Project

Damiano Gasperini - 1796880

Jul 25, 2021

1 Introduction

The project has been devised and implemented by me. It consists in two game steps, in multi-players mode, i.e. it is for two players.
It has been entirely realized using HTML and JavaScript languages, integrated with BabylonJS library and exploiting, for animation purpose, the default CannonJS physics engine.

2 Structure

The structure of the program is the following:

1. Welcome and About pages
2. Level Selection page
3. Gain Moves game
4. 3D Tic-Tac-Toe game

2.1 Welcome, About and Level Selection pages

The first two points in the structure are only HTML pages with JavaScript functionalities. Welcome and About pages should introduce the players to the following game steps, providing instructions.
The three pages offer also a navbar to navigate among them and buttons to proceed in the flow of the program.
In particular the program uses the cookies of the browser to store some useful information:

- in the Welcome page there is the "Go" button: it cleans up the storage from previous data and loads the Level Selection page;
- in the Level Selection page there is the "Continue" button: it creates the *game_tictactoe* JSON variable in the storage, adding as its property the level selected by the user (and other properties later used). Then it loads the Gain Moves game.

Regarding the level selection, it consists in choosing the size of the Tic-Tac-Toe game board (fourth step). The code of this game has been generalized for any size: unfortunately, however, the greater the size, the greater the computational cost, and therefore greater latencies are present, mainly in the interactions and in the animation.
For these reasons I decided to bound the size of the square board between 3 and 6.
The difficulty of the game changes with the size of the board since this is not the classic Tic-Tac-Toe implementation: the moves of the players are limited and the number of moves can be very different from each other. So players must be very smart, optimizing it in order to win.

2.2 Gain Moves and 3D Tic-Tac-Toe games

The last two game steps of the program are the core of the project.
For each of them a different BabylonJS scene has been implemented.
In the next two sections of the report they will be described in detail.

3 Gain Moves game

As the name suggests, the Gain Moves game is the step in which the two players should gain as many moves as possible.
The moves will then be useful for the last step of the program, i.e. the edited Tic-Tac-Toe game.

In this step Player1 will start to play the game first and, when he will finish, the turn will pass to Player2.
The switch between Player1 and Player2 is driven by a boolean property of the *game_tictactoe* JSON variable in the storage, added when it was created in the Level Selection page.
This boolean property is *bool_isfirstplayer*: at first it is initialized equal to 1, but when Player1 will end his turn, it will be set to 0, so that the turn will pass to Player2.

In order to gain moves, the player should drive the car and hit the objects falling from the sky.
Instructions to drive the car are the following:

- Space bar: the car moves forward;
- A key: the car steers left (seeing it from the back)
- D key: the car steers right (seeing it from the back)

For each player the Gain Moves game lasts 40 seconds.

Once the window of the Gain Moves game page has been loaded, a JavaScript function will initialize the canvas variable getting the canvas document element, it will create the BabylonJS engine (in asynchronous mode, through async function, as suggested by the documentation), it will create a new scene and then it will enter into the engine rendering loop.

The *createScene* function is the function that will create the scene. This function is where every BabylonJS element, functionality and animation is implemented. Indeed the scene is the container that will render the game.

First a camera is created: I have used the BabylonJS *ArcRotateCamera*.
It represents an orbital type of camera. This camera always points towards a given target position and can be rotated around that target, with the target as the centre of rotation. It can be controlled with cursors and mouse or arrow keys.
Its position, relative to the target, can be set by three parameters:

- *alpha* (radians), the longitudinal rotation;
- *beta* (radians), the latitudinal rotation;
- *radius*, the distance from the target position.

After this, a function is called in order to have a click event on the scene: this is a little trick I have implemented, since the camera starts to be controlled only after a click on the scene.

