Interactive Graphics project

Spider House

Lorenzo Michieletto 1838277 *Emilian Postolache 1649271*

Introduction

This work is an attempt to make a video game for browser exploiting Webgl. The main idea of this game is to control a spider around a house, with the goal to escape from it by collecting keys. The spider is able to climb up, climb down and cast webs around the walls and the objects.

Technologies involved

The project was developed mainly using the *Three.js* library, which allows to easily manage lights and textures, different cameras and provides the tools to deal with 3D object by using the copious amount of geometric transformation.

Apart from the spider, the keys and two animals, the models loaded in the game have all been created using *Blender*, shading and texturing included, which grants a precise and controlled environment, which is essential for the interaction with the character.

Webpack was used for organizing the code in modules, in order to enable an Object Oriented Approach.

Application structure

The entry point in the program is a *Main* class, which initializes the various Three.js objects (the renderer, the camera, the lighting, etc.) and then calls a *mainLoop* method during each frame.
The basic building block of the application is a modality, which is a subclass of the *Mode* class. The public interface of this class consists of three methods:

- *update()*
- *handleMouseMove(deltaX, deltaY)*
- *handleAction(event, key, value)*

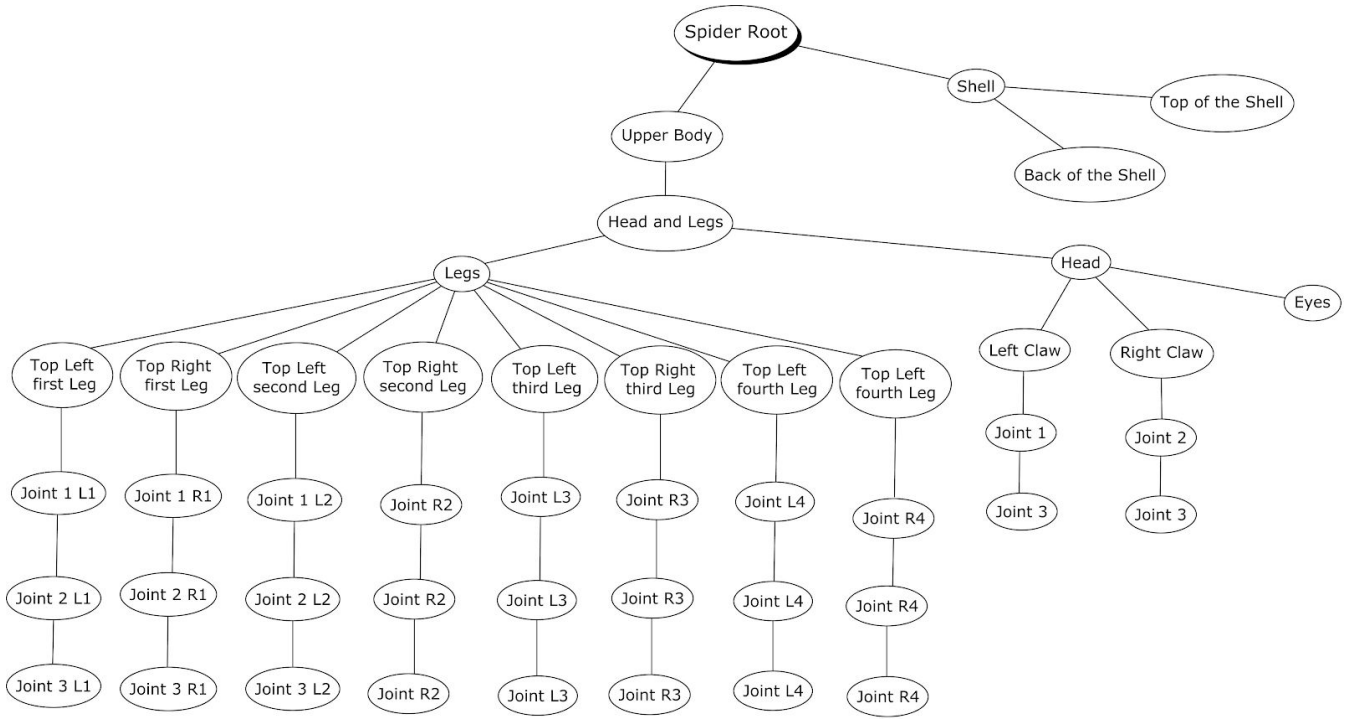
Thus a modality can be viewed as a controller that updates the game logic (more precisely the application logic) through its *update* method, based on the input received through *handleMouseMove* and *handleAction*. A modality is linked to a GUI (a subclass of the GUI abstract class), thus changes are passed back to the client by injecting HTML DOM elements in the page.
Main contains at each time a single modality object as a field and calls its *update* method during each iteration of *mainLoop*.

