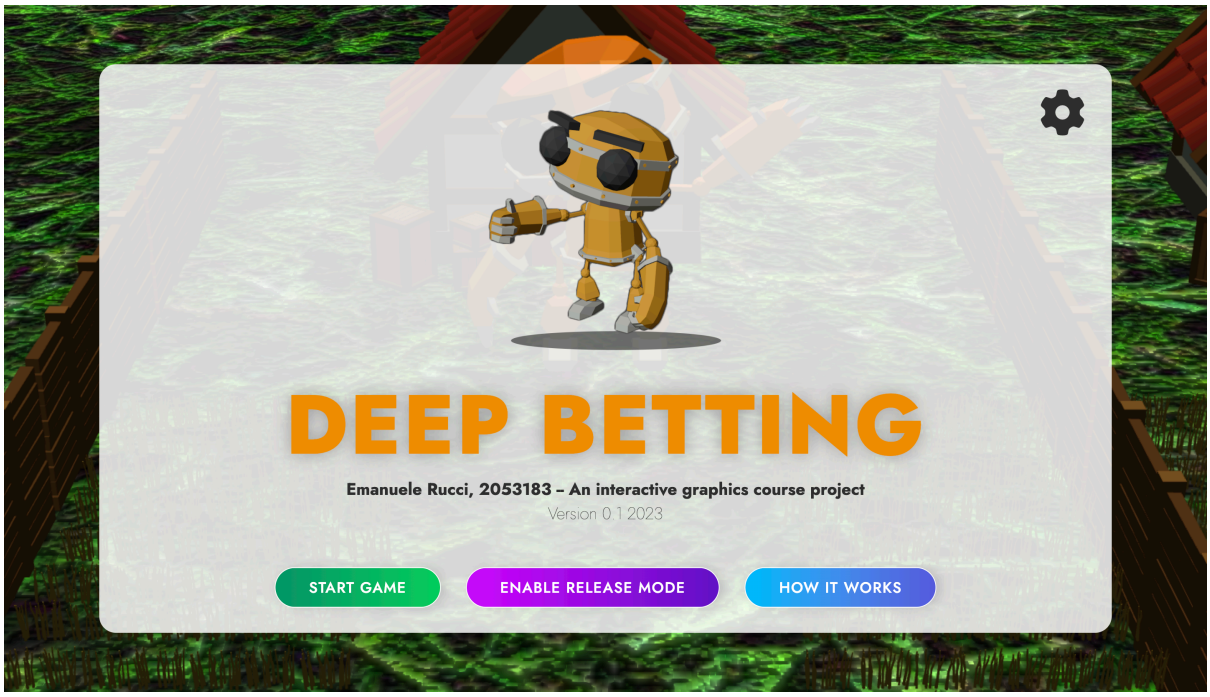
DEEP BETTING

Emanuele Rucci, 2053183
Interactive Graphics project course.
Ai and Robotics 2022/2023.



Introduction

The objective of the project is to simulate a race between five robots, where four of them operate autonomously while the fifth robot responds to user interactions.
The user is provided with a budget to spawn obstacles on the racing path of the other robots using the release mode. During the competition,
the autonomous robots will jump over the obstacles while the user-controlled robot should either punch or jump (using keyboard) over randomly spawned obstacles on its path.

Run the project

Requirements: nodejs installed (used Version 16.15.0), npm installed (used Version 8.5.5)

Instructions:
1. *Open a new terminal*
2. *Clone the repository -> git clone https://github.com/SapienzaInteractiveGraphicsCourse/final-project-deep-betting.git*
3. *Install all the requirements -> npm install (in the root of the repo), this command will look in the package.json and will install the right version for each package required*
4. *Activate the parcel bundler on the index.html file -> npx parcel index.html*
5. *Open the link given in the terminal*

Technologies

The project is a 3D web application based on WebGL technology that allows users to interact with a virtual scene. The following languages and libraries are used to create the application:

- **Parcel:** is a configuration module bundler which has been used to manage all the project's files.
- **Javascript**: It is used to handle user interactions, create the scene, and manage all the project's logic. JavaScript is a versatile language that is widely used to create interactive web applications.
- **HTML + CSS**: These languages are used to create and style the interface and all its components. HTML is used to structure the web page, while CSS is used to add style, layout, and visual effects to the page.
- **ThreeJS**: It is a JavaScript library used to create and display 3D graphics in the browser inside a canvas html element. ThreeJS provides a set of tools and utilities that make it easier to work with WebGL, such as a scene graph, cameras, lights, and geometries.
- **CannonJS**: It is a physics engine written in JavaScript that provides realistic physics simulation for 3D applications. CannonJS is used to add physical behavior to objects in the scene, such as gravity, collisions, and forces.
- **CannonDebugger**: It is a debugging tool for CannonJS that provides a visual representation of the physics simulation. The CannonDebugger is used to help the visualization of the shape associated to each mesh.

