Final Project - Report

Interactive Graphics 2019-2020

Andrea Caciolai
M. Sc. in Artificial Intelligence and Robotics
caciolai.1762906@studenti.uniroma1.it

Contents

1 Introduction

For the final project of the *Interactive Graphics* course, I decided to implement a simple archery game.

1.1 Theme

The user plays as Link, the main character of the famous videogame franchise *The Legend of Zelda*. Link is an archer and is practising his skills by shooting at a target. The target can move, depending on the selected difficulty, and Link has to hit the target in motion. Link's arrows are magical, thanks to the legendary *triforce*, and if he hits the target, the arrow respawns in Link's hand. However, for each missed shot, he has to get a new arrow, and he only has 3 spare arrows left, to try and get the maximum number of hits he can.

1.2 User guide

When starting the game, either being hosted on *GitHub Pages* or locally by running a *localhost server*, the user is shown the game menu. In this very simple menu he is required to choose the difficulty:

- *easy*: the target is still,
- *medium*: the target moves in a linear horizontal pattern,
- *hard*: the target moves in a 8-shaped pattern,

then he can choose between playing in daylight or moonlight. He can also review the instructions, that is just a summary of this and the previous section. Finally, by hitting the `Play` button the game starts. The user can move around with `WASD` and control the camera with the mouse, with which also aiming and shooting are controlled.

More in detail, here are the available controls:

- `W`: move Link forward.
- `A`: move Link to the left.
- `S`: move Link to the right.
- `D`: move Link backward.
- `Mouse`:
  - While in third person, move the camera around Link.
  - While in first person, aim.
  - While nocking, move the bow up/down and also rotate Link in place.
- `MouseLeft`:

  - Press: start nocking the arrow, the more the button is pressed, the more the arrow is nocked.
  - Release: shoot the arrow.
- `MouseRight`:
  - Press: aim (go in first person).
  - Release: stop aiming (go back in third person).
- `ESC`: pause the game, click anywhere on the screen to resume the game.

1.3 Libraries

The project is implemented in WebGL, as per project requirements, with some HTML and CSS code to set up the webpage environment and style, the menu logic and some very simple in-game interface (pause and game over screens, counter of arrows left).

Then, my project relies intensively on the following libraries.

Three.js I decided to use the Three.js library [19] in my project. I used [20] as reference. Three.js is an open-source JavaScript library, built on top of WebGL, that allows to create and render 3D scenes with more ease, providing an extensive API with a large set of functions.

Tween.js Tween.js [21] is a simple but powerful JavaScript library for *tweening* JavaScript properties. Tweening is a shorthand for *inbetweening*, a key process in all types of animation. It is the process of generating intermediate frames between two keyframes, to give the appearance that the first keyframe evolves smoothly into the second keyframe. The programmer sets properties that are to be interpolated between the initial and final keyframes, specifies the interpolation time, and then Tween.js produces the frames in between correctly interpolating these properties.

2 Models

In this section, I will describe the models I used in my project.

In my project, I used 4 main models, plus some others for mere decoration of the scene: Link, the bow, the arrow, and the target.

All the models have been downloaded from the web, then adjusted in some way in Blender [6] and finally exported in the WebGL-compatible *GLTF* format (GL Transmission Format) [9] to be imported in my project.

2.1 Link

In Figure 1 you can see the 3D model of Link I used, downloaded from Sketchfab [1].

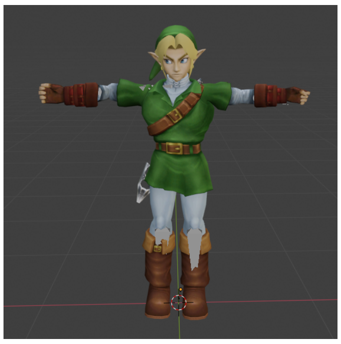
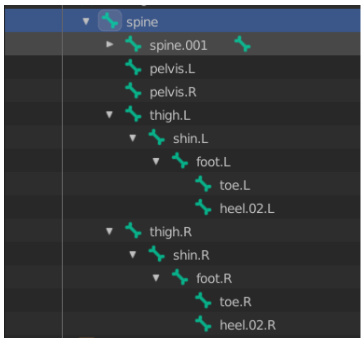


Figure 1: Model of Link in Blender. Notice the bones.

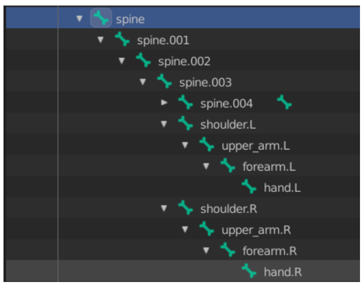
The model came with an animation that I discarded as per project requirements, and although it already had bones to animate the model, they were poorly placed, so I *rigged* the model from scratch in Blender.

