FORMULA DRIVER DOCUMENTATION

Formula Driver is a game developed with the Three.js library. Once the game is loaded in, the player is welcomed with a main menu showing a diorama-like scene where the user's car is undergoing repairs at the pit lane. There, the user is able to change options regarding the difficulty of the game or some of its graphical settings, like the presence of shadows and fog, the resolution and the team the car belongs to. The player can also read about the premise of the game and the controls in the "About" section. When they are ready, the user can start the game by clicking the "Play" button.
Once on the road, the player can move their car left or right in order to dodge incoming traffic, while gaining further speed in doing so, change the camera by switching from a cockpit to a third-person view and vice-versa, and pause (and resume) the game.
The game ends when the player crashes into an opponent, which sends the user's car into a spinning animation, after which a game over screen showing the results is displayed on the screen, with a button that gives the player the option to return to the main menu.
The goal of this document is to explain the technical details behind the game.

SUMMARY:

1. The Menu

1.1 – init():

Once the index.html page is loaded in, and after all the global variables are initialized, the first function to be called is *init()*, which in turn calls the functions *createMenu()* and *animateMenu()*.

1.2 - createMenu():

It changes the title of the web page to "Formula Driver – Menu", then calls the function *setScene()*. The next step, then, depends on if the game still needs to load the models or if the player has already played a game and is now coming back to the main menu after a game over. In the first case, the function *modelsLoader()* is called; otherwise, the models are already loaded in and *tweakModels()* is called. The next two methods are then called in both cases: *createMenuText()* and *assignButtons()*.

1.2.1 – setScene():

This method creates the basic scene in which we display the main menu "diorama". We first create a THREE.Scene with a white background, to which we add a white THREE.AmbientLight, which lights the whole scene, and two THREE.PointLight(s) in two different locations, which light only select parts of the scene and are able to cast shadows. We choose two of them for better presentation purposes.
We then create a THREE.PerspectiveCamera that looks at the center of the scene, and a THREE.WebGLRenderer, which is basically the canvas where the game is displayed on and that we append to the body of the .html file.
The renderer has its antialias and shadowmap settings enabled, allowing it to show an image with clean lines and shadows, if enabled by the user in the menu options.
The renderer has also a size that is equal to the inner width and height of the browser window, but since said parameters can change when the window is resized, *setScene()* adds a listener that calls the function *setResolutionRenderer()* with the objective to resize the renderer should this event happen, while also updating the camera aspect-ratio.
setResolutionRenderer() can also lower the internal resolution of the renderer, while keeping its dimensions intact, by first halving the size of the renderer, and then doubling the size of its internal domElement, which is the canvas part of the renderer: this happens if the player enables a low resolution in the main menu in order to improve the performance.

1.2.2 – modelsLoader():

All the models in the game, from the car itself to the pit crew, were made thanks to the editor available in the official Three.js website.
The player's car, in particular, is a complex hierarchical model with over 50 parts of various geometries, like THREE.BoxBufferGeometry, THREE.CylinderBufferGeometry

and THREE.TorusBufferGeometry, to which we add a standard physically based material called THREE.MeshStandardFormat to create their THREE.Mesh.
All of these models are exported in a .json file, which we then load in the .html file with the THREE.ObjectLoader. Once the process is done (the loader gives the opportunity to keep track of current state of the loading process), the *tweakModels()* function is called.

