Interactive Graphics: Final Project

Mattia Seu
ID: 1871547

University of Rome "La Sapienza" — September 26, 2021

1 User manual

Notice: I have tested the game on the most popular browsers (Firefox, Chrome, Edge), including mobile ones out of curiosity. It seems that out of all of these the desktop Firefox browser suffers severe slowdowns, so it is recommended to experience the game on other platforms.

The project consists in a simple game of basketball in which the player can attempt free throws.

Shoot the ball

To shoot, the user can press a button prompting "Shoot ball" at the bottom middle of the screen. According to how long the button is pressed, a bar above it will fill up, showing how much force will be put into the shot. Going past the bar's limits will simply reset it and start filling it back up.

Game modes

The game starts in Free and Normal mode, in which the player can attempt to shoot the ball into the hoop, getting used to the force necessary to score. Whenever the ball goes through the hoop, a point is scored, as noted in the top left.

For a bit of a challenge, the Timed mode can be activated: this will start a 30 seconds timer in which the player can try and score as much as they can. The high score will be recorded for the session on the top left, but it can be reset by pressing the corresponding button on the bottom right of the screen.

The hard mode button in the same area instead makes the hoop move sideways, adding a degree of difficulty to scoring, which now requires synchronizing the charge up with the movement of the basket.

Scene settings

The bottom left of the screen instead hosts two buttons unrelated to the gameplay itself.

The "Light bulb toggle" turns off the light hovering in front of the board, which showcases its specular texture.

The "Reset camera" button instead, as the name says, resets the camera position to the initial one. It is in fact possible to manipulate the camera by left clicking and dragging on the screen or using the arrow keys to rotate, scrolling the mouse wheel to zoom in and out with respect to the center of rotation and right clicking and dragging to pan around the scene.

2 External resources

In this section the environment, libraries, models and textures I have obtained from external resources will be listed and sourced.

2.1 Environment and libraries

All external libraries are located in the `Common` folder of the repository.

As a framework for the project, I decided to employ **Babylon.js**. Besides the base module, I also used:

- A supplemental Babylon library for the creation of GUI elements, found at `https://github.com/BabylonJS/Babylon.js/tree/master/dist/preview%20release/gui`
- The **Cannon.js** physics engine, provided by the Babylon CDN at `https://cdn.babylonjs.com/cannon.js`
- The **tween.js** library for the animations.

2.2 Textures and models

For what concerns all the visual assets, I have used:

- A color texture for the basketball taken from `https://opengameart.org/content/basket-ball-texture`. In order to generate its normal map, I used the texture generator at `https://cpetry.github.io/NormalMap-Online/`.
- An image from Wikimedia for the halfcourt color texture, found at `https://commons.wikimedia.org/wiki/File:Basketball_Halfcourt_Transparant.svg`. To give the appearance of a wooden gym floor, I also obtained the wooden normal map from `https://3dtextures.me/2019/09/19/wood-020/`.
- The backboard of the hoop uses a specular texture obtained from `https://3dtextures.me/2018/01/06/crystal-001/`.
- The player uses the default dummy model from the Babylon asset library, found at `https://github.com/BabylonJS/Assets/tree/master/meshes`.

3 Technical documentation

Before getting into the implementation details, I'd like to point out that I conducted initial investigations of the Babylon framework exploring their documentation and the attached Playground examples.

The Playground is a Babylon-integrated live online editor offering several tools, including a handy inspector, which presents a graphical interface that I used to initially explore the hierarchical models and data structures offered by the library more intuitively.

My code structure is thus carried on from these initial foray with Babylon, leading me to include the Javascript code inside the HTML file itself, with its proper script container.

3.1 Scene and camera

After instantiating the default Babylon engine, we create and attach a scene to it. A scene can be seen as the rendered space which will effectively be displayed on the screen. Several scenes can be associated to the engine, in case it is necessary to switch between different scenarios, but for our simple game one scene is enough.

After this, we attach a camera to the scene. Babylon offers several types of cameras with different functions and control schemes. Having tried all of them, I opted for the *ArcRotateCamera* type, which behaves as an orbital camera that is always pointed towards a certain target and seemed to fit best the project. I set the latter to be the center of the scene, but it can be moved by panning with the right click and drag of the mouse.

In order to have the best of view of the scene I assigned some initial values to the rotation of the camera, so that both the player, the hoop and the ball trajectory could be properly viewed by the user.

Furthermore, I tweaked the default distance zoomed with the scrolling wheel of the mouse, which was excessive. Now it can be used to better explore the scene if need be.