Rigging is the process of associating a deformable armature (or *skeleton*) to a mesh, that can then be deformed along with the armature. The skeleton of the model is what makes the model *hierarchical*: each bone beside the root one (the spinal bone in the waist) has a parent, and its local space is affected by the transformations (position, rotation, scale) of its parent.



```
spine
  spine.001
  pelvis.L
  pelvis.R
  thigh.L
    shin.L
      foot.L
        toe.L
        heel.02.L
  thigh.R
    shin.R
      foot.R
        toe.R
        heel.02.R
```

(a) The lower limbs are connected to the main bone.



```
spine
  spine.001
    spine.002
      spine.003
        spine.004
        shoulder.L
          upper_arm.L
            forearm.L
              hand.L
        shoulder.R
          upper_arm.R
            forearm.R
              hand.R
```

(b) The upper limbs are connected to a more nested spinal bone.

Figure 2: Hierarchical structure of the model as shown in Blender.

In the importing process (done with the utility class `GLTFLoader` [10]) Three.js ensures that the imported mesh will be a `SkinnedMesh` [18], with a `Skeleton` [17] object with the correct bones associated to it. In this way, by performing spatial transformations (e.g. rotation) to a `Bone` object [7], according to the way the model has been rigged, the vertices in the geometry of the `SkinnedMesh` will be displaced accordingly. In Three.js this will result in a scene graph (shown in the console logs when loading the models as a tree structure) having a `Object3D` [11] as root, then the skeleton as a hierarchical collection of bones, then one or more `SkinnedMesh` objects, with the actual geometry and material.

Actually, to avoid having to carefully match the position of Link's hand with the bow, I import the two models as a single model, so that the bone corresponding to Link's left hand will have the bow `Object3D` as a child in the scene graph.

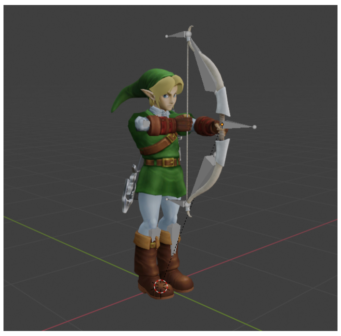


Figure 3: Link with bow in Blender.

Joints The *joints*, or bones, allow moving the models by changing their orientation relative to the parent joint, properly deforming the attached skinned mesh. When loading the model of Link, I create a data structure (see Figure 4) to hold every joint that I will need to tweak to animate Link.

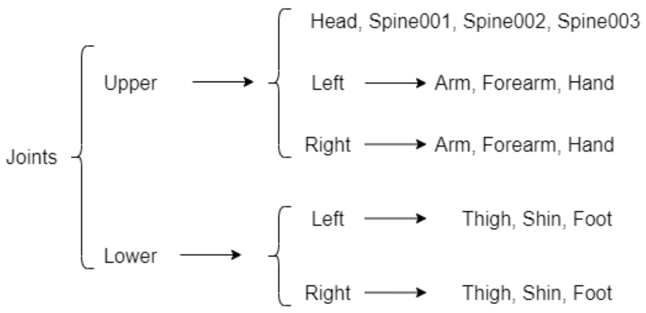


Figure 4: Link joints data structure.

2.2 Bow

In Figure 5 you can see the 3D model of the bow I used, downloaded from [3], a link that I found here[1].

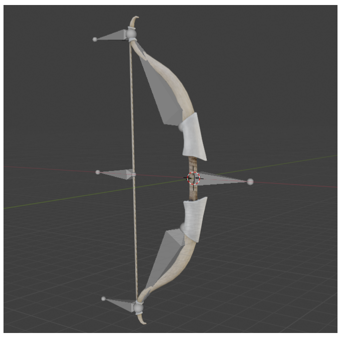


Figure 5: Model of the bow in Blender. Notice the bones.

The model came with a cleverly designed skeleton (so it is hierarchical) that allows the bowstring to be pulled backwards to nock an arrow, and also the bow itself bends backwards if the pulling exceeds a certain threshold, as you can see in Figure 6.

