Multiplayer Racing Template

(Netcode for GameObjects, Lobby, Relay)
v1.0.0

HEAVYART
2024

About

Multiplayer project template. Designed as a starter pack: recommended to import package into a new empty project.
Contains 5 scenes, 3 cars, 3 weapons (machinegun, rocket launcher, mine launcher), 4 elements to pick up (missiles, mines, medkit and nitro refill).
Uses Lobby, Relay multiplayer services and Netcode for GameObjects.
Fixed update time = 0.012. (default = 0.02)

Includes manifest with required packages:
Lobby 1.1.2
Netcode for GameObjects 1.5.2
Relay 1.0.5
Input System 1.3.0

Project structure:
\Content - car models, icons, materials
\Prefabs - car prefabs (player, enemy), pick-up element prefabs, UI, weapons, VFX
\Scenes - all game scenes, including demo scene at \Scenes\Demo\Demo.unity
\ScriptableObjects - pick-up element configs, NetworkPrefabsList (required by Netcode)
\Scripts - all game code

Scenes:
BootScene - Scene to start from. Used as a place to initialize settings and network. Loads only once - on game start.
LoadingScene - Scene that we see, when switching between scenes.
MainMenu - menu, where we can start new game or join existing, set player name, select player color and switch server region.
GameScene - scene with gameplay itself
Demo - demo scene at \Scenes\Demo\Demo.unity

Where to start:
After the installation guide, we recommend to start from SettingsManager on Boot Scene. It's a permanent object (DontDestroyOnLoad) which contains all project settings, therefore it's the best entry point, which would lead to all the parts of the game.
Also we recommend to check PlayerDataKeeper. This script works with PlayerPrefs and contains data about local player.

Running on few instances:
To run several instances on the same computer, could be used ParrelSync for editor and command line arguments, for standalone builds: -authProfileName uniqueProfileName, where uniqueProfileName - unique name for every instance.
If you are using ParrelSync, uncomment code at LobbyDataControl, lines 109-117.

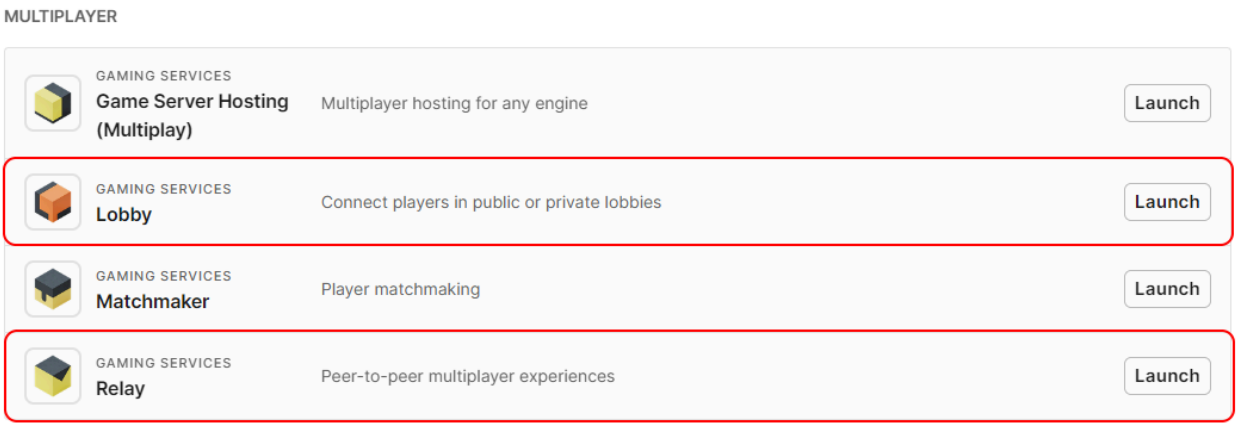
```
108             //Uncomment code below if you are using ParrelSync
109             //#if UNITY_EDITOR
110
111             //if (Application.isEditor && ParrelSync.ClonesManager.IsClone())
112             //{
113             //    string customArgument = ParrelSync.ClonesManager.GetArgument();
114             //    AuthenticationService.Instance.SwitchProfile($"Clone_{customArgument}_Profile");
115             //    PlayerDataKeeper.authProfileName = customArgument;
116             //}
117             //#endif
```

Installation Guide

1. Create project in Unity Dashboard (if it's not created)
   1.1 Select Projects from the primary navigation menu.
   1.2 Click Create project in the upper-right of the Projects page.
   1.3 Enter a project name and COPPA designation.
   1.4 Click Create project.

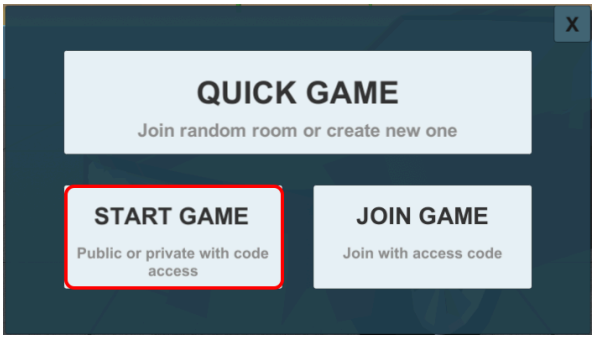
For more information, see the documentation on managing Unity projects.

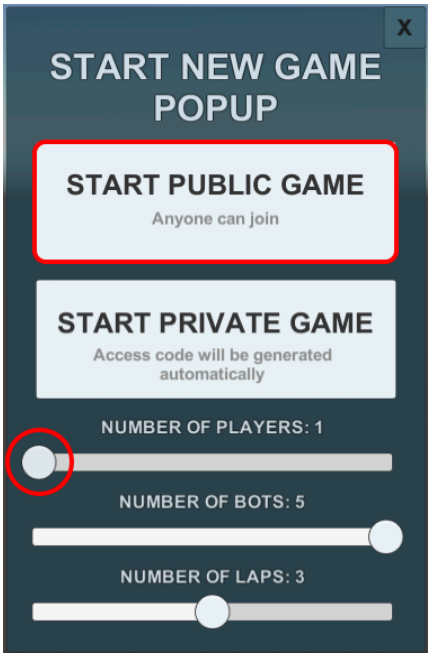
2. Setup Lobby and Relay in Unity Services Dashboard > Multiplayer



3. Launch full game version from **\Scenes\BootScene.scene**
or demo version from **\Scenes\Demo\Demo.scene**

To launch game for 1 player press START RACE, then START GAME, set "NUMBER OF PLAYERS" to 1 (move slider left) and press "START PUBLIC GAME"