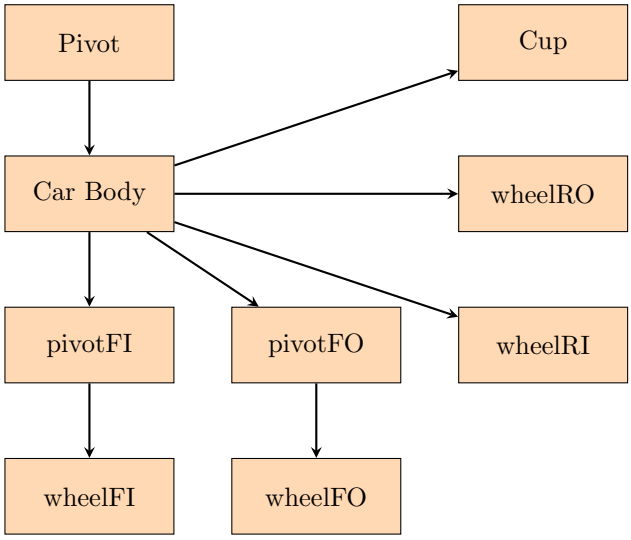
For this game I have initialized three different lights, so to highlight the car in the initial position and to light up all the scene:

- a Directional Light: it is emitted from everywhere in the specified direction, and has an infinite range.
- an Hemispheric Light: it simulates the ambient environment light, so the passed direction is the light reflection direction, not the incoming direction.
- a Point Light: this is the main light and it is a light defined by a unique point in world space. The light is emitted in every direction from this point.

I have added to the scene a simulated sky and ground.

The sky can be added to a scene using a *skybox*: it is a large standard cube surrounding the scene, with a sky image texture painted on each face. Indeed a material has been assigned to the skybox: its *reflectionTexture* property is initialized as a *CubeTexture*. The CubeTexture constructor takes the path of the image and by default appends "_px.jpg", "_nx.jpg", "_py.jpg", "_ny.jpg", "_pz.jpg" and "_nz.jpg" to it, so to load the +x, -x, +y, -y, +z, and -z facing sides of the cube.
The point light position has been set to coincide more or less with the painted sun, having a realistic effect.
Regarding the ground, I used the BabylonJS *CreateGround* function, passing as parameters the size of the ground mesh.
I have attached to it a material, and in particular I used the *TerrainMaterial* constructor. A diffuse texture and a bump texture has been set for this material so to have a realistic grass.
A shadow generator is created in order to represents on the ground the shadows of the car and of the objects falling from the sky.

The car is an hierarchical structure composed of a trapezoidal body, four wheels and a cup (a sphere slice). However there are other three meshes in its structure that are not visible. These three meshes have been mainly used for easily managing the animation of the car. They are the *pivot*, i.e. the car centre of rotation, the *pivotFI* and the *pivotFO*, i.e. the front wheels centre of rotation.
The structure of the car is the following:



The body of the car is obtained by the BabylonJs Mesh Builder *ExtrudeShape* function, that given as parameters the vertices of a trapezium and the depth, it creates a trapezoidal mesh with the desired shape.
A material has been assigned to the car body, with a diffuse color texture, to make the car look like it's made of wood, like a wood car toy.

The four wheels have been created using the BabylonJs Mesh Builder *CreateCylinder* function, that creates a cylinder with the desired dimension, color and texture coordinates for the flat faces of the wheels (the bases of the cylinder). Indeed a material with texture property is assigned to each wheel.

The cup on top of the car has been created using the *CreateSphere* function, taking the desired diameter of the sphere and the slice in input. The same texture of the car body has been attached to the cup material.

The front wheels are positioned a little further away from the side of the body to allow them to turn. The car will be driven forward using the space bar as an accelerator, up to a maximum speed, while the front wheels will be steered pressing A and D keys (with or without caps lock), as already said. The rotation of the wheels will be matched with the speed of the car.
An action manager is used to register a key up and a key down event. As acceleration and turning can take place at the same time, multiple key presses are required. Whenever a key down event occurs, the key is added to an associative array formed from a map object and the array value is set to true. It is reset to false on key up.

In the animation loop, whenever A or D is pressed, the front wheels are turned by a small amount. Since theta, the angle made by the front wheels, is changed, the centre of rotation is re-positioned as well.
When the speed is greater than zero and when theta is not zero, the pivot is rotated and each wheel is rotated depending on its position. Otherwise the pivot is translated in the local negative x direction and wheels are rotated by the same amount.

More in detail, the mathematics behind the car animation is the following.
The front wheel pivots and the rear wheel form a rectangle, with L the axis length (between front and rear) and A the distance between the two front wheel pivots.
When the front inside wheel (*wheelFI*, i.e. left front wheel) is turned by an angle of theta, the centre of rotation is where the line through the pivot point, normal to the wheel, meets the line through the two rear wheel.
The distance R of the centre of rotation from the inside rear wheel can be computed as $R = \frac{L}{tan(theta)}$, while from the middle of the rear wheels axis it is $\frac{A}{2} + \frac{L}{tan(theta)}$.