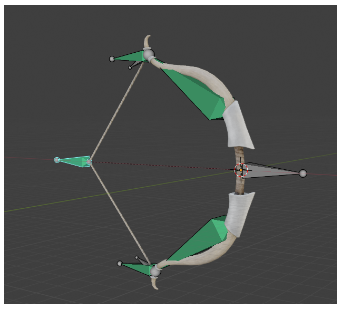
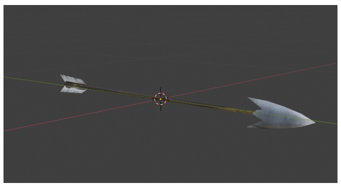


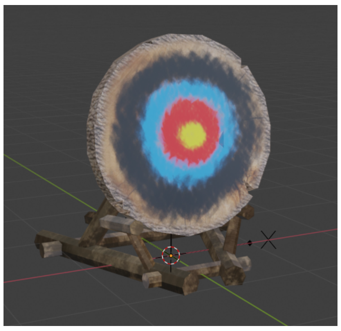
Figure 6: Deformed bow.

The hierarchical structure of the bow will allow a smooth *nocking* (charging the arrow to be shot by pulling it backwards along with the string) animation, that leverages also Link's hierarchical structure (see Section 4.2).

[1]https://www.youtube.com/watch?v=jpsd0Aw1qvA&ab_channel=CGMasters

2.3 Others


(a) Arrow model in Blender.


(b) Target model in Blender.

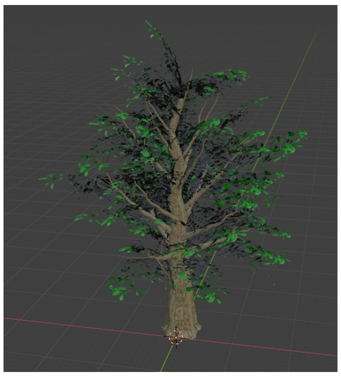

(c) Tree model in Blender.

Figure 7: Other models.

Arrow The arrow in Figure 7a is not a hierarchical model, since it does not require a deforming animation (although it will be animated). I downloaded it from Sketchfab [2]. The model comprised also a bow, but I already had one so I only kept the arrow.

Target The target in Figure 7b is not a hierarchical model, since it does not require a deforming animation (although it will be animated). I downloaded it from Sketchfab [4]. I modified the mesh in Blender so that the object comprises two separate meshes, one for the base of the target and one for the actual target, since I will need to register the collision of the arrow on the target only, not the whole object.

Tree The tree in Figure 7c is not a hierarchical model, it is included just as a decoration for the scene. I downloaded it from Sketchfab [5].

3 Scene

In this section, I will describe how the scene in the game is set up.

3.1 Three.js

To render a scene, Three.js requires the user to define a *scene graph*. A scene graph in a 3D engine is a hierarchy of nodes in a graph (or rather a tree) where each node represents a local space. This is what we did explicitly in the second homework, but by using Three.js we do not have to worry about defining the explicit transformations from parent to child space, and performing explicit tree traversals to render a hierarchical object since the library does all that work in our stead.

So, one starts by defining a `Scene` object [16], that will hold the scene graph, and then can add objects, lights, cameras to the scene, either by adding it to the scene object, so that they will be an immediate level below the root node of the graph, or also to other objects already added to the scene, so defining a deeper graph. `obj1.add(obj2)` will make `obj1` the *parent* object of `obj2`. As such, `obj2` will have a `matrix` property that is the (4×4) *local transform matrix*. Each object will also have a property `worldMatrix` with the *global transform matrix* and a property `modelViewMatrix` used by the shaders to render the scene from the current camera.

While in the previous homeworks we had to carefully define all these properties by ourselves, now it is as simple as adding an object to another and then displacing and rotating it, letting Three.js update the matrices properly.

3.2 Scene graph

My scene comprises several objects, 2 lights and 3 cameras. Of the objects, some of them (the trees) are just displaced and rotated copies of the same object, so share the same geometry and material. In Figure 8 you can see (a reduced version of) the scene graph of my project.