1.2.3 – tweakModels():
This function alters some aspects of the models that are loaded in, like hiding some objects that mustn't show up in the menu, changing some aspects of their materials (like the colors and the roughness), and adding textures and THREE.BoxHelper(s).
We load the former by using the THREE.TextureLoader and then we map them to the matching materials: textures include the logos that show up on the body of the car and on the signs found on the track, and normal maps that gives depth and a more realistic feel to the car itself.
The latter, instead, are invisible boxes that we add to the front of the player's car and to the back of the opponent cars, and that are essential for calculating the collisions later during gameplay. We resize these boxes to be smaller so that the player can get away with some extreme passes that should have caused collisions, in a way to not make the experience over punitive and improve the flow of the gameplay at the same time.
tweakModels() also calls the function *wheelScreenText()*, which adds a texture to the screen inside the steering wheel of the player's car. This screen normally shows the current number of passes and speed, but since we are in the main menu, it will display the sentence 'READY TO START'.
In order to do that, we create a new canvas element that we won't add to the body of the document: instead we extract from it the CanvasRenderingContext2D, which is a 2D visualization of what's inside the canvas, and where we write the string that needs to be displayed on the wheel after choosing font and color. We then create a texture that represents this canvas and map it to the screen's material.
If the player has enabled shadows (the game remembers its settings when the user goes back to the main menu after a game over), *tweakModels()* will also call *setShadows()*, which makes some of the objects able to cast and receive shadows, and sets the shadows' mapSize to 512x512: higher values give better quality shadows, but at the cost of a worse performance. It must be said that the opponent cars don't cast shadows in order to not slow down the game too much.
We then add the models to the scene.

1.2.4 – createMenuText():
This function creates all the menu screens that go on top of the scene, which the player can interact with to change options, read about the premise and the controls, and to start the

game. It essentially adds a series of 'div' and 'button' elements to the body of the document, whose parameters (or at least most of them) are detailed in the style.css file.

1.2.5 – assignButtons():
This function assigns click events to all the buttons defined in *createMenuText()*. The buttons assigned to the "Options" sub-menu have the objective to update the global variables so that the user's preferences can be remembered across consecutive games, and call functions such as *setResolutionRenderer()* and *setShadows()* to apply the graphical settings requested by the player.

1.3 – animateMenu():
Once the entire scene has been created, *animateMenu()* simply rotates it around the Y axis, then requires an animation frame on *animateMenu()* itself, so that the canvas can be updated at each frame, and finally renders the scene using the renderer.

2. The Gameplay

2.1 – createGameplay():

This function sets the title of the web page to "Formula Driver – In Game", stops the spinning animation on the menu and removes all text boxes, hides those objects that needed to be shown only on the previous screen while making visible the ones that are necessary now, and changes some rotation and position parameters on the camera and scene, whose background's color we change from white to light blue.
If enabled by the user in the options, a gray THREE.Fog is also added to the scene.
Then the starting speed of the user's car is initialized according to the chosen difficulty setting, while *spawnNewCar()* is called through *requestAnimationFrame*.
An 'div' element that shows the score is added to the document, while we update the text on the steering wheel's screen to tell the same score with *wheelScreeenText()*: of course, for now, this score is 0.
createGameplay() then finally calls the function *setControls()* and *animateGameplay()*.

2.1.1 – spawnNewCar():

This function initially checks if the game has been paused: in affirmative case, it doesn't do anything. Otherwise, it updates an internal timer: if this timer has reached its max value, which changes constantly during the game since it depends on the current player's car's speed, then it is set back to 0 while a new opponent car is spawned on the far end of the road at a random position and with a random color.
When we load the models in *modelsLoader()*, there's actually only one opponent car, so in order to spawn a new car, we need to clone that object.
spawnNewCar() then checks if there's been a game over: if no, the function calls itself again through *requestAnimationFrame*, otherwise it stops, so that no new opponent cars can be spawned in the future.

2.1.2 – setControls():

This function adds listeners for keyboard inputs: pressing 'a' or 'd' updates a global variable that conveys the direction in which the player wants to turn to, while releasing them means the players wants to return to a neutral direction.
The player can also switch between the two camera-views by pressing 'v': when this happens, the THREE.Camera's position changes to a new set of coordinates, while the 'div' showing the score is hidden if playing with the cockpit view, or made visible if playing in third-person view.
Finally, the player can pause the game by pressing 'p', which updates the title of the document to "Formula Driver – Paused" and stops the request of new animation frames. Of course, the game can be resumed by pressing 'p' again.

2.2 – animateGameplay():

This function calls *animateGameplayPlayer()* and *animateGameplayRoad(),* then it checks if there's been a game over: if so, it stops the spawning of new cars at the end of the road by halting *spawnNewCar(),* and then calls the function *gameOver()*. Otherwise, it renders the scene and requests a new animation frame for *animateGameplay()* itself, so that the animation can keep going.

