Interactive Graphics Final Project

Animation of a Transformer Robot in a Farm

Lorenzo Papa 1699806 & Nicolas Zaccaria 1904366

July 13, 2020

1 Introduction:

The project that we have developed is an interactive animation that has been made using the potentiality of the recommended libraries; *three.js* and *tween.js*, to learn how they work. The first library was used to manage hierarchical models and the whole scene while the second one to have smooth animations, as we will see in the following chapters.

The simple story behind the animation finds a transformer robot as the protagonist; we have chosen this type of robot to show the complex structure used for its transformation, from robot to cube and vice versa. It with the help of the User will have to find the parts of its spaceship, hidden in the environment, a rural scene, to repair it and then to be able to leave.

The realized set, may recall one of the first scenes of the film *"Transformers 4 - The era of the extinction"*; in which the protagonists find in their barn a transformer, and then they help it to escape from the arrival of the army.

Before analysing the implementation, let's quickly see the home page that the user will encounter. This is a simple guide to the game which contains all the commands and information needed to the user for the interactions

with the different components. Also in the center, there is a circular button on which is represented the face of our robot, it is the *"start"* button for the game. All those information can also be recalled by the user via the menu present on the top right part of the screen

There are two important features to note: the first one is that all the images were made by us and loaded as vectorial (*.svg*) in order to better adapt to the user's screen. The second is that the start button when pressed rotates and emits a sound similar to that of an explosion referring to the accident of the starship. We have to say that all the sounds present in the interactive animation are downloaded by *Freesound*[1]; as the one just introduced.

We can now proceed by analysing the implementation; it is organized into the following chapters:

- The GUI: in which we briefly describe the user interface we implemented.
- Hierarchical models: in which we focus on the three main models, the robot, the environment and the starship
- Interaction System: in which we explain all the possibilities that the user have to interact with the animation
- Collision System: in which we explain and show why is needed, how it is implemented, with a short comparison with the possible different approaches
- Animations: in which we analyse the concept behind all the animations for the robot, the starship, environment and the day/night cycle.
- The solution of the game: in which we describe the possible solutions to complete this interactive animation.

[1] https://freesound.org/

2 The GUI:

The GUI was created using the specific module of the *three.js* library, called *dat.gui.module.js*. We chose this menu for its simplicity and easy integration with the rest of the code. The menu contains five elements; the first three: *Help*, *Commands*, *Hints*, are used to help the user.

These perform the following operations: the first, already shown when the loading of the game is completed, recalls the short introduction where is described what happens and what the user has to do; the second recalls the commands already seen in the first screen and, the last one, is a list with a possible sequence of things to do to complete the animation. The last button, *Exit*, allows you to terminate the game and return to the home page.

We also decided to leave an entire sub-section of the debug menu; this to give the possibility to the user to better understand the different functions implemented and their effective functioning, those are also well explained in the following chapters.

To be noticed the possibility to activate and deactivate the two collision systems; also with the activation of the *"Nerd parameters"*, the user will have the possibility to visualize the fps and the two arrows represented the rays used for those two collision systems. The other commands are instead for simpler functions such as removing or activate the audio, the shadows, to make the animation more fluid, if needed, and to stop the day/night cycle.

Problem: We noticed that using a laptop, in some cases, the animation will not be really fluid probably because when the battery is low il will automatically reduce the performances.

Solution: Also, for this reason, we decided to keep the debug menu. In fact, if it will happen we suggest checking if there are at least 20fps; otherwise, the user can simply plug in the pc or directly from the menu removing shadow and stop the day/night cycle.

3 Hierarchical models:

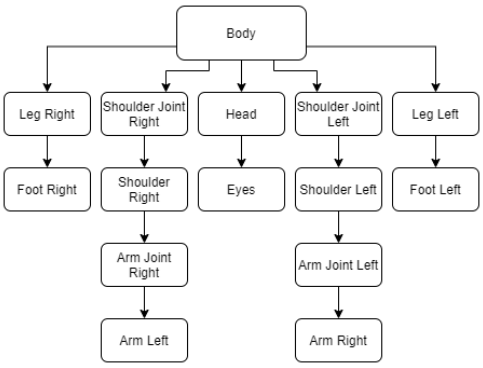
In this chapter, we will introduce the three hierarchical models used: the robot, the starship, and the environment; but before entering the merit of each of those parts; we have to say where we have taken and how we modified them to better fit in the project.

