPhongMaterial({color: 0xF0F8FF}),
    new THREE.MeshPhongMaterial({color: 0xF0F8FF}),
    new THREE.MeshPhongMaterial({color: 0xF0F8FF}),
    new THREE.MeshPhongMaterial({color: 0xF0F8FF}),
    new THREE.MeshPhongMaterial({map: loader.load('./textures/logo.png')}), // visible face
    new THREE.MeshPhongMaterial({color: 0xF0F8FF}),
];
const poster = new THREE.Mesh(geometry, materials);
```

4.4.2 Red Carpet

The red carpet uses a more energy expensive material called MeshStandardMaterial that supports Physically Based Rendering (PRB), the texture applied contains: color, normal, displacement, roughness and ambient occlusion map, all of them shown in the following. (CREDITS: https://3dtextures.me/2021/01/25/fabric-knitted-004/)

(a) (b) (c) (d) (e)

Figure 5: In order: color, normal, displacement, roughness and ambient occlusion map.

The carpet is built using a PlaneGeometry geometry type, its definition is the following:

```
const geometry =  new THREE.PlaneGeometry(600, 900);
const materials = new THREE.MeshStandardMaterial({
        map: base_color_map,
        normalMap: normal_map,
        displacementMap: height_map,
        roughnessMap: roughness_map,
        aoMap: ambent_occlusion
    });
const carpet = new THREE.Mesh(geometry, materials);
carpet.rotation.x = -Math.PI/2; // horizontal plane
```

All maps are loaded using the classic TextureLoader, and setting repeat wrapping for three times in both texture direction S, T. For example:

```
[...]
const loader = new THREE.TextureLoader();
const base_color_map = loader.load('./textures/carpet/Fabric_Knitted_004_basecolor.jpg')
this._set_repeat_texture(base_color_map)
[...]
 _set_repeat_texture(texture){
    texture.wrapS = THREE.RepeatWrapping;
    texture.wrapT = THREE.RepeatWrapping;
    texture.repeat.set(3,3);
}
```

The result is shown in Fig.6

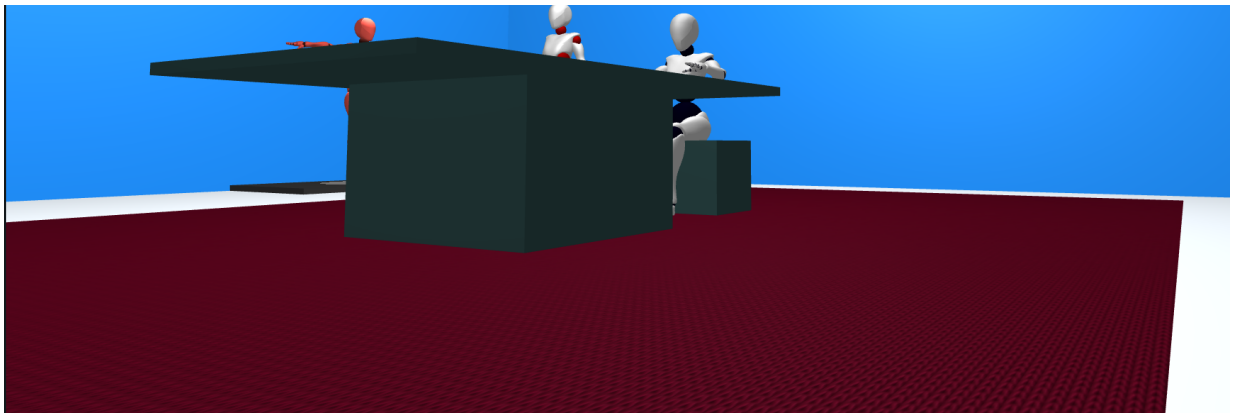
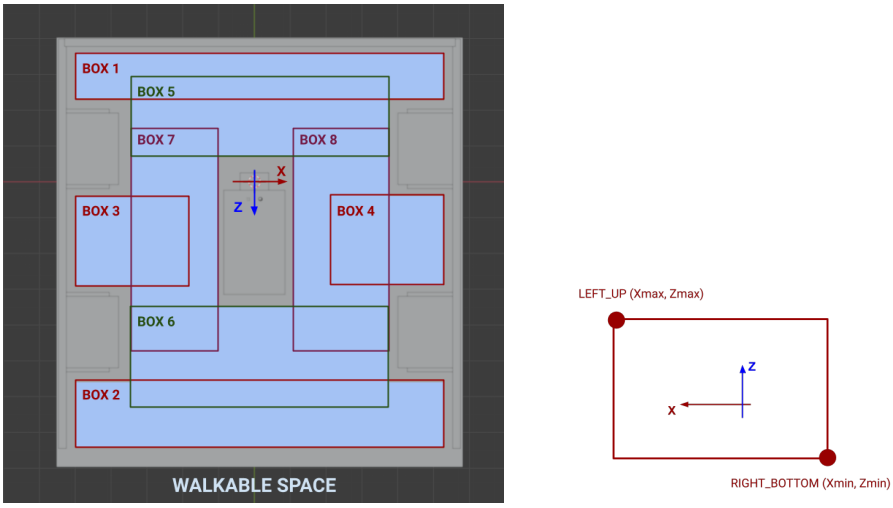


Figure 6: Carpet texture

4.5 Walkable Space

In order to avoid obstacles during walk, it is necessary to define the walkable space, to do that this space has been divided into eight boxes. The walkable area is shown in Fig.7a, instead, Fig.7b shows how a box is defined and in the following there is an example of Box1 definition:

```
let box_1 = walk_animation.get_box_template(); // get blank box structure
// coordinates in the world reference
box_1.right_bottom.x = -469;
box_1.right_bottom.z = -318;

box_1.left_up.x = 489;
box_1.left_up.z = -244;
```



(a) (b)

Figure 7: Walkable space (a) and definition of a box (b)

4.6 Environment.js

Environment.js is the module that deals with the environment and takes care of initialize lights, texture and all materials in the scene. This is its interface, init() function requires the main scene and the walk_animation object used (we will see this later).

```
constructor Environment();
async load(model_path); // similar to X_Bot.js
init(main_scene, walk_animation);
```

init() function is composed by several steps:

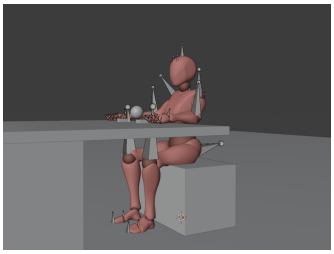
```
init(main_scene, walk_animation){
    this._init_parts(); // setup materials and init parts like X_Bot.js
    this._init_lights(main_scene); //add light to the main scene
    this._init_poster(main_scene); // init poster texture
    this._init_carpet(main_scene); // init carpet texture
    this._setup_walk_space(walk_animation); // init walkable space
    main_scene.add(this.scene); // add model loaded to the main scene
