Defense 2 : EpiKart

Undergraduate (1st Year) EPITA

27/04/2022

TEAM BINBINKS

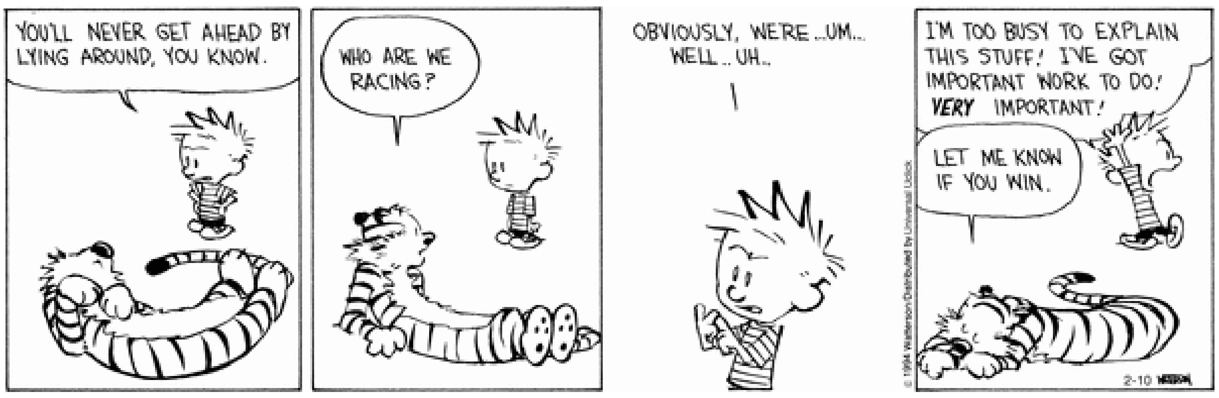
All the members are from A3

Eliott Flechtner (Project leader)

Lancelot Doucet   Titouan Verhille

January - June 2022

Summary


YOU'LL NEVER GET AHEAD BY LYING AROUND, YOU KNOW.
WHO ARE WE RACING?
OBVIOUSLY, WE'RE ...UM... WELL.. UH..
I'M TOO BUSY TO EXPLAIN THIS STUFF! I'VE GOT IMPORTANT WORK TO DO! VERY IMPORTANT!
LET ME KNOW IF YOU WIN.
© 1994 Watterson/Distributed by Universal Uclick
2-10

1 Introduction

1.1 Book of Specifications Recap

This document constitutes the **Second Defense Report** for the second semester project of the Team BINBINKS, which runs from January to June 2022. The name of the project is EpiKart and is an **arcade-style racing video game** with a focus on fun and friendliness. The game features **online multiplayer as well as local play and artificial intelligence**.

The aim of this project is to put us in the condition of an important project realization as in a company. This simulation is directed within the framework of the creation of a software or a video game and will allow us to **familiarize ourselves with the development tools related to the latter**. It also will make us learn more about the creation of a website, the respect of the dates and deadlines, the work in team and its management, the preparation of oral presentations in group and the creation of official documentations.

First of all, we see this project as an opportunity to learn more about **time organization, management and communication as a team and an individual**. The project immediately brings to the forefront the importance of meeting intermediate and final deadlines. It is mandatory to respect these, and even create our own, to adequately deliver the final product. We learn to project and foresee precisely **when every feature needs to be finished for the correct development of the project**. It is an essential quality when working in a company, whatever the sector. Moreover, respecting one's instructions and deadlines is not something that comes to everyone naturally. Thus, **the partitioning of the given time** must also be done by taking into consideration the intensity and quantity of work of the regular classes and evaluations. The project also teaches us to plan for the economic aspect and therefore potentially allocate a budget for our needs. Finally, it teaches us to **document and record its progress as it occurs** and to construct clear and professional reports accordingly through various media such as a Book of Specifications, an oral presentation or a website. This information must also be conveyed simplistically and supported by visuals such as charts, video clips or live demonstrations. Being concise, when necessary, **is also part of the job**!

The particular feature of this type of racing game is **the combat between vehicles to spice up the race** as well as the **zany and varied designs** of the circuits. Players can pick up bonuses during races to slow down others for their own benefit. In addition, mechanics such as drifting and speed boosts pads on the ground allow a player to momentarily increase his speed. Finally, the goal of this kind of game is to get the first place in each race, thus a ranking of the players' times per lap and on the whole circuit **is always displayed and pushes the player to go faster each time**.

To conclude, the EpiKart project is a way for us to progress in all aspects: **social, team management, group work and technical**. Moreover, it teaches us to create a large project over a long period of time in an unguided way to get **as close as possible to life in a company**.

1.2 Second Defense plan

For this Second Defense, we had planned to **clean up the entire code of the latest implementations such as bonuses**. In addition, we had anticipated to also have completely rethought the way we **generate the colours of the karts** in the game room to make it purely random. We also wanted to **allow players to choose their nicknames** when they join the game! We were also expected to try to fix the obvious lag in player movement on the server and add new animations and many in-game particle effects. The kart was supposed to be able to **drift to gain speed momentarily** and its wheels be animated to turn in the direction the user chooses. The in-game user interface will also have been improved by adding a better design for the inventory and information about **the players' positions in the race**. Finally, currently built tracks will be enhanced to make them more polished than the current one!

2 Achievements & Technical aspect

2.1 Project Cleaning

2.1.1 Justifications & interests

When we started working on the project again a few weeks after the first defense in mid-March, we realized that the project **first needed to be completely cleaned up**. This had 3 main interests for us: first, it allowed us to start from a fully functional, standardized, documented and flexible base. Secondly, it allowed us to revise some systems and algorithms by questioning the way they were originally implemented in order **to make them more optimal, manageable and less computationally expensive**. We also had to reconsider many small aspects and processes concerning the way **we had conceptualized complex infrastructures** or simply group decisions. Examples are the synchronization and creation of networked instances or designs concerning the choice of certain colors or the placement of elements on the user interface. Finally, from a group cohesion standpoint, it allowed us **to take a step back in order to clarify all the unclear areas** and to ensure the good understanding of the way our game components interact with each other by all the members. All these modifications were not a waste of time since they allowed us to identify some weaknesses in our way of approaching the implementation of an element we wanted to have in our project. The further development of the project was therefore **greatly facilitated by these key changes to the game**, as we thought.

2.1.2 Changes

We thus undertook the revision of the project over a whole week, keeping a written record of the decisions and changes made. First of all, we all agreed on a **naming convention for the different resources** used in the Unity editor. As we had all worked on different parts of the project, everyone had established their own way of naming variables, scripts and objects in the different scenes. **Homogenizing all of them** allowed us to find our way more easily in the hierarchy of the instances present in each scene, especially when most of our scripts depended on the name of the said resources to access certain internal properties of the latter. This step was thus directly linked to **the reorganization of the code** by adding many comments and indications in our scripts so that any member of the group could quickly and easily understand the role of each method, function and classes but also the link between them. We also took the opportunity **properly sort and organize each resource in specific folders with precise names** to have a quick access to them according to our needs. Then, we took the time to check that each of the objects instantiated on the network **had the right number of components linked to our Photon network system** and that they were all correctly set up. Far from being useless, this step allowed us to address some assignment errors that could have impacted the good development of the project in the future. The player's object was then the object of some changes concerning the way the position and rotation values of some internal components of the 3D model were retrieved, **such as the individual detection of the wheels on the network**.