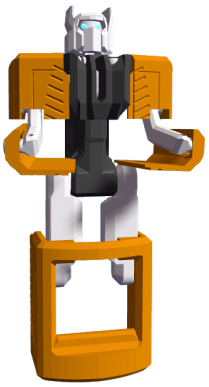
The model of the robot composed of all its parts was downloaded from *"GrabCAD Community"*[2] and subsequently modified by us using *SolidWorks*.

The environment model was instead downloaded from *"Sketchfab"*[3] and subsequently decomposed on *Blender* to use it as a hierarchical model.

Finally, the model of the starship was entirely developed by us in *SolidWorks*; to place the robot, in the closed position, inside for the final animation.

The hierarchical model of the Robot is composed of sixteen different parts where the root node, as we can see in the image reported below, representing its structure, is the body. An important note is that to simplify the positioning and the motion of all the components, for the different animations, the reference system of each of them is placed at the exact point where we expect that it will rotate.

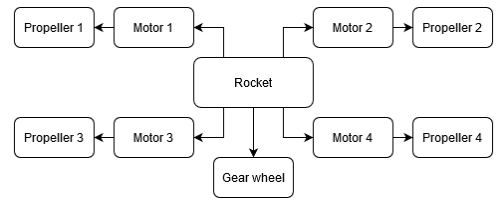


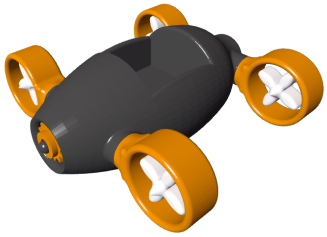


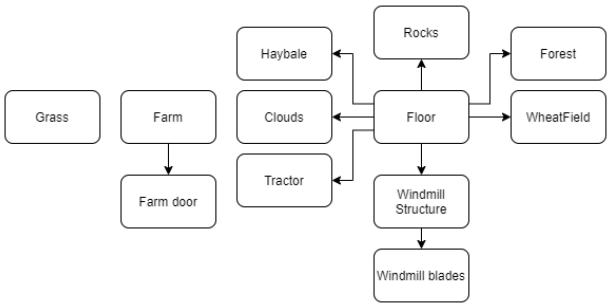
A similar process is done for the hierarchical models of the spaceship and the environment; the first one is composed of ten elements where the root node is the main structure. The second one is composed of twelve elements where we have used three different main nodes. As for the robot, their schemes are reported in the images below.

[2]https://grabcad.com/library/makerbot-transformer

[3]https://sketchfab.com/3d-models/low-poly-farm-879d61d8dfc048548ee380cace6f79d3







To conclude, each root node of those three main elements is setted as child of the scene that contains all the parts that have to be rendered.

After this long introduction to the three hierarchical models, we can now analyse the code. Each model is defined as a dictionary containing the different parts; also, each part is composed as a dictionary composed by its specific properties such as position, rotation ...

Each model is then loaded into the scene through a specific loader, which after loading the mesh, will attribute the specific properties above introduced. There are two types of loaders used: one for *gltf/glt* format files, used for the environment components, and the other for *stl* format files, used for the robot and starship parts.

To conclude, after each 3D element is ready, those are called in a traverse function to define their hierarchy; thus defining the hierarchical model.

We would like to notice that the initial loading phase that starts the game, usually, with a short duration, ends only when all three models just introduced are defined and completely loaded. To do this we use the *loading manager* made available by the used library, *three.js*. This avoids undesired effects during the loading phase.

4 Interaction System:

In the beginning, we imagined a simpler project where the user can interact with the elements in the scene by clicking on some interactive objects and activating some scripted animation implemented for that specific object. During the coding phase, we started to imagine a more complex game structure where the previous interactions are kept with the possibility to make the robot walk at any desired point.