3.2 Lighting

To illuminate the scene, I instantiated two directional lights that shine on the center of the scene from two opposite diagonal directions, highlighting the floor's normal mapping.

Besides these two lights, there is another spotlight used to simulate a light-bulb. This light is turned on by default, with an hovering animation in front of the hoop. It was in fact created to better showcase the specular texture applied to the backboard of the basket. More on the light-bulb in the next section regarding meshes.

3.3 Generating meshes

To represent the simpler objects in the scene, several meshes were generated.

The ball and the light-bulb were created with the *CreateSphere* method of the *Mesh* type offered by Babylon.

The ground was generated with the *CreateGround* method, the rim of the hoop with the *CreateTorus* method, whereas the backboard used *CreateBox*.

The set of methods belonging to the *Mesh* type only create very simple and symmetrical shapes, thus the other meshes were originated with *MeshBuilder*.

This is the case for the backboard of the hoop, which is a box with different dimensions created with *CreateBox*, the torus representing the rim of the hoop instantiated with *CreateTorus* and a disc mesh, which will only be used for collision detection and thus made invisible, generated with *CreateDisc*.

To each of these meshes, we can attach a material. As a starting point I used the *StandardMaterial* present in the Babylon library itself.

For the ball I reduced the specular factor to better represent its rubber material; after this I attached a *Texture* object with a diffuse texture for the color of the ball and a bump texture to give the impression of the rough surface of the ball under the lighting.

The ground received a similar treatment, with the halfcourt color texture and a wooden normal map, which required tiling adjustments through the access of the `uScale` property of the texture.

The backboard of the basket had some adjustments to its color components along with a specular texture assigned to it.

The rim, other than changing the material diffuse and specular component, was also rendered with alpha lower than 1. This means that it appears as slightly transparent. I decided to go for this approach since applying a proper physics bounding box to it was resource intensive, so I wanted to communicate visually that it would not interact with any other physical object.

Finally, the light-bulb sphere material was set to be emissive to represent its giving off light. Other than this, its alpha is also less than 1 to give the impression of the material of a light-bulb.

After all this, the proper positions and rotations are set for each mesh.

In order to have them be moved together with only one transformation, I also set up a hierarchical structure, so that the rim and the light-bulb are children of the board; in turn the spot-light inside the light-bulb is child to the light-bulb mesh itself.

3.4 Physics

To represent more realistically the throwing of the ball and its interactions with the scene, I enabled a physics engine. As mentioned in the libraries section, the Cannon engine is used.

In order to simulate the physical interaction of the scene, it is necessary to assign to each object an *Impostor*. Each engine has different types of impostors: in the case of Cannon, we can use a sphere or box impostor to be attached to the ball, the backboard and the ground. Both the player mesh and the rim thus don't have any physical interactions since it would slow down the application dramatically, forcing us to use unrealistic approximation to have proper performance.

At the end of the player shooting animation, we can then use the *applyImpulse* method of the sphere impostors to impress a force on the ball, giving the impression of being thrown.

Initial testing had some weird interactions happening, leading me to find out that the ball carries angular and linear velocity values from the animation and also across pauses of the physics engine itself, so it is best to set them back to 0 after each shot to prevent unrealistic accumulation of velocity across different shots. As an additional measure to prevent such problems, I turn off the ball physics when not needed.

Notice: There seems to be a bug on the Cannon engine's end: sometimes the ball just penetrates the floor without colliding and bouncing on it. I couldn't find a specific trigger for this so it stays as a bug for the project.

3.5 Importing mesh and skeleton

The player consists of a model imported from the Babylon asset library, called `dummy3.babylon`, which uses its own format to store the necessary mesh and skeleton information. The model also comes with animation, but as indicated in the project instructions I did not load them and instead relied on the tween.js library, as shown later.

We can then directly import this object with the *SceneLoader* component of Babylon. We choose to use its *ImportMeshAsync* method, so that we can use a "promise.then()" structure for our code. This ensures that all interactions pertaining the player happen only after the model is actually loaded in the scene.

Our import has returned both a mesh and a skeleton, which we load into their own variable for ease of access. The skeleton represents all the hierarchical relationships between the single meshes that compose the model. They are connected, so moving the skeleton will move the relative mesh.

Each component of the skeleton is called "bone", so we assign all relevant bones to individual variables in order to access them more easily during animation.

At the end of this procedure, we go on to attach the ball to one of the hands, so that it is moved during the animations along with the skeleton.

