

Interactive Graphics

Spring 2019 – Final Project

Luca Maggio, Matteo Rizza, Nicola Di Santo

Interactive Graphics	1
Spring 2019 – Final Project.....	1
Luca Maggio, Matteo Rizza, Nicola Di Santo	1
Introduction.....	3
Usage Guide.....	3
Structure of the Project.....	4
Three.js Description.....	4
External Models.....	4
Environments.....	5
Bedroom.....	5
Garden.....	5
Kitchen.....	5
Character Model.....	6
Skinned Mesh	6
Customization.....	7
Glasses Draw.....	7
Glasses Load	7
Wizard-Hat Draw and Loading.....	8
Point and Click – Raycaster.....	8
Animation	8
Tween.js for Skeletal Animation.....	9
Rotate the Character to into the Direction of Walk	9
Collision Detection.....	10
Raycaster	10
Collision detection with boxes	11
Requirements-Implementation Matrix	12
Speedup Advice	13
References.....	13

Introduction

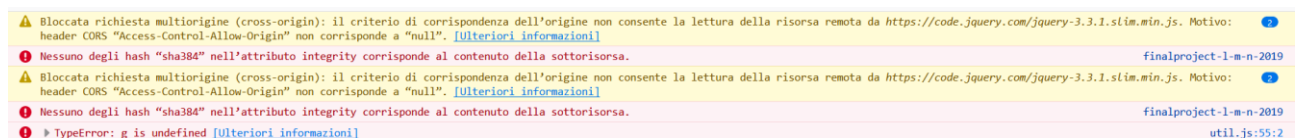
As final project we have decided to implement a basic character that can move around in an environment. The project can not be defined a game, since there is no goal, it can be considered as an interactive animation.

To handle the graphic part of the project we decided to use the following libraries:

- Three.js as core graphic library;
- DAT.GUI.js and OrbitControl.js for better and faster debug, but they are not used in the final project;
- GLTF Loaders from Three.js extensions have been used to import external models ;
- Tween.js as useful tool to implement better animations.

To design the introduction page of the project, we have used the bootstrap framework to achieve a UI style that is intuitive and that is uniform among the majority of devices and browsers.

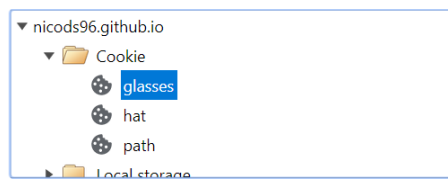
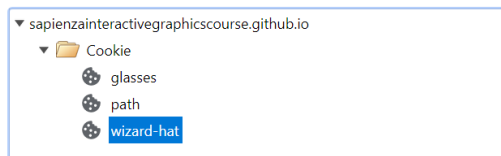
NB. bootstrap framework has been imported with CDN. Within the last Firefox update, we have encountered some problems in loading resources through CDN due to CORS blocking policy. If for some reasons a CORS warning like the one showed below appears, please use Google Chrome.



Usage Guide

The first page loaded when clicking on the project link is the index.html page. The page is a simple html page with two buttons: New Game and Customize. By clicking 'customize' button one button will appear and user can add glasses. The user can make this button to disappear again by clicking on customize or can remove clicking on the remove button that will appear once the object is loaded.

By clicking on 'New Game' you will start the loading of another page, index2.html, that will load the character inside an environment. Since HTTPS is stateless, to transfer the choices of gadget (glasses and hat) to customize the character to the html page responsible for drawing the scene, we have used cookies. In particular, the cookie will contain the string 'glasses=yes;wizard-hat=no' if for example we have added glasses but we didn't add the hat.

			
Nome	glasses	Nome	wizard-hat
Contenuti	yes	Contenuti	yes
Dominio	nicods96.github.io	Dominio	sapienzainteractivegraphicscourse.github.io
		Percorso	/finalproject-1-m-n-2019

Cookies set to store customization choices.

It will be then very important to enable cookies for the domain of the project.

