ERICK PASSOS AND WAN SOUZA

PUN RALLY

MULTIPLAYER RACER TUTORIAL BY SERTÃO GAMES

PUBLISHED BY SERTÃO GAMES

SERTAOGAMES.COM

Tutorial Version 1.0.6, February 2016

Contents

1 Introduction

Figure 1.1: PUN Rally is a multiplayer racing game the is distributed as a complete project on Unity's Asset Store. The game source code, which is fully commented (and complemented by this tutorial), teaches you how to develop a racing game with realistic physics, smooth network synchronization with dead reckoning, lobby, rooms, and RPCs for GUI and gameplay control.

This document is the accompanying tutorial for PUN Rally, a multiplayer racing game project for the Unity 3D engine. The project can be obtained from Unity's Asset Store, and comes complete with source code and assets.

PUN Rally uses Photon Unity Network (PUN) to handle the multiplayer bits of the game; and Unity's built-in PhysX for simulating realistic car physics. The project is a good starting point for dealing with the challenges of developing multiplayer racing games.

In this chapter, we introduce Photon Network, talk about some features of PUN Rally, and give you more details about the tutorial contents.

What Photon does

Figure 1.2: PUN is the easiest way to develop your multiplayer game in Unity. Each game client connects to the cloud servers that are used to relay communication even in firewalled environments.

Photon is a complete network solution for multi-player games, featuring seamless integration with several platforms (including of course Unity), real-time cloud-based communication, match-making and many more.

PUN (Photon Unity Network) is a Unity plug-in that makes building multi-player games with this powerful engine a breeze. It gives you full access to Photon cloud-based relay servers, so your multiplayer game clients are guaranteed to communicate even under firewalls or NAT Internet access.

There are of-the-shelf options for 3d object synchronization (for real-time games) and remote procedure calls (good for turn based games or other types of messages between peers, such as chats, inventory items, etc).

The PUN Rally Game

PUN RALLY IS A MULTIPLAYER RACING GAME for up to 4 (four) players. You choose a nickname, decide to host or join a race, pick your car color, and race to the finish against other players. It is a straightforward racing game that demonstrates what you can do with Photon for Unity. Controls are simple: use the arrow (or WASD) keys to control throttle, braking and steering, while space is the handbrake and left shift flips the car back up if you happen to roll it over (remember we're using realistic physics).

Figure 1.3: Car color selection interface. We use the PUN player-properties feature to store and synchronize each player's selected car color to all client computers. Choosing the color actually defines a car prefab for the player, so you can easily use this feature to have different cars for the player to choose from in your game.

The game was released as a complete project for the Unity Asset store, so you can learn by studying the source code together with this tutorial. The code was designed and written to be simple to understand yet fully functional. We tried to use standard assets as far as we could, and the game features all the basic concepts for a good racing multiplayer:

- Easy-to-setup realistic car physics based on standard wheel colliders, rigidbody and only a few custom scripts;
- Real-time car synchronization with dead-reckoning (prediction of movement even under network instability);
- Peer-to-peer communication with distributed load (each peer only computes complete physics for the local player's car);
- Race lobby, with nickname, create/join race and car color selection (see Figure 1.3);
- Complete GUI including an end-of-race summary with positions and individual race times;
- Simple project structure with only two scenes and easy-to-extend features;
- (new - bate) A simple weapon system using the same over-the-internet features of the physics, good as a template for combat-based games;

What to Expect from this Tutorial

THE GOAL OF THIS TUTORIAL is to teach you how to use PUN to create a basic multiplayer racer. It is expected that you have familiarity with the Unity 3D engine and the C# programming language.

You will also need to create a free account for the Photon Cloud service (we included the instructions for this in the next chapter) before

testing your game project, so it can connect to the relay servers that handle network communication.

The project has only two Unity scenes, *Menu* and *Race*, designed to teach you the basics for using PUN in different situations. The following subjects are explored in the project code, tutorial and accompanying video, so you can expect to learn about them after reading the tutorial (and testing/modifying the project):

- Setting up realistic car physics with Unity's standard wheel colliders, rigidbody and a few custom scripts. You are especially encouraged to play with the car parameters found in the *CarController* script and wheel colliders (see Figure 1.4), so you can learn by experimenting;
- Using PUN to synchronize the cars of several players over the Internet. In PUNRally, each client computer is responsible for computing the physics of the local player's car and synchronizing it with the remote clients (other player's computers). You will learn how to synchronize position and rotation while keeping a fair load balance between peers;
- Using dead-recknoning techniques to handle network instability, so remote cars race smoothly even in the presence of latency. You will also learn how to make the wheels, fx, accell and braking look realistic for remote cars by synchronizing input;
- Other basic PUN concepts such as Remote Procedure Calls (RPCs), lobbies, rooms, player properties and error handling are used and explained in various parts of the project, such synchronizing car color choices, spawn points, raing positions, player disconnections, etc.

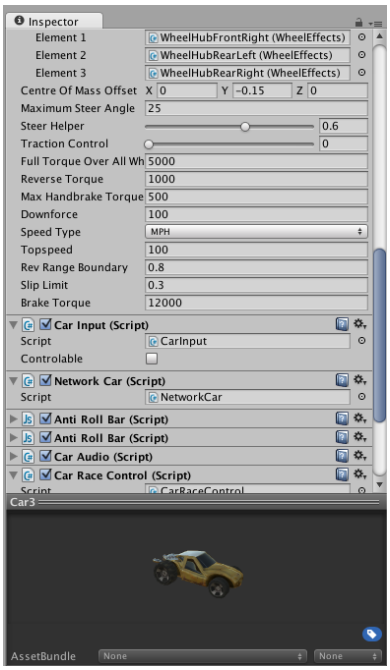


Figure 1.4: We used Unity's stock standard assets *CarController* and played with the parameters to achieve a buggy-like behavior for the car. Try to modify them or even dare to completely overhaul our parameters and 3D models, making your game feel unique.

The Next Chapters

Next chapter introduces PUN basic concepts and teaches you how to handle connection to Photon Cloud, lobbies, and dealing with players joining and creating rooms; The following chapter brings the discussion to the physics simulation of the cars and how to synchronize its state, input and fx between game clients in a fluid manner using dead-reckoning; After that, we explain other important features such as race control, position calculation, and the weapon system on a multiplayer racing game. Finally, in the concluding chapter, we present some ideas for future improvements of your own racing game (and a few directions on how to develop them). We remember that some important features of a more complete racing game are not included such as: AI drivers, (controversial) rubber-band physics, balancing cars with different characteristics, etc.

2 Basic Photon Network Concepts

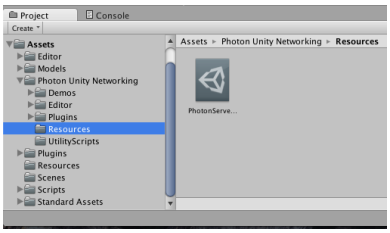


Figure 2.1: The *PhotonServerSettings* asset is located inside the *Assets/Photon Unity Networking/Resources* folder. Just click on it to access its properties.