}
```

5 Animations

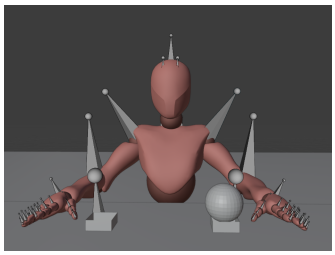
An animation is composed by the interpolation of several poses, each pose has been designed using Blender Pose Mode that allows to easily modify the configuration of an armature. Then the poses have been exported using a specifically designed Python script that we will see later.

5.1 Red Ball Animation

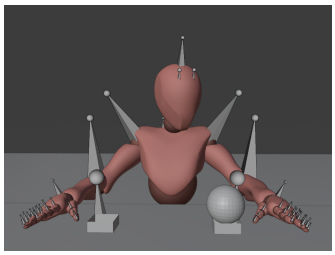
Let's see first the animation of the robot that move the red ball on the table. Red ball animation is composed by 28 keyframes shown in Fig.8 and Fig.9, the first represents the keyframes necessary to move the ball from robot left to its right, the latter instead, returns the ball to the starting point by moving it from right to left, note that this is a cyclic animation, indeed the first keyframe is equals to the last one. During animation, legs are in a fixed pose shown in Fig.8a.



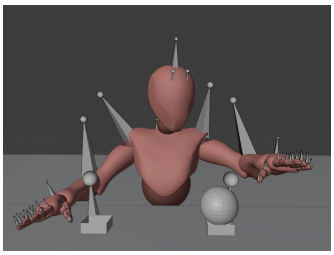
(a) Fixed legs pose



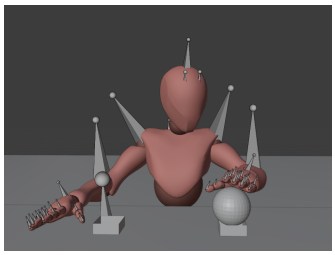
(b) START



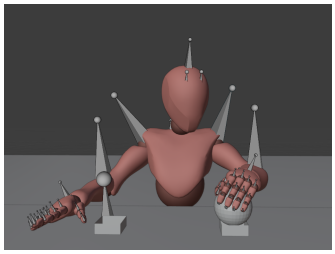
(c) LOOK_LEFT



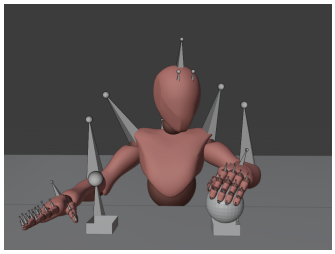
(d) UP_LEFT



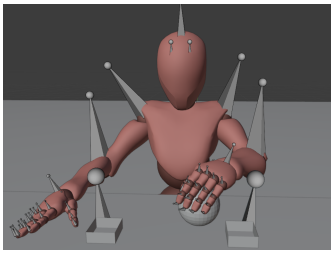