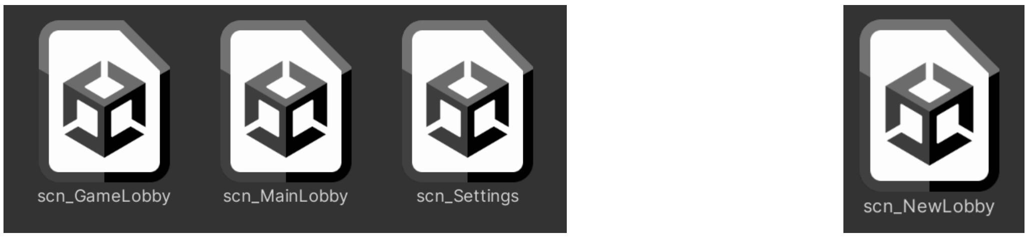
2.2 User interfaces & Menus

2.2.1 Menu Refactor

During the development of the project for the First Defense, we realized that it was mandatory to implement a button to go back in the menus, otherwise the user would have to restart the game each time to return to the main menu. However, adding such a feature required **a major redesign of the way animations**

and transitions between scenes and lobbies were handled. Many other minor flaws led us to believe that an almost complete overhaul of the menus was necessary. We thought that this should not be a problem for the first presentation, but now it was something had to be addressed.

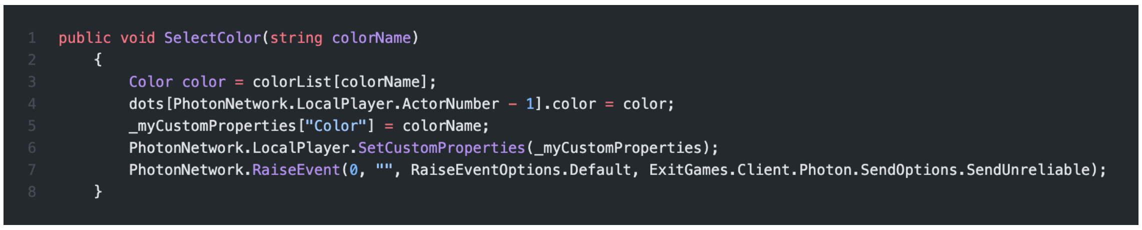
The main problem with our previous implementation was the number of scenes used to represent the menu. There were three of them *(See Figure 1.a Below)* : one for the settings, one for the main menu where the game mode was chosen and one for the waiting lobby once a game was created or joined. We changed the room every time we wanted to change the screen visually. Thus, the state of each of the rooms we left and joined again was not saved, which made it impossible to implement a back button. Our new version of this menu now only **takes place in one scene** *(See Figure 1.b Below)* . The previous rooms are now placed on top of each other in properly separated and easily accessible groups. Everything is managed using **a stack structure**. As soon as the user presses the back button, our program actually retrieves **the last action performed at the top of the stack and applies it in reverse**. In this way, we could navigate completely freely between the different menu options even when a lobby on the network had been created.



a.Previous Scene Organization b.Current Scene Organization

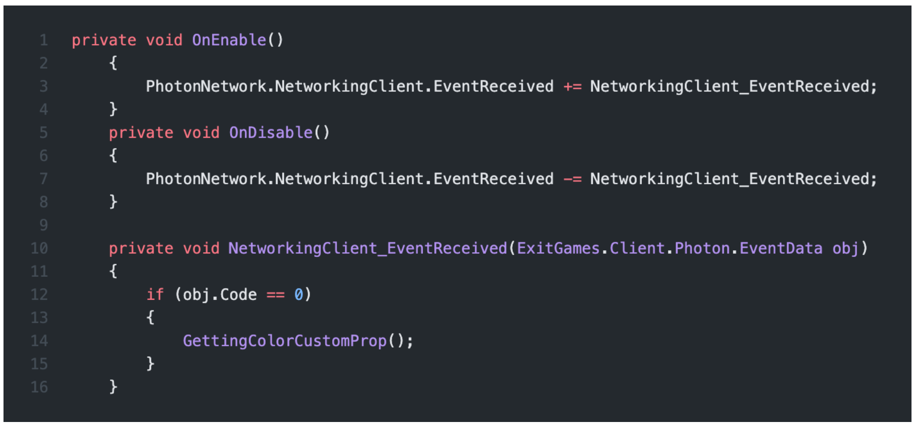
Figure 1: Reworked scene organization

As promised, we completely redesigned the way choice and application of colors for the different networked karts were implemented. Previously, we used the color associated with the player in the menu to give them their in-game color, but the color distribution **was not random**. Moreover, none of the players could change their color from the menu or in the game scene. This is no longer the case with our new system since we have discovered **the Custom Properties offered by our Photon servers** *(See Figure 2. Below)*. They are a kind of variable attached to each client connecting to the global server, reachable and readable by all components instantiated on the network. They allow **to synchronize the appearance** of each one online and to transmit their values easily *(See Figure 3. Below)*. We then implemented a new system to **randomly choose colors for each player** and assign it on the network *(See Figure 4. Below)*. We also allowed each player to be able to change his color once in the waiting room using **a custom pop-up window** *(See Figure 5. and 6. Below)*.



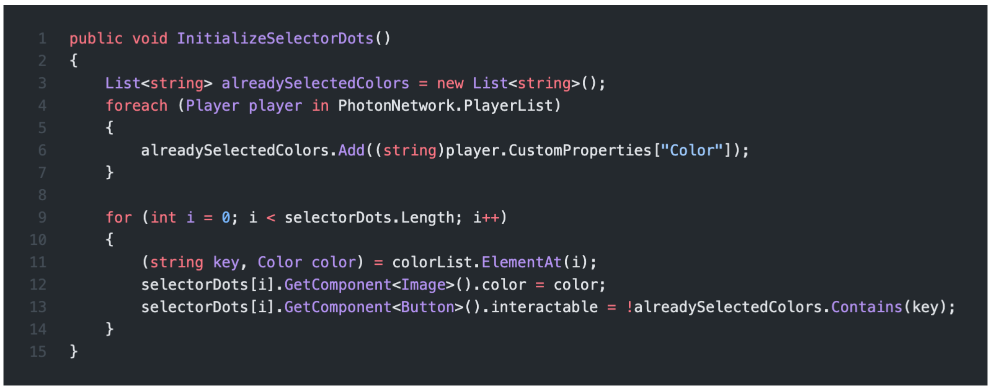
```
public void SelectColor(string colorName)
    {
        Color color = colorList[colorName];
        dots[PhotonNetwork.LocalPlayer.ActorNumber - 1].color = color;
        _myCustomProperties["Color"] = colorName;
        PhotonNetwork.LocalPlayer.SetCustomProperties(_myCustomProperties);
        PhotonNetwork.RaiseEvent(0, "", RaiseEventOptions.Default, ExitGames.Client.Photon.SendOptions.SendUnreliable);
    }
```

Figure 2: SelectColor script (synced with Photon)



```
private void OnEnable()
    {
        PhotonNetwork.NetworkingClient.EventReceived += NetworkingClient_EventReceived;
    }
    private void OnDisable()
    {
        PhotonNetwork.NetworkingClient.EventReceived -= NetworkingClient_EventReceived;
    }

    private void NetworkingClient_EventReceived(ExitGames.Client.Photon.EventData obj)
    {
        if (obj.Code == 0)
        {
            GettingColorCustomProp();
        }
    }
```

Figure 3: EventColorChange script (synced with Photon)



```
public void InitializeSelectorDots()
{
    List<string> alreadySelectedColors = new List<string>();
    foreach (Player player in PhotonNetwork.PlayerList)
    {
        alreadySelectedColors.Add((string)player.CustomProperties["Color"]);
    }

    for (int i = 0; i < selectorDots.Length; i++)
    {
        (string key, Color color) = colorList.ElementAt(i);
        selectorDots[i].GetComponent<Image>().color = color;
        selectorDots[i].GetComponent<Button>().interactable = !alreadySelectedColors.Contains(key);
    }
}
```