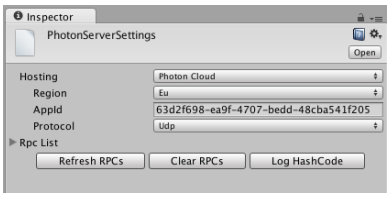


Figure 2.2: Here you'll only need to fill your AppID (see main text), but you can check other the other properties: *Hosting* lets you choose the type of service. We'll use Photon Cloud here, since this is the easiest one, but depending on your plan, you can even host your own authoritative servers. You can choose your *region*, which will give you the best latencies depending where your players are, and the *protocol*. We chose UDP for its good performance. The RPC list shows all RPC methods you have in your project.

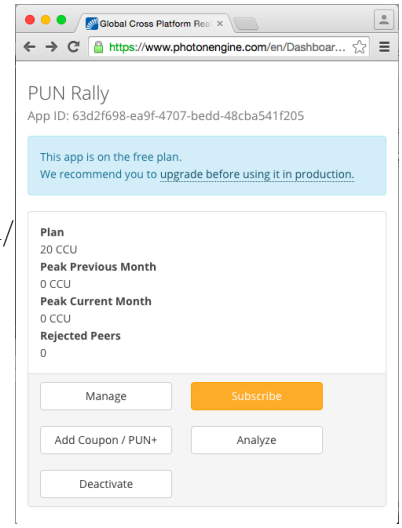


Figure 2.3: The realtime dashboard, showing your newly created AppID.

Developing a multiplayer game involves a lot of things besides synchronizing 3D objects data. In this chapter we give you a first glimpse on the basic stuff that you need to setup PUN for your Unity project, and then teach you how to deal with connecting to Photon's lobby; room creation/joining; using player properties to handle car color selection, updating GUI on player connection, etc.

Setting up PUN for your project

If you've downloaded the project from the Asset Store and opened it, you still need to configure PUN to use your personal credentials to connect to the Photon servers. The process is very straightforward: there is an asset in PUN's bundle that holds the server connection settings. Figure 2.1 shows where you can find the asset, named *PhotonServerSettings*.

The *PhotonServerSettings* asset has all the properties you need configured to use the Photon Network services. Figure 2.2 explains the properties. The most important one for us is the AppID, which is a unique identifier for your game. We'll tell you how to create one now.

Now you need to configure Photon Cloud to act as your multiplayer game's communication platform. It's simple and you can setup a free plan, so your players can race against each other. Follow these steps to have it done:

- If you don't have a user account yet, go to https://www.photonengine.com/ and create one;
- Now follow on to your realtime dashboard and click on the *Create new realtime App* at the bottom of the page;
- Fill in your application name (there are other fields as well but they're optional) then click *create* to register your app.

That's it, you can now see your newly registered application and its AppID (see Figure 2.3 for reference). You can change your billing

plan at anytime (it defaults to the free plan, which is perfectly good for development and small-scale testing).

Remember to copy the AppID and fill it in the correspondent property on your *PhotonServerSettings* asset, as shown in Figure 2.2. You're ready to test your game. Open the *Menu* scene and hit play.

Connecting to lobby and creating/joining rooms

Now, lets see how Photon handles multiplayer games. There are two important concepts that you need to understand to develop a multiplayer game with Photon: lobby and rooms. The lobby is a virtual place where players connected to (and stay) before entering or creating a room. A room is where a multiplayer game actually happens, this time to a smaller sub-group of them.

The lobby is normally unique for each game, so every player who connects to the Photon Cloud service within your game will be in the same lobby. The exception is that you might want to have separate lobbies for different versions of your game. Lets say your latest updated version is not compatible with the old one: you might use a new lobby for the updated release, so players with both versions are able to play (only not together).

Now we'd like to show you how PUN Rally deals with connecting to the Photon servers, entering the lobby (using an initial ID/version) and enabling the players to create or join rooms. In PUN Rally, each room a player creates represents a race, and is configured to accept up to four (4) players, including the creator (called the master client, if you learn to speak Photonish).

In the *Menu* scene, there is a game object named *PUNMenuController*, which has the *PUNMenu* script attached to it. This script has several attributes that are used to hold references to GUI elements that need to be enabled, disabled and managed (see Figure 2.5). The order in which the GUI elements are enabled and used is associated with the sequence of network events that happen in the game, which are listed in the following table:

Figure 2.4: If you can see this screen it means PUN Rally correctly joined the lobby.

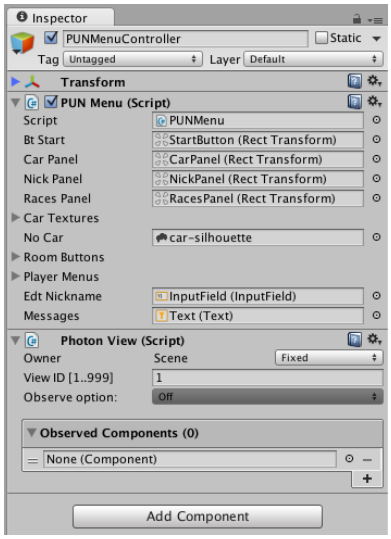


Figure 2.5: The *PUN Menu* script in the *PUNMenuController* game object holds the references to all GUI elements in the *Menu* scene and manages connection, lobby, and room creation/joining by players. Notice the *PhotonView* component as well, which is needed because we use RPCs, explained in the next section.

Event	GUI elements and actions
Enters nickname	connects and joins lobby
Joined lobby	create-room button; rooms-list
Created/joined room	car-selection; player-list; start-button
Pressed start	load race scene

Now open the *PUNMenu* script in the editor and look for the *EnteredNickname* method, called when the player types a nickname. It has the code needed to connect to the Photon servers and looks like this:

```
public void EnteredNickname() {
        PhotonNetwork.player.name =
            edtNickname.text;
        PhotonNetwork.ConnectUsingSettings("v1.0");
}
```

The code basically calls the method *ConnectUsingSettings* [1] using an initial version name as parameter, as explained earlier. It also sets the string *"Connecting..."* to a GUI label, so the player knows what's happening. When connected and having joined the master lobby, the following callback is invoked by PUN [2]:

```
// When connected to Photon Lobby, disable
    nickname editing and messages text, enables
    room list
public override void OnJoinedLobby () {
        nickPanel.gameObject.SetActive(false);
        messages.gameObject.SetActive(false);
        racesPanel.gameObject.SetActive(true);
}
```

The method *OnJoinedLobby* is one of many callbacks PUN has to deal with network events [3]. We use this one to manage some GUI elements. In this case, we disable the messages label, and enable the GUI panel containing the nickname edit-field and OK button, so the player can type his name in.

Then, the existing room listing is enabled (see Figure 2.6). The OK button was set to call a method in the *PUNMenu* script that enables the room listing panel, which also contains a button for the player to create a new one.

We chose to have a static list of up to 4 GUI elements for available rooms, which are filled with the data from the first ones returned by PUN. If there are less then 4 available, the remaining GUI elements are disabled. We use the method *GetRoomList* to list the available rooms and fill the GUI elements with their data with the following code, part of the *OnGUI* method in the *PUNMenu* script:

```
int index = 0;
foreach (RoomInfo game in
    PhotonNetwork.GetRoomList()) {
        if (index >= roomButtons.Length)
                break;
        RoomJoiner button = roomButtons[index++];
        button.gameObject.SetActive(true);
        button.RoomName = game.name;
        string info = game.name.Trim() + " (" +
            game.playerCount + "/" +
```

[1] We'll keep the code examples in this tutorial very small, so we can explain each important method. We also recommend you to take a deeper look in the code, which is fully commented. We tried to keep all scripts fairly small, but some of them, especially the ones that deal with GUI, can be a bit long.