The available modes are described:

- *LoadMode* is the first one being used and it assures that everything is loaded before starting.
- *MenuMode* presents the game and the user can click a Play button
- *GameMode* is the most important mode since is the one in which one can actually play: it is responsible for acquiring all the inputs while playing, for animating the objects and for checking whether an event occurred which is then handled by an *EventHandler*. The core of the game resides in the *SpiderController* class, where the controller, assumed to be a virtual spider, is moved according to the input commands (it computes the next state in which the spider will be). Moreover, it decides whether to accept a new action by making some checks, per each frame, after which the position of the spider can finally updated. Basically the spider controller contains every animation and also all the necessary controls to be done before these. The core functions of this controller regards updating the walk (and thus all possible rotations and also the climbing down action), the jump (by casting a net), the claws movement and also the turning action of the spider.
- *Riddle*, *Pause*, *Tutorial*, do not render on the canvas but inject HTML so that sentences and buttons are added and placed around through their CSS.

Animations

The model of the spider is loaded from a *.gltf* file. To show its structure a function has been imported from the web to print in a clear way the hierarchy of the object, which, with more familiar names, appears as the following:



Walking animation

The aim is to achieve a realistic animation of the eight legs while walking, so that for each side the first and the third leg (starting from the front) move forward and backward at the same time, and the same happens for the second and the fourth respectively. This movement has been achieved by rotating the top nodes of the legs around their local z axis within an empirically evaluated interval. The increment and decrement of these movements is updated for each walking command and is different for standard walk and accelerated walk.

This first part described allows the legs to rotate around the y axis of the spider; after this the legs are also supposed to raise and to lower down. At this attempt the first joints of each leg are rotated around theirs x axis in a way that when a leg moves forward it starts from the ground, it gradually raises up to a point and it eventually descended on the ground when the leg is as forward as it can get.

This alternated movement is set by two linear interpolating functions evaluated empirically which have the legs passing from the base point to the highest point and vice versa while they keep proceeding.
When the legs move backwards instead, the tip of the leg remains fixed on the ground. Walking backward implies having the same animation but in the opposite verse.

Casting the web

This animation involves both picking a point and interpolating a rotation and a translation of the 3D spider object.
First of all, by using a raycaster, a ray is cast from the camera to the point selected by the mouse, and the first intersection with an object is taken. The intersection allows to access all the information regarding the “touched” face:

- the point on the surface of the object
- the object itself
- the picked face (an instance of the Plane Three.js class) which contains, among its properties the normal vector. This vector is brought to world coordinates by applying the object’s world quaternion.

The spider has to move toward the picked point and to rotate its y axis (pointing upward) so that it will coincide with the pointed normal.
For a smooth animation, the translation to the target point and the rotation of the spider needs to be to done with linear interpolation. The former is easy to do since the initial and final positions are known, while the latter is more complicated because the final quaternion is not known in advance. To solve this problem, the rotation to reach the final orientation is done, the final quaternion of the spider is stored, and, after this, the spider is rotated back to its orientation before the jump. Up to this point is finally possible to apply the quaternion interpolation through the function *slerp* Three.js function.
Another issue regarding the orientation of the spider arises from the fact that its orientation in the final plane has a degree of freedom. To set the final orientation on the plane, a good idea is to consider the vector pointing from the spider to the target point and to project it on the plane defined by the normal of the target face; this orientation will be the final z of the spider, thus its forward direction.

It is not possible to cast web from any place to any point, otherwise the spider might pass through objects: a control is done again with the use of a raycaster, which, this time, is set to follow the vector defined by the initial spider position and the target position. If an intersection is found before the object point with the mouse, then the jump is not done.

Last but not least, a cylinder mesh is created and oriented for the representation of the web thrown and it lasts for just the animation steps. After this, it is removed from the scene.

Rotation animation

The spider, while moving, has to know whether it is about to walk on something with a different inclination, so that it can react to it and rotate in the proper way to keep moving. This virtual spider does not have any eyes to see this but it can, once again, raycast to see where it is about to end up.

A good way to manage this is to set a controller which is “in the future” compared to the spider. This controller receives a movement given by the user, elaborates the next position and sees whether this new position needs a rotation. If so, it evaluates whether the next object is downward or upward.