Figure 4: InitializeDots script (synced with Photon)

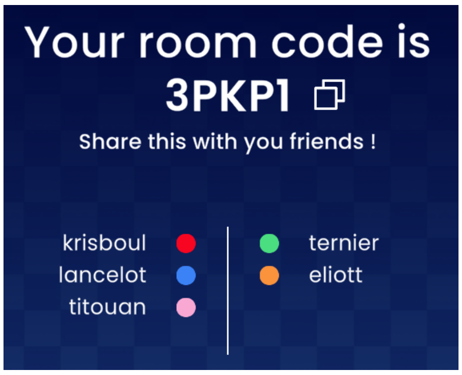


Figure 5: Peculiar Game Lobby

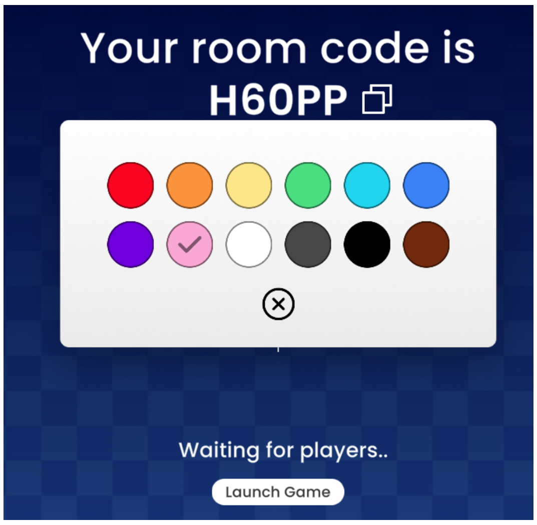
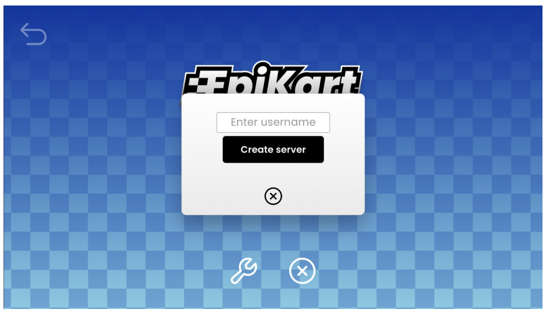
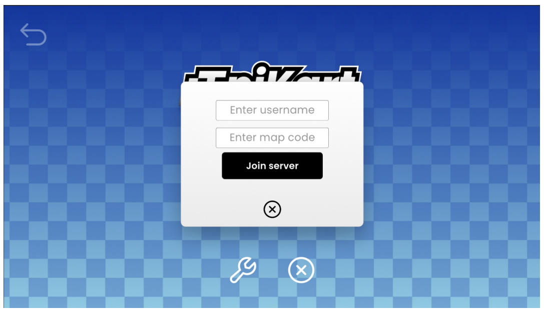


Figure 6: Color picker in Game lobby

Finally, we added the possibility for the player to **choose their username before creating or joining a room** on the network using a regularized expression limited to 15 non-special characters but which can include numbers *(See Figure 7. below)*. Photon already had a system to keep in memory and on the network the name of each player: we only had to **modify this attribute for each player**. We took care to prevent a user from sending a specific command that could exploit our Photon server in an unwanted way.



a.Screen username



b.Screen username + map code

Figure 7: Reworked scene organization

2.2.2 In-Game UI

Regarding the in-game User Interface (UI), we went through a design and research phase during which we were inspired **by many racing games to find what best fit our expectations** *(See Figure 8. below)*. We then simply added icons found in online databases and plugged in the different statistics of each player such as his rank in the race, his number of laps, his inventory and the time per lap *(See Figure 9. below)*.

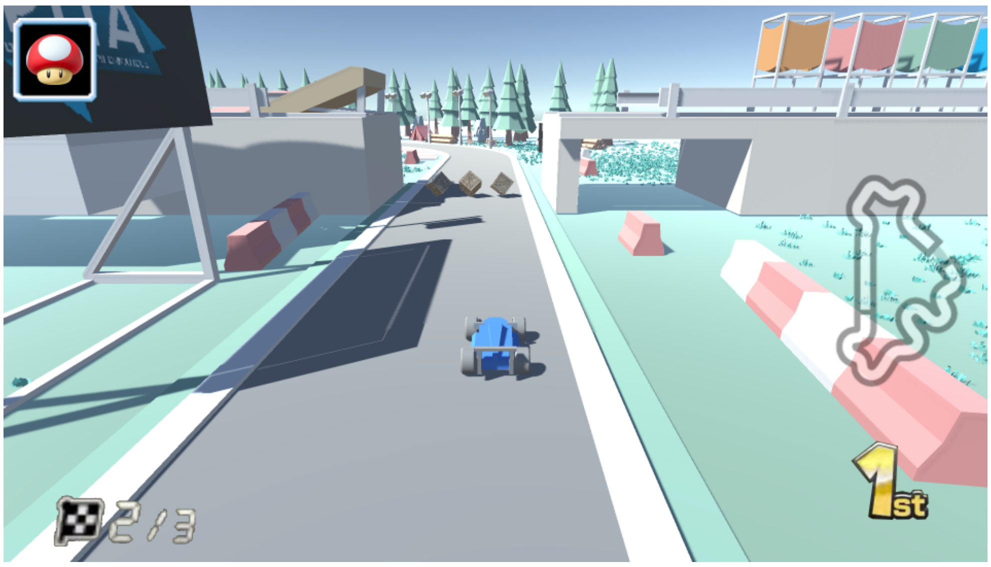


Figure 8: Inspiration for In Game UI

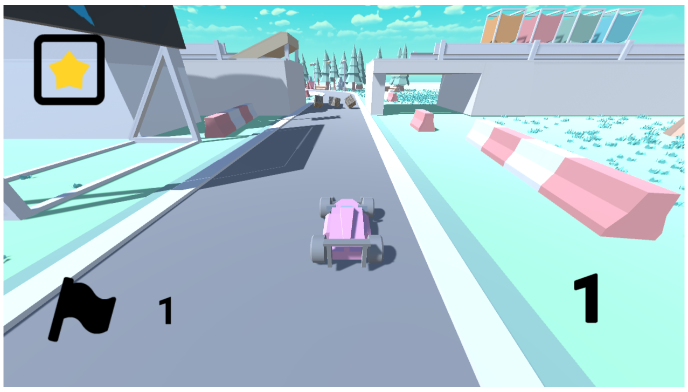


Figure 9: Actual render of In Game UI

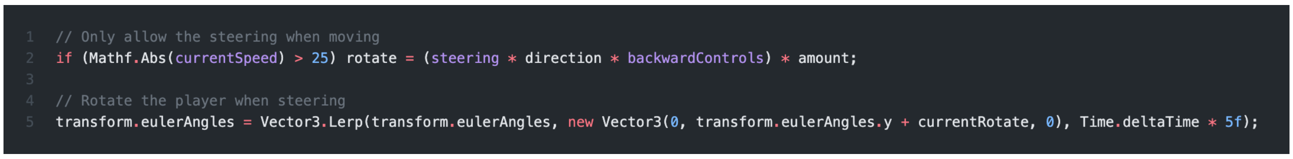
2.3 Kart movement

2.3.1 Bad controls & User's feeling

Since the beginning of the project, we had settled on developing a basic system of movement and control of the player and thus of each kart. We tried to make it **as easy to handle as possible** so that a third party not familiar with racing video games could immediately play. However, despite the time and attention paid to the player's movement script, there were still some significant problems. Most notably, when the user pressed one of the directional keys to go either left or right, there was **a significant delay before the car could turn**. This made driving and steering really complex to master as the delayed response of the keyboard inputs forced the user to anticipate a change in direction long before it was actually needed. On top of that, the animation related to the rotation of the 3D model of the kart when turning **was not smooth enough to look natural and simulate the real movement of a racing car**. What was although missing above all was a real feeling of speed in our players' movements. Indeed, we realized that the game lacked a lot of dynamism and energy: in its current state, we could not make the players feel the speed at which the karts were moving... To really succeed in simulating this experience for the players, we had to think again about the defining elements that would enhance it. After a lot of online research into the subject and watching a variety of gameplay footage from other racing video games, **we made a non-exhaustive list of things we could do to improve our game**. These included improving the player's controls while increasing the speed at which the player moves around the track. We also thought about making the camera more dynamic by increasing or decreasing its *Field Of View* (FOV) during acceleration or deceleration **as well as adding many particle systems to improve the visual aspect**.