[2] Notice that we're using the override keyword. It means this behaviour extends the provided PunBehaviour, getting the virtual implementations os PUN's callbacks.

[3] PUN has several method callbacks which are called in network events such as lobby joining, room creation/joining, player connection, etc. We'll use them extensively in this tutorial. You're encouraged to read PUN's documentation and other examples so you can feel confident about them.

Figure 2.6: This panel shows a list of up to 4 available rooms/races, each one showing the number of connected players. There is also a button for the player to create a race.

```
            game.maxPlayers + ")";
        button.GetComponentInChildren<Text>().text
            = info;
}
```

To actually join a room, we created a simple script called *RoomJoiner* that is attached to each of the four available room button game objects, which also have a *Button* GUI component. The script has a single property named *RoomName*, which is filled with the room name as just shown in the code above. When a room button is clicked, the following method is called in the *RoomJoiner* script, which tells PUN to join the correspondent room [4]:

```
// Called when this room-button is clicked
public void Join () {
    PhotonNetwork.JoinRoom(RoomName);
}
```

On the other hand, if the player clicks on new race button instead, the following code is called in the *PUNMenu* script, which tells Photon to create a room for up to four players, using the player's nickname as room name:

```
public void CreateGame () {
        RoomOptions options = new RoomOptions();
        options.maxPlayers = 4;
        PhotonNetwork.CreateRoom(edtNickname.text,
            options, TypedLobby.Default);
}
```

Both when joining a room or creating a new one, the *OnJoinedRoom* event callback is called in *PUNMenu* [5], which enables the player list and car color selection panel, which are explained in the next section.

[4] This code also implies that the client machine that is joining a room will wait until this operation is completed by Photon, which then executes the *OnJoinedRoom* callback (see next notes an follow the main text for more details.

[5] Only when creating a new room, the *OnCreatedRoom* callback is called before *OnJoinedRoom*, the contents of which are similar to another callback, so we decided not to explain it right now.

Managing players

After a player creates or joins a room, Photon is actually ready to synchronize game objects attributes and exchange messages between the client computers in the same room. Before starting the proper race, which is another Unity scene, we still want the player to be able to choose the car color and see who else he will be racing against (see Figure 2.7).

There are two important attributes we want to keep for each player in the lobby (besides the nickname, of course), which are the selected car color, and the order in which they will be placed at the start of the race (this will be use to place each car in its correspondent spawn point).

Figure 2.7: This panel shows a list of the players currently connected to the room/race. The local player can select his car color, and only the master client (the one who created the room) can start the race.

The master client computer is responsible for setting the spawn position for each connected player. Selecting the car, however, has to be an option for each player. We made use of Photon's custom player properties for both.

Custom player properties

Custom player properties are a collection of values (each associated to a key - hence a *Hashtable*), that is attached to every connected player. PUN uses its own implementation of *Hashtable* for custom properties, and the most convenient feature of it is that when you update the properties of a certain connected player in one computer, all others are synchronized and receive the updated properties for the player in question [6].

We implemented a method named *SetCustomProperties* to handle the two custom properties we use for the player who creates a room, now the master client, and for every other player that connects to that room. As default, *car* is set to zero (0) and *position* is set as the current number of players minus one. The following code shows the two callback methods in *PUNMenu* that call *SetCustomProperties* for the master client (the one who creates the room), and for every other player who joins that room [7]:

```
public override void OnCreatedRoom () {
        btStart.gameObject.SetActive(true);
        SetCustomProperties(PhotonNetwork.player,
           0, PhotonNetwork.playerList.Length - 1);
}

public override void OnPhotonPlayerConnected
   (PhotonPlayer newPlayer) {
        if (PhotonNetwork.isMasterClient) {
                SetCustomProperties (newPlayer, 0,
                   PhotonNetwork.playerList.Length
                   - 1);
                photonView.RPC("UpdatePlayerList",
                   PhotonTargets.All);
        }
}
```

The actual *SetCustomProperties* method, shown bellow, updates the custom player properties [8], receives the values for car color index (we have 6 different colors, so the index ranges from 0 to 5), and index for the player's spawn position.

[6] Every client computer connected to the same room has an updated list the correspondent player objects (instances of the *PhotonPlayer* class). Storing individual properties in them is very useful because there are straightforward methods for listing all players, or only the local one, and also checking if a player is the master client, etc.

[7] Notice the if branch that makes sure custom properties are handled by the master client computer only. Most callbacks are invoked in all client computers (see Photon's documentation for more details), so we need to pay attention to this.

[8] The *Hashtable* used for custom properties can store any type of object both as key and as value, so be very careful or you might get confused with the types.

```
private void SetCustomProperties(PhotonPlayer
    player, int car, int position) {
        ExitGames.Client.Photon.Hashtable
            customProperties = new
            ExitGames.Client.Photon.Hashtable() { {
            "spawn", position }, {"car", car} };
        player.SetCustomProperties(customProperties);
}
```

When player properties are changed, the method *OnPlayerPropsChanged* is called in all clients (see source code bellow), which basically calls for the player list to be updated on the GUI:

```
public override void
    OnPhotonPlayerPropertiesChanged (object[]
    playerAndUpdatedProps) {
        UpdatePlayerList ();
}
```

Figure 2.8: These are the car prefabs in PUN Rally. They're all the same, except for the color, but they could be completely different cars in behavior, properties and visuals. When a player changes its car color, we actually change a color index number, which is used to decide what car prefab is loaded in the race, so this feature can be extended for selecting actually different cars.

Car selector

PLAYERS LOVE TO CUSTOMIZE THEIR RIDES in racing games. Choosing car model and make; custom painting; applying special parts and accessories or changing suspension and engine parameters are common examples of this trend.

We wanted to give you a glimpse on how this can be done with photon, so we included the option for the player to select the car color [9]. Underneath, by picking a color, the player is selecting which car prefab will be instantiated when the *Race* scene is loaded (see Figure 2.8).

[9] We included in the scripts a template code for synchronizing custom colours to be applied in car materials and its remote copies. Take a look at *CarGUI.cs* and the *CreateCar* method in *PunRaceManager.cs*.

When called, the *UpdatePlayerList* method that we discussed in the previous section, enables two arrow-like buttons only for the local player car icon. These buttons are set to execute, when pressed, the methods shown below, which increment or decrement a color index:

```
public void NextCar() {
        carIndex = (carIndex + 1) %
            carTextures.Length;
        SetCustomProperties(PhotonNetwork.player,
            car, (int)
            PhotonNetwork.player.customProperties["spawn"])
}

public void PreviousCar() {
        carIndex--;
        if (carIndex < 0)
```

```
                carIndex = carTextures.Length;
        SetCustomProperties(PhotonNetwork.player,
            car, (int)
            PhotonNetwork.player.customProperties["spawn"])
}
```

Both methods call *SetCustomProperties* in the end, which is responsible for updating the local player's custom properties with the new color index chosen. Remember that Photon synchronizes player properties to all client computers in the same room.

Another interesting option would be to have the car prefabs, with controls disabled, in a carousel for the player to chose. This would be similar in implementation, but with a nicer look, and with the possibility to show more detailed features of each car in the GUI, such as engine power and other attributes. We encourage you to try this for your own game.