2.2.1 – animateGameplayPlayer():

This function concerns with all the animations of the player's car: when the user turns left or right, the program controls if the player is stepping out from the boundaries of the road. If so, no animation takes place; otherwise, the body of the car, its tyres and steering wheel (alongside with the arms of the driver) all turn to the direction determined by the user's input with a series of rotations applied to their matrices (but only until a max steering value is reached), while the road itself gradually translates to right/left in order to simulate the movement: that's because the car is always still at the center of the scene, while it is everything else to translate around it.

When the user releases the key, signaling their intention to stop turning, then this animation is reversed, so that the car can again point straight, while the changes applied to the road's position are kept.

Finally, this function applies a rotation around the Z-axis to all four tyres, so that they can keep spinning during the gameplay, giving the impression of speed and movement even if, again, the car is completely still.

2.2.2 – animateGameplayRoad():

This function translates all objects on the road, like the opponent cars and the "Pirelli" sign, towards the user.

When an opponent is close enough, two THREE.Box (s) are created from the THREE.BoxHelper(s) that were previously added to the player's car's front and opponent car's back. These boxes are the actual collision boxes, and they are intersected to check if there's been a collision: if so, the game refreshes a global variable that signals a game over, while the position of the steering wheel is remembered so that the appropriate spinning animation can be played.

If, instead, an opponent car is behind the player, then it is deleted from the scene, in order to free up some resources, while the number of passes is increased: if a multiple of 50 is reached, then the speed of the car is increased as well, causing the objects on the road to translate even faster. After this, the score is updated on both the HUD and steering wheel , the former thanks to *wheelScreenText()*.

This function also checks if the "Pirelli" sign has reached a predetermined spot behind the player: if so, the sign is translated again ahead of the user, on the far end of the road.

2.3 – gameOver():

This function sets the title of the document to "Formula Driver – Game Over", removes the HUD and adds a new 'div' containing the player's final score, with a button that gives the user the option to return to the main menu: this game over screen is initially hidden, and becomes visible only at certain stage of *crashAnimation()*.

Then it hides the front of the player's car, while making visible an alternative front that shows the damage of the collision and that is missing a wheel, and that was loaded in the program back in *modelsLoader()*.

The camera is then added as a child to the front cockpit (which is an object) of the car and given a new set of coordinates, so that it can spin together with the car during *crashAnimation()*: otherwise, the camera would remain still while the car moves around.

The screen on the steering wheel is updated with a string that says "TERMINAL DAMAGE" thanks to *wheelScreenText()*, and then *crashAnimation()* is called.

If the player presses the button to return to the main menu, some global variables are re-initialized, the game over screen and the renderer are removed from the document, *crashAnimation()* is stopped and *init()* is called, restarting the game: the models, however, won't be loaded again, and all the options enabled by the user will carry across.

2.3.1 – crashAnimation():

This function spins the player's car towards the direction they were facing before crashing into an opponent: this is achieved by simply rotating the car around the Y-axis until a max value is reached. Once the car stops spinning, the game over screen created in *gameOver()* is made visible.

The player can still move the steering wheel during this phase, exactly as seen in *animateGameplayPlayer()*, however the remaining front wheel won't turn, instead it will slowly rotate back and forth around the Z-axis because of the damage.

Lastly, *crashAnimation()* renders the scene and requests an animation frame on itself, unless the player has decided to go back to the main menu.

3. Appendix

3.1 – Testing:

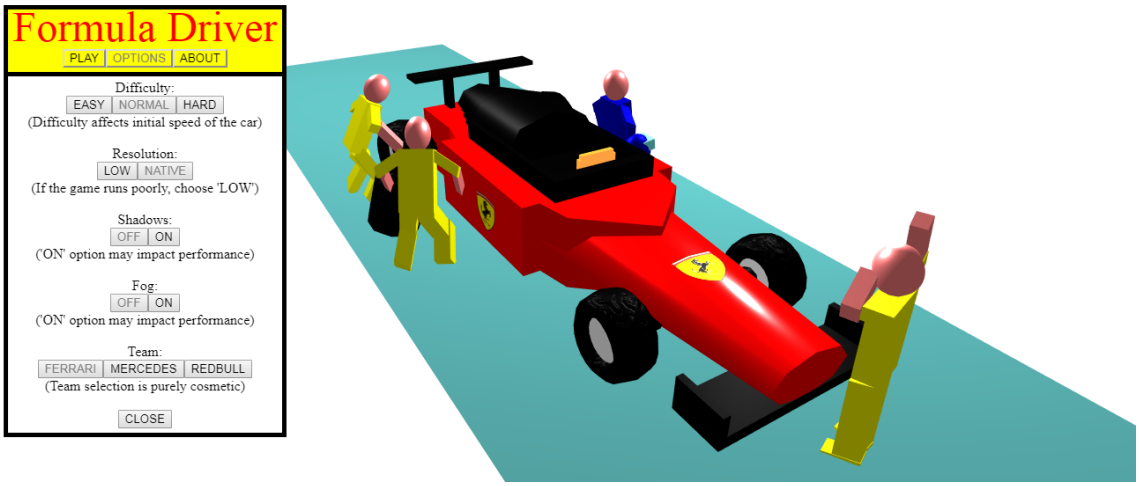
The game was tested using Chrome and Internet Explorer across various devices with different processing speeds.

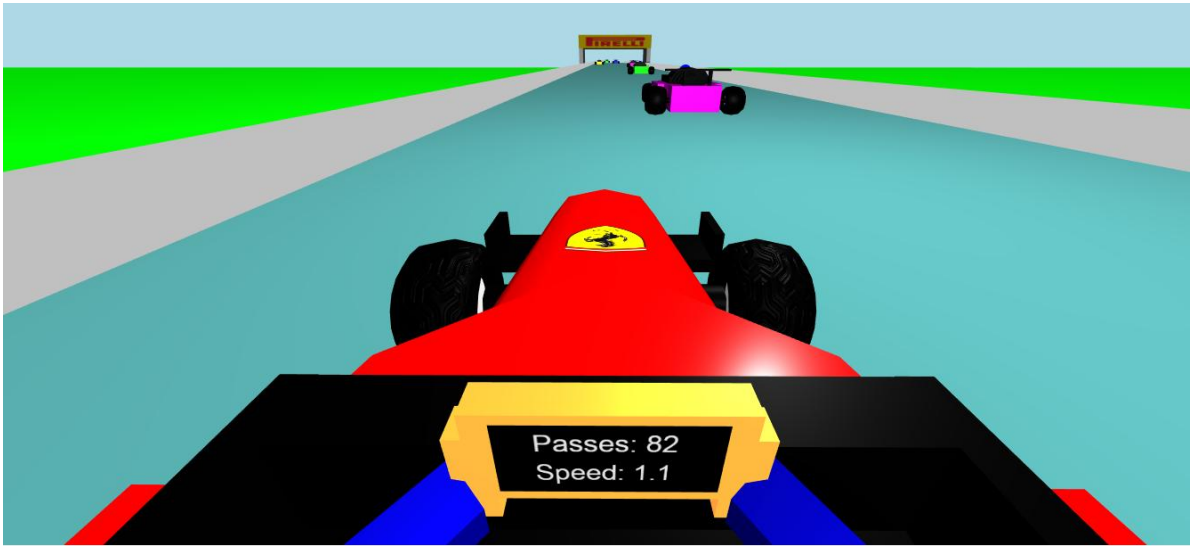
On Chrome the game looks exactly as it should, while Internet Explorer has a slightly different way to render the buttons under the "Formula Driver" title in the menu screen, probably due to some differences in how IE handles text and interprets .css parameters.

Meanwhile, the various processing speeds have obviously affected the frame rate, especially during the gameplay portion: in order to improve performance on slower devices, I decided to give the option to disable shadows and lower the resolution, which has worked in most of cases.

It also must be noted that the game targets 60 frame per second, so playing with a browser capped to 30 fps may introduce issue that I didn't encounter by first-hand experience.

The images below show how the game looks on my end on Chrome, with default settings.