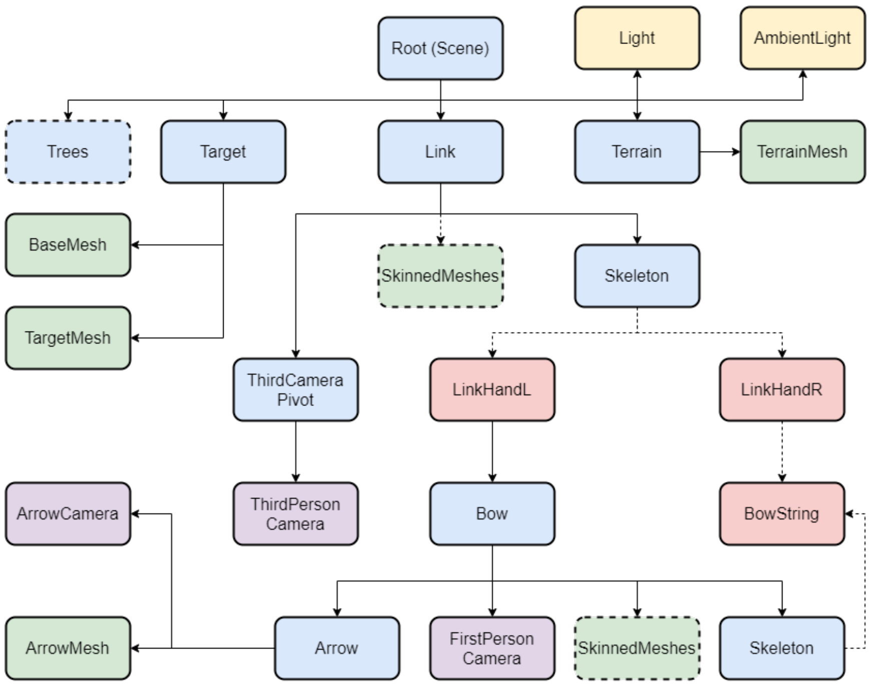


Figure 8: The scenegraph of my project. The nodes in blue are Object3D (or similar entities), in red are Bones, in yellow Lights, in purple Cameras. The dotted nodes represent more than one node (e.g. several trees), while the dotted arrows mean that the child node is not an immediate child of the parent node, but only a descendant.

In the game, all the models are located on the terrain mesh, a simple square plane mesh, with a (square) image texture of some grass.

3.3 Lights

There are two lights illuminating the scene: a directional and ambient light.

The directional light represents either the light from the sun or from the moon, that are actual objects with a very simple spherical mesh. The direction is therefore just the distance vector from either the sun or the moon to the origin of the scene.

If the user chooses to play in daylight, then both the ambient light and the directional light will cast simple white light, both specular and diffuse; otherwise, they will cast a very dark shade of blue. In this latter case, the directional light will be five times less intense, while the ambient light will be twice as intense, to simulate the illusion of the light coming from a starry sky.

Shadows The directional light casts shadows, and all the rendered objects are able to receive and cast shadows.

Three.js by default uses *shadow maps*. As we know, this involves rendering the scene from a camera at the light source, storing the resulting depth buffer (shadow buffer) and then for each fragment (of an object that receives shadows), transforming its depth to light space and comparing the distance to the corresponding location in the shadow buffer, to understand whether the fragment is in shadow or not. This is a costly process since it requires $n \cdot m$ renderings for n objects and m light sources, so often one turns to fake shadows. However, this is not a problem for my project, since I have just a few objects and two light sources, with just one of them casting shadows.

Since to render shadows an (orthographic) camera is implicitly defined, to properly render the shadows of all the objects on the scene I had to properly set the view frustum of this camera, i.e. the top, bottom, left, right properties are set as to match the dimensions of the terrain.

3.4 Cameras

There are 3 different cameras to render the scene from. They are all perspective cameras, with 45° field of view, standard aspect ratio and a view frustum such that all the scene can be rendered from any of them.

As I explain in Section 5.2 the user controls what camera is used as the current camera to render from in the render loop.

Third person camera When starting the game, the user sees the scene as rendered from the third-person camera. This camera, as you can see in Figure 8, is added to Link but indirectly, through a *pivot* object. The camera looks at this object, that is placed approximately at the position of Link's neck, while the camera is positioned a little higher and further away along the z-axis.

As explained in Section 5.2, the user controls the rotation of this pivot, and so I can avoid performing cumbersome calculations to have the camera orbiting Link (basically updating the camera position in spherical coordinates, but also offset), but instead Three.js does that for me.

First person camera When aiming, the scene is rendered through the first-person camera. This camera is added to the bow, that in turn is added to Link's right hand, that the user controls indirectly by moving Link's arms so that the user can see along the same direction along which the arrow will be shot.

Arrow camera Since the arrow is a relatively small object, it was hard to see where it actually landed. Also, I thought it would be fun, so I added a third camera on the arrow, to view in first person where the arrow will land.

3.5 Textures

I personally included only one colour texture, specifically of a patch of grass, repeated along both texels coordinates 32 times to cover the whole terrain and give the illusion of a continuous clearing in a forest.

The other textures used in the project are the ones already included with the models' materials.

4 Animations

In this section, I will describe what objects move and how in my project.

4.1 Link walking

Although in an archery game it is not essential, I decided to grant Link the ability to move. Movement is controlled by the user (see Section 5.2), and it involves both translating Link on the map and also some animations of its limbs so that it does not look like it is floating around while remaining still.