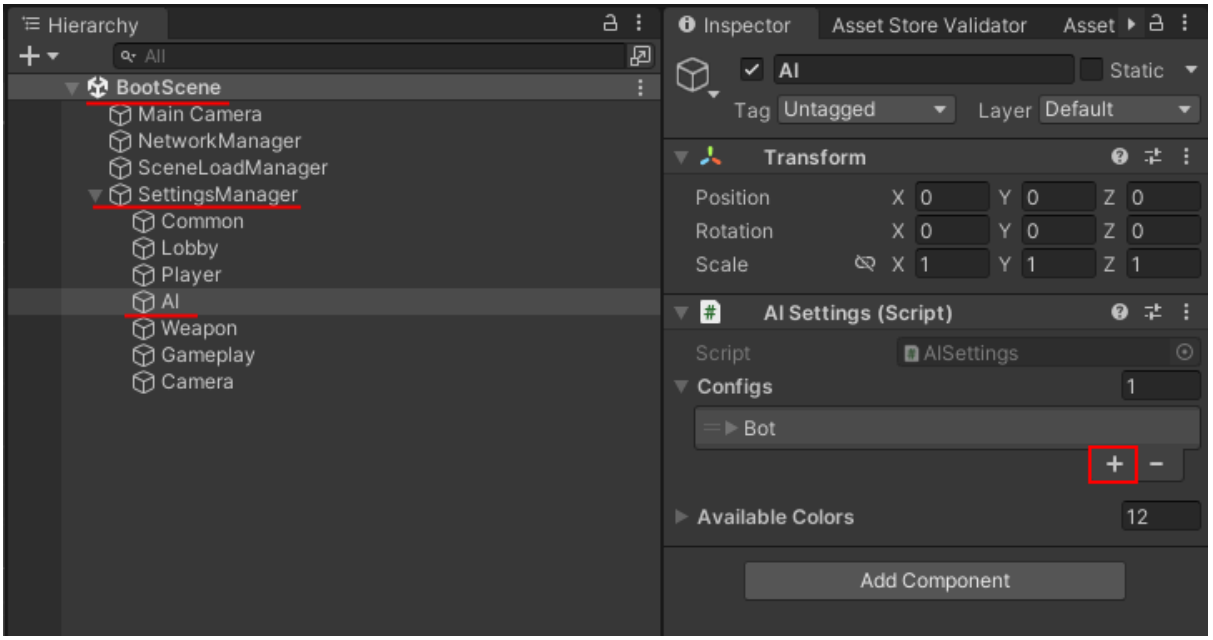




How To

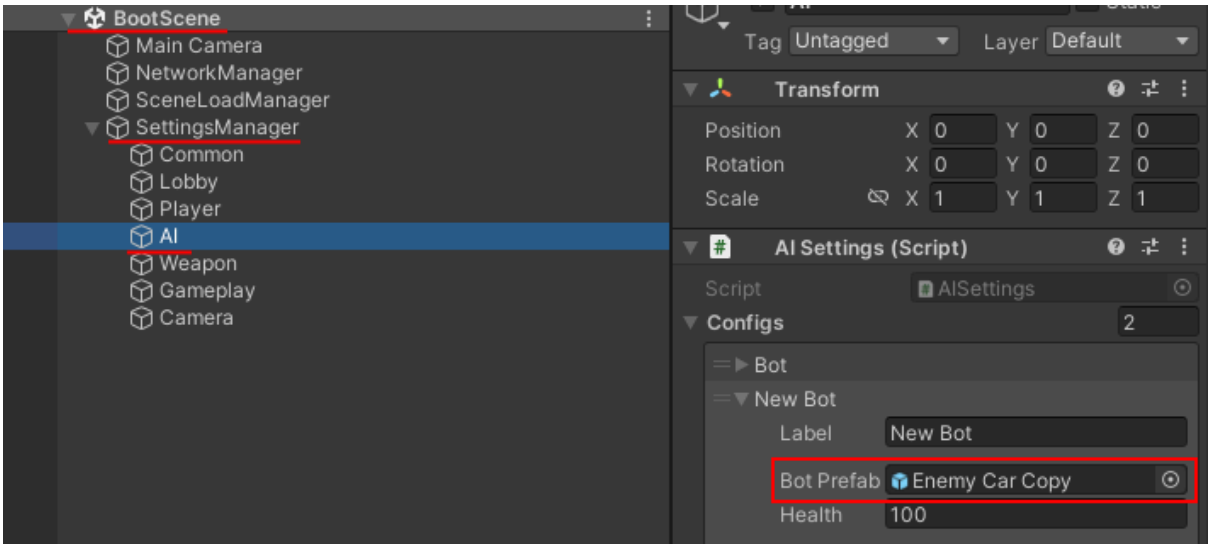
How to add new bot

- Open boot scene.
- Find SettingsManager. Select AI child object.
- Press **+** below Configs collection. It will automatically copy last bot config and add it to the end of Configs collection. Now, it's possible to change its settings.



To add new AI prefab:

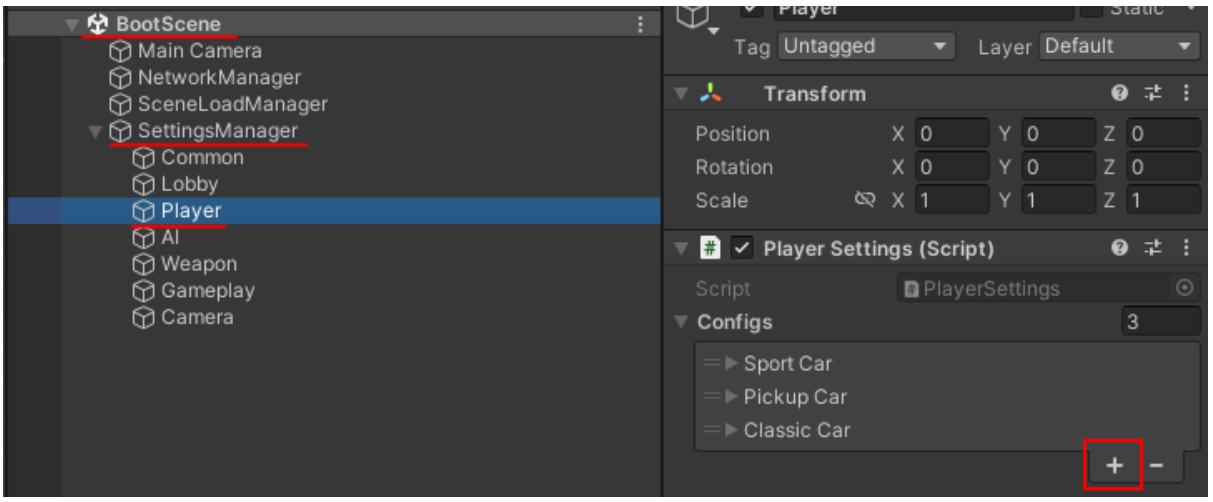
- Copy existing enemy car prefab and put its copy to Bot Prefab field.
- Add new prefab to NetworkPrefabsList at \ScriptableObjects\



When game spawns a bot, it selects one, randomly from the configs list.
For more details, check source code (NetworkObjectsSpawner.cs).

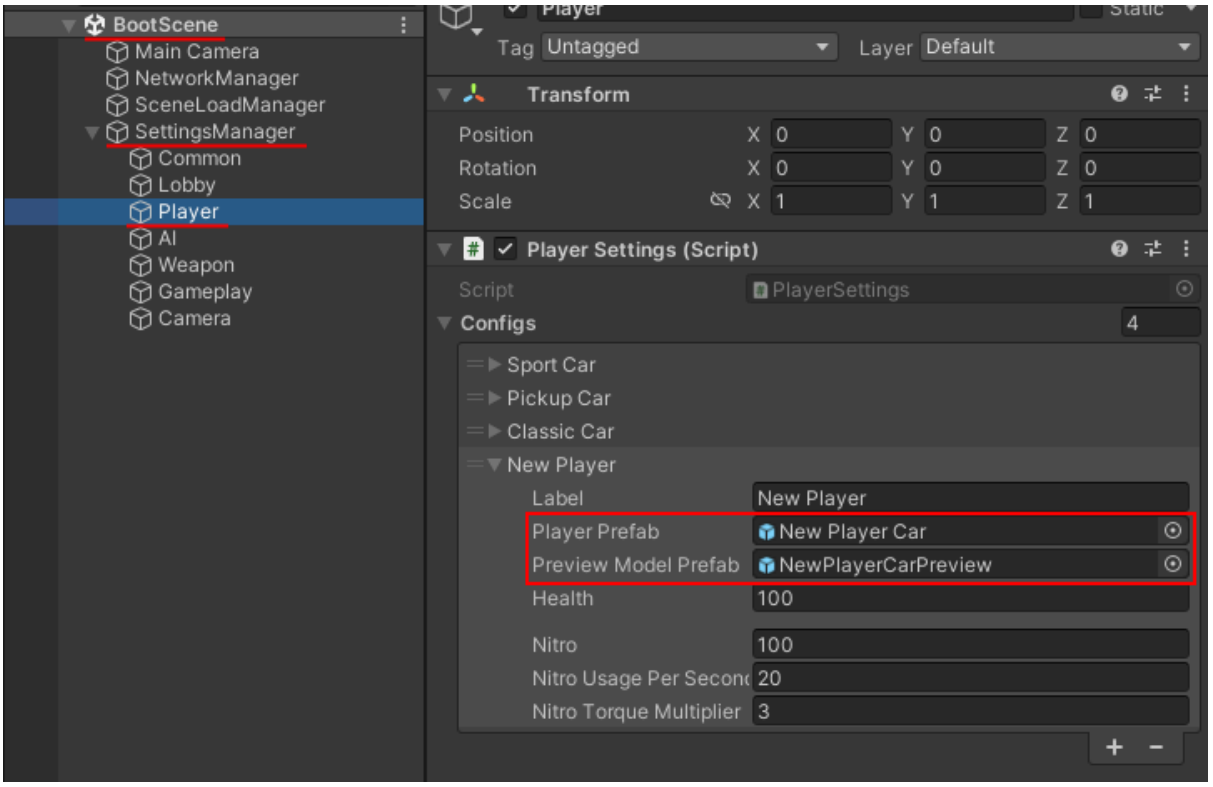
How to add new player's car

- Open boot scene.
- Find SettingsManager. Select Player child object.
- Press **+** below Configs collection. It will automatically copy last player config and add it to the end of Configs collection. Now, it's possible to change its settings.