Once the model is loaded inside a room, it will be possible to click with the mouse inside the room, and the character will move till the clicked point. Button will be showed to change room or exit as well as button to perform further animations like dabdance, hello movement, affermative movement with the head.

Structure of the Project

The project is organized into folders, each of them representing a semantic area:

- The root folder contains the index.html, garden.html, kitchen.html and bedroom.html that are the pages containing, respectively, the user interface to customize the character and start the interactive animation, and the core page at which we are redirected after the character customization, that will host the threejs scene.
 - The folder models will contain all the .gltf models used in the project. The models are all taken from <https://sketchfab.com/> except for the character that is taken directly from the three.js examples folder on github <https://github.com/mrdoob/three.js/tree/dev/examples/models/gltf/RobotExpressive>. They are all free models with license for free usage for non-commercial purposes.
 - The folder 'lib' contains all the libraries used and listed above, except for the bootstrap (<https://getbootstrap.com/>) framework that is imported though CDN
 - The 'js' folder, containing all the javascript written from the group for this project. This folder contains an internal subdivision too:
 - 'Environments' folder, with the scripts to initialize the environments in which the character will move. They have been implemented one for project member.
 - 'Animations' containing all the animation implemented with the help of Tween.js
 - The 'img' folder, containing the pictures used as textures in the project.

Three.js Description

Three.js is a JavaScript Object Oriented framework that is used to create and display 3D animated graphics in a web browser. This library helps the creation of video games, animations or 3D graphics as part of the internal browser without the use of external plugins. All this is thanks to the use of WebGL by the same library. Programming WebGL directly from JavaScript to create and animate 3D scenes is a very complex and error-prone process. Three.js is a library that makes this a lot easier.

Furthermore, this library is easily set with HTML5 elements such as canvas. Three.js also has pre-built components and helper methods you can take advantage of to get started faster.

External Models

The external model used in the project are all GLTF models.

GLTF (GL Transmission Format) is an open format specification for efficient delivery and loading of 3D content. Assets may be provided either in JSON (.gltf) or binary (.glb) format. External files store textures (.jpg, .png) and additional binary data (.bin). A glTF asset may deliver one or more scenes, including meshes, materials, textures, skins, skeletons, morph targets, animations, lights, and/or cameras. (Three.js Documentation, s.d.)

NOTE: IE11 is not supported yet, please use Mozilla Firefox or Google Chrome.

Environments

To draw environment, we have decided to draw each environment within an .html page. This is done within the hypothesis that a user will spend more time using the character inside an environment than switching between scenes. This approach also ensure isolation between javascript code and between environments, freeing up programmers of lot of intense design to draw three scenes into a single page in isolation.

To draw environments, three function are available at location `/js/environments/`, that are `bedroom()`, `garden()`, `kitchen()`, located respectively in `bedroom.js`, `kitchen.js`, `garden.js`.

Bedroom

For the bedroom environment we have first built a scene. Then a perspective camera has been added to the three.js scene as well as both directional and ambient light. The loader then is initialized with the path of the model and the on-Load, on-Error and on-Progress call-backs. The model contains floor, a bed, a desk, and a training ball.

The on-Load call-back performs the core operation on the model. In particular it will traverse the model and, for each mesh inside it, it will enable the cast shadow mode (by default is disabled). After that, with the aid of the THREE.Box3 utility, we set up a cube mesh that can contain inside it the loaded model and that will form the room skeleton: each side of the cube will be a wall/roof/floor. To make it look real, a cubemap has been used to add texture to each side of the cube, to allow it to reproduce a room without furnishings.



Outside and inside view of the cubemap

The call-back to load the character inside the environment is a little bit more structured. We are going to give all the details into the [character chapter](#). Here we just limit the description to what follows: we need to traverse the whole model, extract the skeleton and reconstruct it to use animation with skinning, retrieve the customization options from cookies and, if needed, load the ornamentation models. Once those operations are completed, the animations are loaded, and the room is ready to be used (i.e. it is possible to point and click to let the model move).