Moving Link around Making Link move is quite straightforward. Motion involves changing the position of something, and like every `Object3D`, Link has a position vector $\mathbf{p}$ (`.position` property, implemented by Three.js with the `Vector3` class [22]) associated to it. So, I keep track of another vector, the *velocity* vector $\mathbf{v}$, and update the position vector simply by

$$\mathbf{p} \leftarrow \mathbf{p} + \mathbf{v}dt \tag{1}$$

where dt should be an inifinitesimal time, in practice is the time interval between two consecutive render iterations.

The velocity vector in turn is updated with two rules:

$$\mathbf{v} \leftarrow \mathbf{v} + s\hat{\mathbf{d}}dt \tag{2}$$

$$\mathbf{v} \leftarrow (1 - \mu dt)\mathbf{v} \tag{3}$$

where $\mathbf{d}$ is the direction vector, s, μ are parameters, namely Link *movement speed* and *ground friction*, so that if the user stops holding down the movement keys (again, see Section 5.2 for details), Link's motion stops smoothly.

Immediately after the position update, the position vector is clipped back into the region of the map, to avoid going beyond its prescribed borders.

Animating Link movement I implemented a very simple walking animation, leveraging Link's joints and Tween.js. I identified a walk "step" as divided into three phases, starting from the rest position: right leg goes forward (*forward phase*), left leg goes forward (*backward phase*), back to rest position (*rest phase*). However, we do not walk like robots, so we also move other parts of our body in the meantime. So, I defined two sets of three `tweens` to be chained one after the other to implement a walking step. Each `tween` interpolates values for the rotation about the "principal axis" of Link's joints between the initial value and final value for the phase, with the 2nd one starting from the final values of the 1st, and the same for the 3rd and the 2nd. The first set implements the animation for the upper body, while the second set for the lower body.

The animation consists, for each involved joint j, that in Link's rest pose would have angle ϕ_j about its principal axis, in this joint being rotated to a certain angle $\phi_j + \theta_j$ in T_1 time in the forward phase, then to $\phi_j - \theta_j$ in $T_2 = 2T_1$ time in the backward phase, then finally back to its rest position ϕ_j in $T_3 = T_1$ time. The involved joints are: thighs, shins, feet, obviously, but also the arms and forearms to emulate the movement with which we are used to "balance" our walking, and also the third spinal bone to emulate a slight rotation of the torso during the walk. Finally, since we tend to keep looking straight while walking, I "undo" this joint rotation by rotating the head back of the same amount.

The third `tween` chains back to the first so that we have an infinite loop of animation. By calling `tween.stop()` on the considered tweens, I can stop the animation when the motion stops, namely when Link's velocity falls below a certain threshold.

4.2 Link nocking the arrow

The main animation for Link is the *nocking* animation. To nock means to prepare the arrow to be shot, by pulling the bowstring, to which the arrow is fixed, backwards, so that when released by the archer, the elastic bowstring will spring forward, accelerating the arrow with it.

Since, as you can see in Figure 8, the bow is a child of Link's left hand, and the bowstring is a child of Link's right hand, I can perform this nocking operation literally. Well, actually, I somewhat cheated since the arrow is not a child of the string, as it should (see Section 4.3). Nonetheless, Link's upper joints are involved to make this nocking happen. I have just a single `tween` for this purpose, that interpolates joints rotation values between the rest pose and the pose that Link would have in the position of "maximum nocking". In particular, I rotate Link's torso clockwise and extend his right arm, while keeping the bow fixed with the other arm (this implies adjusting the left-arm angle wrt to the torso, that is being rotated in the meantime).

4.3 Arrow flight

As you can see in Figure 8, the arrow is not a child of the bowstring object, as it should. The reason for this choice is that it was practically impossible for me to devise an animation that brought the bowstring perfectly backwards, without going sideways a little. So, since the arrow is shot in the forward direction, and the forward direction is twisted wrt to the bow, that is the object over which the user has control, the user would see the arrow going in a different direction than the one he has aligned its camera with. To counteract this, I made the arrow simply a child of the bow, so that the two would share the same forward direction.

Once the arrow is shot, a simple `tween` undoes the displacement done by the nocking `tween`, translating the arrow and the bowstring back to rest position in a short amount of time, to emulate the bowstring springing forward and shooting the arrow. Once this (barely noticeable) animation is completed, I start animating the arrow flight.