To add new player prefab:

- Copy existing player's car prefab, and put its copy to Player Prefab field.
- Copy preview model prefab, and put its copy to Preview Model Prefab field.
- Add new prefab to NetworkPrefabsList at \ScriptableObjects\

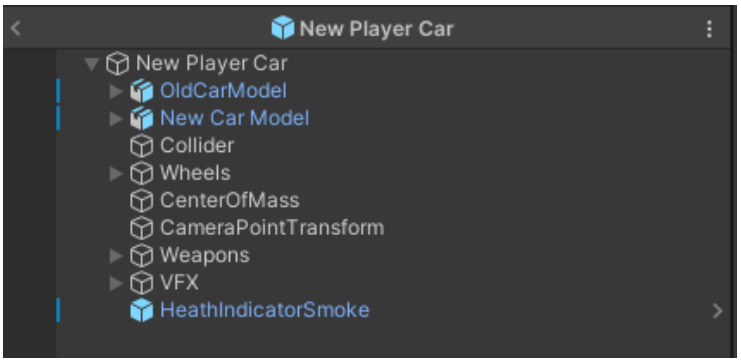


Done! Now you can run the game and check your new car.

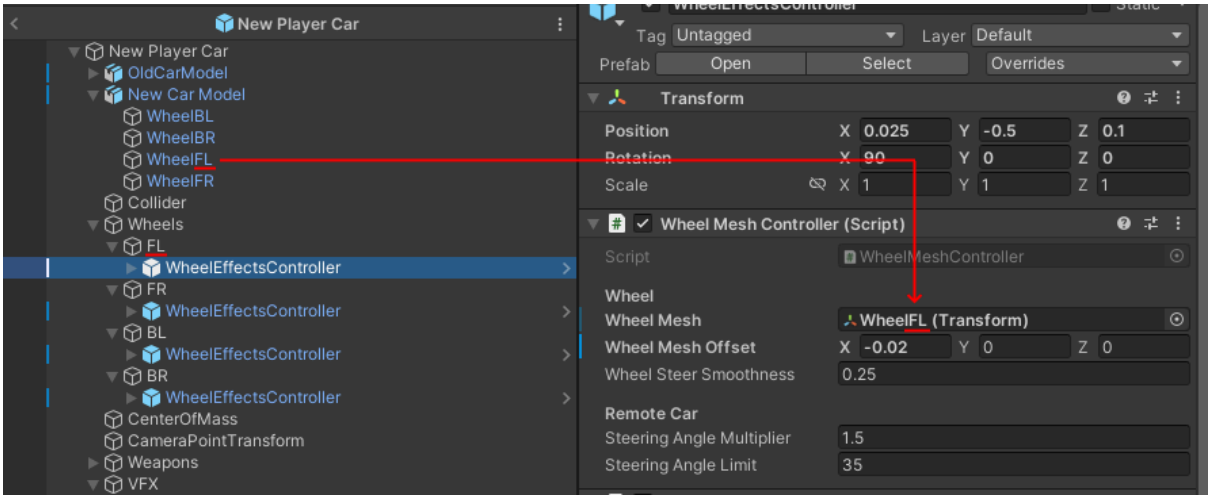
For more details, check source code: PlayerDataKeeper.selectedPrefab property and ServiceUserController.GetLocalPlayerSpawnParameters() method.

How to change car model (Player, Enemy)

- Open car prefab.
- Put a new car model, next to the old one. Setup its position and rotation. Suppose to be close or equal to zero.

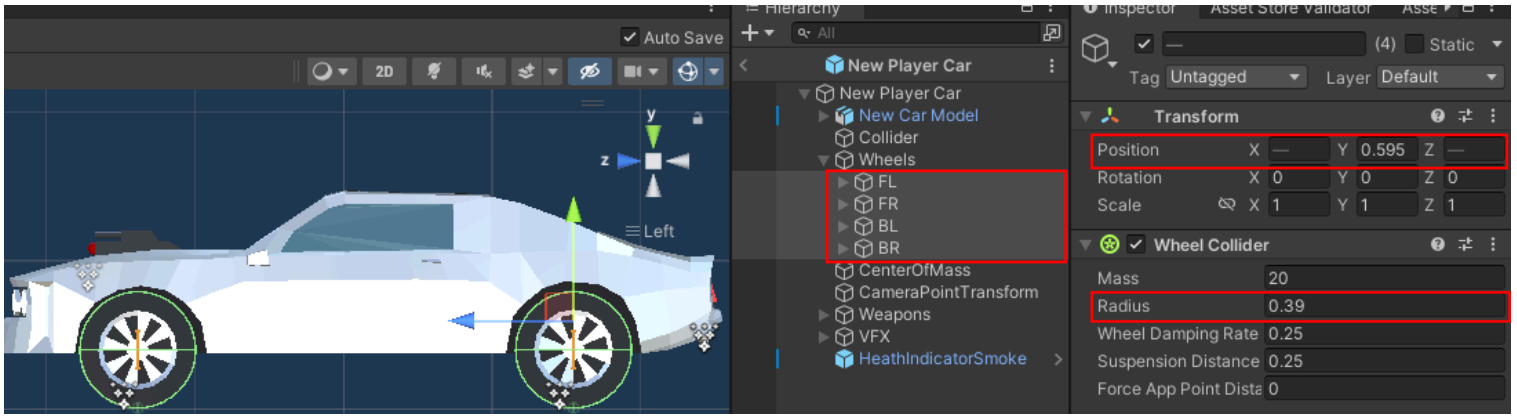


- Link car wheel models with Wheel Mesh Controllers. Put the wheel model transforms to the Wheel Mesh fields.
FL - Front Left, FR - Front Right, BL - Back Left, BR - Back Right

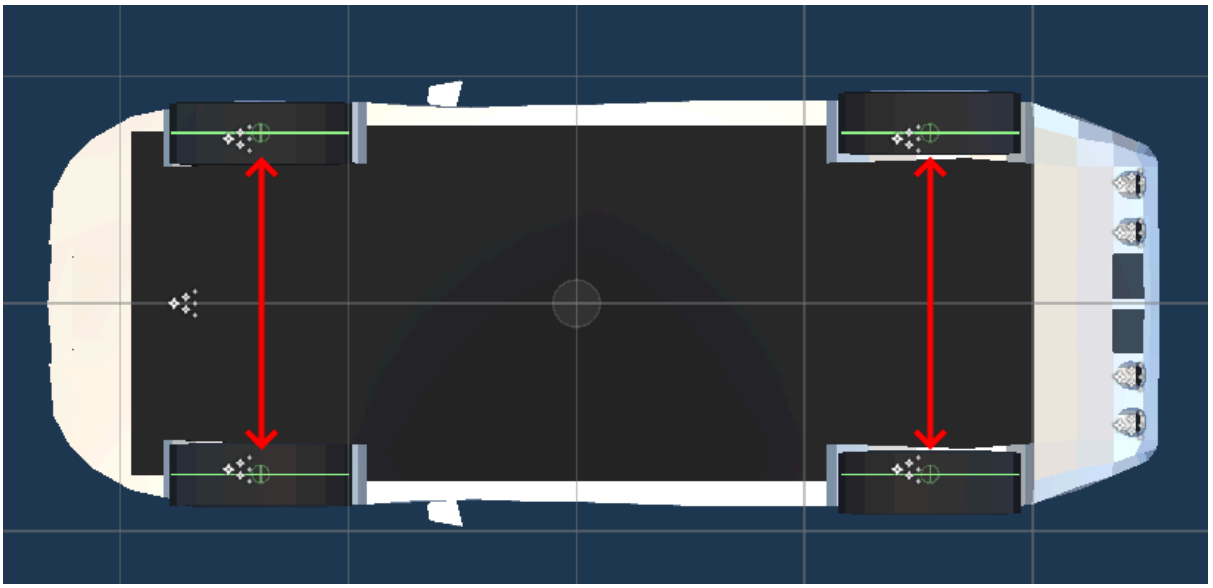


Here, you can also adjust wheel model offset in the Wheel Mesh Offset field.
For more details, check WheelMeshController prefab.