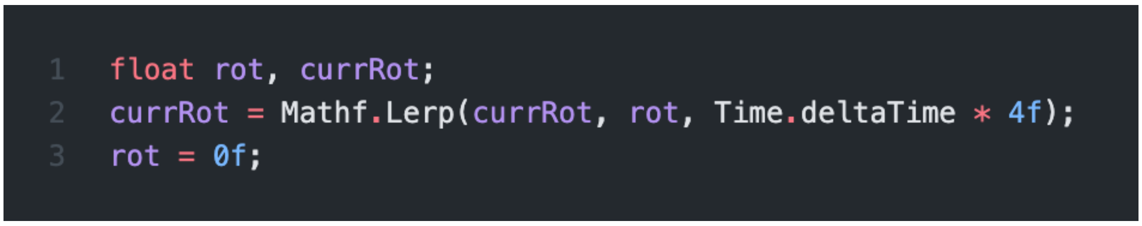
2.3.2 Rotation delay fix

First, we needed to address how the change of direction on the player's horizontal axis was programmed. Previously we used **Euler angles to determine and change the orientation of the kart** *(See Figure 10. below)*. We used two floating variables to define the local rotation of the 3D model to be performed *(See Figure 11. below)*. First, the variable *rot* acting as a "clock" that went from an arbitrary and manually defined maximum value to 0 depending on the user inputs. The second variable named *currRot* stored the current rotation angle of the car and used the linear interpolation function Lerp from the *Mathf* library. It was applied on the *rot* variable to make the rotation of the 3D model evolve **in a linear and progressive way**. This was then updated with each new image of the game and modified the Euler angles of rotations constituting the angles relative to the scene's World. This system is also very close to the system currently used to **calculate the acceleration and deceleration values of the kart**.



```
// Only allow the steering when moving
if (Mathf.Abs(currentSpeed) > 25) rotate = (steering * direction * backwardControls) * amount;

// Rotate the player when steering
transform.eulerAngles = Vector3.Lerp(transform.eulerAngles, new Vector3(0, transform.eulerAngles.y + currentRotate, 0), Time.deltaTime * 5f);
```

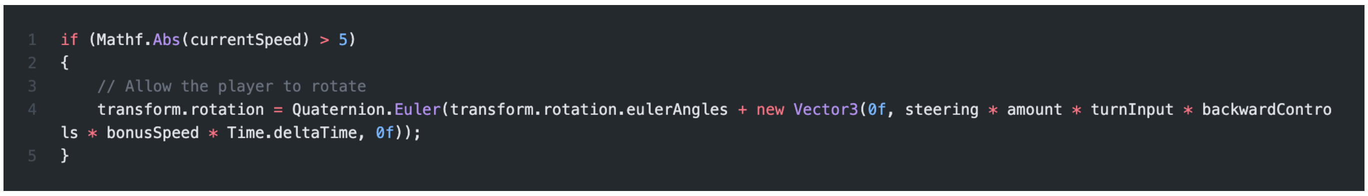
Figure 10: Old version using Euler angles



```
float rot, currRot;
currRot = Mathf.Lerp(currRot, rot, Time.deltaTime * 4f);
rot = 0f;
```

Figure 11: Floating variables rot & currRot

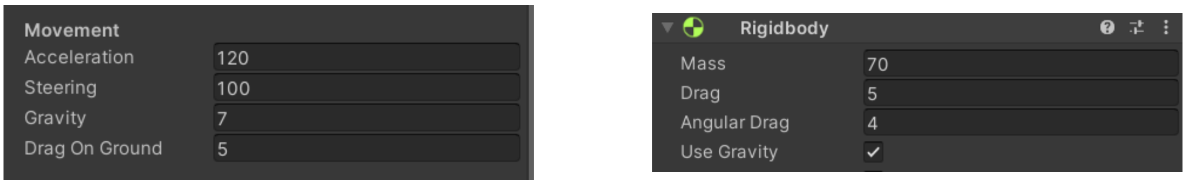
Although this method worked mathematically speaking, it had one major problem: the order in which the angles were processed in Unity. Indeed, the use of Euler angles - returning a three-dimensional vector - **applied first the rotations on the Y axis before those on the X and Z axis**. However, the 3D model of our kart was originally oriented to swivel locally around the Z axis, which was processed last in our code. The direct consequence was the progressive amplification over time of an unbalance of the values then **used in the calculation of the linear interpolation of the kart rotation**. Visually, this resulted in the perception of a certain delay during a change of direction when in reality it was merely a computational miscalculation. We decided to modify this system so that we no longer had any variables that handled the car's rotation. Instead, we used the ***Quaternion* structure to represent rotations in Unity** *(See Figure 12. below)*. By combining it with the Euler method, we were able to process the car's rotation angles in the desired order, here along the Z, X and Y axes respectively.



```
if (Mathf.Abs(currentSpeed) > 5)
{
    // Allow the player to rotate
    transform.rotation = Quaternion.Euler(transform.rotation.eulerAngles + new Vector3(0f, steering * amount * turnInput * backwardContro
ls * bonusSpeed * Time.deltaTime, 0f));
}
```

Figure 12: New *Quaternion* rotation structure

Changing the force and way with which the player was turning directly allowed us to solve the problem of the player's movement speed *(See Figure 13. below)*. Indeed, after many experiments, we managed to modify the maximum speed value until we reached suitable ones. At the same time, we took the opportunity to increase the value responsible **for the player's ability to make more or less sharp turns**.



Movement	
Acceleration	120
Steering	100
Gravity	7
Drag On Ground	5

a. Updated movement variables values

Rigidbody	
Mass	70
Drag	5
Angular Drag	4
Use Gravity	✓

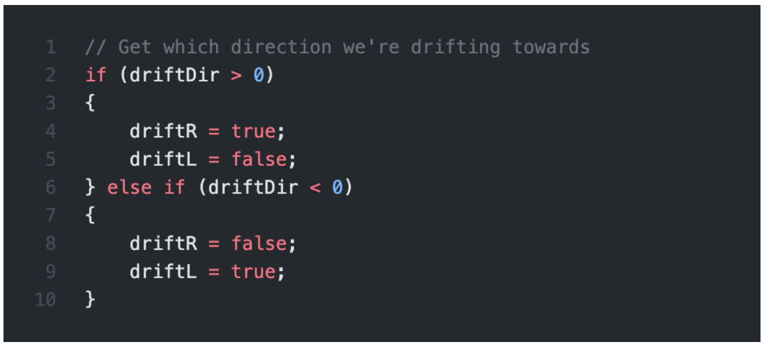
b. Changed physics parameters

Figure 13: Reworked Player & Collider values

2.3.3 Drifting mechanic

Finally, we wanted to implement a drifting mechanic to allow experienced players with the highest skill level to spice up their races. Ideally we wanted to be able to **give a speed boost** for a short time after each drifting to get players to use the feature more often. However, we felt that the focus on correcting the basic movement errors of the players was more important. So we decided not to implement this mechanic for this Second Defense. Instead, **we have already started to develop the premises of the algorithm that manages the player's ability to drift**. However, we thought it was important to describe what has been done with the implementation so far.

