The ”Runner” Game Documentation

by Herson John Nolasco Ruiz
George Paul Pislariu

June 13, 2019

The ”Runner” Game

We choosed to implement a ”endless-runner” game, a type of game in which the player’s character is constantly running forward, avoiding obstacles and trying to go as far as possible. The user controls the character by moving it left or right or by making it jump or slide.

1 User Manual

When the user loads the index page, has the possibility to choose between two modes that differ each other in the textures loaded for the objects of the scene and in the sounds played during the character’s actions.
After one of the modes has been selected, to start the game the player must press any key.
Once the game has been started, the goal of the player is to run as far as possible avoiding the obstacles that move closer to the character.
In order to dodge the obstacles the character can be controlled using these commands:

- ”Up Arrow” or ”W” to jump;
- ”Down Arrow” or ”S” to slide;
- ”Left Arrow” or ”A” to move left;
- ”Right Arrow” or ”D” to move right;
- ”P” to pause the game.

There are three types of obstacles: ”wood”, ”iron” and ”bar”.
The ”wood” and the ”iron” obstacles can be avoided by jumping over them or by moving the character in a different lane of the obstacle’s one.
Instead, the ”bar” obstacles can be avoided by sliding under them or by moving the character in a different lane of the obstacle’s one.
If the game is paused, the player can resume the game by pressing the ”P” key again. The game can be paused an infinite number of time.
The game is over when the character collides with one obstacle. After the collision the final score is displayed in the top-right corner of the window and the player has the chance to retry and reload the game in the same mode by pressing the ”R” button.

2 Technical Aspects

The project folder contains the main executable html file and five folders:

- **fonts**: stores the fonts for the text;
- **images**: stores the backgrounds and the .gif of the main page;
- **sounds**: stores the sound effects;
- **src**: stores the html, js and css files.
- **textures**: stores the textures of the objects.

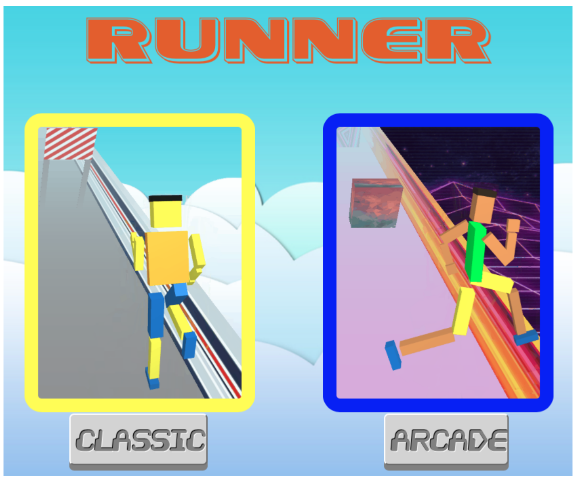
2.1 The HTML Files

2.1.1 Index Page

The index page introduces the user to the game giving the previews of the two selectable modes: "**classic**" and "**arcade**".
The page is organized in a "div" containing two blocks. Every block gives to the user a preview of the game scene. The user can choose a mode by clicking on the respective button.
The "onclick" event on the buttons invoke the functions "**button1**" (for the classic button), "**button2**" (for the arcade button). Invoking "button1" we store a variable "booleanMode" in the localStorage with "true" value, instead calling the second function we store it with "false". This variable will be used as a flag in the .js file to distinguish between the two themes.
In both functions after the assignment in the localStorage, we replace the current page with the "main" game page.
For the title text of the game we used the "Xcelsion Punch" font and for the buttons we used the "Videophreak" font.

2.1.2 Main Page

The body of the game page is organized in three "div": one for the current status of the game, one for the score and one container for the rendered scene.
The properties of these components are defined in the "style.css" file.
When the page is loaded, the function "changeColorScore" is called. This function reads from the localStorage the value of the "booleanMode" variable set previously in the index page.
If the variable is equal to "true" the text are set to "black" color else to "white".
The "panelGame" div gives to the user the current state of the game. The fonts used in this <div> are "Videophreak" and "Webpixel Bitmap".
The "panelScore" div instead, gives to the user the current score. The font used is "Videophreak".
The "container" div will contain the rendered game scene. The details of the game scene will be presented later.



The main page

2.2 The GAME.JS File

The main library we used is **Three.js**. In this case we used the minimized version "three.min.js" stored in the "src" folder.
The core of the game is represented by the **"game.js"** file that is in charge of creating the objects, move them, compute the score, render the scene and other tasks that will be presented.
The first function that the script calls is the "init" function that reads the "booleanMode" variable in the local storage in order to set the various textures, background and sounds that will be rendered. This function then initializes all the variables that will be used and also initialize the 'keydown', 'keyup', 'focus' event listeners.