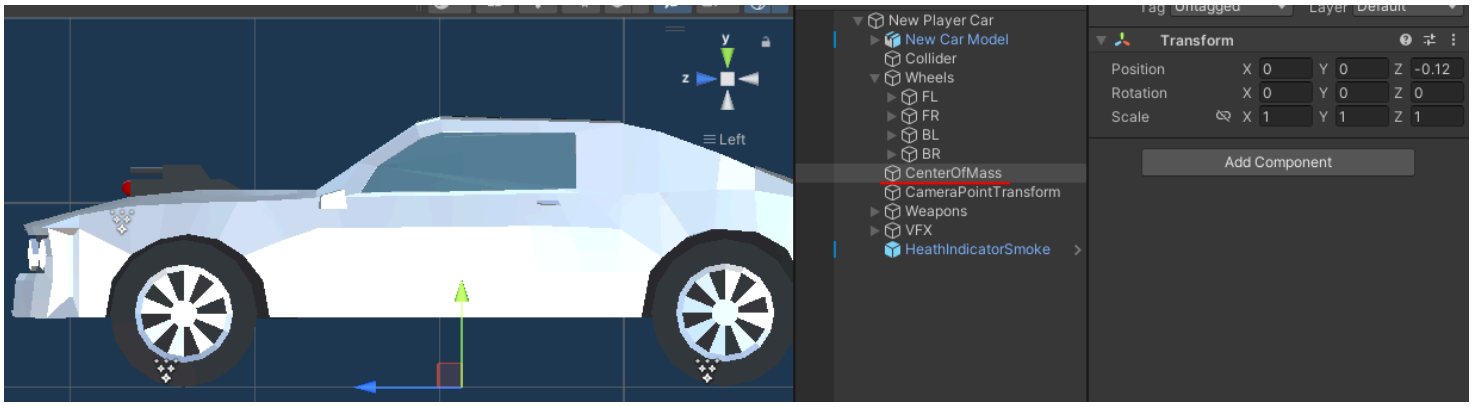
- Remove Old Car Model from prefab.
- Adjust wheel radiuses and positions, to make it fit to the new model.



- Keep the same distance between front and back pare wheels.

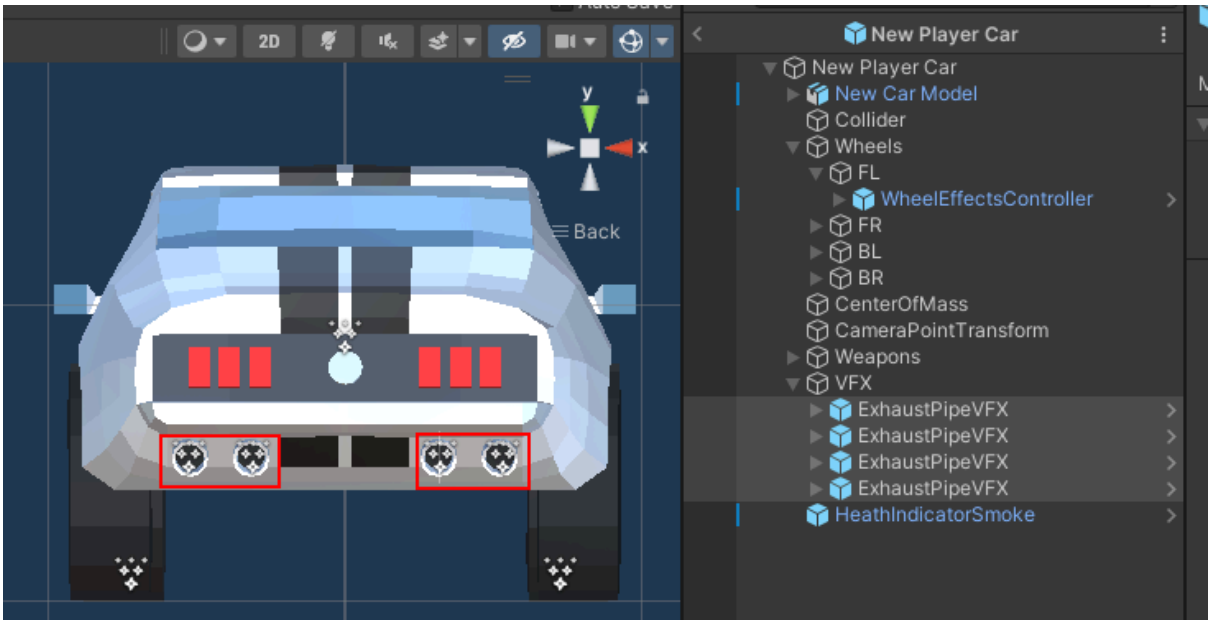


- Place CenterOfMass below the car, at the height where wheels hit the ground.

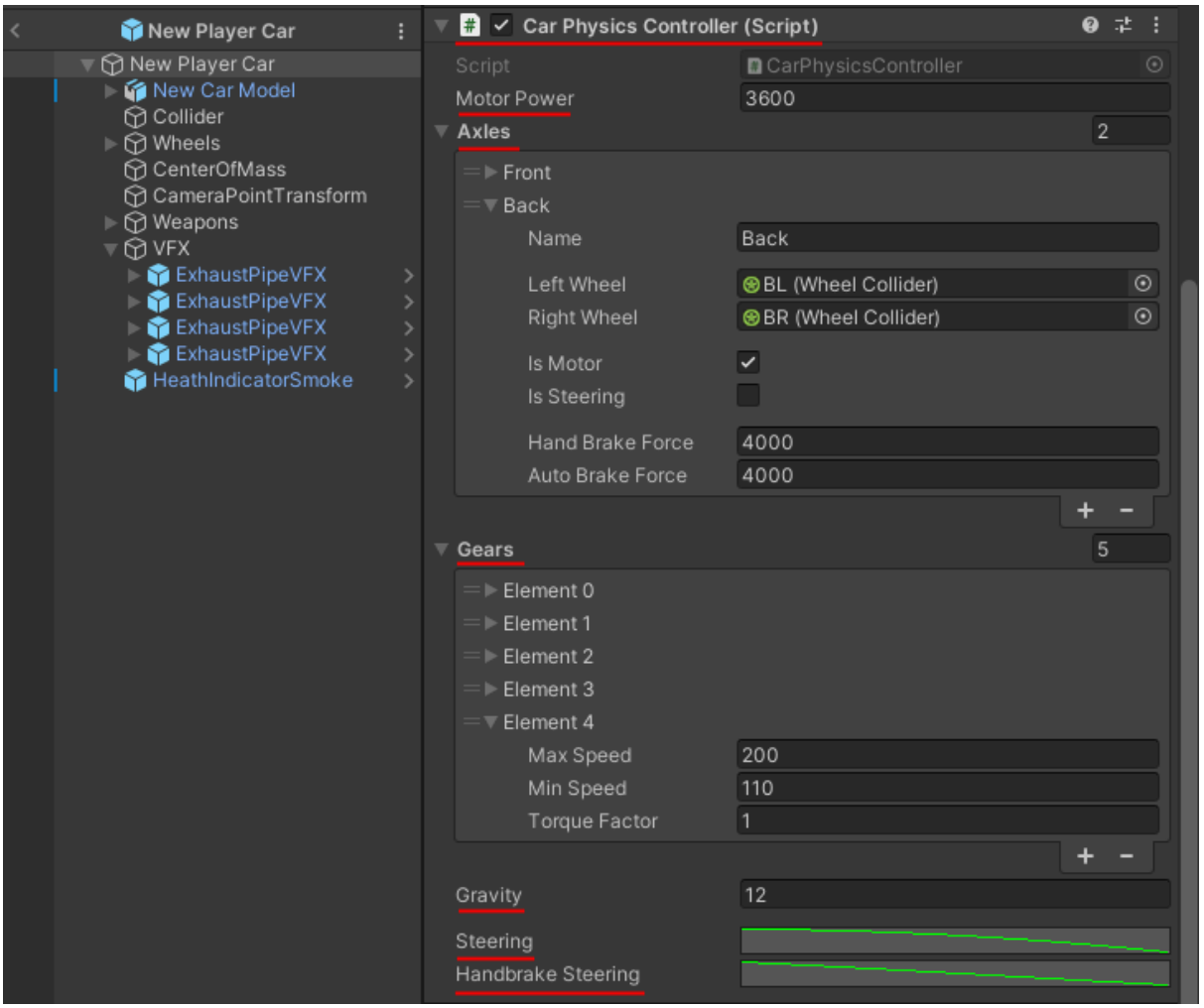


Note: Moving center of mass forward and backwards, changes cars behavior.
On (0, 0, -0.2) car feels heavier, on (0, 0, 0.2) it feels more lightweight.