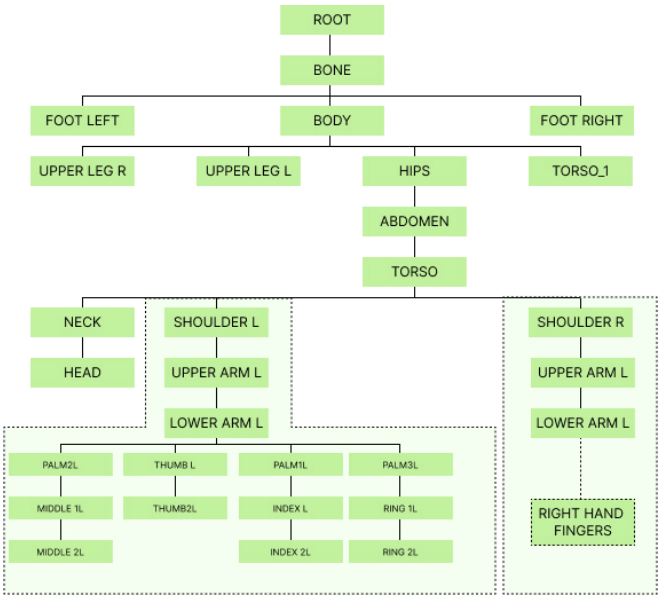
- **DAT**: used for easily create sliders and other controls that can be used to manipulate variables in real time.
- **GSAP**: It is a JavaScript animation library that provides a powerful set of tools for creating complex animations and transitions. GSAP is used to animate objects in the scene, such as moving, rotating, or scaling.
- **TweenJS**: It is a JavaScript library used to create tween animations. TweenJS is used to create smooth transitions between different values of an object's properties, such as position, size, or opacity.

Models, textures and sounds

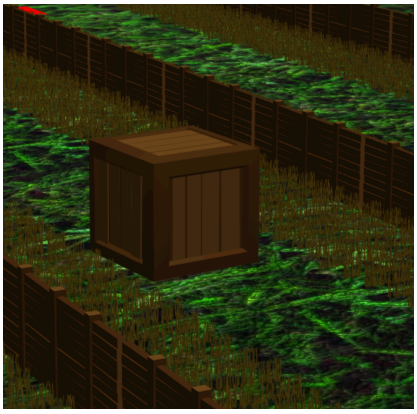
All the models are imported with the *GLTFLoader* class which is also provided an asset Loader which allow to manage the asset animation progress bar to be displayed before that the scene is fully loaded.

- **ROBOT**

Created inside the robot class, present a hierarchical structure that allows to move the bones in hierarchical way, for example, moving the hips would result in moving the abdomen, the torso the neck and the head and both the arms, hands and fingers.



On the right is presented the hierarchical structure of the model used.



- **OBSTACLE**

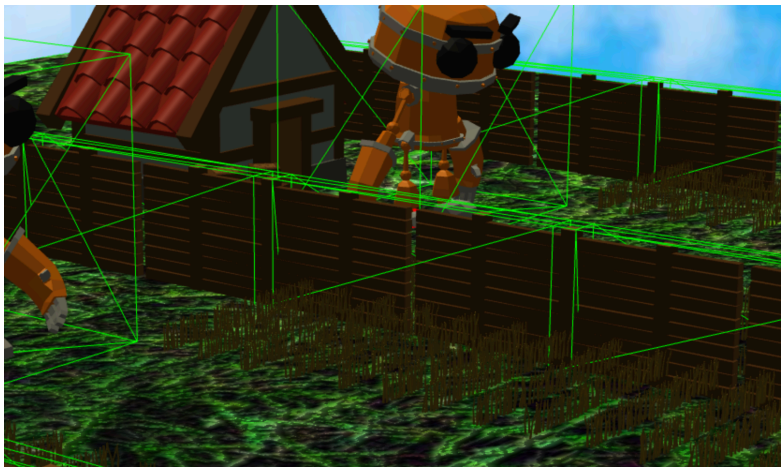
This object owns a cannon physic body that allows the mesh to receive force and moving according to it.

The obstacles has its own class that then has been used in the main.js file.