Garden

For the realization of the garden environment a scene and a perspective camera was created. After that, a hemisphere light was implemented to faithfully reproduce sunlight. the environment is realized inside a cubemap and personalized through various models such as the fountain, the gazebo and the tree. After this, our robot is inserted into the scene, checking for any customization options (hat, glasses). Finally the movement animation is implemented thanks to the use of raycasters.

Kitchen

The kitchen section was created by creating a scene and initializing a perspective room. Lights (directional and ambient) were also loaded into the scene. A loader has been added that allows us to add models to the scene: in particular, models in GLTF format have been added, a kitchen

composed of a kitchen countertop, a hood, a sink, objects such as crockery, carrots and a chocolate jar, the tiles and a sideboard and some objects (as you can see in the figure below) like flowers and chairs. A table has also been added. The method of loading the model is identical to that described in the previous two rooms. Then the cubemap that represents the wrapper of all the models inserted inside (including the small robot that will represent the character) has been initialized. To make it look real, to make it look to add texture to the side.



Outside and inside view of kitchen cubemap

Character Model

The character is also a GLTF model. By default the model has built in animations that are completely ignored into the project. The interesting part of using this model is the possibility of use skinning and bones for animations.

Skinned Mesh

First of all, while traversing the model, we look for an object whose name is *'RobotArmature'*. The RobotArmature object is the root bone of the model's skeleton. We store it into a global variable and, once the model is fully loaded, we use the root bone to initialize a skeleton. Mentioning three.js documentation, we can say a skeleton is an object that use an array of bones, it can be used by a SkinnedMesh.

A SkinnedMesh is a mesh that has a Skeleton with bones that can then be used to animate the vertices of the geometry. The material must support skinning and have skinning enabled. Using a SkinnedMesh to animate the character is very useful since the whole process will reduce to a series of joint rotations or bones translation. Moreover, a skeleton has a hierarchy, and then we can translate the whole character translating its root bone or we can rotate an entire arm by rotating only the shoulder joint.

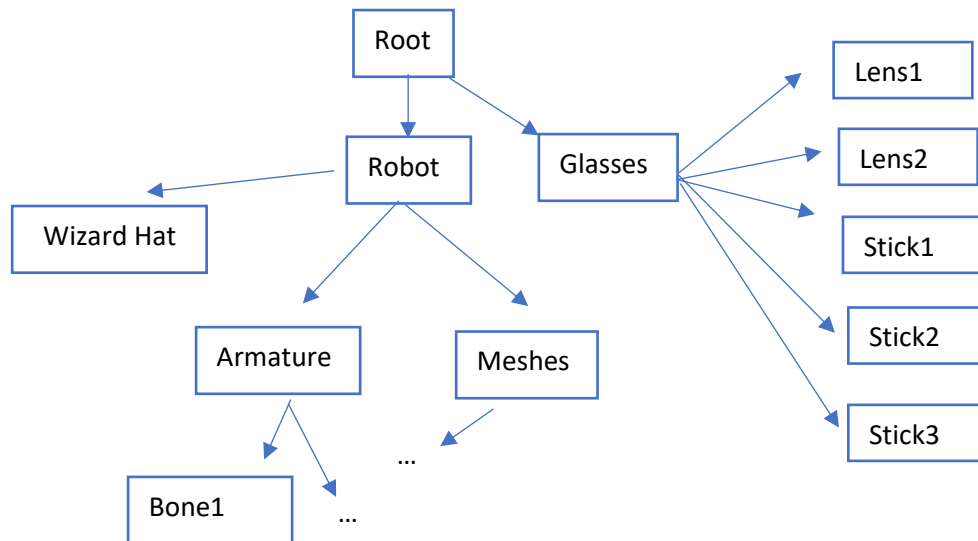
This technique is used by constructing a series of 'bones,' sometimes referred to as rigging. Each bone has a three-dimensional transformation from the default bind pose (which includes its position, scale and orientation), and an optional parent bone. The bones therefore form a hierarchy. The full transform of a child node is the product of its parent transform and its own transform. So, moving a thigh-bone will move the lower leg too.