- Adjust ExhaustPipeVFX positions.

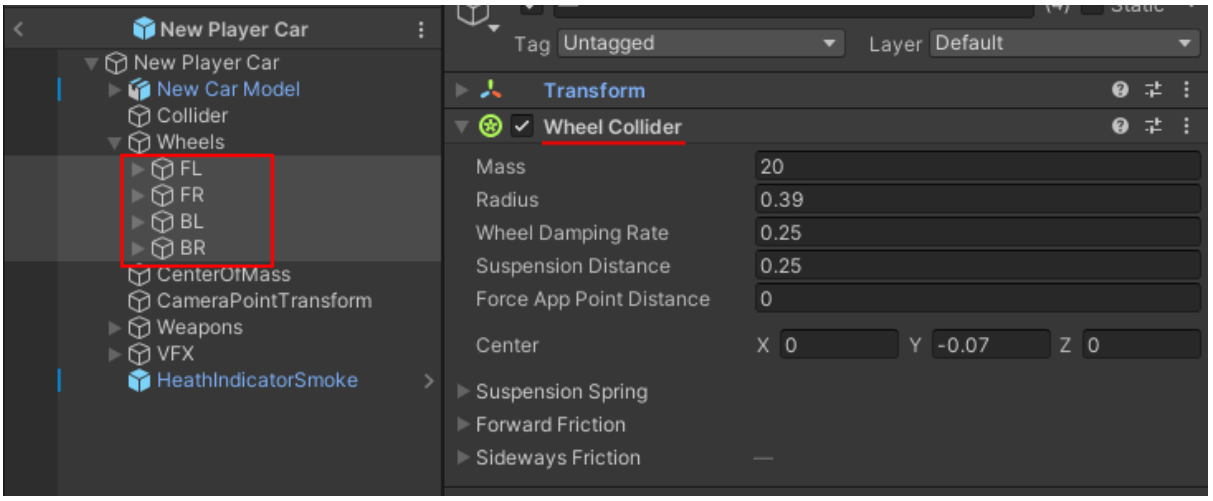


How to change car parameters (speed, brakes, friction):
- Open car prefab.
- Find Car Physics Controller component on its root object.
It controls the car's engine and wheels.



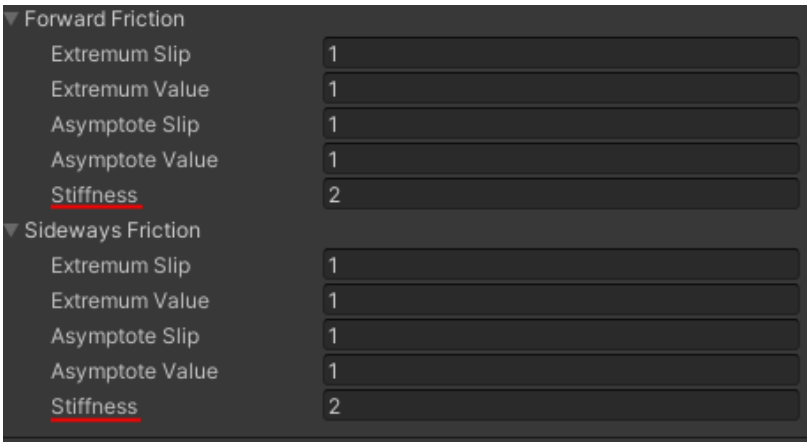
Here we can change next parameters:
Motor Power - Car's motor torque. The higher value, the faster the car.
Axles - Wheel pairs control. Handles wheels toque, brakes and steering. Auto-brakes activate when the car starts to gas in the opposite direction. Helps to stop the car faster.
Gears - Basic gears for further extension. Every gear has a speed range and motor torque multiplier to adjust acceleration. Gears switch automatically, with no delay.
Gravity - Custom gravity for better results.
Steering - Graph to set up common steering behaviour according to car's speed.
Handbrake Steering - Graph to set up additional steering during active handbrake (according to car's speed).

To adjust suspension and wheel friction we have to configure WheelCollider components. Every wheel has its own WheelCollider.



We recommend to check the [Unity Official Wheel Collider Documentation] before adjusting. It contains all the necessary data about Wheel Collider parameters.

However, we recommend to skip wheel friction curves setup and set friction values like on the image below. It's the best parameters to start configuring a car.

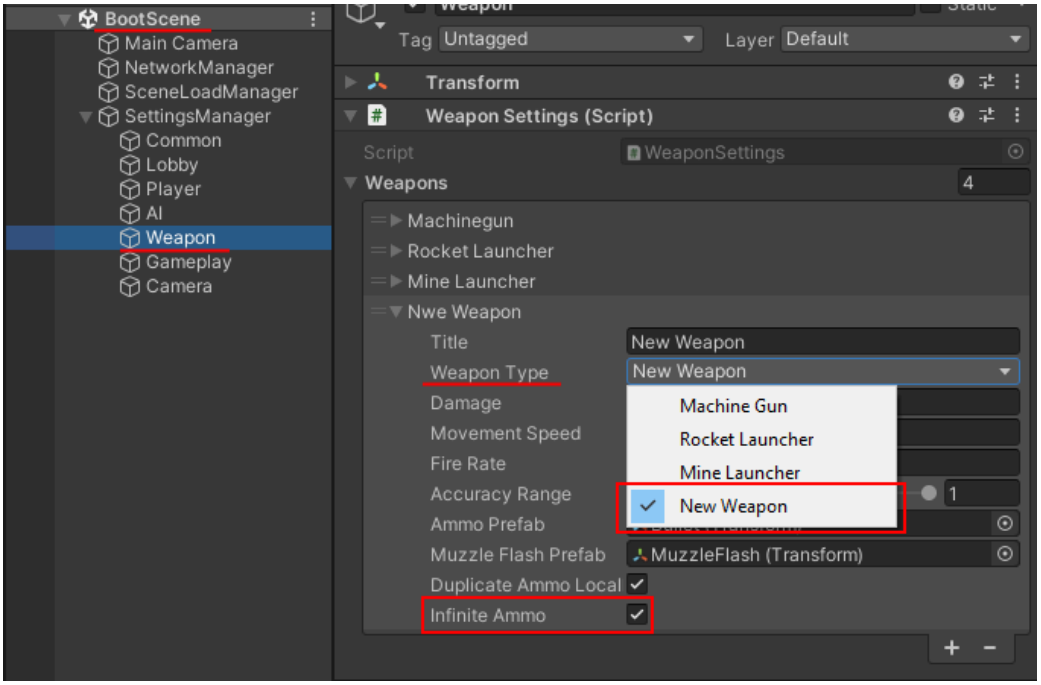


For steering wheels, more stiffness gives more control (faster steering).
For motor wheels, more stiffness gives faster acceleration.
Less stiffness gives more sliding (drifting) for all the wheels.
Note: Center of mass affects wheel stiffness behavior.

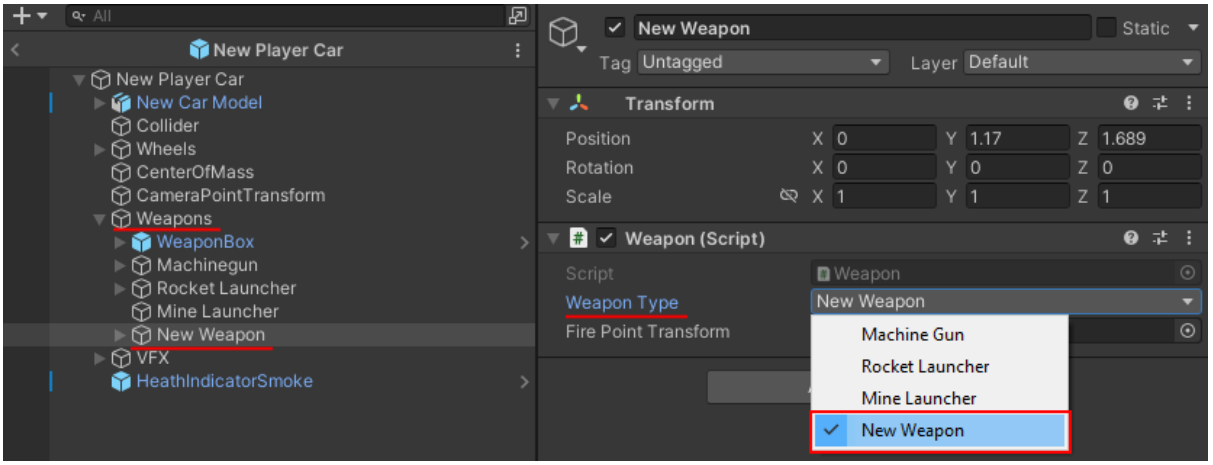
Check car prefabs for more details.