Track Selection

To select different tracks, we're using the same principle used for the car selector: a local variable (an attribute in PUNMenu.cs) to store the currently selected track number. The methods *NextTrack* and *PreviousTrack* are accessible from the GUI (for the master client only, the creator of the race).

To synchronize the selected track number, we use a Remote Procedure Call (RPC), a feature that is explained in the following section, together with

Loading Scenes

When is time to start the race, the player who created the room will press a button. But we still need to send this command to all other client computers, so everyone will load the same Unity scene. We accomplish this by the use of Remote Procedure Calls (RPCs). The following code, which runs in the master client, shows the GUI callback that asks Photon to remotely execute another method in all client computers:

```
public void CallLoadRace() {
        photonView.RPC("LoadRace",
            PhotonTargets.All);
}
```

RPCs, as the name implies, are a technique to remotely execute some function, method or procedure. In Photon, RPCs are used to execute the same method in several computers on the same room. The last argument in the *RPC* method even serves to tell in which computers we want the desired method to be executed. As shown above, in this case we want all connected players'computers, including the master client that is asking for this execution [10].

For a method to be called by *RPC*, it has to be marked accordingly with the *PunRPC* annotation. The following code shows how this is done for the *LoadRace* method, which makes all computers load the same scene, that represents the selected track ("Race" + trackIndex: Race0, Race1 or Race2 in our example project) [11]:

```
[PunRPC]
public void LoadRace () {
        PhotonNetwork.LoadLevel("Race" +
            trackIndex);
}
```

Now that we loaded our race scene corresponding to the selected track, lets talk about car physics and understand how a the proper multiplayer racing simulator part works.

[10] Methods called by *RPC* can have parameters too, in which case there is a third argument that is a collection of objects - the parameter values.

[11] Notice that we use the *PhotonNetwork.LoadLevel* function instead of Unity's usual *Application.LoadLevel*. This is important because we might need to instantiate game objects synchronously in the race scene, and we'd face issues because of network and local disk latency causing clients to load at different times. By using this version of the *LoadLevel* method, Photon takes care of delayed clients and makes sure all instantiations happen correctly.

3 Multiplayer Car Physics

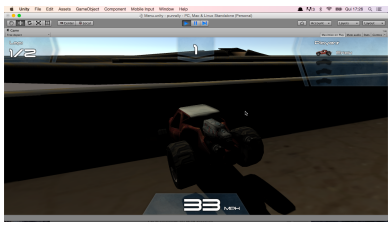


Figure 3.1: The car behavior in a simulator has to be predictable in the way a real car is. Even without a proper damage model, when colliding against a wall, the car shouldn't fly or explode, but actually change it's speed accordingly and maybe sppining out of control depending on the angle of attack. This is why it is a good approach setting up the car with a rigigbody physics engine, such as Unity's built-in PhysX.

Racing games are a very popular genre, and one we in particular like a lot. With the advances in computer games tech over the last decades, two main approaches are generally identified: arcade and simulators. Arcade racers are all about fun, without any respect to coherent physics models, etc.

Simulators, on the other hand, are all about being similar to the real world. The physics have to feel like a real (or at least imaginable) car would. If the car rolls over, it has to do in a manner that is coherent. Ideally, we should be able to set any car parameter in the same way we do in real cars, and they should be deformable too (but this deserves its own tutorial).

We decided to follow the simulator route, although a very simple one for that matter. In this chapter, we'll show you how we managed to deal with two challenges that come with this choice:

- Setting up and calibrating a reasonably realistic car using the physics engine without any type of cheating (such as fake acceleration or braking forces, below ground centers of mass, etc);
- Synchronizing these cars among all client computers in the race as smoothly as possible, also keeping the wheel behaviors and fx coherent (if a player turns his car wheels right, all other players should see the wheels of his car turning in their own computers as well).

Setting up the car behavior is the most challenging task in creating a racing game, and the one you probably spend most of your time. We do recomend you to start a racing game by doing this, because you'll need time to have it right. In this tutorial, we started talking about lobbies and rooms because that's what comes first for the player, but we actually started of by creating and tweaking the cars, and we still do.

Car Physics 101

The way a car feels to drive is the most important thing a racing simulator, so this is the first thing you should make right. In this section, we will show how to assembly a physics based car using components that come bundled within Unity, and setting it up to feel more realistic. Even if you didn't like the feel of the car in PUN Rally, we recomend you to read this section to understand how you can make it the way you want.

Structure of a physics car

WITHOUT ITS WHEELS, A CAR IS BASICALLY A RIGIDBODY, and considering we're not implementing a complex damaging model (even Gran Turismo doesn't do it properly), this is what we will start with: a game object with a *Rigidbody* component attached to it (see Figure 3.2).

We also attach separate box colliders to the car structure and set them up to resemble the car shape. This will make the rigidbody use them as its bounds, and is lighter than having a complex mesh collider. For the wheels, there is a special type of collider, the *WheelCollider* component, that works by casting rays downwards (from the car's local coordinate system perspective). Look at the figure bellow and see how the box colliders are set. Remember to make them non overlaping, because this would mess up the computation of the center of mass:

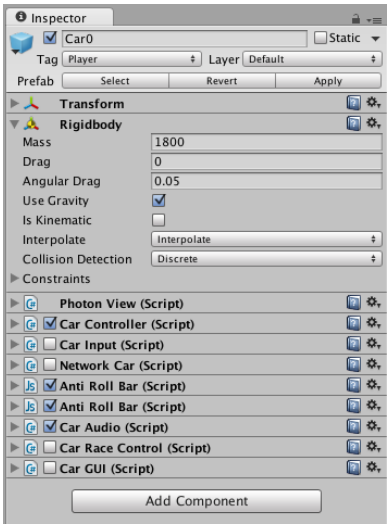
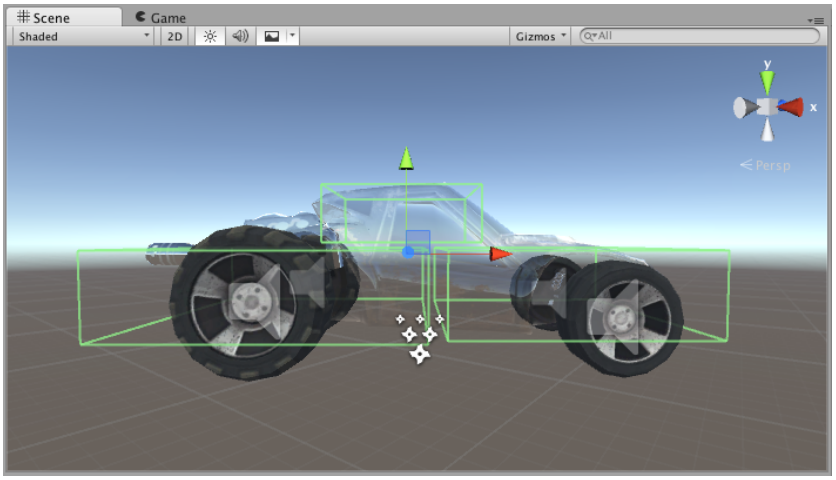


Figure 3.2: The components of the top level car game object: the rigidbody will be used to simulate the physics of the car body, with a proper mass, and letting PhysX do all the heavy work. All components are standard and come bundled with Unity, except for the disabled scripts, which we developed for the tutorial and will explain one by one.