In order to detect whether the player was intending to drift or not, we first had to detect the pressing of a certain key on the keyboard or controller. We arbitrarily decided **to assign the 'space' key** as an indicator that a player was about to start drifting. We then had to detect which side the player wanted to drift towards: two boolean variables named *driftL* and *driftR* were used to determine whether one of the directional keys was currently pressed by the user *(See Figure 14. below)*. The control of the kart would then change according to both their values. We therefore had to modify the previously described **method of calculating the rotation of the player model**. In the case where none of the directional keys were pressed then the movement was not affected. We therefore had to study video clips to better visualize how a racing car would look when drifting. Thus, if the player was detected as having activated the drift ability, **the local rotation of the car was accentuated so that the player was now moving across the track**, almost perpendicular to the road. In this way, the rear of the car pointed in the opposite direction to where the user was drifting to better simulate a real race car



```
// Get which direction we're drifting towards
if (driftDir > 0)
{
    driftR = true;
    driftL = false;
} else if (driftDir < 0)
{
    driftR = false;
    driftL = true;
}
```

Figure 14: Detection of Drifting direction

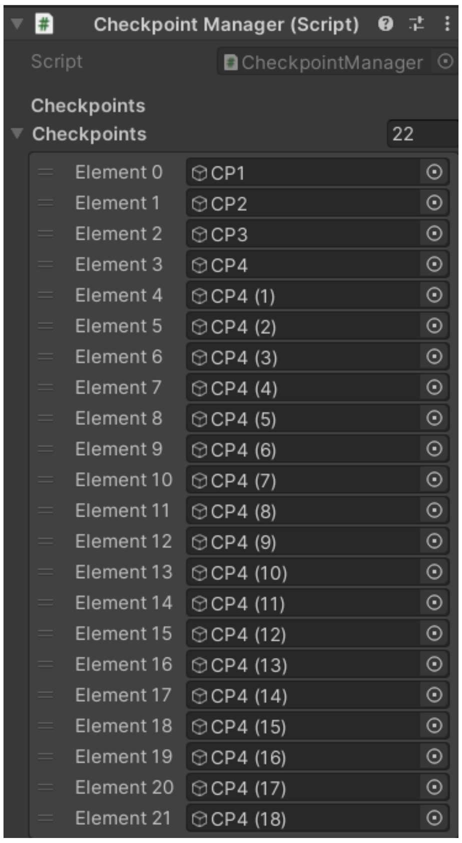
.

2.4 Checkpoints, Laps & Ranking system

2.4.1 Checkpoints & Respawn points

During the First Defense, we had developed a very basic respawn system for the player that consisted in detecting a collision with a certain object named "Killzone" that teleported the player to his initial position at the beginning of the race. Although this method was functional, it was still very simplistic and didn't take into account the player's progress in the current round. We had to do something about it by implementing a much more complex checkpoint system.

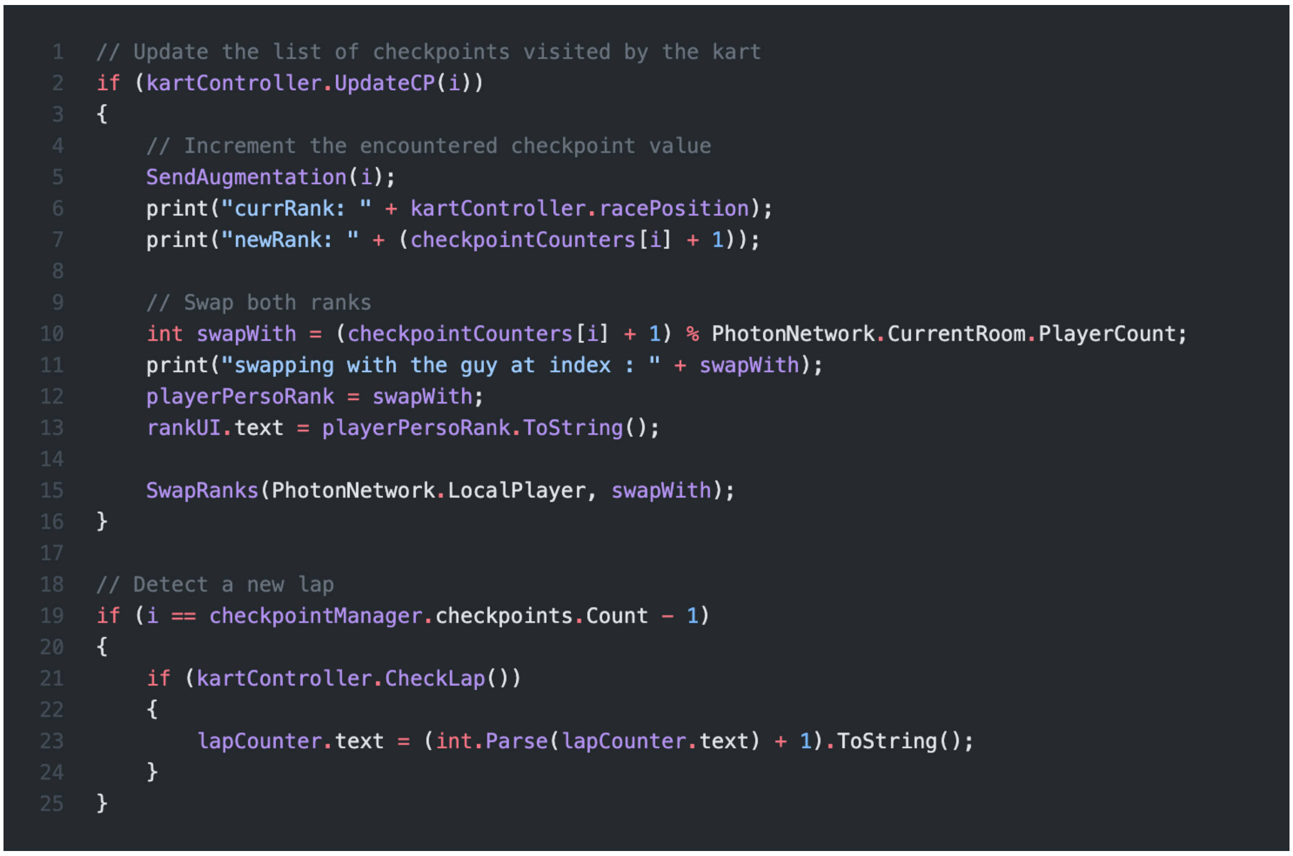
We thus created an object always present in the game scene that stores all the checkpoint objects *(See Figure 15. below)* - which are just invisible cubes stretched to the dimensions of the track - in a public list. When each player detects a collision with one of the checkpoints on the circuit, it first detects **whether the player has not cheated and skipped a certain number of checkpoints before**. If it is not the case, the counter associated with the collided checkpoint is increased by one and is sent to the network thanks to an event system integrated in Photon. The checkpoint is then marked as "seen" by the player and constitutes **the last possible respawn point of the latter**. At the end of a turn, our algorithm detects that the just-seen checkpoint is the last one in the list. If it is the case it resets the list of checkpoints seen by the player **while increasing his lap counter by one**. *(See Figure 16. below)*