I start by reparenting the arrow to the scene so that it is easier to compute its trajectory. Again, moving the arrow means updating its position vector $\mathbf{p}$ in a proper way, with its velocity vector $\mathbf{v}$. In this game, gravity matters, so I also introduce the *acceleration* vector $\mathbf{a} = \Big(0, -g, 0\Big)^\top$, so that the update rule for the arrow position and velocity is:

$$\mathbf{v} \leftarrow \mathbf{v} + \mathbf{a}dt \tag{4}$$

$$\mathbf{p} \leftarrow \mathbf{p} + \mathbf{v}dt. \tag{5}$$

The initial velocity $\mathbf{v}_0$ for the arrow is a vector that has the arrow forward direction and the magnitude $\|\mathbf{v}_0\| = v_0$ computed as the result of an *impulse*, that as we know results in a change of momentum:

$$J = F\Delta t = m\Delta v = m(v_f - v_i) \tag{6}$$

and since the arrow is still we have $v_i = 0$ so that if we call v_f the initial velocity v_0 we have

$$F\Delta t = mv_0 \implies v_0 = \frac{F\Delta t}{m} \tag{7}$$

where m, F are parameters, namely the arrow mass and force, while $\Delta t = t_2 - t_1$ is the *charge*, i.e. the length of the time interval in which the user lets Link nock the arrow, computed as a difference of the two time instants when the mouse button is pressed and released, obtained with `performance.now()`. The maximum charge is 2 seconds.

Of course, an actual arrow also bends in its flight, changing its orientation. The angle of rotation about its own x-axis is directly proportional to the y component of its direction vector, i.e. the normalized velocity vector. However, undesired effects came up when modifying the `rotation.x` property of the arrow object directly, if the arrow was shot in a skewed direction (not straight forward). My guess is that when reparenting the arrow from the bow to the scene, the Euler angles with which the arrow rotation is expressed (wrt to the scene during its flight) are set in such a way that one should also change some other component to get this done properly.

My solution to the problem has been to reparent the arrow to a *pivot* object, that has all the transform properties of the arrow (position, rotation, scale), but the rotation about the x-axis (when still parented to the bow, that is the angle I want to change), that is set to 0. The pivot object is then added to the scene, instead of the arrow. Accordingly, the arrow transform properties are all "undone", meaning that in the pivot space the arrow sits at the origin, with scale 1, and has no rotation except wrt to the x-axis. Now, during flight animation, I update the position of the pivot object and can change the x-rotation of the arrow and obtain what I wanted.

Once the arrow lands, a short dummy `tween` allows the user to see for a second the location where it landed, and then the third person camera is restored as the current camera.

4.4 Target movement

To add some difficulty to the game, I animate the target by simply moving it around, depending on the selected game difficulty.

Easy The target is still. No animation.

Medium The target moves linearly horizontally. This is achieved by a set of three `tweens`, the first that interpolates position values (along the x-axis) from 0 to l in T_1, the second from l to $-l$ in $T_2 = 2T_1$ and finally the third from $-l$ back to 0 in $T_3 = T_1$. The three `tweens` are chained one after the other, with the third chaining back to the first, to attain an endless animation.

Hard The target moves in an 8-shaped pattern. What I mean by that is best explained by a simple drawing.

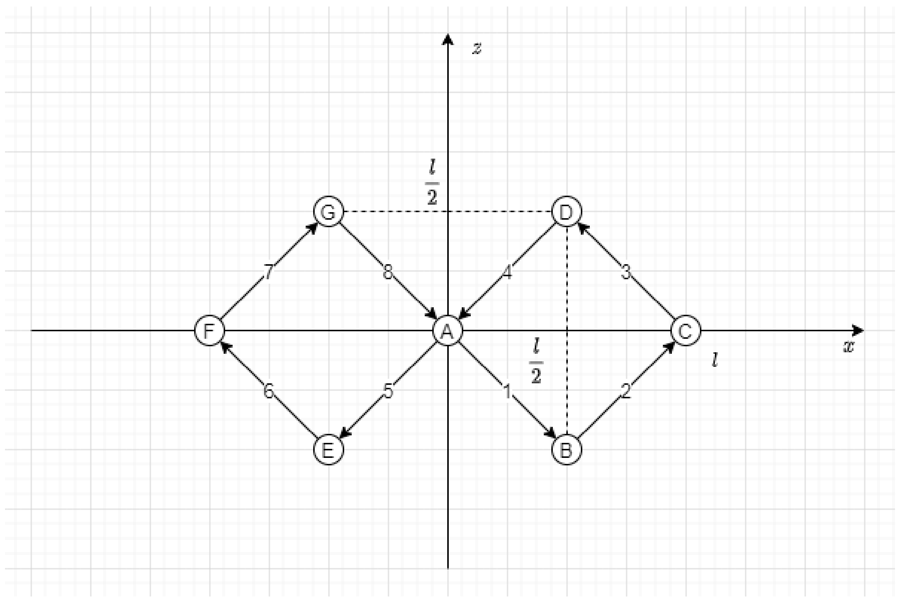


Figure 9: Pattern of the target in hard difficulty. Nodes represent waypoints, edges represent translations.