2.2.1 The Character

The character is defined and initialized in the **Character()** function.
Inside this function, we can set the parameters such the colors, jumpDuration, jumpHeight and the slideDuration.
The character is built following this hierarchy:

- Character (self.element)
  - Head
    - Face
    - Hair
  - Torso;
  - LeftArm
    - LeftLowerArm
      - LeftHand
  - RightArm
    - RightLowerArm
      - RightHand
  - LeftLeg
    - LeftLowerLeg
      - LeftFoot
  - RightLeg
    - RightLowerLeg
      - RightFoot

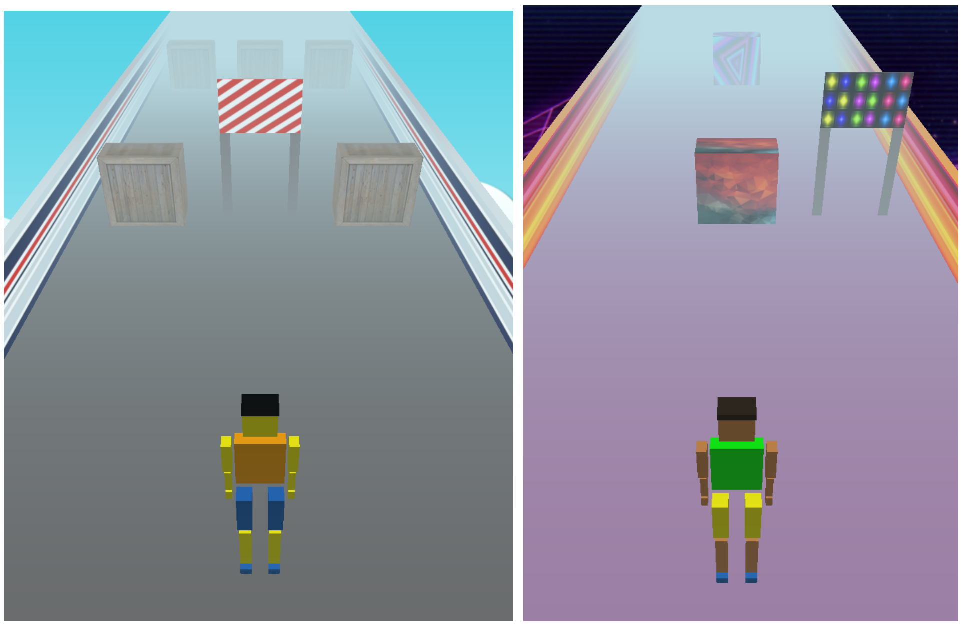
The hierarchy above is build in depth-first order through the "init()" that is in charge also of initialize parameters of the character such isJumping = false, isSliding = false, isSwitchingLeft = false, isSwitchingRight = false, currentLane = 0 (the character moves through three lines -1 is the left one, 0 is the center, +1 is the right one).
The arms and the legs of the player are built through the auxiliary function "createLimb" that creates and returns a limb with an axis of rotation at the top.
The head and the torso instead are built using the auxiliary function "createBox" that uses the parameters in input to establish the size, color and position of the box.
The "update" function to simulate the movement of the character will be presented later in the "Animation" section.

2.2.2 The Obstacles

We implemented three types of obstacles: "Iron", "Wood" and "Bar". These types of obstacles differ each other for size, texture and way to avoid it.
The "Iron" obstacles are built using the auxiliary function "Obstacle_Iron" that uses the parameters in input to establish the size, texture, position of the box and add it to the scene.
The only way to avoid this type of obstacle is to jump or move to another lane.
The "Wood" obstacles are built using the auxiliary function "Obstacle_Wood" that uses the parameters in input to establish the size, texture, position of the box and add it to the scene.
The only way to avoid this type of obstacle is to jump or move to another lane.
The "Bar" obstacles instead are made of three parts: leftPole and rightPole created using "createBox" and the horizontalPole created using "createBox_texture".
The only way to avoid this type of obstacle is to slide or move to another lane.
For each type of obstacle is defined also a "collides" method that models the obstacle as a box bounded by the given coordinate space and detects if it is colliding with the character.

2.2.3 The Scene

The scene is built through the function "createScene()".
Inside this method the scene is instantiated and its width and height are set equally to the window ones and the background set to the one depending to the mode selected.
The renderer is instantiated and the fog (with color 0xbadbe4) is created and added.
The camera used is a perspective one and its position is set to (0, 1000, 150).
For the camera is defined also the method lookAt(new THREE.Vector3(0, 730, -190));
The ground is added in the scene using the auxiliary function "addWorld" that use "createBox" to create the main ground and "createBox_texture" to create the left and right borders.
The light used in the scene is a simple hemisphere light defined as:
light = new THREE.HemisphereLight(0xfffafa, 0x000000, .9).
The createScene() function adds the character to the scene using the constructor defined before and also adds multiple row of obstacles using a new function called "createRowOfObstacles".
"createRowOfObstacles" takes as input parameters the position of the new row of obstacles and the scaling factor for the obstacles.
The spawn of the obstacles is pretty simple: for each lane it computes a random number using the Math.random() method and if its value is captured by the if-else if statements it builds the obstacle, stores it in an array that contains all the obstacles and then adds it to the scene.