Checkpoint Manager (Script)

Script	CheckpointManager
Checkpoints	
Checkpoints	22
Element 0	CP1
Element 1	CP2
Element 2	CP3
Element 3	CP4
Element 4	CP4 (1)
Element 5	CP4 (2)
Element 6	CP4 (3)
Element 7	CP4 (4)
Element 8	CP4 (5)
Element 9	CP4 (6)
Element 10	CP4 (7)
Element 11	CP4 (8)
Element 12	CP4 (9)
Element 13	CP4 (10)
Element 14	CP4 (11)
Element 15	CP4 (12)
Element 16	CP4 (13)
Element 17	CP4 (14)
Element 18	CP4 (15)
Element 19	CP4 (16)
Element 20	CP4 (17)
Element 21	CP4 (18)

Figure 15: CheckPoint Manager in Inspector

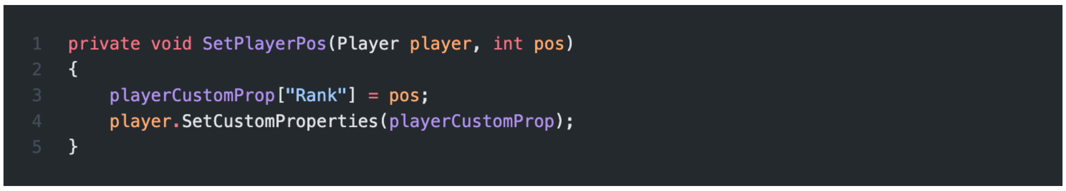


```
// Update the list of checkpoints visited by the kart
if (kartController.UpdateCP(i))
{
    // Increment the encountered checkpoint value
    SendAugmentation(i);
    print("currRank: " + kartController.racePosition);
    print("newRank: " + (checkpointCounters[i] + 1));

    // Swap both ranks
    int swapWith = (checkpointCounters[i] + 1) % PhotonNetwork.CurrentRoom.PlayerCount;
    print("swapping with the guy at index : " + swapWith);
    playerPersoRank = swapWith;
    rankUI.text = playerPersoRank.ToString();

    SwapRanks(PhotonNetwork.LocalPlayer, swapWith);
}

// Detect a new lap
if (i == checkpointManager.checkpoints.Count - 1)
{
    if (kartController.CheckLap())
    {
        lapCounter.text = (int.Parse(lapCounter.text) + 1).ToString();
    }
}
```

Figure 16: Update CP KartController

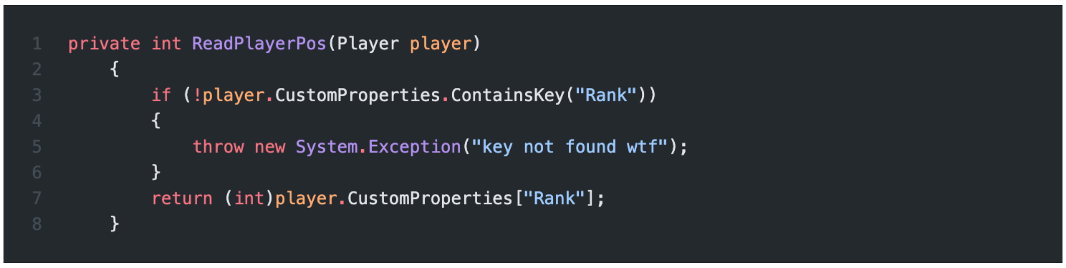
2.4.2 Ranks detection

Concerning the way we managed the ranks between the players, each player gets the event sent by Photon at each new checkpoint collision and **modifies his position live according to the others**. We thus use a function to swap the ranks of two players taking as arguments the values recovered from Photon being the involved player, **its current rank and its new one**.



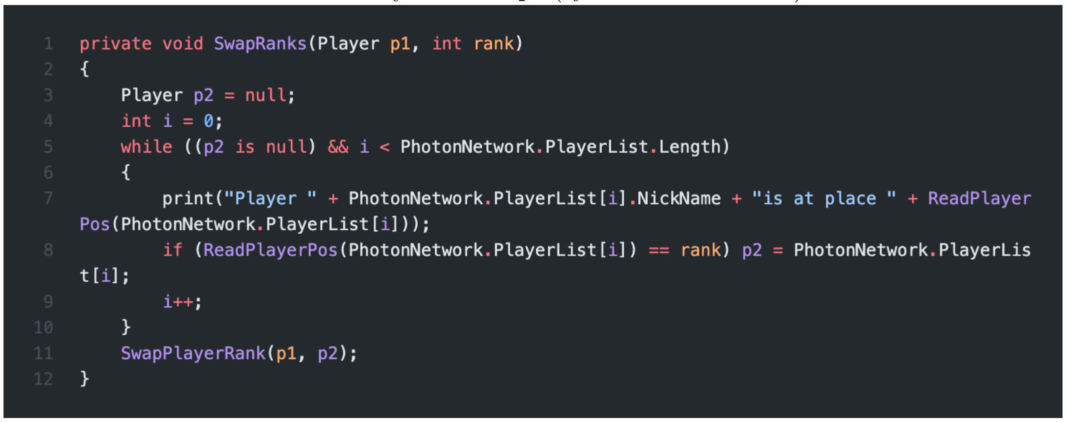
```
private void SetPlayerPos(Player player, int pos)
{
    playerCustomProp["Rank"] = pos;
    player.SetCustomProperties(playerCustomProp);
}
```

a. SetPlayerPos script (synced with Photon)



```
private int ReadPlayerPos(Player player)
    {
        if (!player.CustomProperties.ContainsKey("Rank"))
        {
            throw new System.Exception("key not found wtf");
        }
        return (int)player.CustomProperties["Rank"];
    }
```

b. ReadPlayerPos script (synced with Photon)



```
private void SwapRanks(Player p1, int rank)
{
    Player p2 = null;
    int i = 0;
    while ((p2 is null) && i < PhotonNetwork.PlayerList.Length)
    {
        print("Player " + PhotonNetwork.PlayerList[i].NickName + "is at place " + ReadPlayerPos(PhotonNetwork.PlayerList[i]));
        if (ReadPlayerPos(PhotonNetwork.PlayerList[i]) == rank) p2 = PhotonNetwork.PlayerList[i];
        i++;
    }
    SwapPlayerRank(p1, p2);
}
```

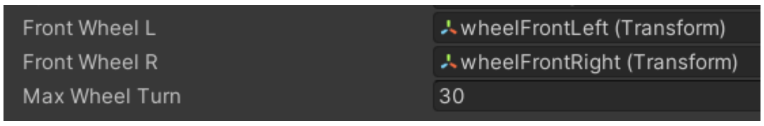
c. SwapRanks script (synced with Photon)