If the spider is moving forward and it is at the edge of a stair, the controller takes the next forward command and moves in front of the spider; here the controller sends two rays backward. These two rays are displaced of an *epsilon amount* with respect to the virtual position of the controller: one is a bit more upward and the other is a bit downward. The intersections of these rays are considered only if having a distance lower than the actual speed of the spider and they should point out where is the following surface to climb on. In case of conflicts the upward direction is preferred.

Given the result of the raycaster which presents an intersection, the normal of the new object is taken and now the strategy to have the spider rotate is the same as the one adopted when casting the web: its final quaternion is set to have as the y axis the normal found on the intersected object. The interpolation follows the same steps described for the previous animation.

When movements taken from the controller are backward instead, the rays are cast in the opposite direction: from the back of the spider towards it, and, whether an intersection occurs, the spider position is not updated.

Climbing down

The same line of reasoning, which sets the final quaternion of the spider by just knowing where an actual axis of the spider will point, is used for climbing down.
This movement of the spider is feasible only when walking on the ceiling of a room and it allows the user to move with the regular commands thanks to a net which stays still on the ceiling.

Since moving forward and backward is allowed while the spider is hung on the web, the mesh of the net is continuously built and dismantled.

Climbing down animation

Camera

One of the most challenging part of this project was defining a camera able to change in a proper way when the view is obstructed by an object, which is very likely to happen in an indoor environment.

First of all two simple kind of cameras are defined: the constrained camera and the orbit camera. They are both two perspective cameras with a *lookAt* position and a camera position both moving.

The first one places the camera right behind the spider and is set to look right in front of it. The position is updated for every movement of the spider so that it follows the change of its direction. Unfortunately this camera is not well suited for the rotations on the walls, thus when one is happening the camera is automatically switched to the orbital one. Furthermore, apart from the case in which the spider walks on a plane surface, whose normal is the world y axis, the rotations of the camera are not set to follow the pointing direction of the spider.

The second kind of camera is the orbital one, which is placed in spherical coordinates around a moving center: any movement of the mouse defines the angles *theta* and *phi* which rotate the camera with regard to the position of the spider (its center indeed).
The main issue regarding this camera concerns moving around narrow areas like the corridor or when climbing on a seat.

To see whether an object obstructs the camera a ray is sent from the camera position to the the spider position and if an intersection reaches the maximum radius of the camera, it means that the view is not free so the camera is switched to a position which is placed right above the spider; if this position is still obstructed by an object, the position is changed again to the left side of the spider.
This scheme proved to be quite robust for most of the cases found in the environment.
A more advanced system to control the camera would require to cast too many rays which is computationally expensive, since this control ought to be done within just one frame.

Environment

All the environment has been constructed from scratch using Blender in the *world.blend* file. These *world.blend* file is exported with blender to a *.glb* (binary) file that is imported in Three.js with the *GLTFLoader* class.

Geometry

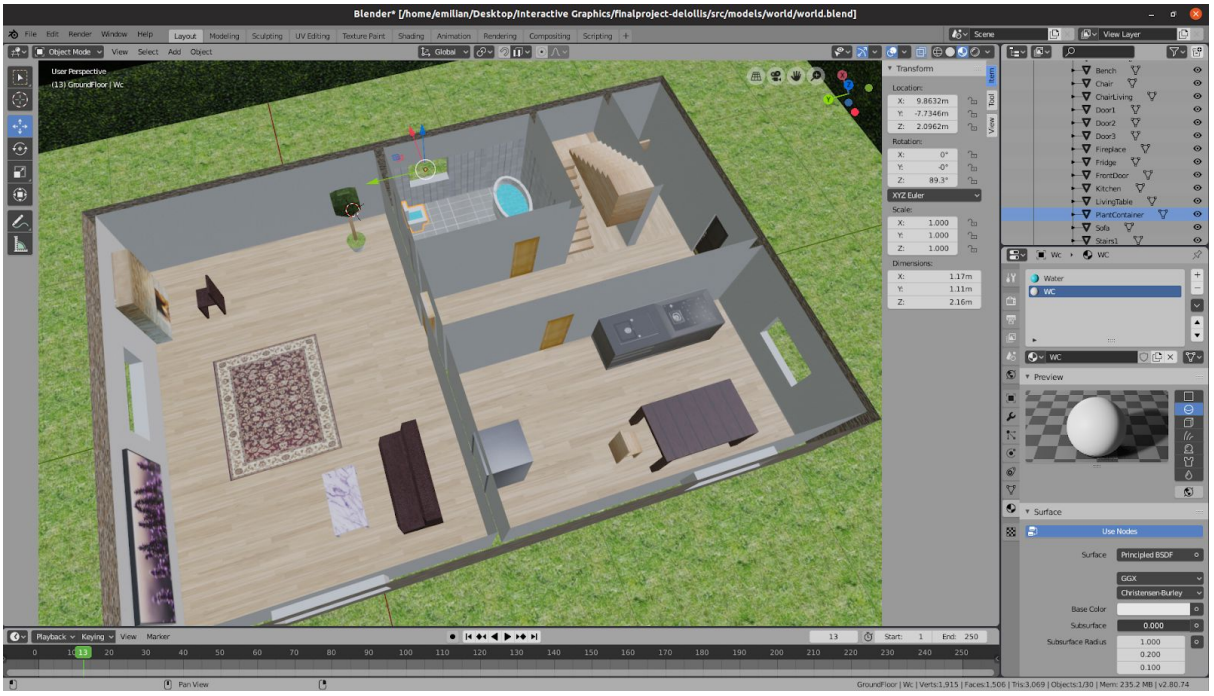
The world in which the spider moves is a one-story house that is surrounded by a garden with a pool. The ground floor is divided in a living room, a kitchen, a bathroom and a hall that connects them all. Instead the second floor is divided in a bedroom, an empty closed room and two magic rooms. A sky-box textured cube has been used to draw the sky.