When the BabylonJS engine is running at F frames per second, the time for one frame is $\frac{1}{F}$ seconds. If the speed of the car is D units per second, the distance travelled by the car is $\frac{D}{F}$ units.
When the car is driven forward (i.e. $theta = 0$) at a speed of D units per second, the distance travelled by the car in one frame is $\frac{D}{F}$ units.
Let the radius of each wheel be r units. In one frame a wheel travels a distance of $\frac{D}{F}$ units and turns by an angle of psi radians where $r \cdot psi = \frac{D}{F}$ and so $psi = \frac{D}{F \cdot r}$.

On the contrary, when the car is turning (i.e. $theta \neq 0$), each wheel will rotate at different rates, as they travel different distances in the same time.
Assume the front wheel pivots as the centre of the front wheels and that the rear inside wheel travels a distance D, and let phi (radians) the angle of rotation turned by the car for one frame.

The rear inside wheel travels a distance $R \cdot phi = \frac{D}{F}$, and so $phi = \frac{D}{R \cdot F}$.
For one frame, let this wheel rotate by an angle of $psiRI$, then $r \cdot psiRI = \frac{D}{F}$, and so $psiRI = \frac{D}{r \cdot F}$.

The rear outside wheel travels a distance $(R + A) \cdot phi$.
For one frame, let this wheel rotate by an angle of $psiRO$, then $r \cdot psiRO = \frac{D \cdot (R+A)}{F}$, and so $psiRO = \frac{D \cdot (R+A)}{r \cdot F}$.

The front inside wheel travels a distance $\sqrt{(R^2 + L^2)} \cdot phi$.
For one frame, let this wheel rotate by an angle of $psiFI$, then $r \cdot psiFI = \frac{D \cdot \sqrt{(R^2+L^2)}}{F}$, and so $psiFI = \frac{D \cdot \sqrt{(R^2+L^2)}}{r \cdot F}$.

The front outside wheel travels a distance $\sqrt{((R + A)^2 + L^2)} \cdot phi$.
For one frame, let this wheel rotate by an angle of $psiFO$, then $r \cdot psiFO = D \cdot \sqrt{((R + A)^2 + L^2)}$, and so $psiFO = \frac{D \cdot \sqrt{((R+A)^2+L^2)}}{r \cdot F}$.

At any point, when the front wheels are turned, the centre of rotation and the turning radius R need to be recalculated. As said before, the centre of rotation will be an empty mesh, the pivot, which is the parent of the car. The pivot's position depends on the angle theta of the front wheels. The car has been created with its front pointing towards the negative x axis. When the pivot rotates to turn the car, its local negative x axis will always be the direction of travel and the centre of rotation will always be along its local positive z axis. R is initialized at a certain value and the pivot's position is initially set at R, along the local positive z axis. When $theta = 0$, any movement will be linear and the pivot will be translated by a distance of $\frac{D}{F}$, along the local negative x axis per frame.

When theta changes, a new value NR for the radius of the circle of rotation will be calculated: the centre of rotation will be translated by a distance $R - NR$ along the local positive z axis, while the car will be translated by a distance $NR - R$ along the local positive z axis. This makes the car not changing position and the centre of rotation being at a distance NR from the car. R is then set to NR.

Moreover I have used *Ray* and *RayHelper* in order to know when the car moves out of the ground area: if this

happens, the car will fall down, so to simulate gravity, and once it will have reached a certain negative y value the game will be over for the player.
Regarding the objects falling from the sky, they are torus shape meshes. In order to manage so many objects, I used the BabylonJS *SolidParticleSystem*. Overall, they are 200 objects that cyclically are reused by means the implemented *recycleParticle* that reset the particles at the emitter origin.
Particles are initialized with a random orientation and velocity (the velocity has an upper limit).
The *updateParticle* function is where their animation is implemented: it consists in moving down the particles as they was subjected to gravity. Their position is updated according to their velocity and their rotation is updated according some coefficients among all three axes.
In this last function is implemented also the core functionality of the game: through the *intersectMesh* function, when a particle hit the car a counter is incremented, representing the gained moves.