Strengths of this approach: a bone represents a set of vertices (or some other objects, which represent for example, a leg), bones are independently movable the animator controls fewer characteristics of the model and it can focus on the large scale motion, an animation can be defined by simple movements of the bones, instead of vertex by vertex (in the case of a polygonal mesh). Weaknesses of this approach: a bone represents a set of vertices (or some other objects) and does not provide realistic muscle movement and skin motion and that is the reason why our character is a robot.

Customization

To customize the aspect of the character and to implement a more complex hierarchical model, we have both grouped together the imported model (character) and we have built two objects (glasses and wizard hat) starting from grouping three.js basic meshes and then we have again grouped the objects hierarchical model with the character model. To build the hierarchy the `THREE.Group()` utility has been used. It is quite like an `Object3D`, with the difference that allows us to deal with multiple object easily.

If all the gadgets are added to the model, then the hierarchy is something like:



So, the hierarchy can be also defined dynamic, since it can change during the execution. The root element is a box (invisible), we will explain why into the collision detection chapter.

Glasses Draw

To draw the glasses, we can check the procedure `drawGlasses()` into the file `/js/drawAccessories.js`. To build lens, we have used a cylinder geometry with a very low height. We have also used a low number of vertices to give lens a eave shape. To simulate glass material, a `MeshBasicMaterial` has been used with transparency and opacity settings:

```
//glass
var material = new THREE.MeshBasicMaterial( {
  color: 0xe0e0e0,
  opacity: 0.8,
  transparent: true
}); //E0E0E0
```

Sticks instead use a box geometry and a simple dark green mesh basic material. After all the components are positioned, they are added to a group and returned to the loader procedure that invokes the `drawGlasses()` function.

Glasses Load

To Load glasses, we first checks the cookies to see if they must be on the character, then if yes we call the `drawGlasses()` procedure and we will periodically check to see if the model has been loaded. Once it is correctly loaded, we can adapt them to the model and add them to the model hierarchy:

```

if( getCookie('glasses') == 'yes'){
    glasses = drawGlasses();
    checkGlasses = setInterval(function(){
        if(glasses == null){
            //console.log('glasses null');
        }else{
            //console.log('glasses not null');
            clearInterval(checkGlasses);
            glassesAreLoaded = true;
            var box = new THREE.Box3().setFromObject( modelChar );
            glasses.scale.set(0.1,0.15,0.15);
            glasses.rotation.x += Math.PI/2;
            glasses.position.z += box.getSize().z +1.5 ;
            glasses.position.y -= box.getSize().y /2 + 0.2 ;
            hier.add(glasses);
        }
    }, 500);
}

```

Wizard-Hat Draw and Loading

The hat is defined completely similar to the glasses. The main difference is that to customize the hat, instead of playing with material properties, we have chosen to customize it with a simple texture:

```

var geometry = new THREE.ConeGeometry( 0.5, 20, 32 );
var material = new THREE.MeshBasicMaterial( {map: new THREE.TextureLoader().load('textures/txtuhat.jpg')} );
var cone = new THREE.Mesh( geometry, material );

```

Point and Click – Raycaster

The key interaction with the user is the point and click to specify where the character should move. To obtain the point from the mouse click, we exploit the raycaster.

Ray casting is the use of ray–surface intersection tests and has a variety of application, especially in computer graphics. We have used it to ‘draw’ a ray that goes from the camera to the point clicked by the mouse. This ray is going to intersect some object into the scene (it have to since the camera is fixed and surrounded from a box). We are sure that, by design of the environment, the last object intersected will be the cube, representing the room’s foundations. We will then get the coordinates of the cube point intersected and will use the planar coordinate x and y to tell character where it has to translate.

```

mouse.x = ( event.clientX / window.innerWidth ) * 2 - 1;
mouse.y = - ( event.clientY / window.innerHeight ) * 2 + 1;

// update the picking ray with the camera and mouse position
raycaster.setFromCamera( mouse, camera );

// calculate objects intersecting the picking ray
var intersects = raycaster.intersectObjects( scene.children );
var point = intersects[intersects.length -1].point; //cube

```

Then an animation is started, that goes from the actual character position to the point get from the raycaster. Follows the code, more detail on the animation in the Animation chapter.

```

var rootInit = { x : hier.position.x , y : hier.position.y };
var rootFinal = { x : point.x , y : point.y };

```

Animation

In our Project animation in the key of user interaction. Three button are available to trigger an animation. The technique used to animate our model is Skeletal animation together with tweening.