Comparison between the two scene "classic" and "arcade" in order

2.2.4 The Sound

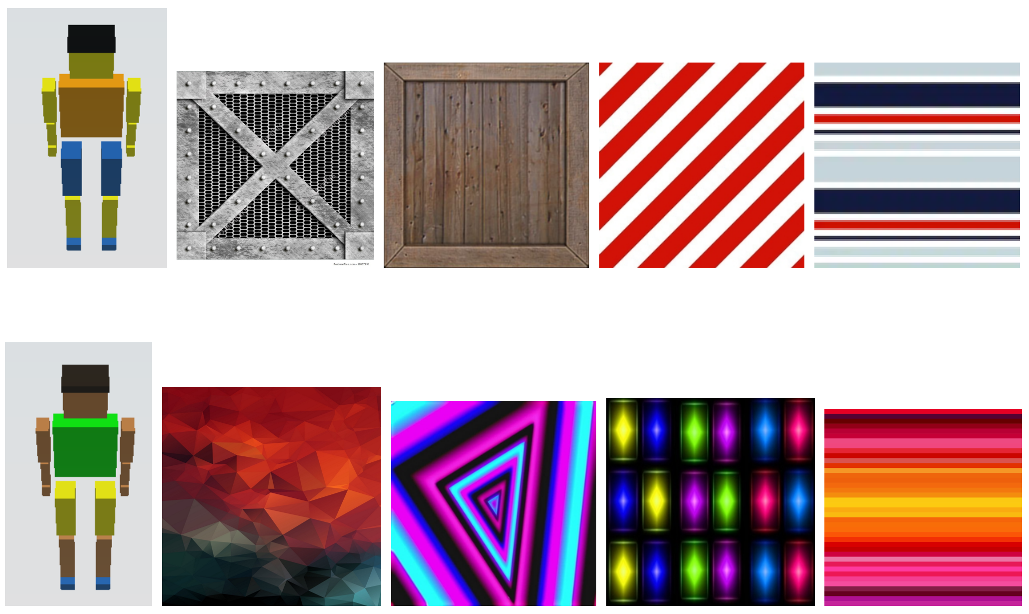
We implemented 4 types of sound:

- backgroundSound played during the game;
- jumpSound played when the character jumps;
- slideSound played when the character slides;
- gameoverSound played when the character collides with an obstacle.

The sounds are stored in the "sounds" folder and they are imported and set in the "init" function as the page loads.
To import the sounds we defined a function called "sound" that takes in input the source of the audio file and creates two methods "play" to play the audio and "stop" to pause the audio.
The "play" method is used when the character performs a jump or a slide and also in the game to play the background audio while the game is not paused.
The "stop" method instead is invoked only when the character collides with an obstacle and the game is over.

2.2.5 Colors and Textures

We declared a variable "Colors" that contains the association between the color's name and their hex code.
We decided to use the colors for the character's parts and for the ground and textures for the obstacles and the borders of the ground. The textures used are stored in the "textures" folder.
The "init" function as the page loads is in charge of switching the colors and the textures depending on the choice that the user makes when a mode is selected. The objects that use a simple color are created using the "createBox" function while the objects that have texture use the "createBox_texture" function.
The "createBox" function creates MeshPhongMaterial using the color passed as input.
The "createBox_texture" function use the "TextureLoader" to load the texture and creates a MeshBasicMaterial using as the map the texture loaded.



Comparison between "classic" and "arcade" modes
In order: character, iron texture, wood texture, bar texture, borders texture

2.2.6 The Animation

The animation of the character is managed in the "update" method defined in the character constructor. The default movement that the character executes is the "running forward".
The running animation is obtained through the auxiliary function "sinusoid" that generates the values of the limbs that vary the positions sinusoidally.
When the player press a key to perform an action, the event listener push the corresponding action in an array "queuedAction" where the action are executed in order.
During the jumping action, the y position is computed as the product of the jumpHeight and sinusoidally parameters.
For the slide action the rotation of the limbs are fixed and the y position of the character is lower then the running one.
For the switching action, we modify the value of "currentLane" by subtracting 1 if the player wants to switch left or by adding 1 if the player wants to switch to the right and we modify the character x position.
When the game loads the variable "gameOver" is set to false and "pause" set to true. Once the game has been started, "pause" is set to false and the scene is updated using the requestAnimationFrame(update).
The function "update" is the main function that is in charge of creating new obstacles, compute the score and check if the game is over or not.
When the game is not paused, it checks if the last obstacle spawned overcome the z position -11000. If the condition is satisfied a new row of obstacles is spawned at z = -14000.
Then the "level" is computed as the "difficulty" (increased every time that the condition is satisfied) divided by the "levelLength" fixed a priori.
When the level is captured by the switch statement the "speed" is increased.
At every new frame the obstacles move closer to the character depending on the speed of the current level and the obstacles that the character passed are deleted from the "obstacles" array that contains all the obstacle in the current scene.
The collision of the character with the obstacles is computed using the "collisionDetected" function that models the character as a box too and check if it collides with the obstacles by calling their "collides" function. If a collision is detected the game is over.