In order to track the time, a timer is implemented.
When time is over the gained moves of the player will be stored as a property of the *game_tictactoe* JSON variable in the storage. Moreover if both players have played the Gain Moves step, then the 3D Tic-Tac-Toe will be loaded.
The scene includes two dynamic panels, so that the player can see the time left and the gained moves.
Depending on the PC, the interaction can sometimes be slow and the rendering may appear jerky. This is because the high computational cost demanded by the shadow generator and because the high number of particles that have to be animated.
For greater fluidity I recommend zooming in on the browser window so as to lower the resolution.
Moreover, to better appreciate the car animation, in particular the wheels rotation, I suggest to zoom in with the camera.

4 3D Tic-Tac-Toe game

The 3D Tic-Tac-Toe game is not the classic implementation of the tic-tac-toe game.
The first main difference is that players can play the game on six boards at the same time, one for each face of a big cube.
Moreover the boards will have dimension $n \times n$, where n is the size chosen at the Level Selection step.
Changing the board size the logic of the game changes as well, since to win a player will have to put n consecutive pawns, i.e. aligned in horizontal, in vertical or in diagonal.
An other important difference is that players have limited moves: to make the game not so difficult, each player will have n^2 moves at any match, plus the corresponding moves gained at the Gain Moves game.
Players must be very smart in order to optimize their moves to win, or at least not to make the opponent win in the case of great disparity in the number of moves.

In the game logic, Player1 is the player "X", while Player2 is the player "O". The turn switches between Player1 and Player2, every time a player makes a move.
A move consists in clicking on the desired board location.
Player1 will start first, unless the red button has been pressed, i.e. the Restart Button: in the case of restart, the first player will be the opposite of the previous match.

Once the window of the 3D Tic-Tac-Toe game page has been loaded, a JavaScript function will:

- initialize the canvas variable getting the canvas document element;
- read the storage and in particular the *game_tictactoe* JSON variable, in order to initialize the size of the boards, to add the corresponding gained moves and to check who is the first player;
- create the BabylonJS engine (in asynchronous mode, through async function, as suggested by the documentation);
- create a new scene;
- enter into the engine rendering loop.

The *createScene* function is the function that will create the scene. This function is where every BabylonJS element,

functionality and animation is implemented. Indeed the scene is the container that will render the game.

In the scene I have implemented a camera of BabylonJS *ArcRotateCamera* type.
Moreover two lights, respectively an *HemisphericLight* and a *PointLight*, are present.
The default CannonJS Physics engine is created and enabled in the scene, setting null gravitation, like in the space.
The huge cube hosting the six boards is an hierarchical structure. It is composed of small cubes, in particular there are $(n+2)^3$ cubes.
An empty mesh, positioned at the center of the big cube, is the parent of all the cubes composing the big one.
To easily create this structure, i.e. positioning all these cubes, and manage the game logic, five arrays are filled by the program:

- *boxPositions* array: it is a $(n+2)^3$ array containing the computed positions for all the cubes of the structure.
- *clickable_boxes* array: it is a $n \times n \times 6$ array containing the indices of the cubes that compose the six boards, and so that are clickable.
- *position_pieces_boxes* array: it is a *6-dimensional* array of $(n+2)^2$-*dimensional* arrays. In particular each of the six arrays correspond to a face of the big cube: array 0 corresponds to the Top face - array 1 to the Bottom face - array 2 to the Left face - array 3 to the Right face - array 4 to the Front face - array 5 to the Back face. All these arrays contain the indices of the small cubes that compose that face.
- *gameState* array: it is a $(n+2)^3$ array that keeps track of the moves performed by the players.
- *winningConditions* array: it is a $((n+n+2) \times 6)$-*dimensional* array of *n-dimensional* arrays. In particular each array in it contains the n indices of same consecutive pawns, necessary for winning the game.

All the small cubes composing the structure are enabled to interact by the physics engine: a *physics impostor* is attached to them, that is a simpler representation of a complex rigid object that cannot be changed during interaction.
The physical parameters of the cubes are:

- unitary mass: it is the object's mass in kg;
- zero restitution: it is the amount of force the body will "give back" when colliding. A low value will create no bounce.
- unitary friction: it is the impostor's friction when colliding against other impostors.