We started the implementation focusing on the camera interaction. We decided to use the *OrbitControls.js* library from *three.js* to implement a camera orbiting around the center of our scene that is perfect to explore the environment. We decided, also, to disable the pan movement and to limit the min and max distance to fit our idea of camera movement. The camera controls include the possibilities to rotate the camera holding down and dragging the right mouse button, the zoom in and out functions, scrolling the mouse wheel, and the panning by holding down and dragging the left mouse button if the option is enabled from the debug menu.

We moved, then, on the interaction with the environment, the objects and the robot. There are three possible interactions:

- Single click on an interactive object
- Hovering above an object with the mouse pointer
- Double click on the environment

The single-click required to register the mouse down event, to get the coordinate of the click, and the mouse up event to check the amount of time that the click gets. This is necessary to distinguish between a single click to interact with an object or a click and hold to move the camera if the pan is enabled. To do it we have defined a threshold value; if the time is lower than this value, the event is considered as a single-click, a ray is cast from the mouse coordinates in the canvas and with the same direction of the camera. At this point, we check if the ray intersects an object belonging to a list that contains every interactive object existing in the scene at that moment. If the intersection happened then the callback function defined for that specific object is called.

To help users in this interaction system, we added a hover interaction that checks the mouse position at each cycle, casts a ray as we did for the

single-click command, and if an intersection happens between the ray and an object in the list of interactive objects then the emission of that object is changed to highlight it. When the mouse pointer is moved away from that object, it returns to its original emission value. In this way, the player knows always if he is pointing an object in the right way before clicking on it and he is helped in founding key elements in the scene.

The last interaction is the double click used to make the robot move. As before, it is based on the use of a ray to compute the intersection between the ray that is cast from the mouse position and the floor. Once the intersection point is known, we compute the angle and the distance between the robot position and the destination point; while the angle is used to make to robot turn around toward that point, the distance to define the time the robot will take to get to that point, making the speed of the walk animation always the same.

5 Collision System:

We decide to give the possibility to the user to move our protagonist in any place of the realized scene; so a collision system is required.

Unfortunately, *three.js* marked as deprecated the libraries used to compute it. So we tried some different approaches starting from the bounding-box, a physic library, *physics.js*, and using ray-caster.

The first method didn't fit our case since we have some model, like the trees or the rocks, that are modelled as a single object that covers the majority space of the environment. A bounding-box in this case would make impossible any movement in the scene since a collision would always be reported.

The second system would be better but it can't be used because we want to introduce physics only for the collisions, and not for the whole animation since our project doesn't require a complex physic engine also avoiding heavy computational cost. Another problem is that the physic library doesn't cooperate well with the *tween.js* library we used for the animation since this last one update at each cycle the values of the vector used to express, for example, the position of an object while a psychic engine computes animation in terms of force, acceleration, velocity, etc...

With all of this informations; we decided to implement a collision system based on ray-caster. The main concept is that: since the robot always rotates before translating, collision happens mostly[4] in front of the robot. So we decided to implement a ray-caster that has an origin point in the robot body and directed accordingly to the body rotation. Also, due to the characteristics of our scene, we decide to make the ray pointing not just parallel to the floor but a little farther than the robot feet instead so it can detect collisions with low height obstacles such as rocks. As a final improvement, we made the direction of the ray oscillating along the horizontal plane, based on a cosine function, to detect, also, collision with objects placed more to the side of the robot.

As we can see from the images reported below: the first one shows the detection of the collision, this can be seen by the red arrow, it will change the color from black to red when a collision occurs. The other two images are correlated; because the second one reports the final condition of the robot, it will stop and rotate on its z-axis to avoid the obstacle and the third one

[4]In lower cases happens not perfectly in front of the robot but a little bit on the side

