JumpyRex

Interactive Graphics a.a. 17/18 Final Project

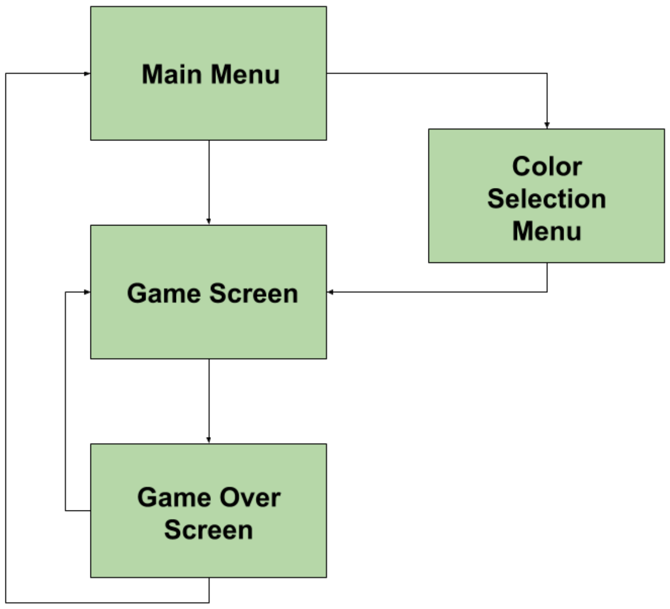
Marzio Persiani & Luca Torresi

Introduction

For our Interactive Graphics project, we decided to implement a 3D version of Dino Runner, the 2D Google Chrome game which starts whenever the network is not available. This seemed a good example of a project in which we could put in practice the theory learned during the course. In fact, we had the possibility of implementing a hierarchical 3D model, animating it, decorating it with various textures, building a simple world in which events drive the interaction between different models and creating a series of user interfaces. To create our project from scratch, we chose to use Three.js library, on top of WebGL rendering engine. Three.js came very handy throughout all the phases of the project: we used it to create our models, animate and decorate them, creating a scene and adding a source of light to it and handling the interactions and collisions between the various models. For the user interface, we instead used HTML and CSS standard code. The game is fairly simple in its dynamics: the player controls Rex, a goofy Tyrannosaurus Rex made of polygons, and tries to run further as she/he can through the desert, avoiding cactuses and flying birds. When Rex hits one obstacle instead of jumping it, the game ends. To add a bit of personalization to the concept, the player can choose the color of Rex among green, orange, purple and blue.

Game Interface: menu creation and navigation

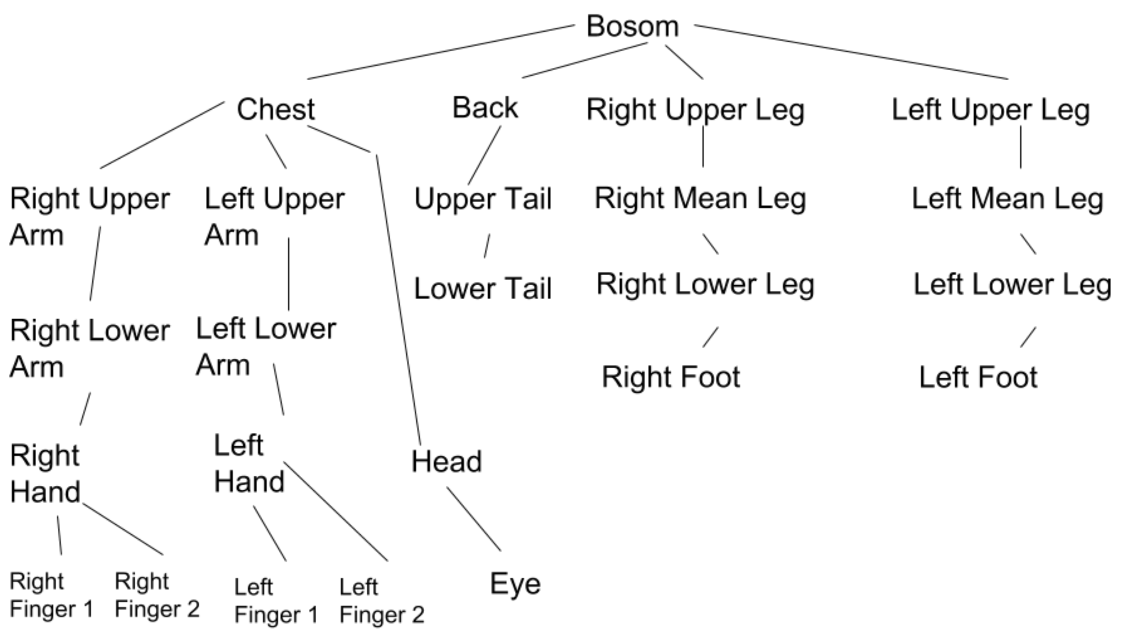
Even the most complex and well-crafted game, would be useless without a practical interface, or set of interfaces, such as menus, instructional screens and so on. Of course, our game makes no exception, so let us see how we implemented our set of interfaces, starting from a general overview of how our game can be navigated:



Aside from the **Game Screen**, we designed every interface by hand and later implemented it by means of pure HTML and CSS language. With html we prepared a series of <div>, the HTML standard container, assigned them a unique id and filled them with their proper content, being it buttons, instructional text and so on. Once we did this, we implemented a series of CSS classes to shape the graphic appearance of the various divs. Every CSS class defines a set of graphic properties like div width and height, background to load, element order and disposition, cursor appearance and so on. Inside every interface, are a series of navigation buttons: they are needed to move between the various menus and the main game, and they all share the same underlying mechanism. Every button has attached to it a callback function to be invoked when the user click it. Every interface has a *visible* property, defining whether it is shown on screen or not. When the user click the, say, *PLAY* button in **Main Menu**, `startGame()` is invoked, which in turn sets the current interface to *hidden* value, making it invisible, starts the various game scene creation routines and makes the **Game Screen** visible to the user. This is only an example, but all the possible navigation sequences, shown in the previous image, work in the same way.

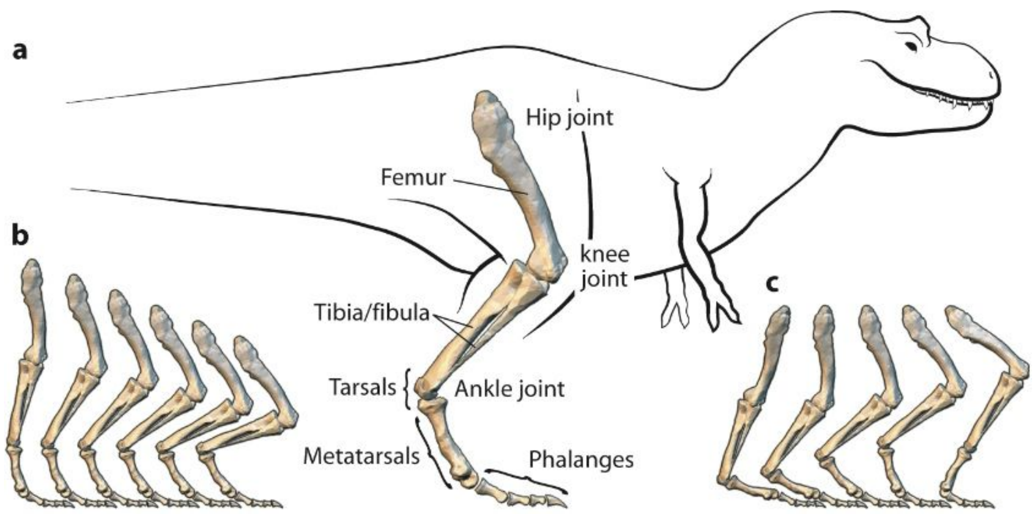
Modelling Rex: a hierarchical approach

The first step in modelling Rex was an old school one: drawing by hand the complete figure and later abstracting it in a set of separate different polygons, each one to be matched by one of the available mesh geometries of Three.js. While we used a `SphereGeometry` for the bosom and eyes and a `BoxGeometry` for the head and the feet, all the other Rex body parts were created using many `CylinderGeometry`, which in some cases (chest, tail parts, legs parts, hands) were modified so to become truncated cones. This effect was obtained by setting different radius values for the two plain surfaces of the cylinders. Once created and placed in the scene, the body parts had to be hierarchically arranged. This is due to the fact that, otherwise, every mesh rotates and translates with respect to its own 3D space, and the resulting effect would be a chaotic set of erratic meshes. Three.js offers a handy function for nesting meshes: it is sufficient to call the `add(mesh)` method on the parent mesh, passing as argument the desired child mesh. Our root mesh is the bosom one, which is later translated in the scene carrying with it all the other descendant body parts meshes. The full body parts hierarchy is the following:

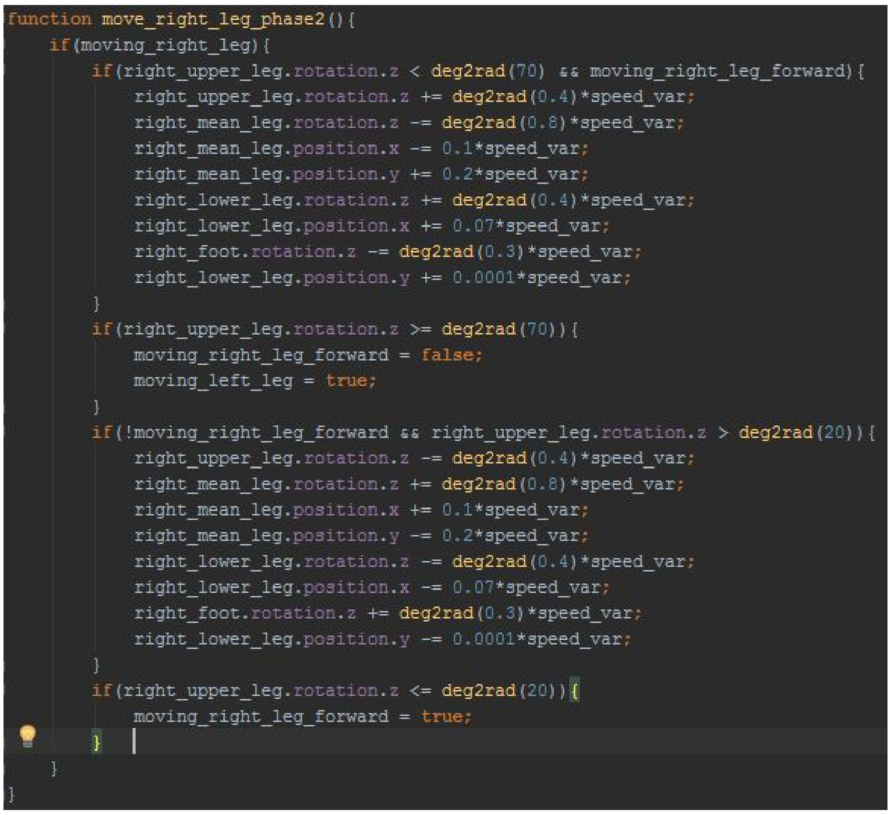


This hierarchy makes so that rotating and translating a child body part implies that its movements are all done with respect to its parent part, which plays the role of anchorage point. Without it, complex movements like T-Rex stride would be nearly impossible to implement.

Animating Rex: body parts in action
In order to animate Rex, we had to make use of two fundamental functions: `requestAnimationFrame(function)` and `update()`. The first is the actual render method which, at very close time intervals, redraws the content of the canvas shown on screen. It takes as argument a function to invoke before every draw step, in order to modify the position and aspect of the various elements present on scene. For this purpose, we passed it the *update()* function, which calls a series of methods designed to make the game proceed and, at the very end, calls recursively `requestAnimationFrame(update)`. The main core of the methods called in `update()` is the set of the functions implemented to move the Rex body parts: different methods such as `updateRexTailMovement()` and `move_right_leg_phase2()` are used in order to rotate and translate the different groups of body parts, in order to implement a movement which is as close as possible to the real one. Let's take, for example, the movement of a leg:

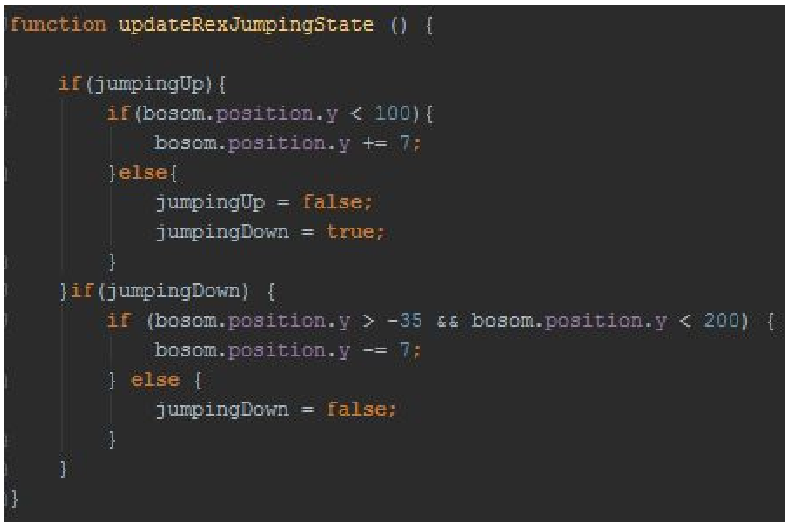


As we can see from the picture, the leg is divided into three longer bones and a series of phalanges which, for simplicity, we abstracted as a single part (the foot). The leg movement is quite peculiar: after a bit of observations, we realized how it follows an s-like pattern. Indeed, when the femur comes forward, the tibia retracts, but in the meantime the metatarsal moves forward, being dragged by the tibia. The foot, in the end, tends to align with the metatarsal during the step, to later form a 90° angle with it when resting. The movement can thus be seen a series of rotations around the z-axis of each leg part, and the movement of each part is relative to the current position of its hierarchical parent. Since we are talking about rotations, the first thing we had to define was a series of *thetas*, i.e. the rotation factors to be added or subtracted from the current z-orientation of the boy part at each `update()` call. To have more control over the final appearance of the movement, we also defined a `var_speed` variable, which uniformly scales the speed of the rotation of each body part. Each *theta* is thus multiplied to `var_speed` in order to set the new position of the leg part on the z-axis. Of course, to implement a realistic walking movement rotation is not enough: each leg part also shows a certain displacement from its original x and y coordinates, and so we also had to properly increment or decrement the x and y components of the translation vector of each leg part. To keep the movement coherent, also in this case we multiplied the displacement amount for `var_speed`. Showing a snippet of code can now come handy to make some final remarks about leg movement:



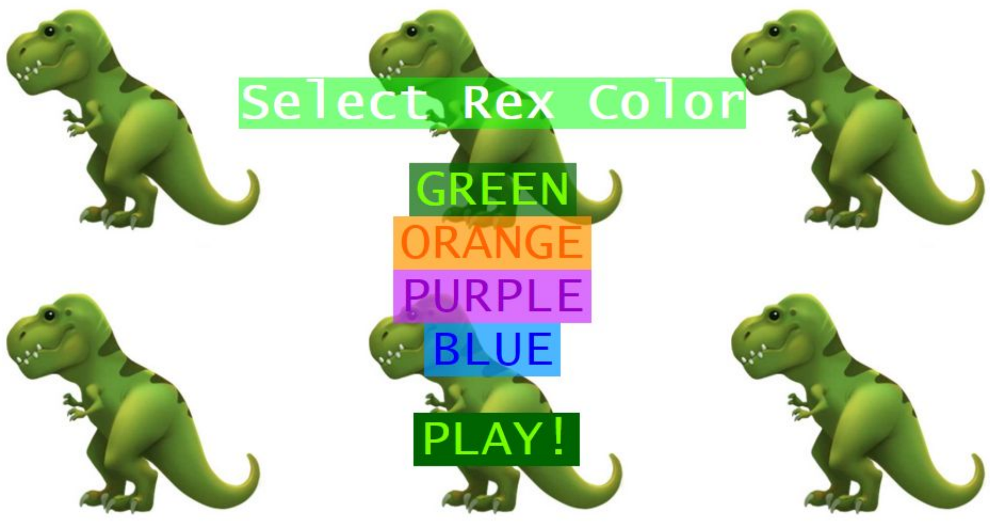
```
function move_right_leg_phase2(){
    if(moving_right_leg){
        if(right_upper_leg.rotation.z < deg2rad(70) && moving_right_leg_forward){
            right_upper_leg.rotation.z += deg2rad(0.4)*speed_var;
            right_mean_leg.rotation.z -= deg2rad(0.8)*speed_var;
            right_mean_leg.position.x -= 0.1*speed_var;
            right_mean_leg.position.y += 0.2*speed_var;
            right_lower_leg.rotation.z += deg2rad(0.4)*speed_var;
            right_lower_leg.position.x += 0.07*speed_var;
            right_foot.rotation.z -= deg2rad(0.3)*speed_var;
            right_lower_leg.position.y += 0.0001*speed_var;
        }
        if(right_upper_leg.rotation.z >= deg2rad(70)){
            moving_right_leg_forward = false;
            moving_left_leg = true;
        }
        if(!moving_right_leg_forward && right_upper_leg.rotation.z > deg2rad(20)){
            right_upper_leg.rotation.z -= deg2rad(0.4)*speed_var;
            right_mean_leg.rotation.z += deg2rad(0.8)*speed_var;
            right_mean_leg.position.x += 0.1*speed_var;
            right_mean_leg.position.y -= 0.2*speed_var;
            right_lower_leg.rotation.z -= deg2rad(0.4)*speed_var;
            right_lower_leg.position.x -= 0.07*speed_var;
            right_foot.rotation.z += deg2rad(0.3)*speed_var;
            right_lower_leg.position.y -= 0.0001*speed_var;
        }
        if(right_upper_leg.rotation.z <= deg2rad(20)){
            moving_right_leg_forward = true;
        }
    }
}
```

