Defense 1 : EpiKart

Undergraduate (1st Year) EPITA

09/03/2022

TEAM BINBINKS

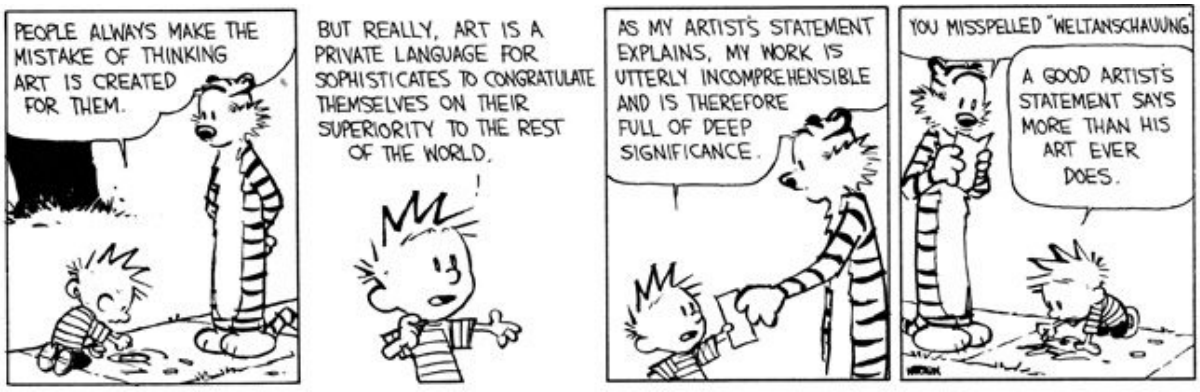
All the members are from A3

Eliott Flechtner (Project leader)

Lancelot Doucet Titouan Verhille

January - June 2022

Summary



1 Introduction

1.1 Book of Specifications Recap

This document constitutes the **First 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 Follow-up and recap

After the book of specifications were handed in at the beginning of the project on January 14th, we started to **come up with clear weekly schedules** *(See Figure Below)* for each member of the group, mixing revision sessions and classes, but also and especially **brainstorming and debriefing sessions**. In addition, a major element of the project had to be completed at the end of each week and each group members had to write a condensed report containing screenshots of their progress and modifications. This way we were able to collect a large amount of data showing the progress of the project development over the last few weeks. However, with **Valentin Berna going back to S1# classes**, it made us rethink the way we were approaching the project. Since Valentin was the most creative team member, he was supposed to the most used to 3D assets modeling. We thus **updated the original Book of Specifications** to represent more accurately our production and to reorganize our tasks to **take into account the loss of a key project member**.

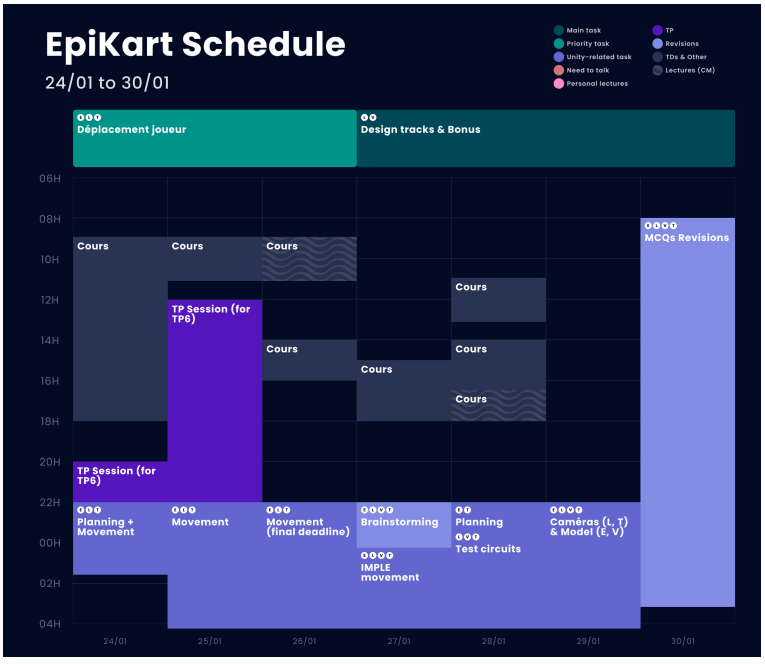


Figure 1: Schedule

1.3 Modifications

First, we updated the percentages of completion, namely, we redistributed Valentin's tasks **between ourselves according to our strengths**. We also fixed a few typos and grammar mistakes we could find and adjusted incomplete or imprecise sentences. We thus had to re-dispatch all the tasks concerning designing 3D models, assets and a major part of the tracks. We took the opportunity to also **modify certain strange choice for task assignation** and most significantly :

- **The track design and 3D modeling was assigned to Lancelot**, as he was the most capable of designing tracks and models after Valentin. Eliott was placed as substitute having a little background with Blender and animating sprites and backgrounds.
- **The player's movement was given to Eliott**, even though Lancelot made the first versions of it. This is also more justified since he built the entire server setup and network system.
- **The menu design and its complete implementation were given to Titouan** since Eliott was too occupied working on the kart's movement.

1.4 First Defense plan

For this First Defense, we had planned to have mostly entirely implemented servers and **online networking** as well as a completely functional menus with basic features to make it possible to the users to create and join rooms. We also wanted to have implemented either **two or three of the bonuses** we had designed together and have a really simple track so we can test out the karts' movements abilities.

2 Achievements & Technical aspect

2.1 Server

2.1.1 Researches and priority

After doing some research to find old S2 projects on the internet, it became clear that the implementation of online games, servers and more generally multiplayer were elements that **should not be neglected and, in contrary, developed early on**. From what we have seen, this would avoid **fragmenting the code** at several levels and would require restructuring a major part of the project. We therefore logically thought of

implementing a simple system to **synchronize the positions of two instances on the network**.