3.2 – Scheme of the hierarchical object "Player's car":

Among the objects loaded in the game, such as the road and the pit crew, the player's car is surely the most complex. Here is a scheme that shows how all the parts of the hierarchical model are linked.

BODY -> CylinderBufferGeometry, Textured
- BACK -> BoxBufferGeometry, Textured
  - RIGHT POSTERIOR WING -> BoxBufferGeometry
  - LEFT POSTERIOR WING -> BoxBufferGeometry
  - UPPER POSTERIOR WING -> BoxBufferGeometry
  - RIGHT POSTERIOR SOSPENSION -> CylinderBufferGeometry
    - RIGHT POSTERIOR TYRE -> TorusBufferGeometry, Textured
      - EXTERIOR INSIDE -> CircleBufferGeometry
      - INTERIOR INSIDE -> CircleBufferGeometry
  - LEFT POSTERIOR SOSPENSION -> CylinderBufferGeometry
    - LEFT POSTERIOR TYRE -> TorusBufferGeometry, Textured
      - EXTERIOR INSIDE -> CircleBufferGeometry
      - INTERIOR INSIDE -> CircleBufferGeometry
- RIGHT SIDE -> BoxBufferGeometry, Textured
  - RIGHT LOGO -> PlaneBufferGeometry, Textured
- LEFT SIDE -> BoxBufferGeometry, Textured
  - LEFT LOGO -> PlaneBufferGeometry, Textured
- FRONT -> CylinderBufferGeometry, Textured
  - LOWER FRONT WING -> BoxBufferGeometry
    - RIGHT FRONT WING -> BoxBufferGeometry
    - LEFT FRONT WING -> BoxBufferGeometry
  - RIGHT ANTERIOR SOSPENSION -> CylinderBufferGeometry
    - RIGHT ANTERIOR TYRE -> TorusBufferGeometry, Textured
      - EXTERIOR INSIDE -> CircleBufferGeometry
      - INTERIOR INSIDE -> CircleBufferGeometry
  - LEFT ANTERIOR SOSPENSION -> CylinderBufferGeometry
    - LEFT ANTERIOR TYRE -> TorusBufferGeometry, Textured
      - EXTERIOR INSIDE -> CircleBufferGeometry
      - INTERIOR INSIDE -> CircleBufferGeometry
  - FRONT LOGO -> PlaneBufferGeometry, Textured
- LOWER COCKPIT -> PlaneBufferGeometry
- RIGHT COCKPIT -> BoxBufferGeometry
- LEFT COCKPIT -> BoxBufferGeometry
- BACK COCKPIT -> CylinderBufferGeometry
- BACK COCKPIT 2 -> CylinderBufferGeometry
- BACK COCKPIT 3 -> CylinderBufferGeometry

BACK COCKPIT 4 -> CylinderBufferGeometry
BACK COCKPIT 5 -> CylinderBufferGeometry
FRONT COCKPIT -> BoxBufferGeometry
WHEEL -> BoxBufferGeometry
RIGHT GRIP -> BoxBufferGeometry
RIGHT HAND -> BoxBufferGeometry
RIGHT ARM COCKPIT -> BoxBufferGeometry
LEFT GRIP -> BoxBufferGeometry
LEFT HAND -> BoxBufferGeometry
LEFT ARM COCKPIT -> BoxBufferGeometry
SCREEN -> PlaneBufferGeometry
SCREEN TEXT -> PlaneBufferGeometry, Textured
TORSO COCKPIT -> BoxBufferGeometry
HEAD COCKPIT -> SphereBufferGeometry
FACEPLATE COCKPIT -> BoxBufferGeometry

Some notes:
- When there is a collision, the entire Front sub-tree is made invisible, while a damaged variant (that is not shown on the scheme) is made visible.
- The opponent cars use a slightly different model: for example their tyres contain less polygons in order to save up resources.
- Body, back, front and sides of the car all share the same THREE.StandardMeshMaterial, so any change to the material of one part is applied to all parts.
- The logos are applied on invisible planes, like the one above the front of the car, so to be more accurate in the placement.
The image below shows the model in the official Three.js editor.