Final project report

Introduction

For the final project it was chosen to implement an endless run game in which the goal is to keep the character running by avoiding all the obstacles: the character can run in one of three parallel lanes and the user can either make it move to an adjacent lane or jump depending on the kind of obstacle present in the scene. On a sidenote, the project was only developed on Ubuntu and tested on Firefox: no other OS or browser were tested, so unexpected behaviour may arise if testing on Windows or in a Chrome browser.

Technologies used

All the models used in the project were made in Blender using a low poly style, which is a style that tries to represent elements through stylized shapes and low polygonal meshes, hence the name. Such choice had a double effect: the overall style of the game resulted cartoonish and the load on the gpu was decreased in order to accommodate for lower specs devices. On a side note, other approaches were also used to optimize the game but they will be discussed more in detail in a further paragraph. The textures used in the project, instead, were either generated procedurally in Blender or were created from scratch in Gimp, more specifically: the colour textures such as the orange and white ground and the red and white obstacles were created in Gimp, while the bump map of the bricks of other obstacles was generated procedurally in Blender through a noise texture. It should be noted that due to the relatively high speed of the obstacles on the screen while playing the game, it is really hard to notice the texture of the bricks, but whenever the character hits an obstacle and the game stops the rough texture of the bricks becomes more apparent. Some solutions were tested such as slowing down the game or making the texture stronger, but none gave acceptable results: on one hand slowing down the game made the texture more visible even when playing the game, but the challenge decreased drastically making the game not engaging at all, while on the other hand increasing the “displacement effect” of the bump map made the bricks look unrealistic when the obstacle was not moving and still did not make a big difference when instead the obstacle was moving. The main library used to render the scene was Three.js, which was useful to deal with both basic and more complex aspects providing many useful functions, from setting the camera and lights, up to importing complex models from gltf files and merging multiple models into a single one in order to reduce the number of draw calls executed at each iteration. The animations

were created thanks to the Tween.js library that allowed an extremely fine-tuned control over them, such as:

- chaining and looping animations;
- defining callback functions to be executed on every update or upon completion of an animation;
- pausing and resuming animations;
- choosing different interpolation functions for each animation.

Models

The character was modelled to resemble a mannequin both to adhere to the low poly style chosen and to make the rigging part easier to deal with by not having to curve the mesh. The mannequin is composed by multiple polygonal meshes that represent major parts of the body such as the head, the rib cage, the bones in the arms and the legs, the main joints such as the shoulders and the knees and so on. This made rigging the character quite easy: after building a basic skeleton for the model, each polygonal mesh was linked to one of the bones so that animating the bones would animate all the meshes linked to such bone. The ground was modelled as two separate planes one over the other: the lower and wider one was given a solid green colour while the higher one was coloured through a texture image to make it resemble an athletics field. Two distinct models were created for the obstacles: one lower obstacle representing the hurdles used on athletics fields and one higher obstacle representing a brick wall. The idea behind those two obstacles was to increase the variety of mechanics in the game because the lower one could either be avoided or jumped over while the higher one could only be avoided. To make the obstacles more realistic some textures were used to provide additional details: the red and white oblique stripes of the hurdle were added through a colour texture while the roughness of the brick walls was added through a procedural noise texture applied to the normals of the bricks. To highlight to the player the boundaries of the track, in addition to the white stripes in the texture of the ground plane, low brick walls were put alongside the field: such walls have the same style of the high brick walls and were essentially created in the same way. In a first iteration of the drawing algorithm, the side walls were drawn together with the obstacles and removed whenever a row would move out of the viewing volume of the camera. Such approach would lead to the creation and destruction of new rows of low walls essentially every few frames, meaning that many unnecessary operations were executed to destroy the nearest row of walls and to recreate a new one in the furthest points of the view volume. Another source of inefficiency was that every wall was treated independently, meaning that for each

mesh a separate draw call would be invoked at every drawing iteration. To solve both problems it was chosen to convert all the separate meshes to a single one using a buffered geometry: instead of progressively drawing one row of side walls at a time, instead during the loading of the page enough rows of walls are added into a buffer such that the full viewing volume of the camera could be covered, then the geometries and the materials are extracted and the former are used to create a buffered geometry while the latter will be applied to the final mesh. The combination of the buffered geometry with the extracted materials is used to create the final mesh that will be dealt with in a single draw call, significantly reducing the number of operations and therefore decreasing the computational load of the page. On a side note, thanks to this approach, instead of creating a new mesh whenever portion of it is shifted out of the viewing volume, it is enough to translate it back with an appropriate value such that the translation is operated seamlessly.