- **GRASS,STABLE**

The grass and the stable are meshes that do not own any physical body; this is to avoid not useful computation: these objects are never going to impact an obstacle.

- **FENCE**



Each fence has a physical object associated, this is because, in contrast to the grass and the stables, it affects a lot the obstacle that receive a force.

- **TEXTURE**

  There are different textures involved in the project:
  - The cube scene has 2 different textures: sky and land, the sky has been applied to 5 out of 6 face of the internal cube, while the land has been applied to the bottom face of the cube.
  - The ground geometry which is a planeGeometry mesh has a double side texture which is a grass texture, as well a bump effect.

- SOUNDS

  There are also different sounds involved:
  - in the countdown are played 2 different sound when each second pass and when the countdown is finished

- in the creative mode when an obstacle is released there is a 'pop' sound and when an obstacle is removed there is a 'trash' noise

- In the competition mode there is the 'eye of the tiger' music.

Code Detail and Implementation Strategy

The code is composed basically by the following files: robot.js, obstacle.js, main.js, style.css, utility.js

Here are explained all the functions and the objective of each of this files

ROBOT CLASS - robot.js

This file is responsible of:

- Define a new robot implementing the constructor method and set some global variables used also in other files: the constructor clones the mesh, it create a new cannon body that associate to the mesh; it adds the mesh to scene and the body to the world, then it calls the *runInitialAnimation* that sets a random initial animation, while it is also calling the *idleAnimation* that manage to set the shoulder of the robot down.

- *SetBones:* used to set class attribute for the bones that could be manipulated in the animations

- Define the character animations using tween: every animation is defined as the orientation of each (or the interested) bones for every

keyframe of the animation, and the duration of each bone's movement passing from one orientation to the others.

- Update the mixer: this is for animate the robot using the clips instead of custom made animations.
- Link mesh and bodies for cannon: *linkMeshAndBodies* that copy the position and the orientation of the mesh to the physics body. This function also apply a displacement on the y axes for the cannon body of the robot to ensure that is positioned exactly where the mesh is in every animation frame.
- Define 2 utilities functions used for debugging purpose: *showBonesGuide* that insert in the DOM sliders from Dat to have an easier way to define all the keyframes animation; *showButtonsForActions* that prints a couple of button for run the jump animation and the punch animation.

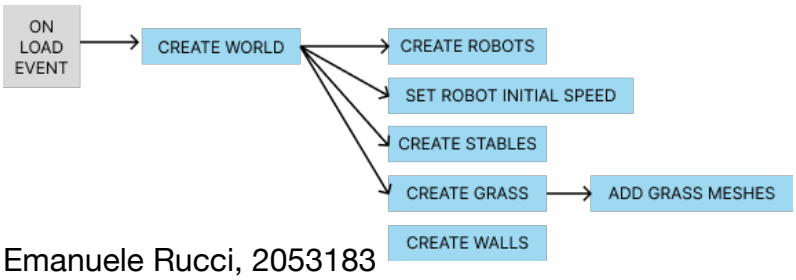
OBSTACLE CLASS - obstacle.js

The obstacle doesn't have nor animation or clips. Its class has just the constructor method that does the exact same operation of the robot's constructor, and the *applyForce* function that is using the cannon's *applyImpulse* function which is called when the robots punch an obstacle, and this is close enough to receive the force.

The constructor here also set a class attribute *is_spawned* to differentiate the obstacle inserted with the creative mode and the object randomly spawned on the myRobot line.

SCRIPT FILE - main.js

This file contains all the logic game, the competition manager, the frontend manager, the collision manager and more.



The light blue boxes represent functions.

- *OnLoad Event:* As soon as the dom is loaded the function createWorld is called.

- *CreateWorld:* this function is responsible of create the robots, the stables, the grass and the walls; The above image represent the order of the calls and each of of this function basically adds meshes to the scene according to a position strategy hard coded in each function; moreover the only function that associate a physical body to the meshes is the create walls: this is because, if an obstacle hits a wall has to respond to the collision physical laws. The function *setRobotInitialSpeed* assigns to each robot a random speed for running and regulate the speed of the clip according to the speed given, while the function add grass mesh is responsible to position the mesh in the scene following two lines in every racing path.
  Then to each robot's physic body is attached a collide event given from cannon js. As last operation, the state is changed to *'waitForSpawn'.*

- *Animate:* this function is the loop of the program and performs the following operations:

  - Switch over the state of the game calling the right procedures for each state;

  - Links mesh and body for the physic engine to update;

  - Calls the mixers updates.