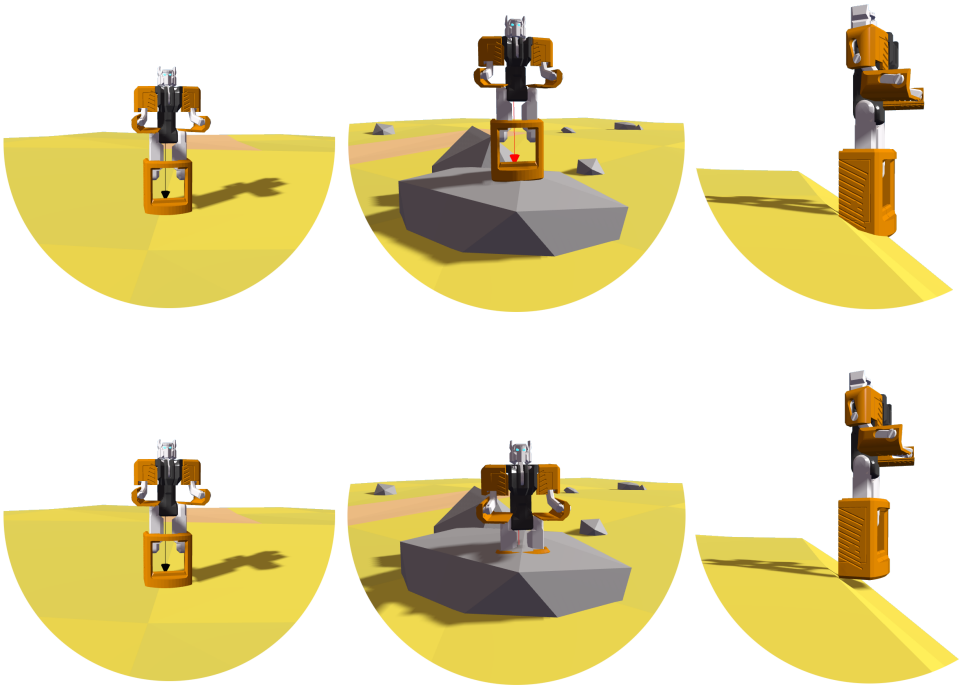
reports the case in which the system is not active, in this case, the robot will pass through the objects. From these last two images, we can appreciate and justify the implementation of this system.

Also since the floor of the scene is not flat but has some bumps we decided to implement a vertical collision system with the floor to avoid the robot going through the ground. This system is modelled starting from the collision system with objects, just explained; but in this case, the ray direction is fixed from the body of the robot orthogonal to the floor. When the distance between the robot body and the intersected point of the floor is less than a minimum threshold then the y-coordinate of the robot position, representing its vertical translation in the scene, is incremented, if the distance is higher than a maximum threshold the y-coordinate is decreased, if it is between the minimum and maximum then the value is correct.

When the robot is in the cubic stance, after the transformation, to define its height with respect to the ground we use the coordinates computed in the previous stance, and then those are lowered by a constant value equal to the length of the legs plus the feet. The inverse process is also applied in the reverse transformation: from cubic to stand stance.

We used, for the floor collision system, the distance measured from the body and not from the feet since they are expressed in function of the respective nodes in the hierarchical model. So computing the distance from the feet requires some extra computations to switch between each reference frame.

We decided to report the six images above to compare the floor collision system just described. The images on the first row show what we obtained when the system is active, unlike, the ones on the second row, show the same



situations when we disabling it. Again, those are reported to show why this system is needed and justify the implementation.

As can be seen by observing the first column of images, those represent the robot in a condition in which the system is not necessary, in fact, the two images are equal and its contribution is null. The second column, on the other hand, shows the case in which the floor is higher than the default condition, in this case, the arrow represents the ray that calculates the distance turns red, underline to the user that the robot has risen; if instead, it was disabled, an intersection between the ground and the robot would be noticed. The last column, on the other hand, reports the case in which the robot is moved downwards following the path of the terrain; in this case, the arrow will turn blue. The images and results obtained justify the implementation of this system.

We can conclude this chapter by saying that in its simplicity the combination between these two rays provides for avoiding unpleasant visual effects of interpenetration and simulating in a very elementary way, a simple physical system needed for this type of project.

6 Animations:

As introduced in the first chapter, we decide to use *tween.js* to implement the main animations. This library is based on the creation of tween objects. Each one is based on a main function that takes two vectors and a scalar: the first one represents the actual position, the second one for the final position, expressed in xyz-coordinates, and the scalar that represents the duration of the animation. Then the created object automatically modifies the values of the actual position at each render cycle until the complete animation is performed in the established interval of time.