Since we knew almost nothing about server requests or how online networks work in Unity, we looked for free and good quality alternatives to implement this system. This is how we discovered the *Photon.Pun* library, which offers **a complete and guided network integration in Unity**. After seeing that this engine was one of the most recommended and easy to use of all those we had found so far, **we decided as a group to use it for our project**. To help us with the installation of the module, we went through the Photon documentation and consulted a number of different tutorials and resources online and on YouTube. We then coupled these two main sources to implement the system in our project.

2.1.2 Setup in Unity

First of all, one of us had to create an account on the Photon website in order to create a "new app" *(See Figure 2 Below)* - basically a new server - with just a few clicks. Eliott, our group leader, got in charge of doing just that since he was the one mainly building the online servers and multiplayer mode. We were also offered to subscribe to Photon's services in order to have access to more server capacity, but we considered **this an unnecessary expense** since we knew we wouldn't be using it to its fullest. Then we just had to download the free Photon Engine library directly from the Unity Asset Store **and add the ID of our newly created server**. In order to make a simple prototype to test if everything was working correctly and if we had understood the documentation. We created **three different scenes**: one to connect to the server *(See Figure 3.1 Below)*, another to allow the user to join a specific game room *(See Figure 3.2 Below)* and finally a game scene that will contain only two cubes **that can be moved manually via the scene editor**. Our goal was then to simply verify that these cubes were moving together on the network and that they could respectively **see each other's movements in real time**.

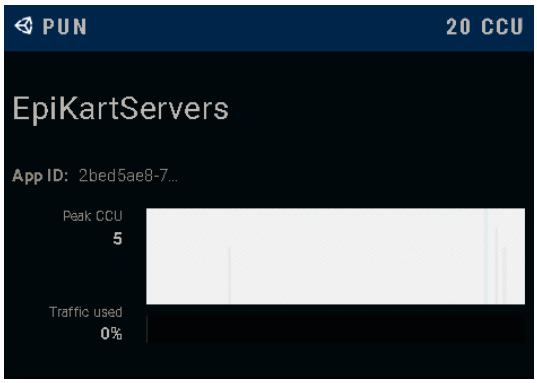
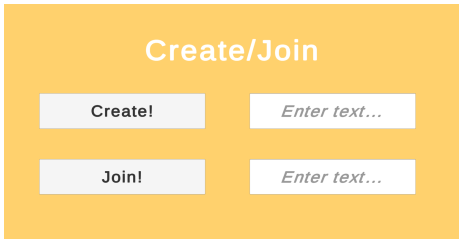


Figure 2: Photon server for EpiKart

1. Loading screen



2. Lobby screen

Figure 3: The beginning of EpiKart

2.1.3 Syncing movement across network

We made sure that a user launching the game window could be detected and added on the network: thanks to the *ConnectToServer* script and its *Start* and *OnConnectedToMaster* functions *(See Figure 4. Below)*, the game window changed to the next scene at the very moment the player was recognized **as a distinct element of network**. Once in the second scene, we quickly implemented two buttons and two text fields respectively to create and join game rooms. When a game room is created by a user, **he automatically joins it** and is able to see his own cube. This was made possible by using the *PhotonNetwork.Instantiate(...) (See Figure 5. Below)* command from the *Photon.Pun* imported library which **allows to generate any game objects on the network**. Spawning objects on the network also required to attach **to every generated game objects a *Photon View* component**.

However, the cubes' positions were not getting updated because **they were not synchronized across the network**. Thus, we had to add the *Photon View Transform Classic* component to each of the elements instantiated on the network, allowing us to specify which properties of the cubes had to be updated on the network.

Now that all the Photon components had been added to the cubes, **their positions, rotations and scales were all synchronized on the network** and all the players connected to the rooms could move and interact with each other without any problems.

```
// Start is called before the first frame update
void Start()
{
    // Connect the client to the server
    PhotonNetwork.ConnectUsingSettings();
}

public override void OnConnectedToMaster()
{
    // Join the lobby as soon as the client is connected
    PhotonNetwork.JoinLobby();
}
```

Figure 4: ConnectToServer C# Script

```
void Start()
{
    // Instantiate a player at a random position on the terrain
    Vector3 randomSpawnPos = new Vector3(Random.Range(minX, maxX), 0, Random.Range(minZ, maxZ));
    PhotonNetwork.Instantiate(playerPrefab.name, randomSpawnPos, Quaternion.identity);
}
```

Figure 5: Instantiation of cubes

2.2 Kart movement

2.2.1 Collider explanation & Model replacement

After finishing the Photon server prototype as detailed above, we replaced the old cube models with the racing karts ones to make our tests with something more convincing. In order to implement the car movement system and to make it **as fun and realistic as possible** regarding the wheel mechanics, we had the idea to replace the rectangular collision box **with a sphere collider inside the car**. We created an empty game object containing the player model and its sphere collider to which a Rigidbody had been added to apply world physics such as gravity or terrain friction. We also added a camera behind the player and attached it to it in the scene hierarchy so **it would follow him wherever he goes**. After multiple adjustments, we found values we liked for the physical properties of the sphere collider and **made a prefab of the player object**. Now that we could place a player anywhere in the scene, it was a matter of actually coding the script for driving the kart.

2.2.2 Movement

Now that we could place a player at any point in the scene, we had to code the script that manages the movement of the kart: this is the role of the *KartController* script attached to the kart model and not to its prefab parent, **since it does not have the *PhotonView*** *(See Figure 6. Below)* component necessary to move the kart on the network. The first thing to implement was the movement of the sphere collider : using the new input system proposed by Unity since January 2021, we managed **to detect when the user pressed the directional keys of the keyboard or oriented the joystick of a controller connected to the machine**. With this information, we were able to modify the model and make it react accordingly by determining the new position and rotation of the kart as well as the acceleration forces acting on it *(See Figure 7. Below)*. However, only the sphere collider had to roll on the ground: the kart **had to follow only the position of its collider**.