A series of if-statements is clearly visible: the first one, `moving_right_leg`, is needed to be sure that the current one is the right moment to move the right leg. This is due to the fact that, since a natural bipede stride requires that the legs move alternatively, when the left leg is being moved the right one must be still and vice versa. In the other if we handle the direction of the leg movement: as it is easy to imagine, the leg must have two alt points which, once reached, can not be exceeded, and need to the system to know that the leg has now to be moved in the opposite direction. To get this result, we use a mix of both boolean variables and control over the amount of rotation of the upper leg body part. As a final note, the `deg2rad(amount)` function is a simple method that takes a quantity expressed in degrees and turns it into its respective number of radians, this is needed because, in Three.js, the rotation vector of each object in the scene is expressed in radians. As a side consideration, the pivotal movement for the game, the jumping one, turned out to be the simplest to implement: thanks to the hierarchical model, it was sufficient to implement a listener which, every time the spacebar or up-arrow keys are pressed, start a y-axis translation routine of the bosom body part. Since it is the root element in the body hierarchy tree, all the children parts follow it accordingly while also keeping on moving according to their rules. Even in this case, we had to implement a jump speed and two halt points. When the top one is reached, the jump enter its second phase and, by decreasing the bosom y value, the Rex comes down to the ground. Here is the jump routine code:



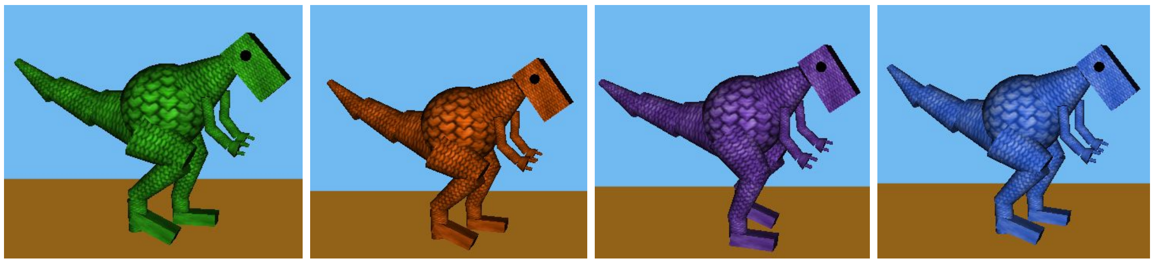
```
function updateRexJumpingState () {

    if(jumpingUp){
        if(bosom.position.y < 100){
            bosom.position.y += 7;
        }else{
            jumpingUp = false;
            jumpingDown = true;
        }
    }if(jumpingDown) {
        if (bosom.position.y > -35 && bosom.position.y < 200) {
            bosom.position.y -= 7;
        } else {
            jumpingDown = false;
        }
    }
}
```

Customizable Rex color: using textures
To give the players of our game a mean of experience personalization, we implemented the possibility to choose Rex skin color. To do this, we exploited the set of Three.js functionalities related to mesh creation and use. First of all, in the main menu of the game, other than the PLAY button, the PAINT REX option can be selected. This gives access to a second menu in which the color of the skin of the T-Rex can be chosen between green, orange, purple and blue.