The library also includes some extra useful methods; all of those that are reported below has been used in our implementation:

- Set a number of repetitions for the specific tween
- Set a delay
- Enabling the backward[5] animation
- Create a group of tween objects
- Write a specific function that will be invoked at the end of the tween animation.

Observing the written code, due to the complexity of our main model, the robot, we had to define the position and rotation for each of its components; a similar process was also done for the starship and the environment. For this reason, we decide to create a JavaScript file, for each hierarchical model, made by one or more dictionaries containing the final position and orientation of each part for each animation.

Subsequently, to simplify the organization of the code, we decided to create other files where we have defined all the animations with their tween objects and specific attributes. In this way, to play each animation, we just need to call the relative function.

Before analysing the animations in more detail, we have to remember that *tween.js* was used only for animations that required a specific initial and final position such as the robot movements, the day/night cycle, etc...

[5]From the final position to the initial one

For the remaining ones, such as the rotation of the windmill blades, the translation of the clouds, the opening or closing of the door, etc... we decided to simply increase or decrease the value of the variable relative to the reference axis in the coordinate of interest.

The main animations that we will analyse are those related to the robot, the starship, and the day/night cycle.

The first one, are generated every time the user wants to interact with the protagonist. These refer to the following actions: the transformation from robot to cube and vice versa, the walking phase, the recovery behavior if a collision occurs, the grabbing of the objects, and the final animation to jump into the starship. Each of these actions is often obtained as a concatenation of others invoked through the methods provided by the library, *onComplte*(); obviously, but very important, they are activated only if all the conditions, defined as boolean variables, are satisfied.

For example, the action of walking that is performed only if the robot is standing, when the user double-clicks on the ground, is characterized by a sequence of phases: the first is when from the *default stand* position the robot moves in the *start walking* position, setting arms and legs with opposite values, then it starts the real *walking* phase until it reaches the desired position; this thanks to the *yoyo*() and *repeat*() methods. Then, when the action is completed it automatically goes back into the *default stand* position.

In the meantime, the collision system is also used, if active. If a collision occurs, the robot is stopped in its current position, and all the generated tweens are eliminated, so as to deactivate the other actions; finally it will rotate on its z-axis until it is free to move in the surrounding area. Also in this case, at the end of the animation, the robot is setted in the *default stand* condition.

Observing the code, we can also notice that some actions incorporate sounds, such as when the robot takes an object. Their reproduction is performed through the library functions, *audioSystem.js*, which can be easily called at the end of the action as previously described.

The same behavior and techniques are also used for the other actions of the robot, introduced above in which we will not go into detail to avoid repetitions.

With the same principles, the animation of the starship has been created. For example, since its simpler structure, the rotations of the propellers are not managed via tweens as the windmill blades. The tweens are used only for the translation from the actual position to a final one, out form the view

volume, once the robot is inside the starship to end the game.

Probably more interesting as behavior is the day/night cycle; also obtained through the use of tween. In this case, however, the value given as input to the function are not the xyz-positions but the values in RGB used for the lights and for the background color. Specifically for the night, we have tried to reproduce the effect given by the light reflected by the moon.

7 Solution:

To conclude this report we describe a possible solution to the animation. There are two objects to be found, the order in which objects can be taken is indifferent: the first is hidden near the trees, behind the windmill, while the second is placed inside the silos, that to be taken the user must first open the door, and then it will be visible. Remember that the interactive objects will start to blink when the robot is close enough or it will remain lighted when the user is hovering on it.

We decided to hide only two objects, to not be repetitive since we want to pose our implementation on a didactic level.

Subsequently, piece by piece the spaceship will begin to move until its complete reconstruction. The robot will then be able to jump into the starship, which, in this moment, it will become an object with whom the user can interact[6]. To conclude, the robot will sit in the starship, we can notice that its body will start to flash; once the user has clicked and the robot is closed, the animation of the launch sequence will start. After the animation the game will end.

[6]It flashes/lights up if the user is pointing it with the mouse