3.6 Animation

The Babylon library has a module that can be used to carry out the animation, assigning it to a skeleton to be triggered when needed. I tried using this, but ultimately decided against it because controlling a great amount of properties at once required creating an individual animation instance for each of them, with most documentation hinting towards a better approach being that of creating animations in external software; this prompted me to explore the solution proposed during the lectures.

The animation of the skeleton is thus carried out making use of the **tween.js** library, which can interpolate several properties at once without issue.

To properly operate the tweens, I created a structure containing all the properties to be animated, called `bone_params`.

After this I created two different tweens, to separate the shooting action into 2 phases:

- a wind-up animation where the player charges up the shot
- a shooting animation where the player releases the energy accumulated and shoots the ball to its front

Each phase uses a different easing function, giving a different impression to the animation.

In the first case, I used a quadratic-out function, which gave a springier appearance to the player crouching, getting ready to shoot. For the second part I instead opted for a faster quartic-in function to simulate how the player releases its energy more quickly to impress force on the ball.

Other than "tweening" the given parameters, we also need to actually assign them to the respective bone properties. This is done in the *onUpdate()* function of the tween, which performs the prescribed action each time the given parameters are updated.

The updating has to be triggered each frame of the animation, by calling the *registerBeforeRender* method of the scene, which executes a given function before the rendering of each frame.

This is also where we update all values that are changed during rendering.

Each tween has an *onComplete* function that can be executed at the end of the tween.

For the shooting tween, I set this to release the ball from the player tree and reactivate its physics; after this the ball velocity is reset to prevent it from carrying over across shots, and finally a force is applied to throw the ball in a predefined direction. Notice that this force is scaled by `impulse_scaling`, which is defined through use of the GUI as the detailed in the next section.

3.7 GUI and user interaction

The Babylon library offers an additional component that provides both 2D and 3D GUI elements. I made use of several of the options offered in its 2D GUI package to manage the user's interaction with the game and scene, as well as some HUD elements to relay information to the user.

Besides the camera that can be controlled with a click-and-drag across the whole screen, every other action can be performed with the use of buttons.

The *Button* class has some useful functions to execute callbacks which I exploited to properly interface the game.

The most interesting usage of this is for the "Shoot ball" button. This makes use of two different tools: *onPointerDownObservable* and *onPointerUpObservable*

The difference between the two is that the first executes callbacks when the button is first clicked, whereas the second one executes it when the click is released.

The "Shoot ball" button thus sets the appropriate flags to direct the code flow in the render loop, resets the ball and reattaches it to the skeleton and finally starts the wind-up animation when the click is pressed down.

One of the triggered flags starts the filling of the power bar, indicating that the shot's power is increasing as long as the button is kept pressed.

Only when the button is released the filling is interrupted, calculating also the power of the shot with a time variable, offset with a value stored previously, when we first pressed the button. A modulo operation instead loops the filling and the scaling back to 0 after it exceeds a certain amount, which I chose so that it would slightly overshoot the basket.

The power bar is composed of two *Rectangle* layers, where the first one is a simple opaque white rectangle. The second one instead starts with a width equal to 0 and only starts raising until it reaches the same width as the white rectangle when the button is pressed. The time computed above for the power scaling of the shot is transformed with a simple ratio manipulation into the pixel width.

The HUD displays the current and high score on the top left with 2 text boxes. In order to score a point, the previously instantiated invisible disc mesh is used. In the same *registerBeforeRender* block we mentioned earlier, we check if the ball's mesh intersect with the disk by calling *intersectsMesh* on it. This also triggers a flag that prevents each frame from registering a score, which is reset only when the user presses the "Shoot ball" button again.

The last interesting component of the HUD is the timer, which is used during the "Timed mode". This timer is calculated by taking the *deltaTime* property of the scene: this relays how much time in milliseconds passed between the rendering of two frames. It is thus enough to translate it to seconds and subtract it from a previously initialized timer variable each frame; it is then rounded by ceiling (so we only display the whole seconds) and finally the text box with the remaining time is updated. Once the timer has run out, the high score cannot be upgraded any more thanks to a flag that is triggered.

The other buttons are rather straightforward, triggering some flags to direct the code flow, starting for example the movement of the hoop from side to side, or resetting some values. The "Light-bulb toggle" also changes the `alpha` value of the light-bulb mesh and the contained light's intensity, so it can effectively disappear from the scene while turned off.

One last detail to point out is that I tried to express the idea of whether an option is activated or not with the background color of each button. A black background means that the related option is disabled, whereas a purple one has the opposite meaning.

Moreover, the free mode and timed mode are mutually exclusive, so pressing one of them changes the color of the other whenever the mode is switched.