(e) UP_LEFT_S



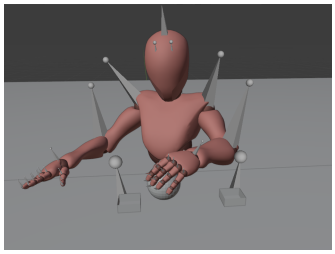
(f) UP_LEFT_GRAB



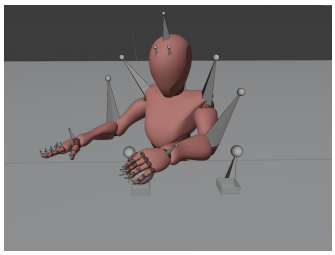
(g) UP_LEFT_GRAB_UP



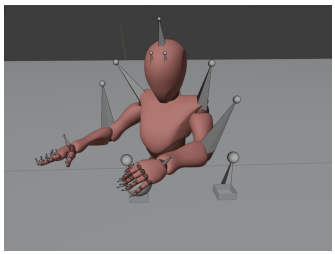
(h) UP_LEFT_TO_DX_1



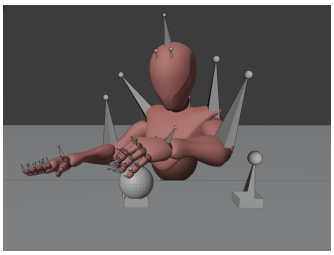
(i) UP_LEFT_TO_DX_2



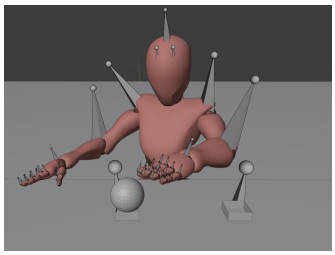
(j) UP_LEFT_TO_DX_3



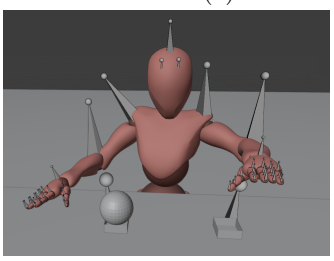
(k) UP_LEFT_TO_DX_4



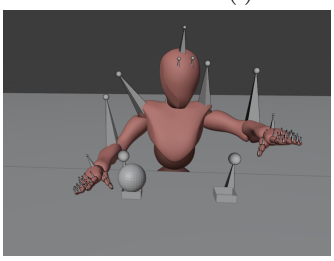
(l) BACK_TO_SX_1



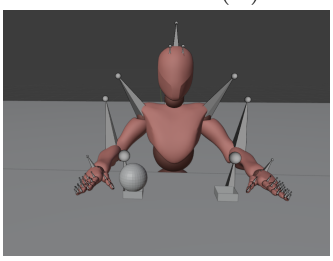
(m) BACK_TO_SX_2



(n) BACK_TO_SX_3

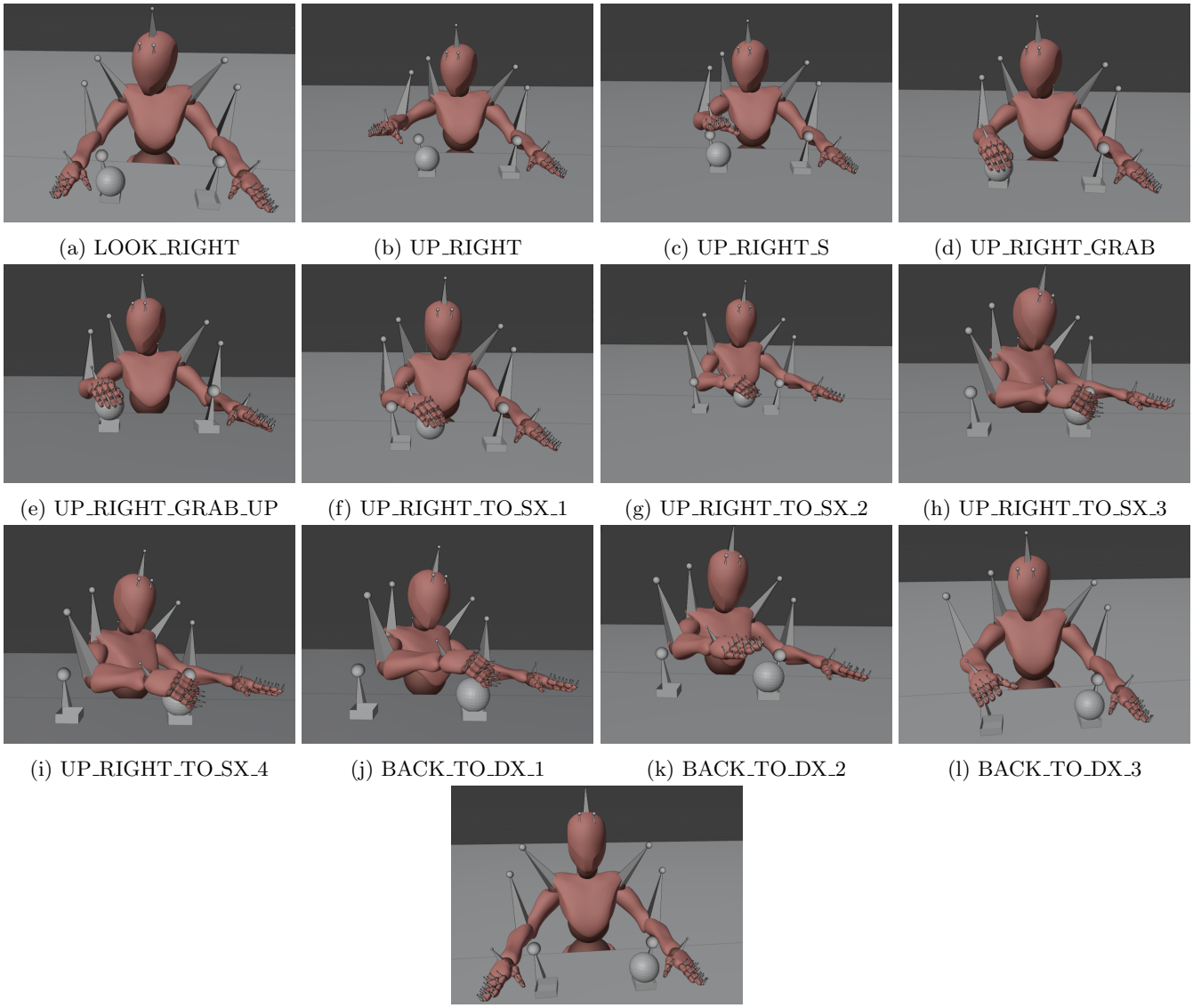


(o) BACK_TO_SX_4



(p) BACK_TO_SX_5

Figure 8: Red ball animation Part 1



(a) LOOK_RIGHT (b) UP_RIGHT (c) UP_RIGHT_S (d) UP_RIGHT_GRAB

(e) UP_RIGHT_GRAB_UP (f) UP_RIGHT_TO_SX_1 (g) UP_RIGHT_TO_SX_2 (h) UP_RIGHT_TO_SX_3

(i) UP_RIGHT_TO_SX_4 (j) BACK_TO_DX_1 (k) BACK_TO_DX_2 (l) BACK_TO_DX_3

(m) BACK_TO_DX_4

Figure 9: Red ball animation Part 2

5.1.1 X_Bot_Sphere_Animation.js

Red ball animation is manage by X_Bot_Sphere_Animation.js module that uses Tween_spline.js to move the sphere and X_bot_Interpolation_Configuartions.js to interpolate previous robot's poses. Module interface is the following:

```
constructor X_Bot_Sphere_Animation(x_bot, env);
init();
// Controlling functions
start();    stop();    restart();    reset();    update();
```

Object constructor required an X_Bot object and an Environment one, this last is necessary to access to the position of the red ball that will be moved during animation. Thanks to the developed libraries, init() function is very simply:

```
init(){
    // init x_bot animation
    this._init_array(); // setup array of configurations
    this.animation = new X_bot_Interpolation_Configurations(this.x_bot);
    let times = [];
    let step= 500; // interval between keyframes
    for(let i = 0; i<28; i++){
        times.push(i*step); // build times array
    }
    this.animation.init_v2(this.array_config, time, true);

    // init animation of the sphere
    this.func = (object) => {
        this.env.object_to_move.position.set(object.x, object.y, object.z);
    }
    this.object_pos = [{x:0.16857, z:0.48385, y:0.9035},
                       {x:0.16857, z:0.48385, y:0.9035},
                        [...] ];
    this.obj_tween = new Tween_spline(times, this.object_pos, this.func,
                                      TWEEN.Easing.Quadratic.None,
                                      TWEEN.Interpolation.CatmullRom);
    this.obj_tween.init(true);
}
```

Sphere animation uses the same tween configuration of X_bot_Interpolation_Configurations, namely, quadratic easing function and Catmull–Rom interpolation function.

5.2 Walk

X_bot_Walk.js is the module that deals with the walk animation and the user interaction (go ahead, turn left, turn right), the first task is delegated to another module called X_bot_Walk_Animation.js, let's see first at all this last one.

5.2.1 X_bot_Walk_Animation.js

Like the previous animation, also the walk one is based on X_Bot_Animation.js module and supports the usual interface: init(), start(), update(), reset().

In Fig.10 are shown X_Bot poses that compose the cyclic walk animation, these keyframes take inspiration from Mixamo Walk animation.

5.2.2 X_bot_Walk.js

The module interface is the following:

```
constructor X_bot_Walk(x_bot);
init(array_boxes); // array_boxes represents the walkable space
update();
keydown_dispatcher(key_name); // key pressed handler
keyup_dispatcher(key_name); // key released handler
```

init() function requires as input an array of boxes as described in Environment section. Keydown_dispatcher is registered in the following way, keyup_dispatcher in a similar manner.

```
document.addEventListener('keydown', (event) => {
    var name = event.key;
    var code = event.code;
    walk_animation.keydown_dispatcher(name);
}, false)
```

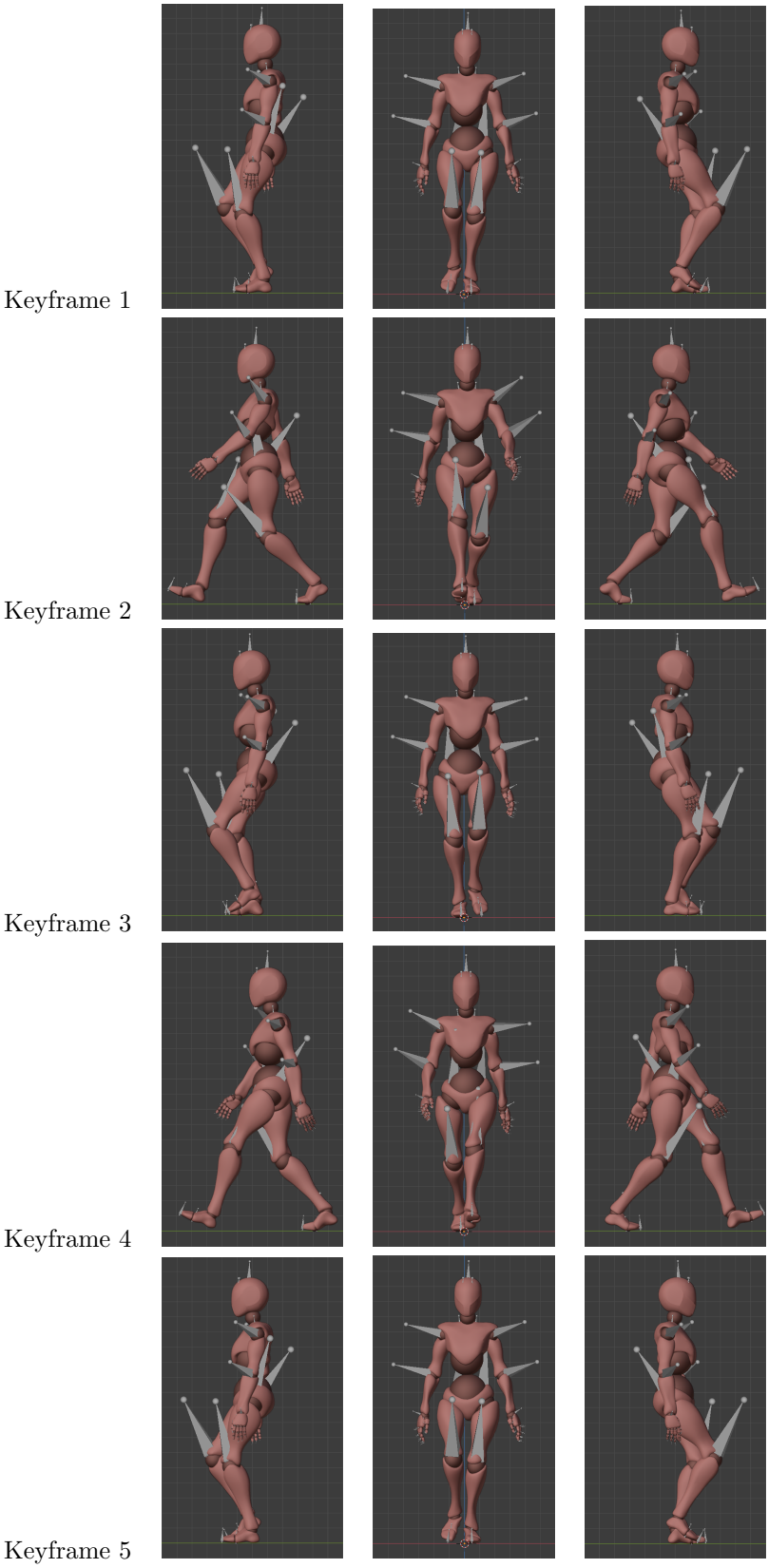


Figure 10: Walk keyframes

Let's now see the implementation of X_Bot_Walk constructor and init() function;