Skeletal animation is a technique in computer animation in which a character (or other articulated object) is represented in two parts: a surface representation used to draw the character (called skin or mesh) and a hierarchical set of interconnected bones (called the skeleton or rig) used to animate (pose and keyframe) the mesh. While this technique is often used to animate humans or more generally for organic modelling, it only serves to make the animation process more intuitive, and the same technique can be used to control the deformation of any object. (Skeletal animation, s.d.)

[Tween.js for Skeletal Animation](#)

In three.js, animation is quite complicated. It's hard enough to orient and translate meshes in 3d space. When you add a fourth dimension and the coordinates of hundreds of vertices, the task seems impossible.

Skeletal animation is objects with skeletons. That includes people walking, cats meowing, robots working, snakes a-slithering, octopi a-swimming, tails a-wagging and anything else with a skeleton-like structure.

If a model has a skeleton, we call it a "rigged" model. A powerful feature of many modeling programs called inverse kinematics can be used with rigged models that calculates realistic movement of a chain of bones. You can read about inverse kinematics for more on that.

Tween.js really is a great tool to use with three.js. It keeps an array of the tweens that are running and discards them when they are complete.

[Rotate the Character to into the Direction of Walk](#)

Before animating the character, we made him rotate in the direction of the point to reach and then we let him walk, to create a more natural walking experience.

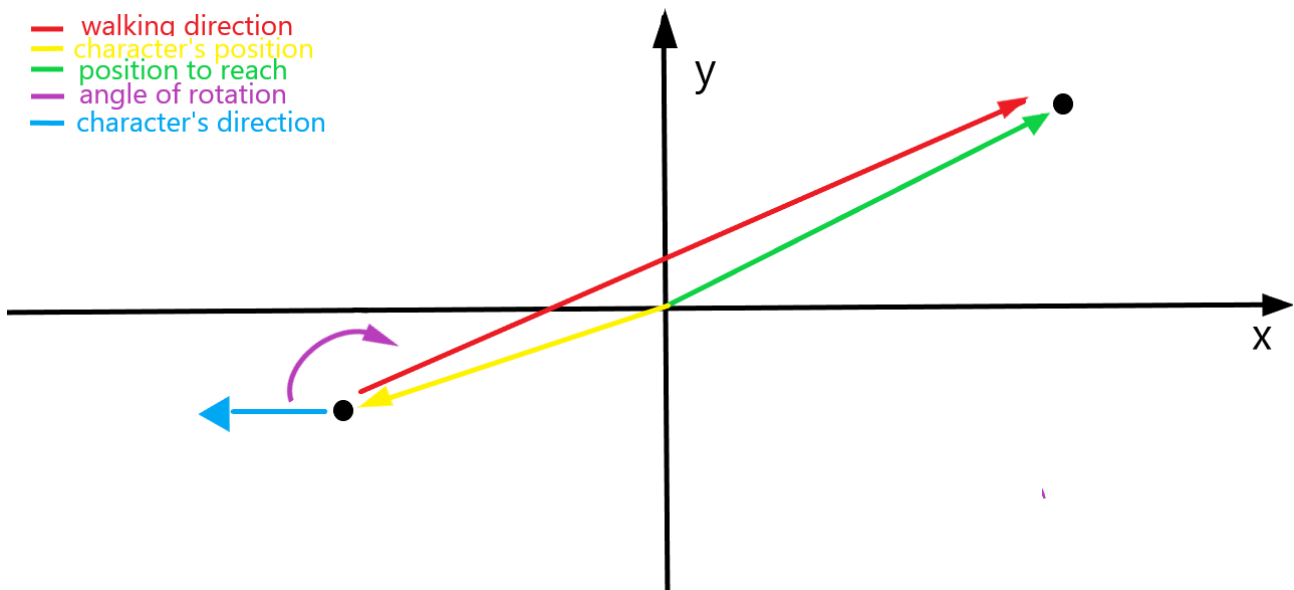
The math behind this is very simple. We have defined the following variables:

- *robotLookingAt*, is the current direction of the character;
- *newLookingAt*, is the direction that goes from the character to the clicked point;
- *angleOfRotation*, is the angle that goes from the current direction to the new direction;
- *point*, are the coordinates of the point we want to reach;
- *charPos*, is the position of the character (in the code its name is *invisiblebox.posiiton*);

The algorithm is simple:

1. $newLookingAt = charPos - point$.
2. $angleOfRotation = \cos^{-1} \left(\frac{\langle newLookingAt, robotLookingAt \rangle}{|newLookingAt| |robotLookingAt|} \right)$, where $\langle a, b \rangle$ is the scalar product between two vectors.
3. Apply the rotation to character its direction.

Here is an image explaining the logic:



And finally, here is the code:

```
var newLookingAt = new THREE.Vector3( );
newLookingAt = newLookingAt.subVectors( point, invisibleBox.position ).normalize();
//CALCOLARE L'ANGOLO TRA LE DUE DIREZIONI
var angleOfRotation = robotLookingAt.angleTo(newLookingAt);

//RUOTARE PERSONAGGIO E VETTORE robotLookingAt
invisibleBox.rotation.z += angleOfRotation;

//update robotLookingAt
var axis = new THREE.Vector3( 0, 0, 1 );
robotLookingAt.applyAxisAngle( axis, angleOfRotation );
```

Collision Detection

Since more than one person have worked to the collision detection, two types of collision detection have been implemented, one using raycaster and the other using boxes intersections.

Raycaster

For the collision detection, invisible boxes have been created adapted to the shapes of the models present on the scene. Within the `tween.onUpdate()` function, the position of the model is updated (updating invisibleBox position we update also the robot model position because it is child of invisibleBox). Then, a check is performed to detect any collisions between the vertices of the box relative to the robot and the boxes of the various models; this functionality is implemented through the raycaster. All the invisible boxes are stored in the `collidableMeshList` variable. When a collision is detected, through the `tween.stop()` function the displacement animation is interrupted to avoid the overlap between the two models.

```

tween.onUpdate(function(){
    invisibleBox.position.x = rootInit.x;
    invisibleBox.position.z = rootInit.z;

    var originPoint = invisibleBox.position.clone();

    for (var vertexIndex = 0; vertexIndex < invisibleBox.geometry.vertices.length; vertexIndex++)
    {
        var localVertex = invisibleBox.geometry.vertices[vertexIndex].clone();
        var globalVertex = localVertex.applyMatrix4( invisibleBox.matrix );
        var directionVector = globalVertex.sub( invisibleBox.position );

        var ray = new THREE.Raycaster( originPoint, directionVector.clone().normalize() );
        var collisionResults = ray.intersectObjects( collidableMeshList );
        if ( collisionResults.length > 0 && collisionResults[0].distance < directionVector.length() ){
            tween.stop();
            tweenHEAD1.stop();
            tweenLEGR1.stop();
            tweenLEGL1.stop();
            tweenARMR1.stop();
            tweenARML1.stop();
            tweenHEAD2.stop();
            tweenLEGR2.stop();
            tweenLEGL2.stop();
            tweenARMR2.stop();
            tweenARML2.stop();
        }
    }
});
tween.start();

```

The raycaster simply checks for each vertex of our character (to be precise of the box including the character) if they intersect any object and stops the animation if true.