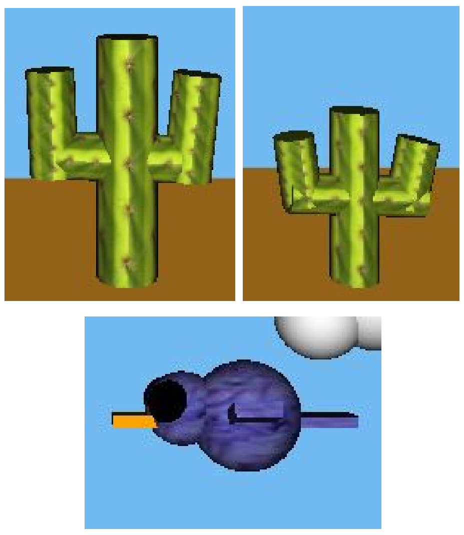
Clicking one of this button will set the global variable `selected_rex_color` by means of the function `setRexColor(index)` with the name of the .jpg file of the texture corresponding to the chosen color option. This variable will then be used in the function `initRex()` to set the texture of each of the body part of the T-Rex. The default Rex color is green, and so in the case in which the user does not select any color the standard green texture will be loaded.
`THREE.TextureLoader().load('textures/reptile_skin_2.jpg')` is the library function we use to load a certain texture, and its return value `texture_skin` will be later

passed to the mesh material by invoking `THREE.MeshLambertMaterial({map: texture_skin})`. Of course, *reptile_skin_2.jpg* is the name of the standard texture file, but will be replaced with the aforementioned `selected_rex_color` variable. Following are all the possible aspects the Rex could have during play:



Obstacles: creation and animation

Up to this point, we focused mainly on the creation and animation of our protagonist, Rex, but the game fun relies on on another major element: the obstacles Rex must avoid while running in the desert. There are three kind of obstacles Rex can meet on its path: birds and bigger or smaller cactus, all depicted below.



All the objects, just as the Rex, are created with a hierarchical approach. Cactuses are composed of 4 `CylinderGeometry` where the root is the vertical principal stem, the horizontal stem is direct child of the root and the two arms, left and right, are its children. Birds, since they present moving body parts, have a more complex structure: we used `SphereGeometry` for the main body, head and eyes and a `BoxGeometry` for the beak, wings and tail. The cactus move in a rigid way on the canvas: the x value of the vertical stem is decreased, so that the whole cactus translates towards the Rex. As previously stated, we instead added a more complex movement routine for the birds, managed through the function `updateBirdWingsMovement()`. This function, called at every update run as long as there is a bird on the scene, manipulates the wings *theta* values in a fashion which is similar in principle to the movement functions for the Rex body parts. The translating movement of the bird is once again obtained through a simple decrease of the x value of its root element, the main body sphere. An interesting point to highlight is how we handled the spawning and lifecycle of the obstacles: when the game starts, all the obstacle objects are generated with an initial position which would be outside the canvas on the right side but are not added to the scene. After the scene initialization is complete, the function `randomObstacle()` is called. This method randomly selects one of the possible obstacle types and adds it to the scene, setting it as value of the variable `curr_obstacle`. At this point, every time the `update()` function is called, the position of the obstacle currently on scene is updated with the method `updateObstaclePosition()`. Once the object reaches a predefined x-position outside the canvas on the left side, the obstacle is removed from the scene, all the variables pointing to or related with it are emptied and `randomObstacle()` is called again.

Collision handling: making the objects interact

Aside from the object movement routines, the main core of the game is represented by the collision detection engine. Collisions can only occur between Rex and the current obstacle, so, when detected, they are the signal that the player failed to avoid an obstacle and the game is over. In the past, collisions had to be checked by hand at every update cycle, checking the position of the various objects to see if they overlapped and, in case, take certain decisions. Luckily, with today's APIs such weights are lifted from the shoulders of the game developers, and many primitive functions are offered to check these conditions. In Three.js, collisions are handled by using the class `THREE.Raycaster()` and the method `intersectObjects()`. What this class does is generating a series of rays starting from the center of mass of the desired objects and extending along the three dimensional axes. The mentioned method thus checks, whenever called, if a series of objects are met by one of the rays attaining to the body of interest. In our particular case, at every update cycle we check whether the ray coming from the current obstacle intersects one of the Rex body parts along the z axis. When this condition is met, the Rex dies, the game freezes and the Game Over screen is shown to the player, which can thus decide to play again or to go back to the Main Menu.