How to add new weapon

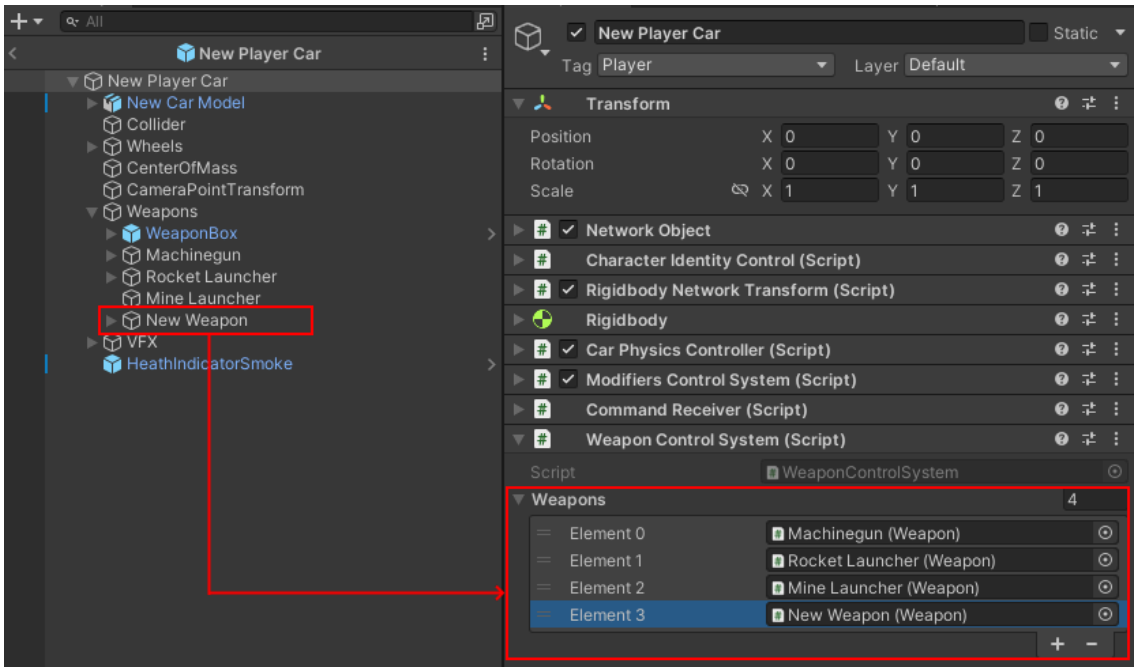
- Open WeaponType.cs enum and add new weapon type.
- Open boot scene.
- Find SettingsManager. Select Weapon child object.
- Press **+** below Weapons collection. It will automatically copy last weapon config and add it to the end of Weapons collection.
- Select new defined weapon type in Weapon Type field drop-down menu
- Set Infinite Ammo = true. This parameter allows us to test weapons without picking up ammo.



- Open car prefab.
- Find weapons.
- Copy one of weapons.
- Set WeaponType to our new weapon type.



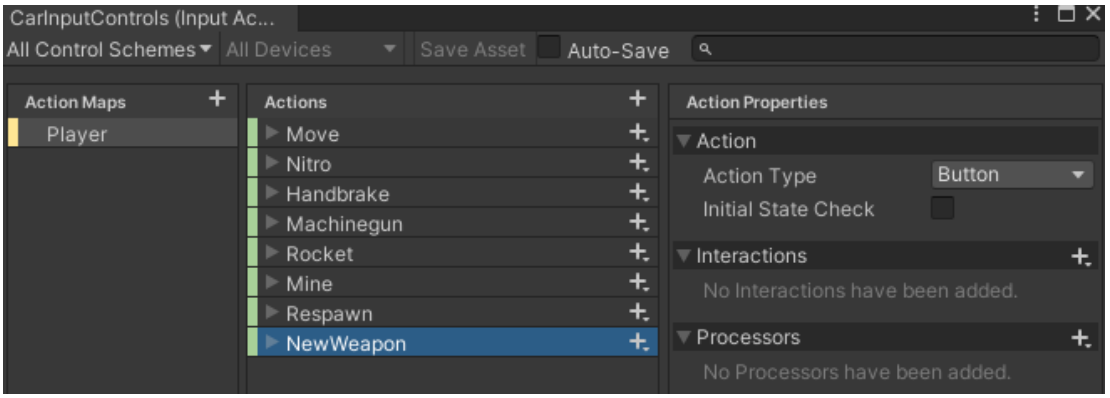
- Drag and drop our new weapon from Weapon Controllers to Weapons collection in WeaponControlSystem component on character root object.



- Add new weapon to PlayerCarBehaviour.

```
//Weapons
machinegunWeapon = weaponControlSystem.GetWeapon(WeaponType.MachineGun);
rocketWeapon = weaponControlSystem.GetWeapon(WeaponType.RocketLauncher);
mineWeapon = weaponControlSystem.GetWeapon(WeaponType.MineLauncher);
newWeapon = weaponControlSystem.GetWeapon(WeaponType.NewWeapon);
```

- Add weapon fire inputs.



- Enable input.

```
//Inputs
inputActions = new CarInputControls();
inputActions.Player.Move.Enable();
inputActions.Player.Handbrake.Enable();
inputActions.Player.Nitro.Enable();
inputActions.Player.Machinegun.Enable();
inputActions.Player.Rocket.Enable();
inputActions.Player.Mine.Enable();
inputActions.Player.Respawn.Enable();

inputActions.Player.NewWeapon.Enable();
```

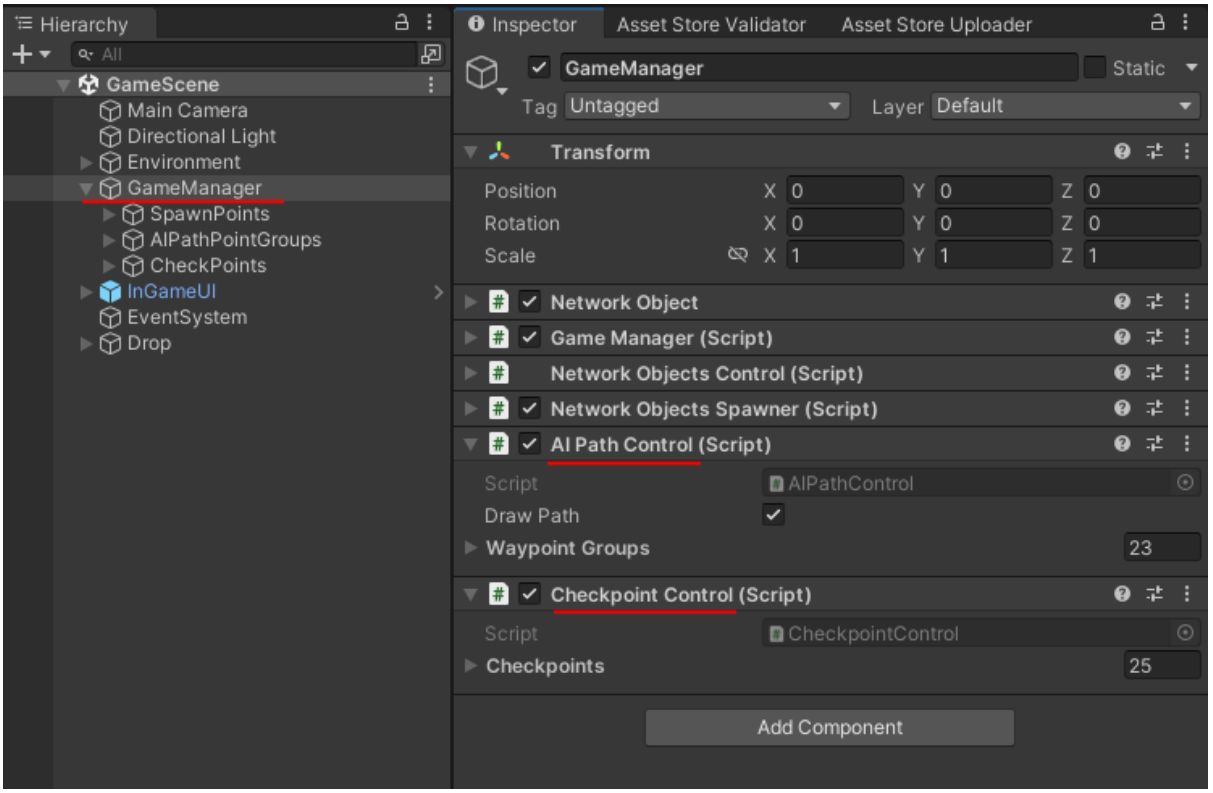
- Add input handling.

```
        //Use machinegun
        if (inputActions.Player.Machinegun.inProgress) machinegunWeapon.Fire();

        //Use rocket launcher
        if (inputActions.Player.Rocket.inProgress) rocketWeapon.Fire();

        //Use mine launcher
        if (inputActions.Player.Mine.inProgress) mineWeapon.Fire();

        //Use new weapon
        if (inputActions.Player.NewWeapon.inProgress) newWeapon.Fire();
```