Besides the box and wheel colliders, game object hierarchy of the car includes the visual meshes (remember that wheel meshes should be separate than the body) and some particle FX that we kept from Unity's standard assets example car. That is basically it. You can see this hierarchy in Figure 3.3.

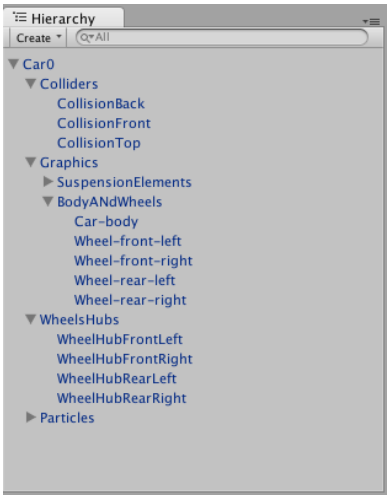


Figure 3.3: Hierarchy of the car game object.

Controlling the car

A CAR MOVES BY HAVING TORQUE applied to its wheel colliders. This will instruct PhysX to compute friction and forces, resulting in the car body moving, rotating and the wheel colliders to change position (up and down).

The engine will also compute the amount of slip and rotation of each

wheel, and this is fundamental because it lets you implement many nice features a real car has, such as limited slip differentials, AWD systems, etc.

For this tutorial, however, we stick to standard assets as much as we can, so we're using Unity's *CarController* script to do this work. It is not optimal, but can be configured to AWD, FWD or RWD modes, and has several parameters that can be changed to tweak the car. We'll talk about some of them in the next section.

The most important scripts that we implemented for the tutorial are *CarInput* and *NetworkCar*. *CarInput* allways sends input values to the standard *CarController* script, but these come from two different origins: if this is the car controlled by the local player, the input comes from Unity's input system. If it is a remote car, the input will come from the *NetworkCar script*, which is explained in a later section. The following code from *CarInput* shows how this is done:

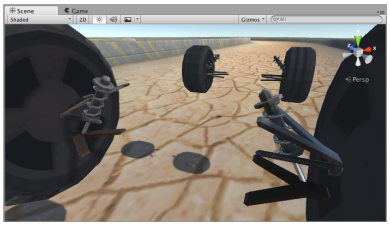


Figure 3.4: The car's suspension arms are a set of separate meshes that can be moved independently of the rest of the car's visual. We created a script (*SuspensionArm*) and attached it to each arm that we wanted to move. The script uses a reference to the correspondent wheel collider, so the suspension travel can be calculated and used to rotate the suspension arm accordingly, so the suspension movement doesn't affect the physics at all, being actually computed based on the simulation.

```
void FixedUpdate() {
        // If car is locally controllable, read
           and cache player input
        if (controlable) {
                Steer =
                   CrossPlatformInputManager.GetAxis("Horizontal");
                Accell =
                   CrossPlatformInputManager.GetAxis("Vertical");
                if
                   (CrossPlatformInputManager.GetButton("Fire3"))
                   {
                        Unflip();
                }
                #if !MOBILE_INPUT
                Handbrake =
                   CrossPlatformInputManager.GetAxis("Jump");
                #endif
        }
        // Allways move the car, independently of
           wether the input values
        // came from (local player or remote).
        car.Move(Steer, Accell, Accell, Handbrake);
}
```

Car physics tweaks and perks

Setting car parameters right is subject of much debate in the Unity forums (and many other game developer communities as well). If

you're going the simulator route as we are, these are the things to look for:

- Setting mass and center of mass as a real car. The center of mass should be as centralized as possible, otherwise the car will behave very erratically when rolling over [1]. You might want to dislocate it a bit in the direction where your car engine is;
- Setting up *WheelCollider* parameters. The radius, mass, spring, damper and target position must resemble those of a real car. The friction parameters, however, are a bit trickier, so we encourage you to start playing with spring and damper before digging into more comple stuff;
- Having a anti-roll bar script implemented and properly set (see Figures 3.5 and 3.6);
- Ideally, implemeting your own car controller script (Standard Asset's one is very limited), and making sure you implement things such as brake balance, engine torque curve, gears that affect wheel torque, wind resistance, traction control, etc. Doing all this is out of scope of this tutorial, but you can take a look at the Unity Forums. We also recommend you to search the internet for good references on car physics such as [2];

[1] Many people set below ground center of mass to avoid rolling over, but this is cheating - a normal car would roll over easily as well, except that it has anti-roll bars, which we discuss below.

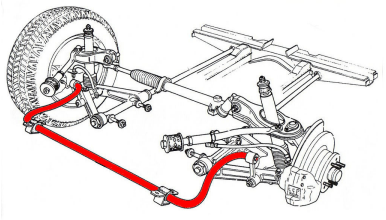


Figure 3.5: Anti-roll bars are also known as anti-sway bar, sway bar, and stabilizer bar. In real life, cars have anti-roll bars attaching left and right wheels of the same axis, especially for independent suspensions. Their function is to distribute the load from the outside wheel to the inside one in a turn, avoiding too much roll.

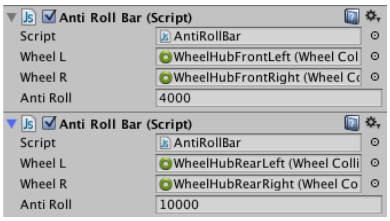


Figure 3.6: One trick we learned from a mechanical engineer who works for a major car maker is that you can change the car grip by having different values for the stabilizer bar torsion strength in the front and rear axel. We set the front axis to roll more (less tortion strength), so the car digs and has more grip on that axel. The rear tortion strength is higher, so the car drifts more easily. Try to play with these parameters and see what you can do.

[2] Marco Monster. Car physics for games. http://www.asawicki.info/Mirror/CarPhysicsforGames/CarPhysicsforGames.html, November 2003

Synchronizing remote cars

In PUN Rally, each player's computer is responsible for instantiating the car prefab for the local player, which is controlled by the *CarInput* script. Photon will instantiate a copy of this prefab in all the other players' computers, but making these remote copies move and behave in sync with the local original doesn't happen automatically.

The idea is pretty simple: run periodic updates that send transform data from the locally instantiated car prefab to all remote computers to be applied in the respective remote copies, which are moved around as *kinematic* rigidbodies. In this section, we'll see how PUN helps us to do this, and what is the data we need to synchronize to have the remote copies move smoothly and coherently.

The PhotonView Component

PUN comes with the *PhotonView* component, which takes care of synchronizing a game object's data, when attached to it. It comes of the shelf with the ability to synchronize the *Transform* component

properties, but we'll need to perform a more detailed synchronization technique, which we'll explain in the following sections.

There are options for sending periodic synchronization (reliable and unreliable) or only when there is a property change (see Figure 3.7).

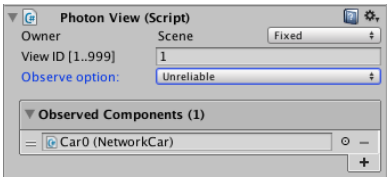


Figure 3.7: The *PhotonView* component and its properties. For custom synchronization, we need to tell which component should be observed for changes and have its callback executed. In our case, we set it to the *NetworkCar* script we implemented. Notice that we chose the *Unreliable* observe option. This will tell Photon to constantly synchronize cars using a fast (but not reliable) network protocol. There are option for reliable (not so fast) and on properties change only (also unreliable).