Each translation in Figure 9 is handled by a specific `tween`, and as usual, the 8 of them are chained one after the other, with the last chaining back to the first, to attain an endless animation.

5 Interaction

In this section, I will describe the ways the user can interact with the game, influencing the flow of events, and also how the objects interact with each other.

5.1 Menu

The menu is the first interaction that the user has with the game. He can press buttons, either to start playing or to review the instructions, and select the desired moment of the day (day/night) to play in, and the desired difficulty. This is achieved with some simple HTML to set up the interface and jQuery to retrieve the values the user has “input”.

5.2 Controls

The user can control what Link does in several ways: he can control his movement, the cameras, and trigger the nocking and shooting.

Controlling Link's movement As said in Section 1.2, the user can control Link's movement by means of the `WASD` keys. As said in Section 4.1, Link's velocity is updated with the direction (unit) vector $\hat{\mathbf{d}}$, and in fact, the user influences directly this vector.

JavaScript handles interaction with the user in a *event-driven* way: the user pressing a key triggers an *event*, and if there are event *listeners* added to the document to listen, then these functions are invoked. So, I set up two listeners, one to handle `keyDown` events (the user pressing a key) and one to handle `keyUp` events (the user releasing a key). When a key is pressed, I inspect the delivered event to check which key has been pressed, and then update the $\hat{\mathbf{d}}$ vector accordingly, e.g., if the `W` key is pressed, then it will be $\hat{\mathbf{d}} = \begin{pmatrix}0, 0, 1\end{pmatrix}^\top$, while if both `W` and `D` are pressed, then it will be $\hat{\mathbf{d}} = \frac{1}{\sqrt{2}}\begin{pmatrix}1, 0, 1\end{pmatrix}^\top$. On the other hand, if a key is released, I update the $\hat{\mathbf{d}}$ vector by setting the proper component to 0 and then renormalizing.

Controlling the cameras As seen before, this is both a first-person and a third-person game. At first, I thought of having just the first-person camera, since after all, this is an archery game. However, then the user could not see the nocking animation, so I decided to have both cameras.

The camera controls are implemented based on a utility class offered by Three.js, that is `PointerLockControls` [14], in turn based on the `Pointer Lock API` [13] of JavaScript. The `Pointer Lock API` provides input methods based on the movement of the mouse over time, and is exploited by the `PointerLockControls` class to define a method `onMouseMove`. This method can access the `event.movementX` and `event.movementY` properties, that hold the desired information on the mouse cursor movement across the screen.

The `PointerLockControls` class is normally used to bind a camera to "follow" the mouse cursor so that an upward movement of the mouse from the user would result in the camera "rotating upward". I defined some child classes (actually, I had to repeat the code, since I struggle to understand how inheritance in JavaScript works...), redefining the `onMouseMove` according to what I needed.

Controlling the first person camera One may think that to control the first-person camera I did not have to change the base `PointerLockControls` class much, since it was intended to this exact purpose. Actually, since the user is supposed to move the camera and look at where he wants to shoot the arrow, Link must move accordingly to do so.

So I used two different control classes, capturing the mouse movement at the same time, one to control Link's in-place rotation (capturing only the x-component) and the other to control Link's vertical movement of the arms (capturing only the y-component). Since the first person camera is a child of the bow, and the bow is a descendent of Link's left arm, the combination of the two controls allows full control (both horizontal and vertical) of the camera, in the prescribed range. In fact, the camera cannot be moved too high or too low, since realistically Link's arms (and head, in fact, to add realism, also the head rotates vertically so that Link actually looks at where he is shooting, and this rotation is controlled in the same way of the arms) motion is constrained.

Controlling the third person camera I wanted the user to be able to control the third-person camera with the same simplicity of the first-person camera, i.e. simply by moving the mouse, and then the camera would *orbit* around a target. Three.js offers another utility class, `OrbitControls` [12], that implements exactly this behavior. However, the class expects the mouse to be pressed for the motion to be registered but the real issue has been that I have not been able to adapt this code to a setting in which the target moves – I was probably doing some wrong transformation from Cartesian coordinates to spherical and back. However, this made me think of a much simpler and effective solution: add a virtual object, a *pivot*, as a child of Link, at the desired target position, so that it gets affected by Link's movement (if I don't want this to happen in a certain situation, I can temporarily reparent the object to be a child of the scene), control the pivot rotation with the same pointer lock logic used for the first-person camera, and add the third-person camera as a child of this pivot. This way, the pivot object is rotated according to the mouse movement, and the camera is rotated as well, orbiting the pivot point and therefore performing exactly the interaction I tried to code explicitly.