Game execution pipeline

Many preliminary operations, some of which asynchronous, need to be executed before the application is able to render the scene: the first function to be executed was set to start after the loading of the page through the window.onload parameter. This first function is responsible to create and setup a renderer, a scene, a camera, a couple of lights and to add the renderer that has just been created to the html page by appending it to the body as a dom element. The last operation executed by this first function is the call to the next function of the pipeline, which is responsible of loading the models from the gltf files. The loading process of the models is executed asynchronously by the three.js class GLTFLoader through the mechanism of promises: when the loader is instructed to load a model from a gltf file, instead of carrying out the task right away, the loader produces a “Promise”, meaning that is commits to execute the operation requested in a distinct pipeline from the main execution of the program. This mechanism is necessary to prevent the page from blocking during the loading process because it is not possible to know how much time it will take, due to the fact that it can depend on many different factors that may even vary during the process, such as: the dimension of the file, the location of the file, the speed of the network in case the file is located on a remote server, the reading speed of the machine the file is located on, and more. Because the loading of the models is asynchronous and therefore there is no guarantee on the order in which the loadings will be completed, the models are inserted into a dictionary instead of a simple array, so that they can be accessed by their name no matter the order in which they are inserted into the structure. The loader produces a promise for every model that is to be loaded, so to start the execution of the next functions in the pipeline, which will need all the models to be correctly loaded

into memory, the Promise.all function was used to set which callback to invoke when all the created promises are completed. Such callback is essentially the third function of the pipeline and is responsible to initialize the scene with all the necessary models and to setup the collision detection module. The first component added to the scene is the ground, while the second one is the mesh containing the side walls, and on a side note it is in this stage that takes place the buffer geometry optimization mentioned in the last paragraph. The third and last component of this phase added to the scene is the character: in this phase the model is added to the scene and then all the bones of the skeleton are added to a dictionary that will be used whenever an animation needs to be created. Moreover, in this phase the base animations are created but this topic will be covered more in depth in the next paragraph. The next step of the current function of the pipeline is the initialization of the collision detection module: to make the application detect collisions between the character and the obstacles it was chosen to avoid a physics engine because it would have significantly slowed the application. Moreover, the vast majority of functionalities that such engine would have provided would not even be used in the final result, so it was chosen to adopt a less powerful and more specific solution: the “Raycaster” class provided by the three.js library. For every relevant part of the body of the mannequin a ray-caster would be created and attached to it, so that at every iteration of the main animation cycle it would be possible to iterate over all of them and check whether at least one of them detected a collision with an obstacle. To reduce the number of checks performed at every iteration, the ray-casters were only attached to the lower bones of the body of the mannequin because they were the only bones that could physically collide with an obstacle. After the creation of all the ray-casters, the third function of the pipeline sets a callback to be performed whenever the user presses a button: this allows the page to wait for the user to press the spacebar before starting a new game. When the user presses the spacebar, a new callback is set to be performed whenever the user presses a button so that the character can be moved by pressing the arrow keys on the keyboard, the fourth function of the pipeline is invoked and the main animation cycle is started. An integer variable called clock is used to keep track of the current timestep so that the application can spawn a new row of obstacles far out from the viewing volume every 75 iterations. To create a new row of obstacles, the following procedure is executed:

- a random integer between 1 and 3 is generated to decide how many obstacles to insert in the new row;
- depending on the number generated in the previous step, that many random numbers between 1 and 2 are generated to decide whether to insert a hurdle or a high wall;

- finally, the obstacles are shuffled so that if less than 3 obstacles are spawned, the lines in which they will be placed will be picked at random.