Custom properties serialization

THE PROBLEM WITH SYNCHRONIZING only Transform data with periodic updates is that the position and rotation of the remote cars would change in a rate which is much lower than the actual frame-rate of the rendering and local physics, leading to heavy stuttering. Network latency only makes it worse, so we need a better option, given a racing game relies on smooth car movement.

There is also the issue of synchronizing wheel rotation, steering, and special FX, which are computed based on wheel rotation and slip rates.

Because of this, we implemented the *OnPhotonSerializeView* callback to send more data than only the Transform properties [3]:

[3] Besides the *Transform* position and rotation data, we synchronize there input attributes, which will be used to spin and steer wheels on the remote cars, and the velocity vector from the car's *Rigidbody*, which will be used for dead recknoning.

```
void OnPhotonSerializeView(PhotonStream stream,
    PhotonMessageInfo info) {
        if (stream.isWriting) {
                //We own this car: send the others
                    our input and transform data
                stream.SendNext((float)m_CarInput.Steer);
                stream.SendNext((float)m_CarInput.Accell);
                stream.SendNext((float)m_CarInput.Handbrake);
                stream.SendNext(transform.position);
                stream.SendNext(transform.rotation);
                stream.SendNext(rb.velocity);
        }
        else {
                //Remote car, receive data
                m_CarInput.Steer = (float)
                    stream.ReceiveNext();
                m_CarInput.Accell = (float)
                    stream.ReceiveNext();
                m_CarInput.Handbrake = (float)
                    stream.ReceiveNext();
                correctPlayerPos = (Vector3)
                    stream.ReceiveNext();
                correctPlayerRot = (Quaternion)
                    stream.ReceiveNext();
                currentVelocity = (Vector3)
                    stream.ReceiveNext();
```

```
                updateTime = Time.time;
        }
}
```

Dead Reckoning

JUST SENDING THESE PROPERTIES here and there doesn't do anything useful yet. We need to use them to smoothly update the remote cars, and find a way to compensate for network instability.

Interpolation smoothing

TO AVOID STUTTERING, we can use interpolation to smoothly update the car position and rotation over the course of a few frames. This will give time for Photon to send/receive the next synchronization update, creating a more fluid movement that can even mask variations in network latency. This would be a version of the *FixedUpdate* method with just interpolation:

```
public void FixedUpdate() {
        if (!photonView.isMine) {
                transform.position = Vector3.Lerp
                   (transform.position,
                   correctPlayerPos,
                   Time.deltaTime);
                transform.rotation =
                   Quaternion.Lerp
                   (transform.rotation,
                   correctPlayerRot,
                   Time.deltaTime);
        }
}
```

However, an aggressive interpolation has a serious issue: any position update takes some time to reach the remote computers, and adding to this the interpolation that distributes the update over several frames makes the remote copies of the cars to always stay behind a few meters. Also, the larger the latency, worse the lag for remote cars.

Using prediction

Because of this, we can't be too aggressive with the interpolation, and find a better way to compensate for network instability. The idea is to try to predict where the car will be be in the next frames based on the

position, orientation, and the velocity vector that tells us the rate and direction the car is moving at the last update. This is the actual code that we use to update a remote car:

```
public void FixedUpdate() {
        if (!photonView.isMine) {
                Vector3 projectedPosition =
                    this.correctPlayerPos +
                    currentVelocity * (Time.time -
                    updateTime);
                transform.position = Vector3.Lerp
                    (transform.position,
                    projectedPosition,
                    Time.deltaTime * 4);
                transform.rotation =
                    Quaternion.Lerp
                    (transform.rotation,
                    this.correctPlayerRot,
                    Time.deltaTime * 4);
        }
}
```

We need two properties to compute the predicted position of the car in a certain frame: the velocity vector, and the time since the last update. The method above computes and stores the desired position for the car in the *projectedPosition* variable. It then uses interpolation, now with a smoothing factor, to update the actual transform data.

For this to work, all remote cars have their *Rigidbody* set as kinematic, so the *NetworkCar* script can change its transform directly. This way, the input data will make the wheels spin and steer coherently, but this will have no effect in the car position and rotation, which are completely controlled by the *NetworkCar* script.

There are other options for dead reckoning, such as using actual physics simulation between update frames. This is much more complex to do, but the FX can be more coherent with the input data.

4 Racing Gameplay Control

What else do we need? We already have a functional *Menu* that connects to Photon and enables players to create/join rooms and select cars. Now we need to load the *Race* and make sure all cars are instantiated in all machines, that the race is started simultaneously, and the car positions are shown in real time. Besides, we also included a simple weapon system, that will be incremented in future updates for combat-based racing games, and is explained at the end of this chapter.

The code we'll show is split between two scripts (we'll specify which one we're talking about every time we show some code): *PUNRaceManager*, which is attached to a single control game object (*PUNManager*) in the *Race* scene; and *CarRaceControl*, that is attached to all car prefabs.

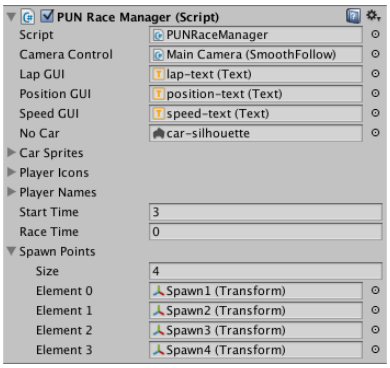


Figure 4.1: The spawn points are referenced in the *PUNRaceManager* scripts, so the car instantiation car use their position and rotation.

Grid positioning

The first thing we have to take care of is where to instantiate each player's car in the scene. We decided to place and orient static game objects to act as spawn points, shown in Figure 4.2.

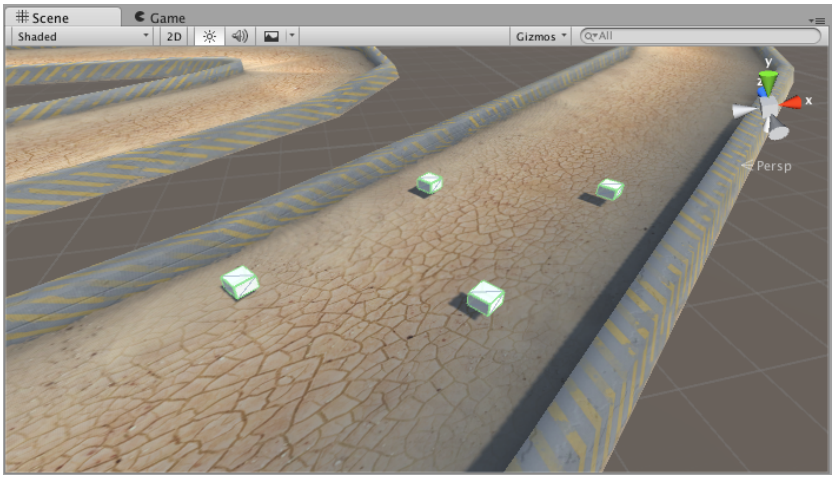


Figure 4.2: We positioned the spawn points and attached them to the race track prefab. Using this approach you can have many different tracks (prefabs), and the cars can easily be positioned at the starting grid.

You should have noticed that, to develop a game, a lot of effort is given to extra features that are not exactly pure game mechanics. It'd be great if we could just implement the racing physics, but in the real world, we need to do everything from GUI control to error checking.