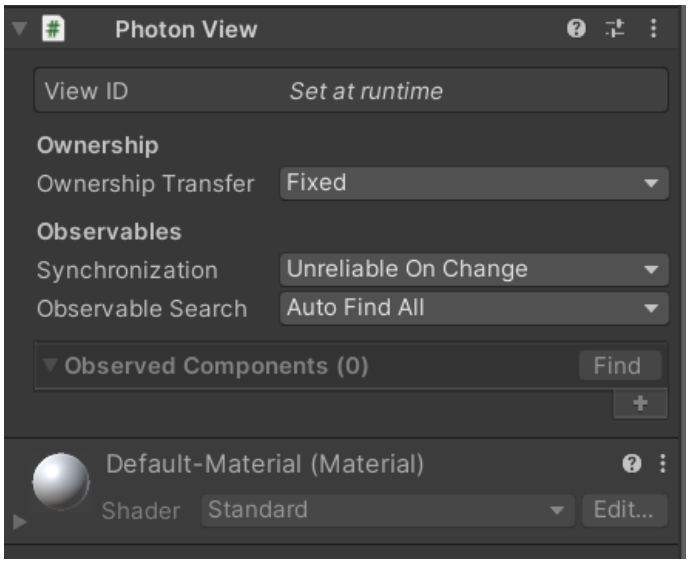


Figure 6: Photon view component

```
// Update current speed & rotation
currentSpeed = Mathf.SmoothStep(currentSpeed, speed, Time.deltaTime * 20f); speed = 0f;
currentRotate = Mathf.Lerp(currentRotate, rotate, Time.deltaTime * 4f); rotate = 0f;

// Move forward when accelerating
sphere.AddForce(-playerModel.transform.forward * currentSpeed, ForceMode.Acceleration);
```

Figure 7: Smooth calcultation of player movement

After creating many public variables to have access to the data values much more easily and directly in the editor, we took care of making the movements of the sphere collider more fluid and created a new function called *Steer* forbidding the player to be able to turn in case he is not completely stopped. Also, since

we wanted to have the possibility for the player to fly over obstacles after driving on springboards, we had to implement a way to detect when the player was touching the ground or not. We managed to get this information by projecting a *RayCast* under the kart's chassis that would return true if it detected any surface under the player's wheels or not. This boolean value was used to only allow the user to control the vehicle once on the ground to make the game more realistic. Finally, for the sake of perfectionism, we implemented the controls to allow him to back up while reversing the controls like in most racing games we had played like *TrackMania* or *Rocket League* for example.

Now we wanted to make sure that the player would be able to adapt to the topography of the different terrains they would encounter such as rounded slopes for examples. We tried multiple ways to implement that but they all **ended up being way too complex**. We finally settled that we would be reusing the previously covered projected *RayCast*, we oriented the car so that **its normal was always parallel to the ground's**. In other words, it was simply a matter of always **making the projected ray under the car collinear with any straight lines perpendicular to the ground** *(See Figure 8. Below)*.

```
// Init current grounded status & cast a beam
RaycastHit hit;

// Update the grounded status of the player & rotate the player according to terrain
if (Physics.Raycast(groundRayPoint.position, -transform.up, out
 hit, groundRayLength, whatIsGround))
{

    playerModel.up = Vector3.Lerp(playerModel.up, hit.normal, Time.deltaTime * 8.0f);
    playerModel.Rotate(0, transform.eulerAngles.y, 0);
}
```

Figure 8: Projected raycast for topography

As already said in the Books of Specifications, we wanted to make it especially playable on a controller because **it was more intuitive for the user** and that is what we did! After struggling to find the exact key code and reinstalling tons of drivers, we managed to drive our karts using gamepad controllers.

All the previous additions concerning player movement worked very well locally, but when we tried to make the karts appear on the network, **the camera view of the last kart to appear in the scene was swapped with that of the game room host**. We just had to consult the Photon server documentation to solve this problem. We had to make sure that the *PhotonView* component currently processed by the *KartController* script was checked to match that of the new kart in the scene. For that we checked the IsMine command *(See Figure 9. Below)* from the inherited from the *PhotonView* component. If this was not the case, **then the current kart view should not be updated**.

```
// Store the ID of the player's view
PhotonView view;

// Access the Photon network for the player
view = GetComponent<PhotonView>();

if (view.IsMine)
{
    // Execute player movement code
}
```

Figure 9: Camera bugfix

2.3 User interfaces and Menus

Before we even started implementing anything in Unity, **we first had to agree on the style we were going to us**e for the menus and the user interface in the game. So, as stated in the book of specifications, we used the **collaborative design software Figma** to design a variety of graphical elements. We separated our workspaces into different files **grouped by themes and elements to make it clearer** *(See Figure 10. Below)*. We then created a series of different screens representing each possible state of the menu as slides *(See Figure 11. Below)*. In order to better understand the different possible paths according to the user's actions, we represented the **links between the screens using a tree** *(See Figure 12. Below)*. Using Figma's interpolated transition feature, we were able to link the different slides together: this was particularly useful as it allowed us to **conceptualize the different animations we would like to reproduce in the game**.