Figure 17: Scripts to manage Ranking with CustomProperties

2.5 Aesthetics, Power-ups & SFXs

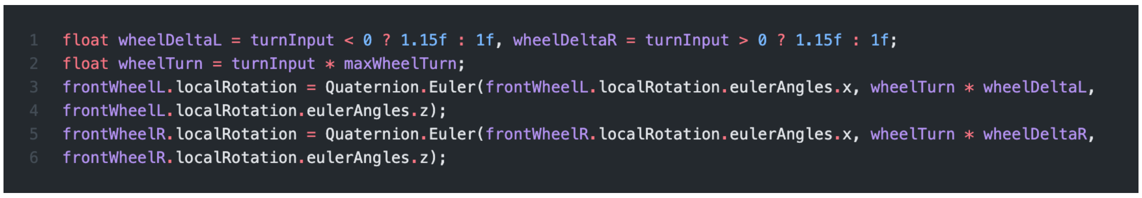
2.5.1 Karts wheels animations

In order to improve the look and feel of the game, we decided together to add a lot of decorative elements and some smaller details to really make this improvement possible. We wanted to add an extra touch to the visuals we had already implemented for the First Defense. First, we felt it was important to make each player's kart more realistic and expressive. We thus identified several elements that we could make more aesthetically pleasing. First, **we felt it was important that the left and right front wheels of each kart's 3D model were animated** according to the current direction in which each player was going. To do this, we declared two new public variables named *frontWheelL* and *frontWheelR* containing the 3D models of the two independent front wheels of the chassis, respectively left and right, of the kart *(See Figure 18. below)*. By accessing the transform components allowing to obtain the coordinates of the latter, we used **the *Quaternion* structure coupled with the Euler module** to modify the local rotation of the wheels. Located only on the Y-axis, we made sure to keep the initial rotation of the two other axes. The calculation of the rotation of the wheels is given by multiplying the integer, positive or negative and designating the user input of the directional keys, by an arbitrary scalar as well as by an angle varying according to the time and other parameters. For even more realism, we made sure **to turn more the front wheel placed on the side towards which the car is heading** since it is on this side that the effect of the turn is most felt on the tires in general. *(See Figure 19. below)*.



Front Wheel L	wheelFrontLeft (Transform)
Front Wheel R	wheelFrontRight (Transform)
Max Wheel Turn	30

Figure 18: Individualized wheels models



```
float wheelDeltaL = turnInput < 0 ? 1.15f : 1f, wheelDeltaR = turnInput > 0 ? 1.15f : 1f;
float wheelTurn = turnInput * maxWheelTurn;
frontWheelL.localRotation = Quaternion.Euler(frontWheelL.localRotation.eulerAngles.x, wheelTurn * wheelDeltaL,
frontWheelL.localRotation.eulerAngles.z);
frontWheelR.localRotation = Quaternion.Euler(frontWheelR.localRotation.eulerAngles.x, wheelTurn * wheelDeltaR,
frontWheelR.localRotation.eulerAngles.z);
```

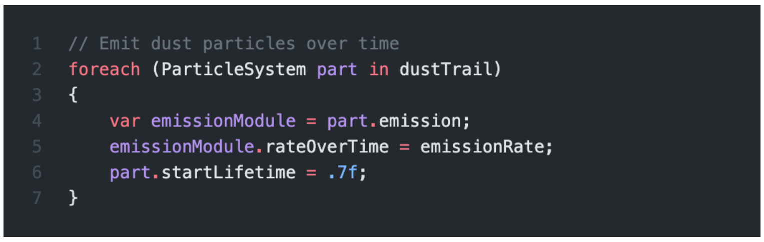
Figure 19: Wheel animation formulas

We also thought about turning the wheels forward or backward to again reinforce the sense of realism and speed we wanted to convey, but the 3D model that karts use at the moment have round wheels **with little to no textures which made such an implementation unnecessary**.

2.5.2 Particles & Skybox

We were also very interested in the particles and its dedicated editor offered by Unity. With the help of resources found online and the official documentation, we managed to create our own smoke particle system. We had to create a new set of particles attached to the player's kart since it should follow him later on. We then proceeded to import a smoke texture as a *PNG* image found online. All that was left to do was to modify the parameters concerning **the shape, the size, the color and the frequency of appearance** of these in the editor integrated into Unity by default. We then placed it under the two rear wheels of the kart to show the player the impact of the terrain the car is driving on. A quick modification of the code in the script linked to the players allowed us to activate and show **the latter only when the player is not in the air or stationary**

(See Figure 20. below). The interest of this method is its ability to be easily modified and to create a multitude of different particles, easily replaceable and accessible.



```
// Emit dust particles over time
foreach (ParticleSystem part in dustTrail)
{
    var emissionModule = part.emission;
    emissionModule.rateOverTime = emissionRate;
    part.startLifetime = .7f;
}
```

Figure 20: Particles activation in player's script

We also took the opportunity to change the sky texture in the game scene, also called skybox. After a long search on the official Unity asset and resource store, we came across a texture pack that we all pretty much liked. After several tries, we agreed on one of them and chose to keep it *(See Figure 21. below)*. We just had to click, drag and drop the texture from the integrated file explorer **on the sky to apply it in the game scene**. However, we had to set the ambient light of the scene again and learn to manage the rendering problems linked to these modifications. It happened for example that we managed to have a rendering in the editor that we liked but, once the program was compiled, **did not render at all as expected**.

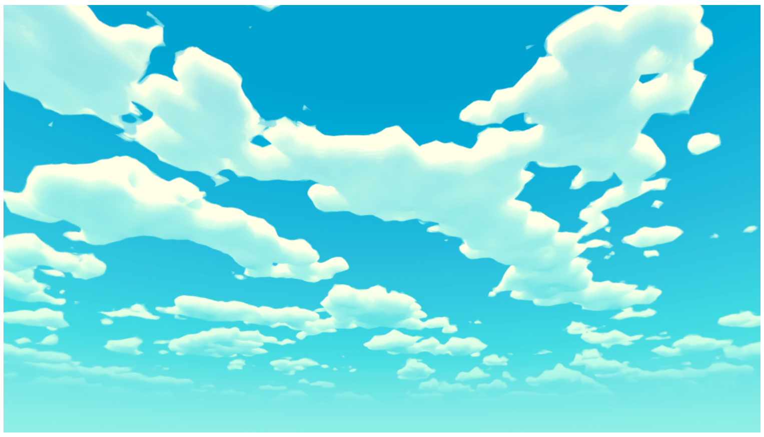
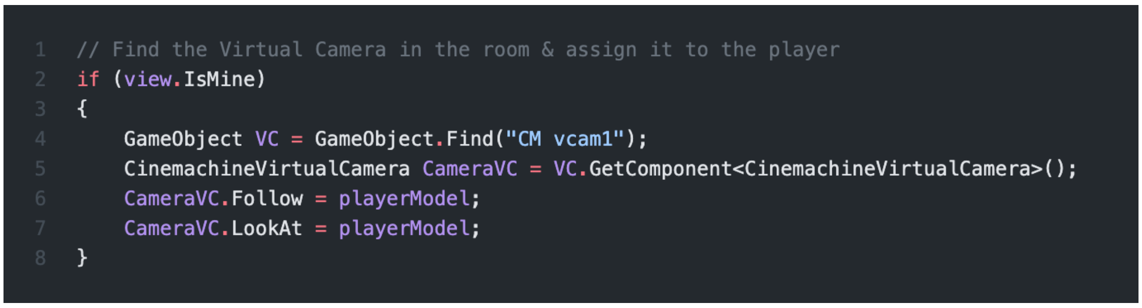


Figure 21: Chosen Skybox

2.5.3 Dynamic camera

We were very happy with the results of the changes and the improvements made, but we still felt that there was something visually missing. We later realized that it was the camera that we found too fixed, static, boring, passive and ultimately not dynamic enough. We then looked for better alternatives for such cameras online and found out about Cinemachine. We were able to download it directly from Unity very easily.