How do we use spawn points?. Take a look at the code below, taken from the *PUNRaceManager* script. It is called when the master client calls an RPC ordering all peers to load the *Race* scene:

```
private void CreateCar() {
        // gets spawn Transform based on player
           join order (spawn property)
        int pos = (int)
           PhotonNetwork.player.customProperties["spawn"];
        int carNumber = (int)
           PhotonNetwork.player.customProperties["car"];
        Transform spawn = spawnPoints[pos];

        // instantiate car at Spawn Transform
        // car prefabs are numbered in the same
           order as the car sprites that the
           player chose from
        car = PhotonNetwork.Instantiate("Car" +
           carNumber, spawn.position,
           spawn.rotation, 0);

        // some code omitted for clarity.
}
```

It takes the *car* and *spawn* custom properties of the local player, which will be used as the car prefab and spawn point indexes respectively. Then ir instantiates the right car prefab in the correct position and orientation. Notice that we used *PhotonNetwork.Instantiate* instead of the regular Unity instantiation. This is mandatory for game objects that you want to have loaded remotely in all peers. We omitted some control code in this method, so take a look at the script in the editor and read all the comments to understand everything that it does after that.

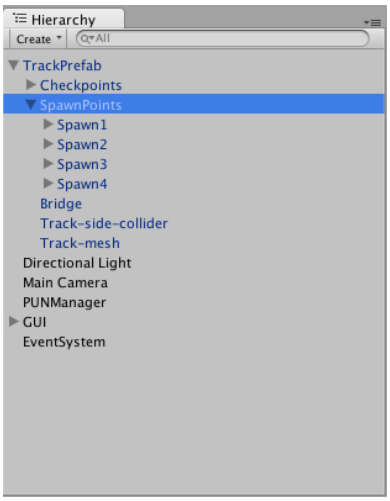


Figure 4.3: The track prefab includes graphics, colliders, spawn points and checkpoints. Spawn points are place to mark where cars will be instantiated. Checkpoints are used by race control.

Race Start, Timer and Finish

Cars controls are disabled when we load the prefabs. We need to do a countdown and start the race roughly at the same time in all client machines. The control code in *PUNRaceManager* uses an enumeration to represent four different race states: PRE_START, COUNTDOWN, RACING, and FINISHED.

The code snippet below, part of the *Update* method, checks (at the master client machine only) if it is time to start the countdown or the actual racing, in which case it then calls an RPC in all machines that will change the race state and release the controls for the local car.

```
raceTime += Time.deltaTime;
switch (state) {
case RaceState.PRE_START:
        if (PhotonNetwork.isMasterClient &&
            raceTime >= 1) {
                photonView.RPC ("StartCountdown",
                    PhotonTargets.All,
                    PhotonNetwork.time + 4);
        }
        break;
case RaceState.COUNTDOWN:
        if (PhotonNetwork.time >= startTimestamp) {
                StartRace();
        }
        break;
```

Pay attention to the clever use of *PhotonNetwork.time* instead of Unity's managed local *Time.time*: in a multiplayer racer, we want all clients to start the race at roughly the same time, so we use an RPC to set a future time (+4 seconds from now) at PRE_START. Then, during the proper COUNTDOWN, we compare Photon Server's managed time to this future time in all computers, this way we don't need to deal with the individual latencies of each computer.

The *StartRace* method runs in all peers. It does three things: sets the race state to RACING; enables controls for the local player's car; and creates a list of all cars to control position calculation:

```
public void StartRace() {
        state = RaceState.RACING;
        GameObject[] cars =
            GameObject.FindGameObjectsWithTag
            ("Player");
        foreach (GameObject go in cars) {
                carControllers.Add(go.GetComponent<CarRaceControl>());
                go.GetComponent<CarInput>().enabled
                    = true;
        }
        car.GetComponent<CarInput>().controlable =
            true;
        raceTime = 0;
}
```

Laps and player position

To count the car laps, each car has a *CarRaceControl* script attached. This script has one important property, named *currentCheckpoint*, that points to the next checkpoint the car is supposed to pass through to count for race progress.

Checkpoints are basically colliders marked as triggers, so they won't interfere with the physics, but the *OnTriggerEnter* callback is called on *CarRaceControl* whenever its containing car passes though one of the checkpoints:

```
void OnTriggerEnter(Collider other) {
        Checkpoint waypoint =
            other.GetComponent<Checkpoint>();
        if (waypoint != currentWaypoint) {
                //Debug.Log ("Missed Check - " +
                    other.gameObject.name);
        } else {
                waypointsPassed++;
                if (waypoint.isStartFinish)
                        lapsCompleted++;

                if (lapsCompleted == totalLaps) {
                        CarInput input =
                            GetComponent<CarInput>
                            ();
                        input.controlable = false;
                        input.Steer = 0;
                        input.Handbrake = 1;
                        input.Steer = 0;
                }
                currentWaypoint = waypoint.next;
        }

}
```

The method checks if the cause of the trigger callback was the *currentCheckpoint*, in which case it increments the number of checkpoints passed, increments the laps if this is the finish line (there's only one checkpoint with this property true), and verifies if the number of laps completed is equal to the race laps, in which case is time to disable controls and stop the car. Notice that it also sets *currentCheckpoint* to the next, given they are organized as a circular linked list.

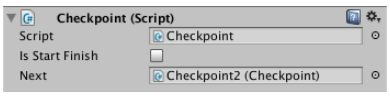


Figure 4.4: The *Checkpoint* script with the reference to the next checkpoint and the boolean property for the start/finish.

Each checkpoint game object has a *Checkpoint* script attached, which

is used just to hold a reference to the next checkpoint in the track, and a boolean property to mark the sole start/finish checkpoint (explained previously).

The laps are counted everytime the car passed by the start/finish checkpoint, but how can em calculate the relative positions of the cars in real-time? The idea is to store how far each car has traveled into the race.

This is the reason we also count how many checkpoints the car has passed. This would give a rough idea of progress for each car. But how can we update the distance traveled in a frame by frame fashion? We'll do this by measuring how far we are from the next checkpoint (the *currentCheckpoint* property).

With each car storing how far it has traveled into the race, all we need to do is sort them in reverse order. The next sections explain how we do this.

Counting the checkpoints

Figure 4.5 shows the placement of the checkpoints in the track. The ccalculation of the distance traveled for each car is done in the *UpdateDistanceTraveled* method, which is called every frame:

```
void UpdateDistanceTraveled() {
        Vector3 distance = transform.position -
           currentWaypoint.transform.position;
        distanceTraveled = 1000 * waypointsPassed
           + (1000 - distance.magnitude);
}
```

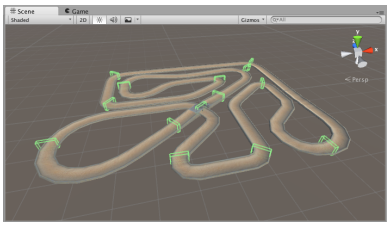


Figure 4.5: These are all the checkpoints for our track. Notice that we positioned them at basically every turn. This is needed for correctly computing how much the car has traveled based on the number of checkpoints passed and the distance to the next. If you want to create a racing AI based on checkpoints, you'd need to place many many more of them along the track.