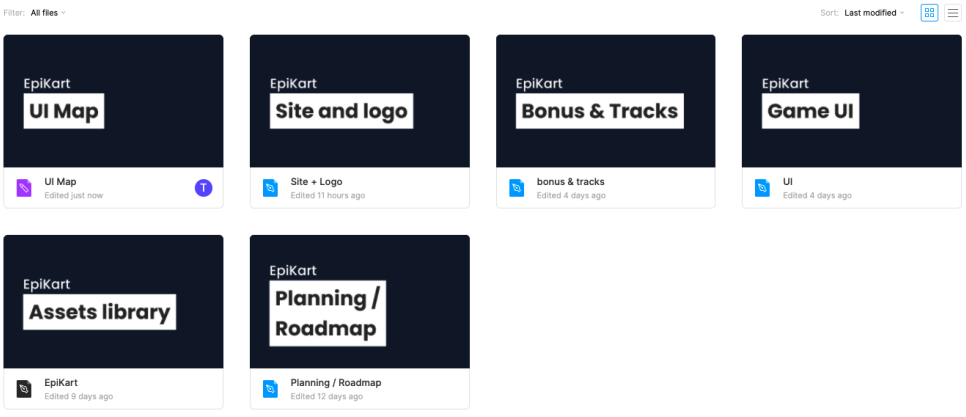


Figure 10: Our collaborative design workspace

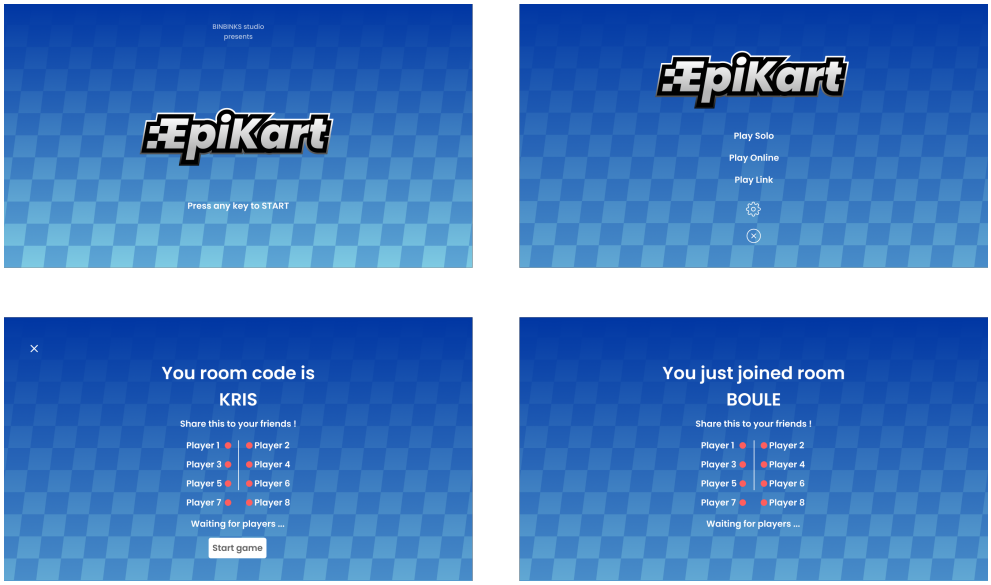


Figure 11: Some Figma experimentations

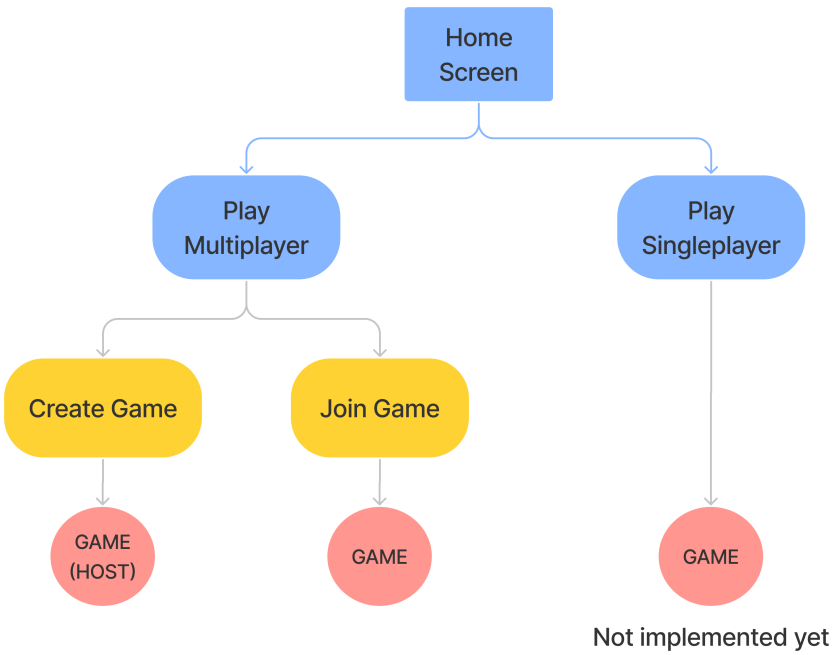


Figure 12: Figma User Interface tree

2.3.1 Animation Launch Script

For the whole menu, we used an empty game object with an *AnimationLaunch* script attached to it which **launches animations made in the Animator panel**. It allowed us to have more control to shape the transitions' speeds for the **interpolation between two key-frames** or saving properties of an element in the scene like position, opacity and scale.