How to add another game scene
- Duplicate existing GameScene. Change anything in the Environment object.
- Add scene to Build Settings
- Set PlayerDataKeeper.selectedScene value, from code, during runtime.
- Start new game session. All joined clients will automatically load scene after you.

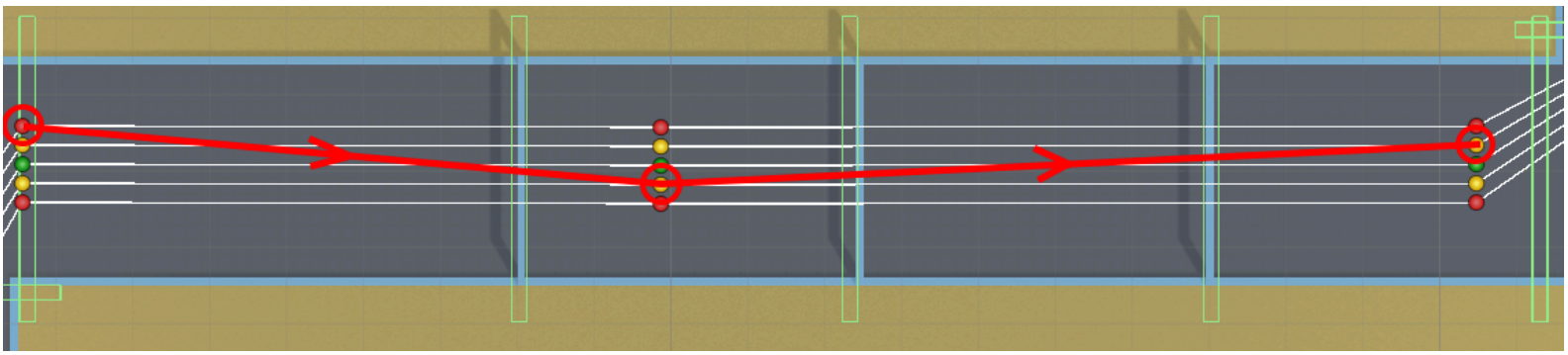
Currently there is only one scene in the project, and value is always “GameScene”.
However, it's ready to expand.

For proper work of the game scene, we have to set up WayPointGroups and Checkpoints.
Checkpoint is a solution to control race progress (place, lap and status) for every car in a race. Every car supposed to hit checkpoints one after another, lap after lap unit finish.
Also, cars respawn on their last checkpoints.
Checkpoints on scene handled by CheckpointControl component on GameManager.



To make bots move in circles, we have to use WayPointGroups.
Bot randomly selects one point from a group and moves to this point.
When bot is close enough, it gets next group in a list and selects new random point to move to. WayPointGroups contain 5 points marked with different colors.
WayPointGroups on a scene are handled by AIPathControl component on GameManager.

Note: If bot turns around after respawn on a checkpoint, add WayPointGroup on this checkpoint. Bot will consider it as passed (cause it’s close enough) and select next group.

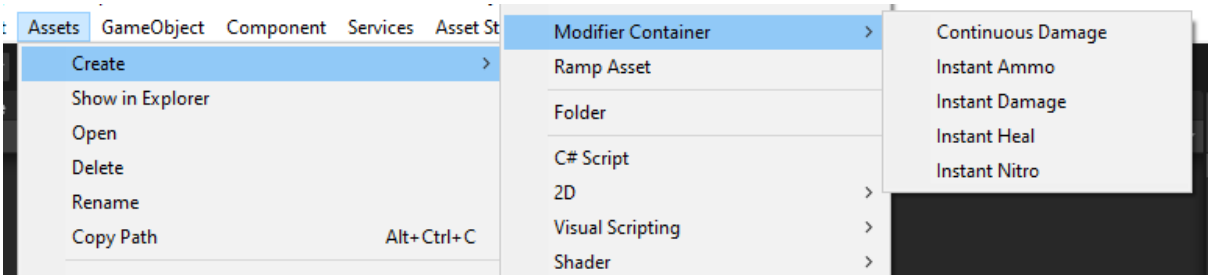


How to add new pick-up item
Pick-ups are made from two parts: Pick-Up Item prefab and Scriptable Object with command.
Project contains Modifiers Control System: solution similar to Command and Visitor design patterns. It used to broadcast and process commands from picking up objects (ammo, nitro, medkit) and receiving damage.
We can use already prepared commands and customize them, or create new (instructions below).

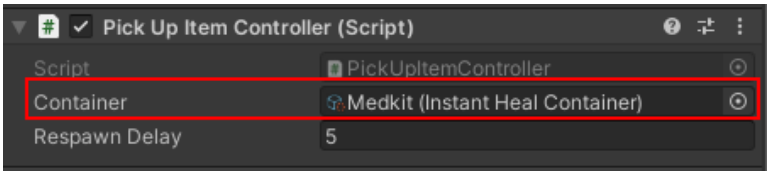
Existing commands	Description
Instant Damage	Damage. Used as bullet damage. Could be mines, etc.
Continuous Damage	Damage (Fire, Poison, etc) with exit time
Instant Heal	Medkit
InstantAmmo	Ammo for selected weapon type
InstantNitro	Refill nitro

Scriptable objects used with pick-ups stored at \ScriptableObjects\Pick-Up Item Configs.
Pick-up prefabs stored at \Prefabs\Pick-Up Items

- Create scriptable object from menu: Assets > Create > Modifier Container



- Copy one of existing prefabs at \Prefabs\Pick-Up Items, and replace its scriptable object (in Container field) with the newly created scriptable object.



- Add prefab to Network Prefabs List. **ScriptableObjects\NetworkPrefabsList.asset**
- To see the pick-up item in game, place it on a scene and save the scene.

How to create new modifiers/commands

Project contains Modifiers Control System: solution similar to Command and Visitor design patterns. It used to broadcast and process commands from picking up objects (ammo, power-ups, medkit) and receiving damage.

To create your own modifiers/commands follow next steps:
- Create new class and inherit it from **InstantModifier** or **ContinuousModifier**.
InstantModifier is a base class for modifiers with no duration (commands).
ContinuousModifier is a base class for modifiers with duration.
- Add **[Serializable]** attribute above class name.
- Define class constructor like on examples below

Instant modifier:

```
[System.Serializable]
1 reference
public class NewCustomModifier : InstantModifier
{
    0 references
    public NewCustomModifier()
    {
        type = GetType().ToString();
    }
}
```

Continuous modifier (contains tag):

```
[System.Serializable]
1 reference
public class NewCustomModifier : ContinuousModifier
{
    0 references
    public NewCustomModifier()
    {
        type = GetType().ToString();
        tag = "ncm";
    }
}
```

All continuous modifiers in project are tagged with first letters of every next word in a name.
Tag for **N**ew**C**ustom**M**odifier could be “**ncm**”.
Tags required to group continuous modifiers with the same effect into one single modifier with longer activity time.

- Override **SerializeModifier** and **DeserializeModifier** methods.