5.3 Collisions

In this game, the user's objective is to hit the target with the arrow; of course, this implies a *collision*.

To handle a collision, one first has to *detect* such collision, and I found two different methods to implement *collision detection* in Three.js: using a *raycaster*, implemented by Three.js in the `Raycaster` class [15], or using *bounding boxes*, implemented by Three.js in the `Box3` class [8]. I was not able to implement correct collision detection with the first method, so instead I turned to the second, less sophisticated method.

When loading the models, for each mesh I let Three.js compute the bounding box of each one of its meshes, with `mesh.geometry.computeBoundingBox()`, then the objects that need collision handling (the arrow, and Link, although Link cannot reach the target so it will never collide with it) are given a `collider` property, that is a function that is executed at each rendering iteration. The collider takes as arguments the object, a list of other objects against which collision detection is needed, and a callback function to be invoked in case a collision is detected. In the case of the arrow, this simply means stopping the flight of the arrow. The actual collision detection is a simple intersection check of the bounding boxes (properly transformed with the object's `matrix`, since they are defined at the level of the mesh geometry so in local space) of the meshes of the two objects, that is implemented by Three.js in the `intersectsBox()` method.

The bounding boxes method is reliable, but being less sophisticated is also less "precise", meaning that the bounding boxes computed by Three.js often are unnecessarily bigger than the bounded object, and the programmer has no control over this. In fact, in the game, it can happen that the arrow stops mid-air in the vicinity of the target when colliding with the target coming from some unfortunate angles.

6 Conclusions

I have reported my work on the project for the Interactive Graphics course, an archery game based on Three.js. Although fairly simple, the game has all the prescribed requirements: lights and textures, two hierarchical models, animated (with Tween.js) by leveraging their hierarchical structure, and user interaction to trigger these animations, and also change lights and difficulty.

There are many possibilities for future work on the project, for instance:

- The gameplay could be developed more, e.g. with more things to shoot;
- A normal map may be included to make the terrain "more interesting", actually I tried to do this and the result was not visually pleasing, since Link's feet would obviously intersect with the displaced terrain mesh, so it would be interesting to implement a physically consistent walking animation on a non-flat terrain;
- Better collision detection could be implemented;
- More animations, and make the existing ones more fluid.

This has been a very fun project, and I have learnt much in the process.

References

[1] *3D model of Link*. https://sketchfab.com/3d-models/link-humping-8c87ee3cadd148938a2d4417f0a3ecce.

[2] *3D model of the arrow*. https://sketchfab.com/3d-models/bow-arrows-787a927eba44437f9bb85741c43c4eaf.

[3] *3D model of the bow*. http://www.cgmasters.net/tutorials/bow_start.blend.

[4] *3D model of the target*. https://sketchfab.com/3d-models/medieval-archery-target-723158d09acf45098a54165ebc24e9f2.

[5] *3D model of the tree*. https://sketchfab.com/3d-models/oak-trees-d841c3bcc5324daebee50f45619e05fc.

[6] *Blender*. https://www.blender.org/.

[7] *Bone in Three.js*. https://threejs.org/docs/#api/en/objects/Bone.

[8] *Box3 in Three.js*. https://threejs.org/docs/#api/en/math/Box3.

[9] *GLTF*. https://www.khronos.org/gltf/.

[10] *GLTFLoader in Three.js*. https://threejs.org/docs/#examples/en/loaders/GLTFLoader.

[11] *Object3D in Three.js*. https://threejs.org/docs/#api/en/core/Object3D.

[12] *OrbitControls in Three.js*. https://threejs.org/docs/#examples/en/controls/OrbitControls.

[13] *Pointer Lock API*. https://developer.mozilla.org/en-US/docs/Web/API/Pointer_Lock_API.

[14] *PointerLockControls in Three.js*. https://threejs.org/docs/#examples/en/controls/PointerLockControls.

[15] *Raycaster in Three.js*. https://threejs.org/docs/#api/en/core/Raycaster.

[16] *Scene in Three.js*. https://threejs.org/docs/#api/en/scenes/Scene.

[17] *Skeleton in Three.js*. https://threejs.org/docs/#api/en/objects/Skeleton.

[18] *SkinnedMesh in Three.js*. https://threejs.org/docs/#api/en/objects/SkinnedMesh.

[19] *Three.js*. https://threejs.org/.

[20] *Three.JS Fundamentals*. https://threejsfundamentals.org/.

[21] *Tween.js*. https://github.com/tweenjs/tween.js/.

[22] *Vector3 in Three.js*. https://threejs.org/docs/#api/en/math/Vector3.