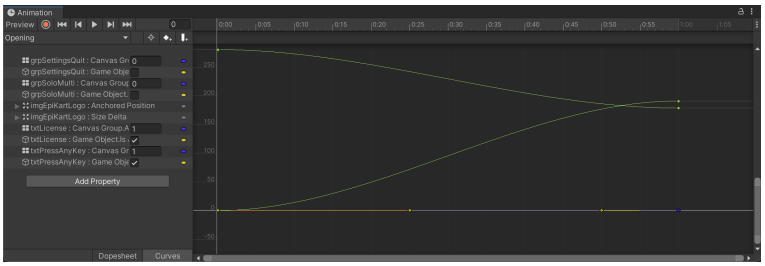


Figure 13: Animation curves

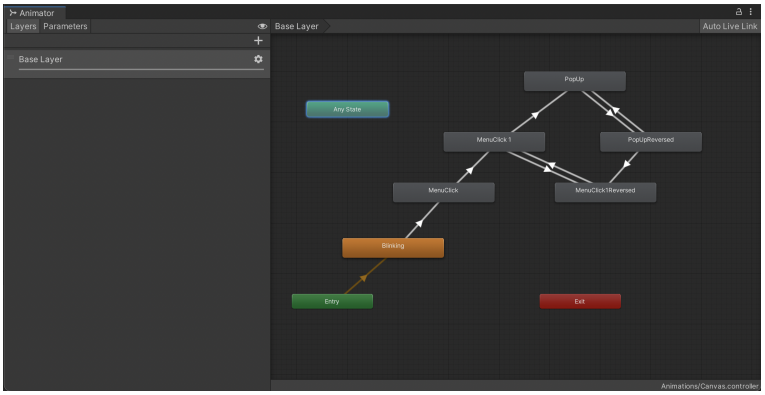


Figure 14: Animator panel in Unity

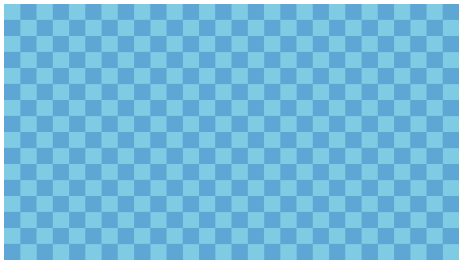
2.3.2 Starting screen

We started by building and designing the first screen the user sees when he launches the game. We took inspiration **from old games' starting screens and created our own** : the user would press any of his keys or click on the screen to transition to the main menu. To make it feel less empty and blank, we took the time to come up with a proper background reminiscent of the appearance of a racing flag. We were happy with the result but found it **needed to be more dynamic** as it was currently just a static background. It was animated within Unity by attaching a simple script called *BackgroundScroller (See Figure 15. Below)* to the background image that would update its position every frame within the scene according **to a passed horizontal and vertical speed**, making it tile and scroll endlessly on the screen. For the sake of clarity and

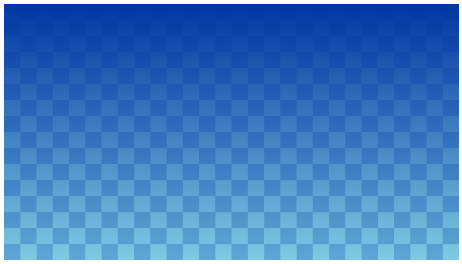
text readability, we added a darker gradient layer on top of it, top-to-bottom *(See Figure 16. Below)*. We then added the **game logo** in the middle of the screen as well as **various texts components** like team license and a periodically-blinking text prompt inviting the player to press any key to proceed. We opted for the *TextMeshPro UI-library* provided by Unity to have more control over fonts and multiple properties of the text elements.

```
public float xSpeed = 0.025f, ySpeed = 0.025f;
void Update()
{
    image.uvRect = new Rect(image.uvRect.position + new Vector2(xSpeed, ySpeed) * Time.deltaTime, image.uvRect.size);
}
```

Figure 15: BackgroundScroller C# Script



Background



Background with overlay

Figure 16: Background of EpiKart

After pressing any key, the menu scene was supposed to **smoothly transition** to the main menu, reducing the game logo while fading out every other text elements and replacing them by two button sets allowing to either enter single or multiplayer mode or to access settings and quit the game. To make this happen, we had to setup **an entire animation state-machine system** relying mainly on the *Animator* component placed in the scene and containing the entire *UI 2D-Canvas* **to access its elements and modify their positions and opacity values in time**. We added in icons for the settings and quit buttons and made it so that they would fill when hovered or selected.

2.3.3 Main menu

Initially, we wanted to make a main screen with all the main options, but after some thought we decided to change and separate the options on several screens to be clearer for the player. Other reasons could be that there would have been **a lot of elements copies from scene-to-scene** and that it would **simplify the animation process**. Therefore, we had to manage the objects in the editor so that they would progressively

activate and deactivate each others. We thus took the opportunity to clean up all the UI game objects since they were all mixed up and **arranged randomly in the editor**. We made a prefab out of each one of them - which can be understood as an easily reusable template for each item - to make it easier to edit and **modify down the line**.

Using the built-in module Unity UI, we linked all the buttons together by creating a path so that the user can navigate through them by **using either keyboard or game-pad joystick controller** and not just the mouse. As we worked through UI elements, we discovered how the buttons' state systems worked and used it to make white rectangular-shaped buttons **to better distinguish them from the rest of the screen which is mostly blue**. We also thought that the button could get darker as clicked to visually confirm to the user that his action has been taken into account. In addition, we added a *Size Fitter* component that allowed **for responsive scaling and adaptation of the button shapes**. We also added a *Canvas Group* component to each part of the UI that needed their opacity to be animated in order to access each values of transparency independently in the *Animator*.