We multiply the number of checkpoints passed by a large number, so it is given priority in the final result. The distance to the next checkpoint is subtracted from another large number, so it will increase as the car travels in the right direction (and would decrease if the car goes the wrong way. Now we need to sort the cars based on this computed distance.

Sorting cars and GUI update

Earlier we showed the *StartRace* method, which created a list with all *CarRaceControl* instances in the race. A C# *List* can be sorted with its *Sort* method, if every element of the list knows how to compare to each other. This is done with the *CompareTo* method below:

```
public int CompareTo(CarRaceControl other)
{
    // end of race code ommited
```

```
        if (distanceTraveled >
            other.distanceTraveled)
                return -1;
        else if (distanceTraveled <
            other.distanceTraveled)
                return 1;
        else
                return 0;
}
```

Now, we only need to sort the *CarRaceControl* list, and use this information to update the GUI objects, referencing the cars in the correct order, so each GUI object can be responsible for updating the car icon, player name, and race time (for the ned-of-race panel only). We do this every frame update in *PUNRaceManager's SortCars* method [1]:

[1] Implementing a *CompareTo* method is a good object oriented programming practice to sort objects that have an intrinsic order. We chose to do this for the *CarRaceControl* script, but you can do it for different classes in all kinds of situations that need sorting to be done. Of course you might want to implement your own version of the *QuickSort* algorithm, but using existing reference implementations from an API is always good practice.

```
private void SortCars() {
        carControllers.Sort();
        int position = 1;
        foreach(CarRaceControl c in
            carControllers) {
                c.currentPosition = position;
                position++;
        }
}
```

The method *UpdatePlayerPanel* is used to set the car references, in the correct position-based order, to the appropriate *CarPanelGUI* components in the GUI objects:

```
private void UpdatePlayerPanel (List<CarPanelGUI>
    guis) {
        foreach(CarRaceControl c in
            carControllers) {
                guis[c.currentPosition -
                    1].SetCar(c);
        }
}
```

Weapon System

MANY RACING GAMES INCLUDE WEAPONS, so we implemented a simple, extensible weapon system beginning with update 1.0.6 (see Figure 4.6). There are three scripts that are used to control the weapon system:

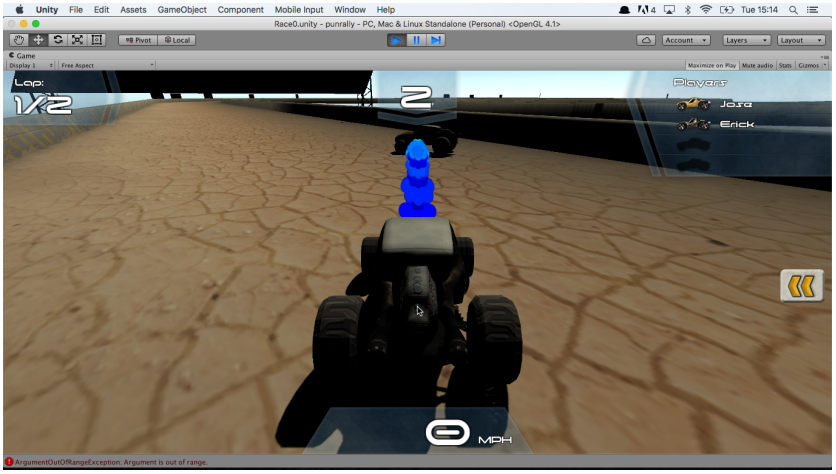


Figure 4.6: The weapon shoots rocket-like projectiles in other cars, applying an upward impulse force on impact.

- *Weapon* - attached to the car prefabs (Figure 4.7), it is responsible for shooting the projectiles, which consists of instantiating a copy of a bullet prefab in all client computers. It also takes care of its own cooldown time, etc;
- *Bullet* - attached to the bullet prefab (Figure 4.8), it is responsible for synchronizing the projectile position over time and synchronizing this over the network. In this first version, bullets behave like rockets. In future updates, we might include other bullet types;
- *Damage* - also attached to the car prefabs, this script detects when a car is hit by a bullet, in which case the damage (in this case an upward force from an explosion) is applied, and a message is sent to the bullet (RPC, by the *Bullet* script), ordering it to destroy itself and show an explosion;

Take a look at these scripts and see how simple they are (none is more than 30 lines of code). The idea is not to have a definitive weapon system, but a head-start for you to implement your own ideas.

This is it for car race control. In the next chapter, we'll discuss some other ideas the you might use to your racing game, including some stuff not covered in this tutorial.

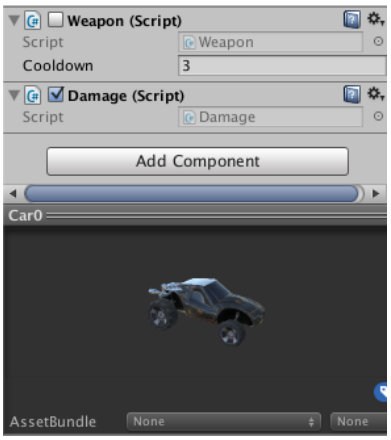


Figure 4.7: The *Weapon* and *Damage* scripts attached to a car prefab.

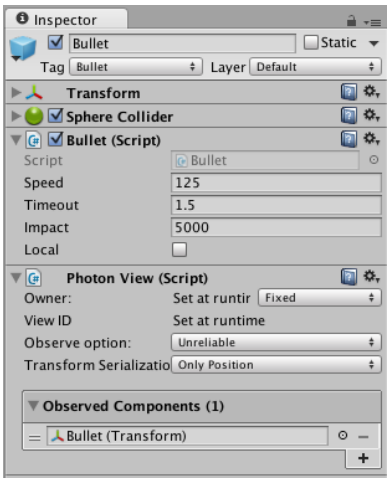


Figure 4.8: The bullet prefab inspector, showing the *Bullet* script and its other components.

5 Conclusion

In this tutorial we've shown you how to create a complete multiplayer racing game. The features implemented and explained here include:

- Photon account setup, connection to Photon Network, and hosting/joining races;
- Player nickname and synchronized selection of car color/prefab;
- A realistic physics car based on simulation with PhysX;
- Distributed car synchronization with dead reckoning (prediction) to compensate for network instability;
- Race start and control and position calculation based on checkpoints;
- A simple weapon system that fire rocket-like projectiles and applies a force on impact;

It is a lot of stuff already, but there's always room for improvement, so we've taken the liberty to suggest you some things you might want to have implemented in your racing game.

What to do Next

One of the characteristics of great games is their uniqueness, so we highly encourage you to think outside the box and try different approaches with your own project.

What about trying an arcade style physics? Can you do this and still keep the *Rigidbody* and *CarInput* scripts? If you can, there no need to change the network code at all. It will still work of the shelf.

But you might also try to implement an authoritative master client, which runs all the physics simulation, receiving input from the peers and giving back the updated positions. This is good to avoid certain types of cheating, but the network latency will be noticed a lot.

Did you know that checkpoints are also the basic structure for doing race AI? If you can store the recommended speed for each checkpoint (lets say by doing trial runs), you can have an AI control script doing the input to steer and accelerate/brake the car along the track.

Would it be difficult to use other racers driving styles to influence your AI the way it is done in Forza Motorsports?

Now it is your turn to code!

Bibliography

Marco Monster. Car physics for games. http://www.asawicki.info/Mirror/CarPhysicsforGames/CarPhysicsforGames.html, November 2003.