The ground walls have been built first by using Loop Subdivide on a primitive cube in order to outline the various rooms, the hollow spaces for the doors and the windows. Then the walls were raised up by extruding the regions. The same has been done for the first floor and for the roof. A staircase has been build in order to connect the two stories.

After the walls have been laid down, various simple objects such as tables, chairs, pools, furniture have been created starting from primitive shapes such as cubes or cylinders.

A key model, a fish and a bird have been imported in the file using the Import utility.

Some difficulties regarding the geometry have been encountered when different surfaces where intersecting: they could cause the spider to change edges in an unexpected way (for example to exit a room by entering the space between the walls). These difficulties have been solved by intelligently removing useless faces or by raising them in order to not be detected by the spider controller.
Another inconsistency has been encountered when an object was scaled in a bad way in Blender: this would invert the normals on the surfaces, making the spider rotate in a bad way. This has been solved by reverting back the badly scaled objects.



The ground floor in Blender

Materials and Lighting

A material has been applied to each face of each object. In total, 38 different materials have been created (such as wood, marble, grass, etc.). Each material has a base texture that is modified by a PrincipledBSDF shader. This shader corresponds to a *MeshStandardMaterial* in Three.js and is a physically realistic shader that takes into account lighting properties such as:

- Metallic
- Specular
- Roughness

These three parameters were manipulated in order to create the various materials. For example grass has low metallic and specular values and high roughness while marble has small roughness and high specular.

The sky-box has been assigned a *MeshBasicMaterial* with Three.js since it doesn't have to be affected by specular or diffuse lighting.
The glass material has a reduced opacity since it must allow to see through the surface.
Two lights have been added to the scene: an ambient light and a point light. These two lights generate a view that albeit not realistic is fun and in theme with the game.

Observations

Developing this game pointed out many bugs and some cases in which the implemented physics is not robust.

As said before, when checking and edge in front of the spider an issue occurs when walking on a surface which hide under it, another surface.

The *epsilon* chosen to cast the upper and the lower rays to detect a collision or a rotation is crucial: if it too high small changes (like a tiny stair) are not detected, while if it is to little, it might intersect the same surface.

A big issue for dealing in a proper way with normals and rotations around the entire environment is working with world coordinates, by taking the world quaternion from the objects to be sure of keeping everything expressed with regard to the same reference frame.

It is worth mentioning that the use of very basic geometries in the entire house (with the exception of few weird objects) does not lower down significantly the frame rate and the initial loading of the world and of the spider is pretty fast.

References

https://threejs.org
https://threejsfundamentals.org/threejs/lessons/threejs-load-gltf.html
http://jblaha.art/examples/characters.html
https://infinitespider.com/spider-legs-work/
http://stemkoski.github.io/Three.js/Collision-Detection.html
https://css-tricks.com/using-css-cursors/

A special thanks to:

Bruyere for the model of the spider
https://sketchfab.com/guurme

Guido for the model of the fish
https://sketchfab.com/guidotondo

a80825 for the model of the chick
https://sketchfab.com/a80825

Phylip Combs for the model of the key
https://sketchfab.com/pcmonster