```
constructor(x_bot){
    [ ... ]
    this.move_x_bot_started = {z_pos:false, z_neg:false, x_pos:false, x_neg:false};
    this.move_x_bot_running = {z_pos:false, z_neg:false, x_pos:false, x_neg:false};
    this.move_x_bot_linear = {z_pos: new Linear_animation(),
                              z_neg: new Linear_animation(),
                              x_pos: new Linear_animation(),
                              x_neg: new Linear_animation()};

    this.walking = false; // if true, walk animation is running
    this.walk_animation = new X_Bot_Walk_Animation(this.x_bot);
    [ ... ]
    this.hit_obstacle = false; // if true, robot cannot go ahead
}
```

As we can see from the constructor, in order to move the robot on the plane XZ, X_Bot_Walk uses four different Linear_Animation objects one for each possible axis (x/z) and direction (pos/neg). In the init() function, these objects are initialized in this way:

```
this.move_x_bot_linear.z_pos.init(speed, this.x_bot.parts.armature.position, 'z');
this.move_x_bot_linear.z_neg.init(-speed, this.x_bot.parts.armature.position, 'z');
this.move_x_bot_linear.x_pos.init(speed, this.x_bot.parts.armature.position, 'x');
this.move_x_bot_linear.x_neg.init(-speed, this.x_bot.parts.armature.position, 'x');
```

To move forward the robot, user has to keep pressed UP_ARROW of the keyboard, and to turn left or right has to click LEFT_ARROW or RIGHT_ARROW respectively. When UP_ARROW is pressed, Walk_Animation uses the proper Linear_Animation object by looking to robot orientation, for example when its rotation respect to y-axis is equals to zero, its head looks at increasing z axis so, it uses *this.move_x_bot_linear.z_pos*.

Now, let's see briefly keydown and keyup dispatcher implementaions.