THE MAIN MENU'

- *The how it works:* a button from the main menu makes accessible a slider that explain how the race works.

- *The settings:* from a button in the main menu is possibile to manage some settings: the camera view, the debug mode that enables or disable the cannons body to have the wireframe visibile in the scene, and the last is the audio enabling or disabling.

- *the Start game button:* when this button is clicked, the function inside the addEventListener is executed. The function shows a menu on the top that allows to enable/disable sound and change camera view during the competition.
  If the sound_enabled setting is true, it plays the 'Eye of the Tiger' sound.
  The state variable is set to "*spawning*". The fov (field of view) of the camera is set to 20, and the projection matrix of the camera is updated. This will change the way the camera renders the scene. Then are shown images that correspond to instructions for the game controls (space for punch and shift for jump).

THE STATES PROCEDURES

- *the 'spawning' state:* the animation function call the *spawnRandomObstacles* that takes in two parameters, a gsap global timeline and spawnAmount which represent the number of obstacles that are going to be spawned on the user-controlled robot's racing path. Then are defined a center point and a radius value that are used by the timeline to animate the camera's position around the center point in a circular pattern. The animation has four parts, each moving the camera to a different point around a circle with the center point as its center.
  Next, the function creates spawnAmount obstacles by generating random positions. It ensures that the obstacles are not too close to each other by checking the distance between the new obstacle and the previous obstacle coordinates values using the *isTooClose* function. If the new obstacle is too close, it generates a new position until a valid one is found. It then creates a new Obstacle object and adds it to the obstacleArray.
  The *isTooClose* function takes in three parameters: position, previousObstacleX, and distance. It checks

whether position is too close to any of the values in previousObstacleX by calculating the absolute difference between position and each value in previousObstacleX. If the absolute difference is less than distance, the function returns true, indicating that the new obstacle is too close to an existing one. Otherwise, it returns false.

As soon as the camera spawning animation is finished, the animate function set the state of the game to '*competition'.*

- *'Competition' State pre-competition:* according to the state of the variable *countDownFinished* the animate function calls two different procedures: when is false calls the *managePreCompetition* that is responsible for managing the countdown and camera positioning before the start of the competition. The function then checks if the camera is positioned behind the robot. If it is not, the function uses the GSAP animation library to move the camera behind the robot. Next, the function checks if the countdown has finished. If it has not, the function starts the countdown by using the tl timeline to animate the countdown element.
  The *countDownCompetitionManagement* function is called during this animation to update the countdown text and play sounds at the appropriate times.
  The countDownCompetitionManagement function takes in one parameter, time, which is the current time of the timeline.The function then checks the time parameter given from the timeline to determine which countdown number or message ("GO!") to display.

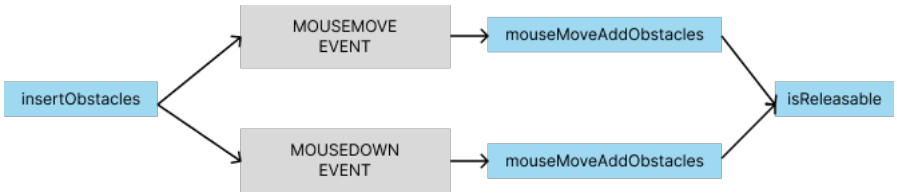
- *'State' State running:* just when the countdown is finished, it becomes not visible, all the robots running clip are enabled, and the state becomes '*running*'. In this state, at animation frame is called the function *manageRunning* that is responsible for managing various aspects of the competition while it is running.
  The function first calls the *myRobotCompetitionManager* function to manage the behavior of the user's robot. It then loops through all the robots in the robotArray and calls the robotCompetitionManager function for each robot except the

user's robot. The robotCompetitionManager function updates the position of the robot's mesh and physics body by adding the robot's speed to its x position.
The function then calls the *manageTimer* function, which calculates the current time of the competition and updates the HTML element to display the time.
Next, the function calls the *checkForWinner* function to determine if any robot has crossed the finish line. If a robot has crossed the finish line, the *printWinnerScreen* function is called to display the winner screen and set the state to "winner".
The *myRobotCompetitionManager* function manages the behavior of the user's robot. The function updates the position of the robot's mesh and sets the position of the camera based on the camera_view option. If the camera view is set to "third_person", the camera is positioned behind the robot and looks at the robot's position. If the camera view is set to "whole_view", the camera is positioned in front of the robot and looks at the robot's position. The orbit.target property is set to the position of the robot being followed by the camera to make the orbit controls center update.