Collision detection with boxes

It has been implemented into the bedroom environment (environments/bedroom.js) inside the function checkCollision(). The function is called each time the translation animation is updated. To implement collision detection with boxes we have defined a series of boxes that wrap the walls and objects in the scene (this is done because the boxes built around all the meshes inside the model were not accurate enough). Here is an example of box definition:

```

BED_BOX = {
    mesh: new THREE.Mesh(
        new THREE.CubeGeometry( 5, 5, 5 ),
        new THREE.MeshStandardMaterial( {
            opacity: 0.3,
            transparent: true
        } )
    ),
    position : {x:-6,y:-9,z:2.5},
    scale : {x:3.2,y:2.8,z:1}
}

```

Then a box around the character is built, referenced into the code as invisibleBox:

```

var size = new THREE.Box3().setFromObject( modelChar).getSize();
invisibleBox = new THREE.Mesh(
    new THREE.CubeGeometry( size.x/2, size.y, size.z ),
    new THREE.MeshStandardMaterial( {
        opacity: 0.3,
        transparent: true
    } )
);

```

The collision mechanism does the following: for each defined box checks whether they intersect or not the invisible box (the box built around the character) using the Box3 method intersectBox(). If there is a collision there is also an intersection. When a collision is detected, the box surrounding the object is displayed for one second and then removed. After a collision is detected the interactive animation ends and the user is redirected to the a game over page and after 5 seconds again he is redirected to menu page. NB in the final version, the *helpmesh*, i.e. the box surrounding the object, is not displayed since it has opacity equal to zero. If you want to modify this value go in js/utils.js and modify it. This helpmesh is displayed only into bedroom and kitchen environments, since they are the two rooms that implements collision with boxes.

Follows the code of the loop that checks for collisions:

```

for (var i = 0; i < boxList.length ; i++) {
    var obj = boxList[i].mesh;
    obj.position.set (boxList[ i ].position.x, boxList[ i ].position.y, boxList[ i ].position.z );
    obj.scale.set (boxList[ i ].scale.x, boxList[ i ].scale.y, boxList[ i ].scale.z );

    secondBB = new THREE.Box3().setFromObject( obj );
    collision = firstBB.intersectsBox(secondBB);

    if(collision){

        var helpmesh = new THREE.Mesh(
            new THREE.CubeGeometry( secondBB.getSize().x , secondBB.getSize().y, secondBB.getSize().z ),
            new THREE.MeshStandardMaterial( {
                opacity: 0.5,
                transparent: true,
                color: Math.random() * 0xffffff
            } )
        );
        helpmesh.position.set(obj.position.x,obj.position.y,obj.position.z);
        scene.add(helpmesh);

        //remove helpmesh
        setInterval(function(){
            scene.remove(helpmesh);
        }, 1000);
        console.log('collision');

        return true;
    }
}

```

The collision detection loop.

This approach is valid if the number of boxes is not huge (this is the case), otherwise it can crate overload to the system.

Requirements-Implementation Matrix

REQUIREMENT	IMPLEMENTATION
HIERARCHICAL MODELS	Main character, skeleton, character customization
LIGHTS AND TEXTURES	Ambient and directional light, texture cube map, simple texture to wizard-hat

USER INTERACTION	Trigger animations with buttons, move the character with point and click interface, change room, customize character
ANIMATIONS	Skeletal animation with bones using tweenjs

Speedup Advice

If for any reason the loading is stuck, this is because it is taking time to draw the model or to download it. We have noticed that loading the project locally speedup thinks a lot, maybe using a browser that allows CORS and opening it as a file starting from *index.html*. If this option is not possible you can use a local server to load it. If this also is not possible you can only wait. By our experience with a 30Mbps connection it takes at most one minute to load and draw the scene.

Any way, you can bypass the loading message by simply typing into the browser javascript console *'loaded=true;'*. We highly recommend to do not take this approach, otherwise you can receive some object undefined errors due to scene not loaded yet.

References

Skeletal animation. (s.d.). Tratto da https://en.wikipedia.org/wiki/Skeletal_animation

Three.js Documentation. (s.d.). Tratto da <https://threejs.org/docs/>