```
keydown_dispatcher(name){
    let rotation_y = this.x_bot.parts.armature.rotation.y;
    if(name == 'ArrowUp'){
        if(!this.hit_obstacle){
            this._start_linear(rotation_y);
            this.walking = true; // go to walking mode
        }
    }
    else if(name == 'ArrowLeft'){
        if(!this.left_key_pressing){
            this.left_key_pressing = true; // one shot event
            this._reset_linear(rotation_y);

            // change armature's orientation
            const tmp = this.x_bot.parts.armature.rotation.y + Math.PI/2;
            if(tmp > Math.PI){
                this.x_bot.parts.armature.rotation.y = -Math.PI/2;
            }
            else{
                this.x_bot.parts.armature.rotation.y = tmp;
            }
            [ ... ]
        }
    }
```

```
        else if(name == 'ArrowRight'){
            if(!this.right_key_pressing){
                this.right_key_pressing = true;
                this._reset_linear(rotation_y);
                // change armature's orientation
                [ ... ]
            }
        }
    }

    keyup_dispatcher(name){
        let rotation_y = this.x_bot.parts.armature.rotation.y;
        if(name == 'ArrowUp'){
            if(!this.hit_obstacle){
                this._reset_linear(rotation_y);
                this.walking = false;
                this.walk_animation_started = false;
                this.going_to_rest = true;  // going to rest
                this.going_to_rest_started = false;
                this.walk_animation.reset();
            }
        }
        else if(name == 'ArrowLeft'){
            this.left_key_pressing = false; // reset one shot event
        }
        else if(name == 'ArrowRight'){
            this.right_key_pressing = false; // reset one shot event
        }
    }
```

As we can see in the previous dispatchers implementations, When UP_ARROW is pressed the robot is in *walk* state, instead, when UP_ARROW button is released robot goes in a new state called *going_to_rest* in which its pose goes slowly from current configuration to the rest one, let's see how it works by looking at update() implementation.

```
    update(){
        if(this.walking){
            [...]
        }
        else if(this.going_to_rest){
            if(!this.going_to_rest_started){ // one shot event
                const current_config = this.x_bot.get_current_configuration();
                const rest_conf = this.x_bot.rest_configuration;
                // create on fly a new animation from current configuration to rest one
                // that takes 400 milliseconds
                this.going_to_rest_animation_walk_animation =
                                            new X_bot_Interpolation_Configurations(this.x_bot);
                this.going_to_rest_animation_walk_animation.init(current_config, rest_conf, 400);
                [...]
            }
            [...]
            this.going_to_rest_animation_walk_animation.update();
        }
        [...]
    }
```

When robot enters in *going_to_rest* status, an animation is built on fly in order to interpolate the current configuration obtainable by X_Bot.get_current_configuration() and the rest one.

In the following instead, the implementation of update() when the robot is in *walk* state.

```
if(this.walking){
    [...]
    let step_will = 10;
    if(this._will_go_outside(step_will, this.x_bot.parts.armature.position.x,
                                        this.x_bot.parts.armature.position.z,
                                        this.x_bot.parts.armature.rotation.y)){
        this.hit_obstacle = true;
        this.going_to_rest = true;
        [...]
    }
    else{
        this.walk_animation.update();
        this._move_forward(this.x_bot.parts.armature.rotation.y); // move armature
    }
}
```

In *walk* state, before updating walk animation and moving forward the robot, a check is performed through this._will_go_outside() that detects if the robot will go to collide with an obstacle (i.e. goes out of walkable space) or not, if this function returns true, then hit_obstacle flag is set to true and the state is set to *going_to_rest*.

_will_go_outside() essentially adds to the current position (x, z) the step given as input and checks if the new position (x+step, z+step) belongs to at least one box that defines the walkable area.

5.3 Blender Pose Library Exporter

Blender allows to use its Python API in scripting mode, to export easily the Pose Library that contains all poses of the X_Bot, the following function has been developed:

```
import bpy
import os
import json
##
def export_pose_library(folder, num_frames):
    def data_path_to_part_component(data_path):
        splitted = data_path.split('"')
        part = splitted[1]
        component = splitted[2].split("'")[0][2:]
        return (part, component)
    ##
    def add_value(dictionary, part, component, axis, value):
        if not part in dictionary.keys():
            dictionary[part] = {}
        if not component in dictionary[part].keys():
            if part == 'rotation_quaternion':
                dictionary[part][component] = {'w':1.0, 'x':0.0, 'y':0.0, 'z':0.0}
            else:
                dictionary[part][component] = {'x':0.0, 'y':0.0, 'z':0.0}
        dictionary[part][component][axis] = value
    ##
    def add_to_pose(pose_dict, data_path, array_index, value):
        part_component = data_path_to_part_component(data_path)
        part = part_component[0]
        component = part_component[1]
        if component == 'rotation_quaternion':
            if array_index == 0:
                axis = 'w'
```