It contained **a huge amount of professional resources to manage everything** from framing and scene composition to creating cutscenes, cinematics and simulating a large number of different lenses. By browsing through some online videos, we first created a so-called virtual camera which is roughly the principal brain of all the Cinemachine components of the current scene. Upon its addition to the scene, our players' camera types changed as they instantly became controlled by the main virtual camera. By clicking and dragging the model of one of the players in the scene onto the main virtual camera, **the latter would automatically start to follow him**. We therefore changed some of the values and properties of the camera-brain, such as the time it takes for it to move towards or away from the player depending on its speed. In addition, we had no trouble implementing a completely new camera system despite the fact that all cameras had to be synchronised on the game servers. A few lines in the player control script were enough to fix any problems very quickly *(See Figure 22. below)*.



```
// Find the Virtual Camera in the room & assign it to the player
if (view.IsMine)
{
    GameObject VC = GameObject.Find("CM vcam1");
    CinemachineVirtualCamera CameraVC = VC.GetComponent<CinemachineVirtualCamera>();
    CameraVC.Follow = playerModel;
    CameraVC.LookAt = playerModel;
}
```

Figure 22: Fix for camera problems on network

2.5.4 New power-ups

We had planned to implement one or two bonuses for this Second Defense, but having spent more time on the development of some parts than others, we could only add one. The new bonus that is now in the game is called "God mode" and simply consists of giving a **short invincibility** to the player who uses it as well as a small speed boost. For example, a player who has activated such a bonus will not be affected by an "Oil Puddle" that is dropped on the ground upon contact. We've also made sure that the "Oil Puddle" is not removed on collision, but remains on the ground to continue to hinder and block the way for other players.

Since we implemented bonuses, we had always decided which bonus a player picked up completely randomly. However, we have been thinking for a long time about changing this method, which does not benefit the last players in the race at all, but rather the first ones. If the distribution of bonuses is completely uniform and **equiprobable** regardless of the rank of the current player, then the leader of the race will always have a fairly high chance of getting a bonus that will increase his lead over the rest of the field, which is not necessary being already first. For example, the bonus described above is a **net improvement in the abilities** of the player who gets it, so we didn't want it to fall into everyone's hands.

We thus decided to write a **pseudo-random bonus distribution algorithm** based on the rank of each kart in the game. It is based on the following rule: the rarity of a bonus is inversely proportional to the strength of the rank given in the arguments of this method *(See Figure 23. below)*. For example, being fifth in a six-player game, we would be much more likely to get a "God Mode" bonus, very rare to get for the first or even the second in the race. Moreover, it allows us to remain in line with our vision of our game, which was not to promote competition, i.e. to distribute the bonuses equally, but rather to favour **friendliness and conviviality**. Thus, using an **unequal distribution** makes sure that all players can have fun whether they are last or first. We would also like to add that with the number of bonuses implemented so far, this distribution is not really felt but it will be when more power-ups have been added.

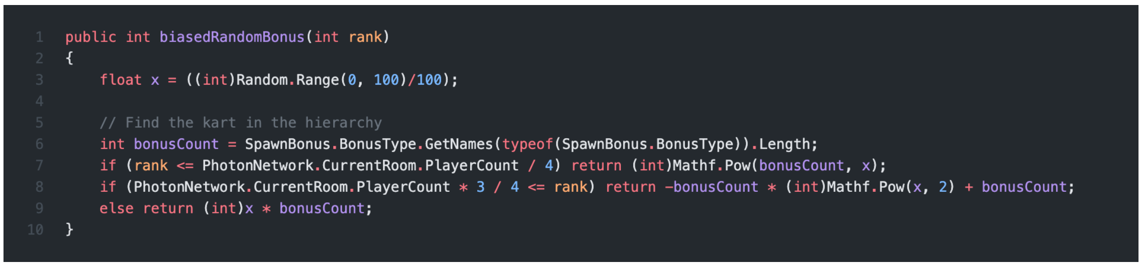

```
public int biasedRandomBonus(int rank)
{
    float x = ((int)Random.Range(0, 100)/100);

    // Find the kart in the hierarchy
    int bonusCount = SpawnBonus.BonusType.GetNames(typeof(SpawnBonus.BonusType)).Length;
    if (rank <= PhotonNetwork.CurrentRoom.PlayerCount / 4) return (int)Mathf.Pow(bonusCount, x);
    if (PhotonNetwork.CurrentRoom.PlayerCount * 3 / 4 <= rank) return -bonusCount * (int)Mathf.Pow(x, 2) + bonusCount;
    else return (int)x * bonusCount;
}
```

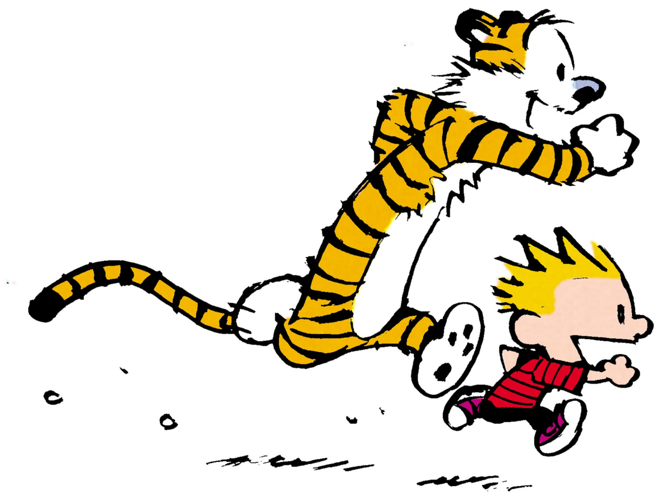
Figure 23: Pseudo-random bonus distribution algorithm

2.5.5 Sound design & noises

Our Book of Specifications indicated that we needed to have some sounds effects and noises for this Second Defense so we took the time to find some **short sound clips** that we could find a use for in the game. We added a lot of sound effects in the menus when clicking on a button or choosing a certain option. Thanks to the new User Interface system proposed by Unity a few months ago, we were able to directly add an Audio Listener component to each button we wanted to add a sound to. After adding our sound effects pack to a specific folder in the Unity editor, we were able to assign them by clicking and dragging each resource where the sound effect was desired. We also made sure that each player could **hear the engine of their own kart** as it started and moved around the track. After some research, we decided to link the pitch of a simple engine sound to the speed at which the player was going, resulting in a rather **convincing effect!**

3 Plan for Final Defense

For the next and **Final Defense**, we have a lot of work to do to finish this project in time and while respecting the program that we had set. First of all, the major part will be the development of the entire **Artificial Intelligence** (AI) part and by extension the 'Singleplayer' mode. This will be a real challenge since none of the three of us have had any experience in this area before. We also plan to add what could not be finished for this support, namely the ability for the player to **drift** as well as new race tracks. We are also expecting to add new **vehicle models**, and not just karts or race cars that can be chosen from the menu before entering the game. On the graphics side, we hope to have developed a more advanced particle system than we currently have and to learn more about the behavior of shaders to really **improve the visual quality of the game**. This also includes improving the User Interface (UI) by creating better, cleaner and more detailed assets ourselves. We also would like to implement new bonuses from the ones we already have designed now that we understood how to deal with Photon servers and data synchronization even more. Finally, it would finally be time to fix the visual lag due to movement over the network that currently prevents a pleasant rendering from the clients side!



4 Appendix

For details and links, see our website at epikart.games

Inspiration & Tutorials

- AnswaCode ”Distribution aléatoire pondérée continue, biaisée vers une extrémité” : Research algorithm for false-random in MarioKart and others racing games :
- Youtube video : Kart drifting and kart movement (OneMinuteGames)
- Youtube video : Particles tutorial (Unity official channel)
- Youtube video : Video Unity Cinemachine (Unity oficial channel)
- Youtube video : Impression of speed (DocGeraud)
- Youtube video : Gameplay for impression of speed (ProfasiaGaming)
- Youtube video : Examples for real kart drift (maxisound74)

Documentation :

- Photon Engine (including its forums)
- Unity 2021 – especially about the particle system and the Unity UI.

Assets :

Kenney car kit

Kenney city kit roads

Kenney pirate Kit

Kenney SFX for menus