It should be noted that, without additional checks, it is possible that a row of 3 high walls may be generated, making the user unable to move on no matter the action taken. To solve this problem, it was decided to force the procedure to insert at least one hurdle in case 3 obstacles were to be placed in the new row. Every new row of obstacles is added to an array containing all the visible objects, so that during every iteration of the animation function all the visible elements can be translated towards the viewer just by iterating over the array. Moreover, whenever a new row is generated, if there is an old row that is out of the viewing volume it is removed from the mentioned array to allow the garbage collector to free that memory space. The last operation executed before rendering the scene is the collision detection check, done by iterating over the array of ray-casters: the check is performed by getting the row of obstacles that is nearest to the character and passing it as an argument to the function “intersectObjects” for every ray-caster. If such function returns an empty array, meaning that no collision is detected, the game moves on by calling the next animation cycle, while if the array is non-empty then the game is stopped and a callback is set so that by pressing the spacebar the user can start a new game. To call the next animation cycle, more than one approach was possible, the most common being invoking “requestAnimationFrame” with the fourth function of the pipeline as the argument. Such method, though, would have made the game run inconsistently from one computer to another and even during the same run, depending on the specifications of the hardware and on the load in every moment. To avoid this inconvenience, it was chosen to use the function “setTimeout” that takes as argument a function and a time expressed in milliseconds and allows to set an asynchronous call to that function after that many milliseconds. Considering that below 30 frames per second the human eye perceives a video as stuttering and that over 60 frames per second the increment in fluidity becomes barely noticeable, it was chosen to set the number of milliseconds to 20, meaning that every second the scene is rendered 50 times, which should yield an acceptable result on a wide range of devices.

Animations

As already mentioned, the animations were created through the tween.js library. It was necessary to implement a total of three animations: one for running, one for switching lanes and one for jumping. For the running animation all the affected body parts were set to follow a sequence of four key frames:

- the first one was needed to move half the body forward and half the body backward by rotating each part involved in the animation appropriately;
- the second one was needed to make all the body parts move back to their default position;
- the third one was needed to move the other half the body forward and the other half body backward essentially mirroring the first timestamp;
- the fourth one was needed to finally move all the body parts to their default position to make the mannequin ready to start another cycle of the animation.

To make the animation look more realistic, in parallel with those four key frames the full body of the mannequin was set to move slightly up and down along the y axis, so that it resembled the body bouncing on the ground at every step. The whole running animation needed to be executed indefinitely until either a new animation was invoked to be run in place of it or the mannequin collided with an obstacle. This effect was obtained by setting every tween belonging to the running animation with the function repeat(Infinity). The switching lane animation, instead, was made as a simple translation along the x axis with no need to stop the running animation. This approach proved both simple to implement and effective to resemble a sideways move. Finally, the jump was made as a two-part animation: the first part relative to the mannequin going up, while the second one relative to the mannequin going down. Because no physics engine was used, the basic animation had to be modelled as a vertical translation along the y axis. By default, tween.js interpolates between timestamps with a linear easing function, but a body moving under the effect of gravity moves with a law that is described by a second-degree polynomial, so keeping the default easing function would have made the animation look not realistic. Luckily, tween.js also allows the user to set the easing function, so the first part of the animation was set to use the “quadratic.out” function, while the second part of the animation was set to use the “quadratic.in” function. As a result, the vertical movement produced by the animation in conjunction with the translation of the obstacles along the z axis made the final result look like if the mannequin was jumping over the obstacles following a parabolic trajectory. In parallel with the vertical movement of the body, the arms and legs were set to move as well in order to make the jump look more realistic: on the first part of the animation the arms were set to move towards the chest and the legs were set to move in such a way to create a 90 degree angle at the knee, while on the second part of the animation the arms and the legs were set to go back to their default positions. Whenever the user presses the up arrow, the running animation needs to be temporarily stopped and the jump animation has to be executed

in its place. Only when the jump is completed can the running animation be resumed. To do this, the sequence of actions that are performed is the following:

- when the user presses the up arrow, all the tweens relative to the running animation are stopped by invoking their “pause” method;
- the two parts of the jumping animation are instantiated and chained together. Moreover, the second part is set to restart the tweens of the running animation when it is finished through the “onComplete” method;
- The first part of the animation is started;
- When the jump is completed, the tweens of the running animation are restarted by invoking their “resume” method.

On a side note, to detect collisions in the right way, all the ray-casters have to be moved alongside the part of the body they represent, meaning that all the aforementioned animations, with the exception of the running one, have to be copied and applied to the ray-casters too.

Possible additions and improvements

In this section we discuss some possible additions and improvements that could be made investing more time in this project. The first set of additions is mostly aesthetic and would serve the purpose of making the environment look more realistic: some bump textures may be applied to the athletics field and the grass; some trees could be generated in addition to the low walls on the sides of the track; more kinds of obstacles could be added, either mechanically similar to the ones already present in the game or introducing new mechanics. A score mechanic could be added depending on how long the user manages to keep the mannequin running. Optionally, a coin system could be implemented to increase the score when picking up coins on the track. Finally, the difficulty could increase over time either by making the obstacles move faster, appear more frequently, making the groups of two and three obstacles progressively more frequent or any combination of those ideas.