DUMP object- utility.js
This files contain a debugging purpose function which allow to print in the console the tree of the bones and the meshes of the gltf model loaded.

Release mode

- *The release mode:* is accessible from the main menu and is enabled using the button ‘Enable release mode’ that calls the function *insertObstacles*; this is the procedure that enable the user to move around the scene and place obstacles on others robot’s racing path.



The first function *insertObstacles* is responsible for inserting obstacles into a 3D scene. It adds the highlightMesh to the scene and attaches event listeners for the mousemove and mousedown events.

The second function mouseDownRelease is an event handler for the mousedown event. It first retrieves the value of an HTML element with the ID cash-value the represent the current budget available. It then checks if there is already an obstacle at the position where the highlightMesh is located. If there isn't, it checks if the player has enough budget to release an obstacle and if the highlightMesh intersects with any other objects in the scene. The intersection is given by the ray caster.
If both conditions are met, it creates a new obstacle object and adds it to the scene and an obstacle array. If the player doesn't have enough virtual currency, it displays a message indicating that the budget is finished. If there is already an obstacle at the position where the highlightMesh is located, it removes the obstacle object from the scene and the obstacle array, restores the virtual currency, and plays a sound effect.
The third function isReleasable takes in x and z parameters and checks if a position is valid for releasing an obstacle. It returns true if the z parameter is one of four specific values (the z for the racing path) and if the x parameter is within a specific range. Otherwise, it returns false.

Autonomous robots

As soon as the robots are created, in the *createWorld* function, a *collide* event listener is attached to each of them. For all the robots, excluding the one controlled from the user, the function basically calls the function *jumpAnimation* that allows the robot to jump above the obstacle and surpass it.

The user-controlled robot

The collision handler function for the user controlled robot manage the speed reduction of the robot given by the collision until a minimum

empirical threshold (0.1) has been reached because of the numerous collisions.

Moreover to manage the action that the robot can perform, there are two event listener attached to the document;
The keydown event, checks if the current state of the game is 'running' (means the competition is running and the robot has to respond to the command given by the user), and according to the button pressed it calls the two different animations which are *jumpAnimation* and *punchAnimation.*

The animations

In the robot.js file have been defined different class methods for the animations. As common, the technique used to define the animation is the keyframe animation that require the animator to specify certain points in time (keyframes) and the desired state of the object (bone in this case) being animated at each keyframe. The library tweenJs (or GSAP for the camera) then interpolates the animation between the keyframes to create a smooth and continuous motion.

The list of animation is the following:

- yesAnimation: moves the head up and down;
- hiAnimation: moves the arm and the shoulder for a shaking gesture;
- danceAnimation: the robot makes a dance that require multiple movement of different bones of the hierarchical model;
- jumpAnimation: the robot fold its legs, moves the arms behind his body and then jumps moving forwards;
- punchAnimation: the animation simulate a shoulder punch loading and then the release of the punch.

There are then other support function to ensure that even if a bone is not manipulated in an animation it has still plausible values.

The first three animation are basically chosen randomly as soon as each robot is constructed and added to the scene, while the other two animation are called programmatically from the main.js file.

The jumpAnimation and the punchAnimation require that no other animation is running, this is to avoid multiple call of the function while its execution has still not reached the end;

Going in detail in the punchAnimation, it require different parameters. This function models the two possible cases: the user-controlled robot punch or the autonomous one punch.
NOTE: to avoid some physics bugs, the case in which the autonomous robot punch never happen because the main file actually calls the jumpAnimation and never the punchAnimation.

In the case in which the user controlled robot punch, the function loops over the randomly spawned obstacle and finds the closest, then , given this obstacle, it is possible to call its method '*applyForce*'. The box receive an impulse and moves away from the robot racing path.

Both at the end of punch and jump animation, the running clip is reset and it starts again allowing the robot to continue its move along the path.

Limits, bugs and conclusion

The project presents a bugs that sometimes compromise the punch action: the physic body of the obstacle is not synchronized perfectly with the position of the shape; this causes a bad behavior of the box which looks like is going behind the ground (y axes), or it is not affected by the force of the punch. Moreover the animations of the robot presents some issues, this is because the definition of the position of the joints is really time consuming and it has

not been possible to define perfectly all the numerous bone's position in each keyframe passing from each animation to the other.

In conclusion the project can be really improved in a lot of ways, but is a good starting point to understand how to combine all these technologies together to build web application based on 3d, which I think it's becoming more and more the new trend in the world of web developers.

Although the project was challenging, I find it to be an incredibly valuable learning experience.