As the cubes that compose the six boards are clickable, while the others are not, they have attached a different material with different diffuse color property. Indeed clickable cubes are blue, while the no-clickable ones are pink.

A first animation of the structure consists in highlighting a clickable cube when the mouse is over it.
To do so, an handling function of the scene event *onPointerMove* has been implemented: when the event is triggered, i.e. when the mouse moves in the scene, a virtual *pick* into the scene happens exactly in the positions where the mouse passed. If the pick hits a clickable cube, then it will be highlighted with a white color.
This highlighting is realized with the BabylonJS *HighlightLayer*, that adds a glow effect around a mesh.
The process to realize this animation is computationally expensive, for the high number of virtual picks that has to be carried out when the mouse moves.

When a player clicks on a clickable cube, the corresponding pawn, i.e. the "X" or "O" piece, is instantiated and positioned in the desired location of the boards.
In order to make this possible, an handling function of the scene event *onPointerDown* has been implemented: when the event is triggered, i.e. when the player clicks, if the game is active and the player has clicked on a clickable cube, then the corresponding piece is instantiated, rotated and positioned on the external (visible) face of the cube.
The right position is computed in a switch loop, checking the *position_pieces_boxes* array.
Moreover, when a click happens, the *gameState* array is modified accordingly. Indeed in this array each element is initialized to the empty string and, when a clickable cube with a certain index i is clicked, the *i-th* element in the array is modified as "X" or "O", depending on the player turn.

The "O" piece is realized with a torus shape, while the "X" piece is realized creating a mesh resulting from the intersection of two cylinders.

Both pieces have a different material and color.

After a click by a player, the turn passes to the other one, but first a function will be called to check if the player has won, and so if his last move was the winner move. This function is called *handleResultValidation*.

The conditions that make the game over are the following:

- a player managed to put n consecutive pawns;
- draw case, if nobody managed to put n consecutive pawns and there are no more clickable cubes available;
- both players have no more moves.

In *handleResultValidation* these conditions are implemented in the following way:

1. a loop checks if one of the winning condition, in the *winningConditions* array, holds; if there is one that is verified, the loop breaks, animations are executed and the game will end, with a winning message written in the dynamic red button.
2. in case of no winning condition verified, a loop checks if there is still a clickable cube available in the game state, otherwise it is a draw;
3. if there is a draw or both players are out of moves the game will end, with a draw message written in the dynamic red button.

In winning case, as said, three animations are implemented in the *handleResultValidation* function.
The first one is about the spread of particles: a particle system is created that emits particles from a point in the scene with a certain rate and random bounded life-time, velocity, size and power.
Particles can have different colors, directions and a texture is attached to them. They will fall under gravity, simulating a fountain effect.

The second animation concerns the camera.
The camera is moved from its position to a goal position. The path, that has to be followed by the camera, is computed by sampling points in the direction of the goal position and then interpolating them with a Catmull–Rom spline.
From the computed path, the key frames of the animation are instantiated, both for the position and rotation of the camera.
In particular, regarding the rotation, tangents, normals and binormals are derived from the path, so to use the *RotationFromAxis* function.
This function, given three orthogonal normalized left-handed oriented Vector3 axis in space (target system), returns the rotation Euler angles (e.g. rotation.x, rotation.y, rotation.z) to apply in order to rotate it from its local system to the given target system.
The computed animations are attached to the camera's animations property and then executed.

The third animation concerns the cubes that compose the big cube.
The physics engine will enable at the same time a vortex and a gravitational field events, both with their properties, like the radius of the area of their interest and their strength.
The cubes, that are subjected to the physics, will decompose the big cube and will start rotating composing a vortex.
In the meanwhile, thanks to the gravitational field, they will be kept together at the center of the scene, for a while.
At a certain time, the vortex and the gravitational field events will be disabled: the cubes, thanks to the accumulated kinetic energy, will spread throughout the screen in different directions, floating, rotating and translating.
In addition, when the two events are disabled, an inscription, made up of cubes, will appear on the screen, declaring the winner.

The scene presents a red dynamic button, i.e. the restart button as already said, the Home button, that will move the player back to the welcome page, and two dynamic panels that show the remaining number of moves of the two players.

5 Note

The program has been tested on Chrome.