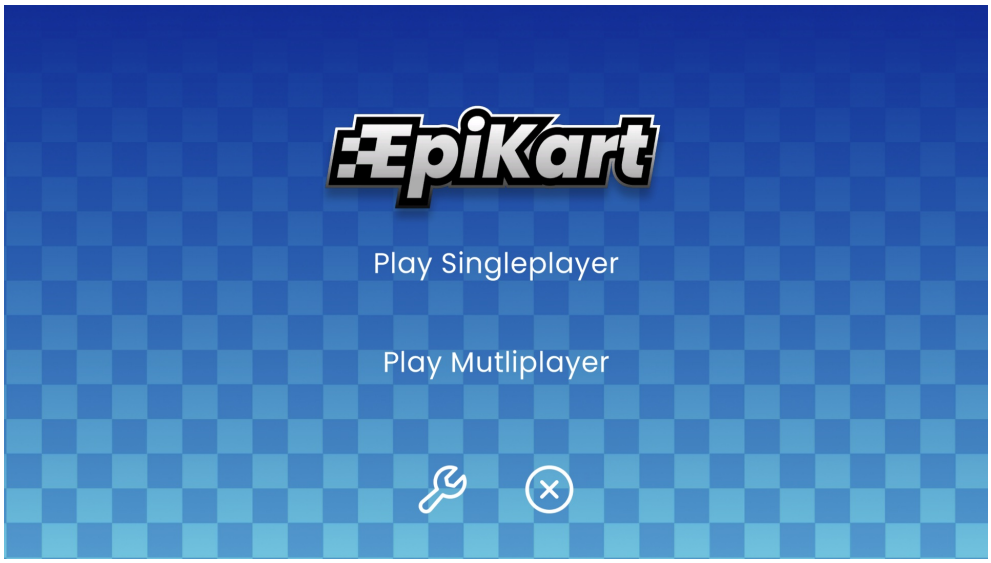


Figure 17: Main Menu

2.3.4 First main menu - Singleplayer & Multiplayer screen

As for the Single Player button, we will focus here on the Multiplayer option. When the user clicks on the latter, the Photon server **connects the player to the network and adds him/her to a list of player IDs currently online and managed by the server instantly and autonomously**. This connection takes only a handful of seconds on the client side and is established with a script using the Photon.Pun library built into the Photon imported package by calling the public functions *Connect* and *OnConnectedToMaster* that we

built. The Photon server **uses customized parameters to launch the server** and, if managed to connect the user, triggers an animated transition between the first main menu screen to the second one. Using the same Animator component, the transition would slide the current sets of buttons to the right-side of the screen while making the new ones slide in from the left-side. In the event of **a lack of stable internet connection**, the bridge between the user and the Photon server **immediately stop** and thus disconnects him/her.

2.3.5 Second main menu - Create & Join screen

In order to be able to play in multiplayer, a host must first create a waiting lobby via the *Create Game* button available at this stage. As each lobby has a unique identifier, we have made sure to generate this identifier randomly in the form of a **5-character alphanumerical code** allowing the creation of the waiting room. We decided to generate a short code containing **only capital letters to facilitate its diffusion when sharing it to other players**. We think that most of the games generating a link or code often were painful to transmit to players wishing to join the game quickly. Furthermore, we added an interactive button that allows, when clicked, **to copy the current room code**. This was done by calling the *CopyManager* script which gets the content of the room code *TMP_Text* field *codeTextField*. It also launches a slight animation informing the player the code was indeed copied. Thus, when the button is clicked, the host is the first to be connected in the waiting room. The name of the room being displayed on the screen, **he can easily transmit it to the other players so that they can join the game**.

On the other hand, when a player wants to join a game, he can this time click on the *Join Game* button which will spawn a pop-up window using the *Animator* component. To create it, we needed to learn more about the **process of 9-slicing an image**, in our case slicing a white rectangular-shaped square with rounded edges using two vertical lines and two horizontal lines to create nine pieces of it in total. In Unity, we set the game object attached to this image to sliced mode, **making it possible to stretch only the middle part**. This was really useful when animating the window because everything had homogeneous and proportional sizes.

After the window has spawned, As the code is only composed of uppercase letters and numbers, we have made sure to **transform all the player's keyboard inputs from lowercase to uppercase using a *Regular Expression*** - or *RegEx* - called each time a new character input is detected. This transformation is done in the *InputFieldRestrictor* script *(See Figure 18. Below)* attached to the text field . Moreover, if the code entered does not correspond to any room already created or that has already been launched, **a text warns the player about it and does not make him join it**.

```
public void onChangeCall()
    {
        warningInputField.text = "";
        joinInputField.text = Regex.Replace(joinInputField.text, @"[a-z]", m => m.ToString().ToUpper());
    }
```

Figure 18: InputFieldRestrictor C# Script

2.3.6 Kart colors

RetrieveRoomName is a script used to retrieve the player's name and colors when you join the game and **keep it in sync if a new player joins along** *(See Figure 19. Below)*. To control the color system of the karts in our game, we made a dictionary of pre-picked colors. Each player is assigned a unique color. When a player joins the room, a controller gets the number of players and their nicknames via the attributes *PhotonNetwork.CurrentRoom.PlayerCount* and *PhotonNetwork.PlayerList[i].Nickname (See Figure 20. Below)*. It then assigns each nickname to the corresponding text field and **assigns the correct color to the circle right next to the nickname**. This color mechanism is used to define the color of the kart, sent to the *SpawnPlayers* script **in charge of instantiating the players and assigns a color in the game scene**.

We had some troubles retrieving pieces of information about the players on the network so managed it differently for the karts. We made it so that each color is synchronized by the script *KartColorSync* and using the Photon callbacks from the function *OnPhotonSerializeView (See Figure 21. Below)*, we define how Photon takes care of transmitting a color change via the server. We also needed to create two functions : one to **transform a Color type data into a string** so that Photon could transmit this data to all the other players and another one doing the opposite when **retrieving it on the client side**.