```
            elif array_index == 1:
                axis = 'x'
            elif array_index == 2:
                axis = 'y'
            elif array_index == 3:
                axis = 'z'
            else:
                print("UNKNOWN INDEX = " + str(array_index) + ' part=' +  str(part))
        else:
            if array_index == 0:
                axis = 'x'
            elif array_index == 1:
                axis = 'y'
            elif array_index == 2:
                axis = 'z'
            else:
                print("UNKNOWN INDEX = " + str(array_index) + ' part=' +  str(part))
        #
        add_value(pose_dict, part, component, axis, value)
        ##
    def export_frame(folder, num):
        filename = os.path.join(folder, './pose_'+str(num).zfill(4)+'.json')
        fout = open(filename, 'w')
        cur = bpy.data.actions["PoseLib"].id_data.fcurves
        pose = {}
        for fc in cur:
            add_to_pose(pose, fc.data_path, fc.array_index, fc.evaluate(num))
        json.dump(pose, fout, indent=4)
        fout.close()
        return pose
    ##
    for index in range(num_frames):
        export_frame(folder, index+1)
```

Once exported all poses, they must be converted into a X_Bot configuration, to do that a Jupyter Notebook (./utils/pose_to_animation.ipynb) has been used.

This notebook performs also another important task, it normalizes all quaternions, indeed when a pose is edited in Blender, quaternion orientation of a part is not stored normalized, then when it is exported using the previous function export_pose_library() it required to be normalized. A quaternion represents an orientation only if it is normalized i.e. its norm is equal to one, if it is not true Three.js deforms object shape during the rendering.

6 User Interaction

6.1 Walk

Pressing UP_ARROW button user moves the standing X_Bot forward respect to the robot, to turn left or right user has to click LEFT_ARROW or RIGHT_ARROW respectively. When the robot can not go ahead due to an obstacle it returns automatically to the rest pose.

6.2 Change Color

Using GUI at right up corner, user can change diffuse and specular components of robot's skin, an example below.

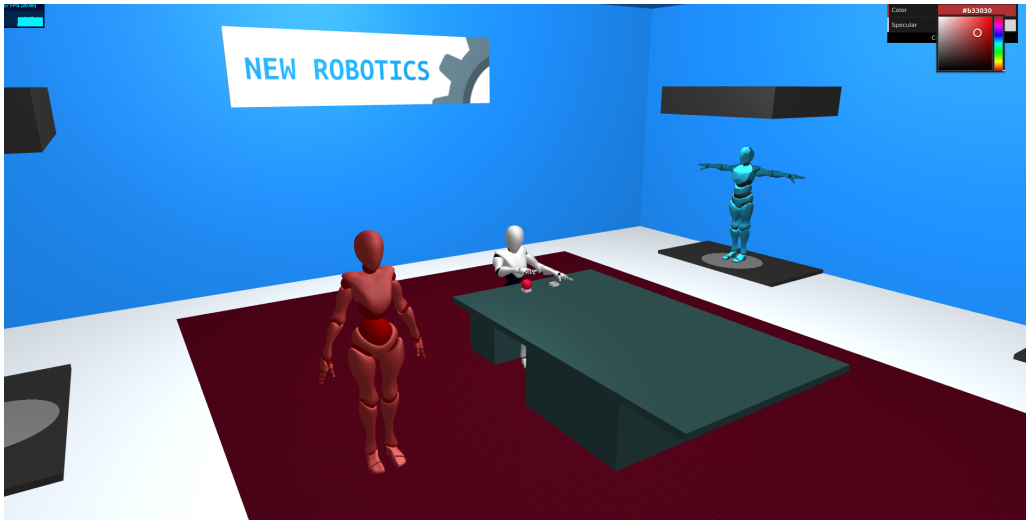


Figure 11: Example color change

6.3 Change View

User can move camera view by pressing left button of mouse and moving it, moreover, can zoom-in or zoom-out using mouse washer.

7 Browser Compatibility

Project has been tested using Ubuntu 18.04 LTS, both locally (Python3 server) and in GitHub Pages via following browsers:

- **Google Chrome version 92.0.4515.159 (Official Build) (64 bit)**
- **Firefox version 91.0.2 (64 bit)**

Note, using GitHub Pages the first loading may take a few seconds.
GitHub Pages Link: `https://sapienzainteractivegraphicscourse.github.io/final-project-gioele_migno/`