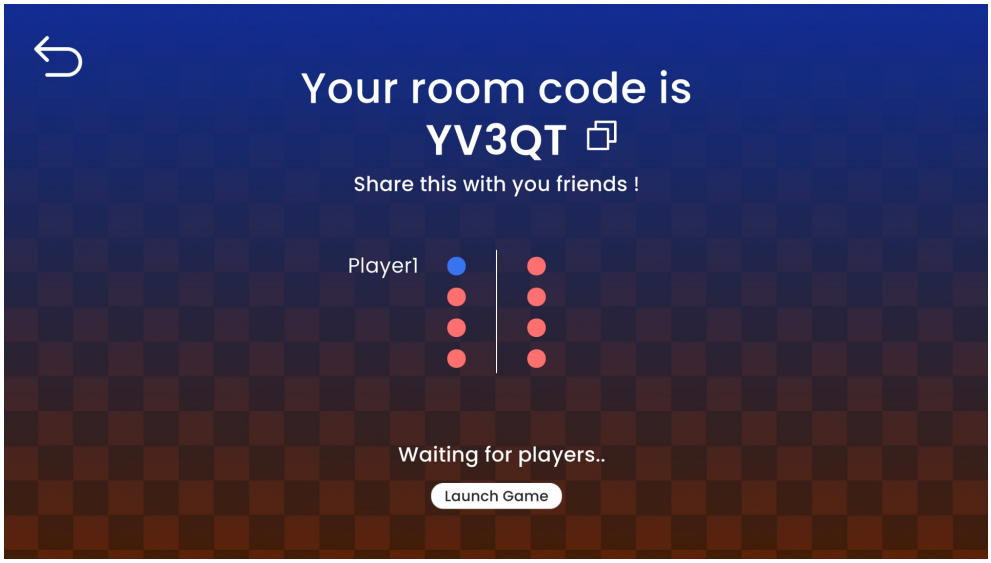


Figure 19: Lobby Menu

```
void Start()
    {
        // Sync all clients to the master
        PhotonNetwork.AutomaticallySyncScene = true;

        // Set the code room text to the server name
        codeTextField.text = PhotonNetwork.CurrentRoom.Name;

        // Update every player slots when joining
        for (int i = 0; i < PhotonNetwork.CurrentRoom.PlayerCount; i++)
        {
            // Update all names accross network
            playerTexts[i].text = PhotonNetwork.PlayerList[i].NickName;
            // Update all colors accross network
            Color currColor = colorList.Values.ToList()[i];
            dots[i].color = currColor;
            playersColors[i] = currColor;
        }
        SceneManager.sceneLoaded += OnSceneLoaded;

        if (PhotonNetwork.IsMasterClient)
        {
            launchButton.SetActive(true);
        }

    }
```

Figure 20: Function of RetrieveRoomName C# Script

```
public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
    {
        if (stream.IsWriting)
        {
            stream.SendNext(GetColorNameFromColor(kartMesh.materials[1].color));
        }
        else if (stream.IsReading)
        {
            string colorIndex = (string)stream.ReceiveNext();

            if (colorIndex == null) Debug.LogWarning("Something wrong happened...");

            kartMesh.materials[1].color = colorList[colorIndex];
        }
    }
```

Figure 21: Function of KartColorSync C# Script

2.4 Bonus & Power-ups

2.4.1 Planned

The aim of the power-ups was to have collectable boxes which grants abilities to the player who collected it. From a list of possible power-ups we brained-stormed as a team, we chose to implement **two for the First Defense**: the **"nitro boost"** which grants a temporary speed boost and the "oil puddle" which drops a spill of oil behind its user on which other karts would slip and be momentarily stopped.

2.4.2 Prefab & Collisions

The first step was to implement power-up cubes which hover over ground, get destroyed when collected by the player, **assign a random power-up to the player and respawn**. To do this, we first created the power-up prefab, by using a basic cube and a wooden crate texture we found online. Then, we created an empty object called *SpawnBonus* **which gets created with the room**. Its purpose is to spawn the power-ups using the script of the same name attached to it. To implement the destruction of the cubes when touched by a kart, we added collision detection on the karts, **which destroy the cube if the collision is triggered.**

2.4.3 Pick-up & Auto-respawn

Instead of using a list or an array, we came up with the idea of using a data structure called an enum for power-ups, which **will contain the name of all possible collectable bonuses in the game**.

The next step was to give the player a random power-up from the enum. Since we had created an inventory variable assigned to each player, we simply assigned the current power-up to it. We generate **a random number between 0 and the length of the enumeration**, and assign the player the power-up at the index chosen to the inventory variable.

Finally, we needed to respawn the power-ups a few seconds after it was collected. We had some troubles managing to achieve that as the scripts **controlling the power-up was attached to itself** and could not be respawned once destroyed. A solution was to attach the script to the kart and call the function *RespawnBonus* in the *SpawnBonus* script to respawn the power-up attached to the empty object *SpawnBonus* which spawned the power-ups *(See Figure 22. Below)*.

```
// Randomly pick a bonus and broadcast it
public void RandomBonus()
{
    // Find the kart in the hierarchy
    int size = SpawnBonus.BonusType.GetNames(typeof(SpawnBonus.BonusType)).Length;
    string bonusName = ((SpawnBonus.BonusType)Random.Range(0, size)).ToString();
     // Add the bonus to the kart's inventory if empty, and change the label on screen
    if (string.IsNullOrEmpty(kart.GetComponent<KartController>().bonusInventory))
    {
        kart.GetComponent<KartController>().bonusInventory = bonusName;
        bonusLabelGroup.SetActive(true);
        bonbusLabel.text = bonusName;
    }
}
```

Figure 22: Random bonus from SpawnBonus C# Script

2.4.4 Effect of power-ups

We created a function which gets triggered when the player **activates the power-up** (presses X on keyboard or the square on the game-pad) which **matches the player's inventory with the corresponding function**.

- Nitro boost : We first added the nitro boost that would double the speed of the player for 2 seconds, giving him an edge over other players. To implement it, we multiplied the kart's speed by a coefficient

initially set to 1. When the power-up is activated, the speed coefficient **is doubled**. After 2 seconds, the coefficient **is reset back to 1** and the player's inventory is cleared.

- Oil puddle : We then added the oil puddle power-up which spawns behind the player who activates it and on which other player could momentarily be stopped if they touch it. The puddle **must be tangent to the terrain's topography** as well as being destroyed when a player rolls over it. We thus needed to add a new entry to the enum previously created. We then created a new game object from a decent 3D model of a puddle we found after scouring the internet, which we reworked to fit our criteria and **added an oil-like shiny texture we created**. This game object was saved and stored in the prefab folder since it needed to be instantiated on the network. When the power-up is activated, an instance of the puddle is spawned behind the player. We **reused the ground adaptation calculation implemented in the *KartController* script** before to orient it along the road tiles. This still needs to be addressed in the future as the oil puddle can still clip through the map. When a collision between a kart and a puddle is triggered, the instance of the oil puddle is destroyed and the kart's speed coefficient set to 0. After a few seconds, **it is reset to normal and the player continues the race**.

2.4.5 Multiple bonus spawning

While testing with multiple players, we realized there was a **devastating bug**: a new set of power-up would spawn for each player joining the room, meaning that if there were 8 power-ups on the track and 3 players, **24 power-ups would spawn**! This bug was painstakingly fixed by spawning a set of power-up only when a player being the host joined the room, in other words, **only once**.

2.5 Track design

We wanted to make a **simple track design for the First Defense**. The game was simple and straight forward, so we downloaded **bundles of low-poly assets** from the internet. We thought about the general shape of the track for some time and finally settled for a simple 8 loop, which features a few height variations, a start position and a jump. Now that we had a general shape, we dragged and dropped road assets onto the editor and manually placed them one by one next to each others. Our first try was disappointing: we realized it was going to take a **non-negligible amount of time** and that we would need to decorate it a lot for it to seem playable enough. Also, the road was too narrow and the track way too big. We scaled it down and changed its shape a bit to better fit to the player scale. It was time consuming because the road assets **had to be aligned perfectly**. Once the road was done, we realized the track looked very bland, so we added a lot

of individual track decorations such as trees, seats, lights, cones, buildings and more. We really liked this part since it was so easy to drag and drop assets everywhere to enhance the current scene. We ended up doing way more than expected and thus were **ahead of schedule**.

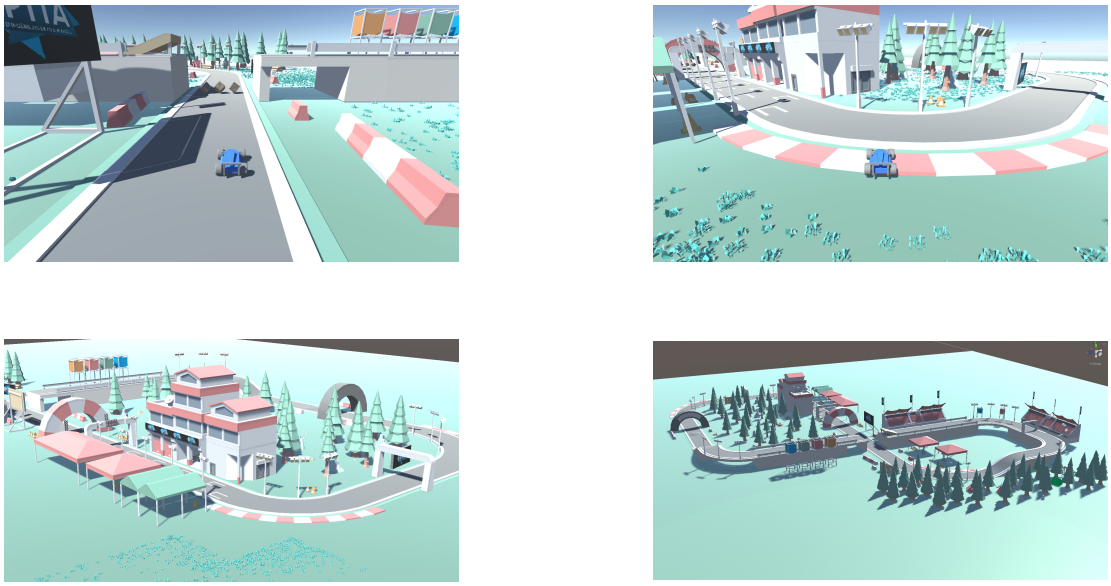


Figure 23: Track screens

3 Plan for next Defense

For the next Defense, we first plan **to clean up the entire code** of the latest implementations such as bonuses. In addition, we will also **completely rethink the way we generate the colours of the karts** in the game room to make it purely random. We will also try to fix the obvious lag in player movement on the server and **add new animations and many in-game particle effects**. The kart will be able to drift to gain speed momentarily and its wheels will turn in the direction the user chooses. **The in-game user interface will also be improved** by adding a better design for the inventory and information about a user's position in the race. Finally, we will add two new themed tracks that are much more polished than the current one and **will allow players to choose their nicknames** when they join the game!

4 Appendix

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

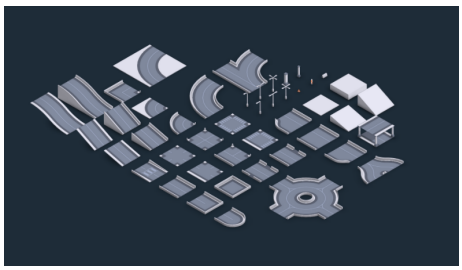
Inspiration & Tutorials

- Photon Youtube tutorial from BlackThornProd
- Kenney Twitter : Kart movement inspiration
- Youtube video : Kart racing game with AI inspiration
- Youtube video : Arcade style driving inspiration

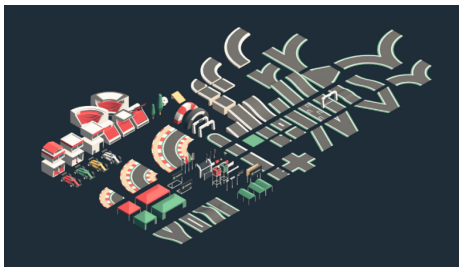
Documentation :

- Photon Engine
- Unity 2021

Assets :

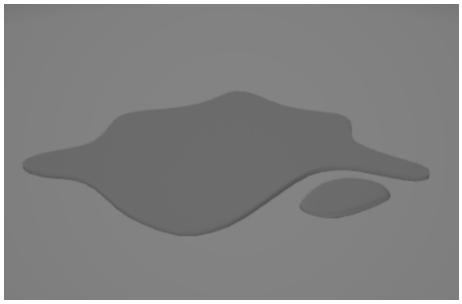


Kenney racing kit



Kenney city kit roads

Kenney car kit



Oil puddle 3D model