Instant Modifier:

```
[System.Serializable]
1 reference
public class NewCustomModifier : InstantModifier
{
    //Custom Parameters
    public int newIntParameter;
    public float newFloatParameter;
    public string newStringParameter;

    0 references
    public NewCustomModifier()
    {
        type = GetType().ToString();
    }
    2 references
    protected override void SerializeModifier()
    {
        //Pack custom parameters to array
        object[] outputData = new object[]
        {
            newIntParameter,              //Index 0
            newFloatParameter,            //Index 1
            newStringParameter            //Index 2
        };

        //Serialize custom parameters
        //serializedData field contains in the base class
        serializedData = Newtonsoft.Json.JsonConvert.SerializeObject(outputData);
    }
    2 references
    protected override ModifierBase DeserializeModifier(string inputData)
    {
        //Deserialize custom parameters
        object[] data = Newtonsoft.Json.JsonConvert.DeserializeObject<object[]>(inputData);

        //Unpack custom parameters
        newIntParameter = System.Convert.ToInt32(data[0]);      //Index 0
        newFloatParameter = System.Convert.ToSingle(data[1]);   //Index 1
        newStringParameter = data[2].ToString();                //Index 2

        return this;
    }
}
```

SerializeModifier and **DeserializeModifier** methods designed to contain instructions to pack and unpack custom data, stored in fields of certain modifiers.
In this example, **SerializeModifier** takes three custom fields (newIntParameter, newFloatParameter, newStringParameter), packs them to object[] and serializes them into JSON string. There could be any other JSON-compatible variables.
Game will use this JSON to send modifier through the network.
DeserializeModifier required to receive modifier and restore it back from JSON.

Continuous Modifier:

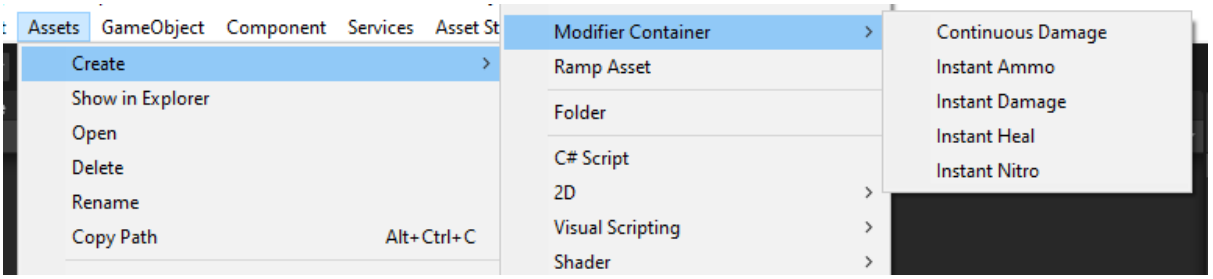
```
[System.Serializable]
1 reference
public class NewCustomModifier : ContinuousModifier
{
    //Custom Parameters
    public float newFloatParameter;

    0 references
    public NewCustomModifier()
    {
        type = GetType().ToString();
        tag = "ncm";
    }
    2 references
    protected override void SerializeModifier()
    {
        //Pack custom parameters to array
        object[] outputData = new object[]
        {
            newFloatParameter,            //Index 0
            duration,                     //Index 1. Contains in ContinuousModifier class
            tag                           //Index 2. Contains in ContinuousModifier class
        };

        //Serialize custom parameters
        //serializedData field contains in the base class
        serializedData = Newtonsoft.Json.JsonConvert.SerializeObject(outputData);
    }
    2 references
    protected override ModifierBase DeserializeModifier(string inputData)
    {
        //Deserialize custom parameters
        object[] data = Newtonsoft.Json.JsonConvert.DeserializeObject<object[]>(inputData);

        //Unpack custom parameters
        newFloatParameter = System.Convert.ToSingle(data[0]);   //Index 0
        duration = System.Convert.ToSingle(data[1]);            //Index 1
        tag = data[2].ToString();                               //Index 2

        return this;
    }
}
```

For more details, check other modifiers stored in **Scripts\Modifiers\Defined Modifiers** folder.

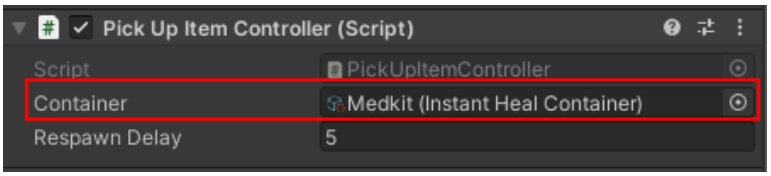
- Create new class **NewCustomModifierContainer** an inherit it from **ModifierContainerBase**
- Add **CreateAssetMenu**
- Override **GetConfig()** method

```
using UnityEngine;

[CreateAssetMenu(fileName = "NewCustomModifierContainer", menuName = "Modifier Container/New Custom Modifier Container")]
Unity Script | 0 references
public class NewCustomModifierContainer : ModifierContainerBase
{
    //Public visible scriptable object parameters
    public int newIntParameter;
    public float newFloatParameter;
    public string newStringParameter;

    2 references
    public override ModifierBase GetConfig() //Upcast to ModifierBase type
    {
        return new NewCustomModifier()       //Create and return modifier of our new type - NewCustomModifier
                                             //Don't confuse with scriptable object type  - NewCustomModifierContainer
        {
            //Take public scriptable object parameters and set them to modifier values
            newIntParameter = this.newIntParameter,
            newFloatParameter = this.newFloatParameter,
            newStringParameter = this.newStringParameter
        };
    }
}
```

- Create scriptable object from menu: Assets > Create > Modifier Container



- Copy one of existing prefabs at \Prefabs\Pick-Up Items, and replace its scriptable object (in Container field) with the newly created scriptable object.



- Add prefab to Network Prefabs List. **ScriptableObjects\NetworkPrefabsList.asset**
- To see the pick-up item in game, place it on a scene and save the scene.
- Add **NewCustomModifier's** handling method to ModifiersControlSystem.

```
0 references
public void NewCustomModifierHandlingMethod()
{
    double serverTime = NetworkManager.Singleton.ServerTime.Time;

    for (int i = 0; i < activeModifiers.Count; i++)
    {
        ActiveModifierData container = activeModifiers[i];
        ModifierBase modifier = activeModifiers[i].modifier;

        //Skip if it's too early
        if (container.startTime > serverTime) continue;

        //Check command by type
        if (modifier is NewCustomModifier)
        {
            //Custom logic here

            //Mark as expired (for instant modifiers only)
            container.PrepareToRemove();
        }
    }
}
```

This method has to process modifier's logic. Also, it's supposed to be called outside ModifiersControlSystem, from the component where it's required.

How it works: For every modifier type (or few) we have special handling methods stored in ModifiersControlSystem.
When we need to check something related to this modifier (status, value, etc), we call this method. Usually, it's called every frame.
Inside the method, we check if we have required modifier at all, if we do - we run custom logic and return the result. If we don't - return default value (or nothing).

Character can get modifiers from bullet(rocket, mine) hit or from picking up elements.

ModifiersControlSystem receives every modifier from CommandReceiver component.
Check ModifiersControlSystem and CommandReceiver for more details.

Modifiers could:
- take arguments and return updated data, like **HandleHealthModifiers** method.
- return multipliers
- do nothing and work as flag: ContainsModifier<**NewCustomModifier**>()
- work with callback delegates
- do any other custom logic
So, long story short: modifier is a command object. :)

Note: ModifiersControlSystem is a component. Every character has its own ModifiersControlSystem with its own active modifiers.
However it could be used anywhere else, outside character logic.

Lobby

Class name	Description
LobbyManager	Singleton. Contains methods to create and join lobby, including private rooms with password. Contains methods to host new game, when lobby is full and ready to start game session.
LobbyDataControl	Part of LobbyManager. Private object. Contains all the necessary properties, events and methods required by lobby to work with data (players, IDs, etc).
LobbyGameHostingControl	Part of LobbyManager. Private object. Contains methods to host or join hosted game using Relay service. Receives join/host commands from LobbyActivityControl. Also contains available regions and its updates.
LobbyActivityControl	Part of LobbyManager. Private object. Contains logic to update lobby timeouts. Required by lobby services, to know, lobby is still alive. Contains logic to check player statuses, before game start. Requires to make sure no one was disconnected (in unhandled way) while waiting, otherwise lobby updates its status to waiting for players.

Game

Class name	Description
GameManager	Singleton. Contains methods and events to control game status. Also contains list of user scores.
NetworkObjectsControl	Part of GameManager. Public object. Stores lists of all characters on scene: players, bots, service objects. Used to have easy access to any type of characters.
NetworkObjectsSpawner	Part of GameManager. Public object. Used to spawn/respawn characters and bots.
AIPathControl	Contains and manages WayPointGroups.
CheckpointControl	Contains and manages Checkpoints.

Character

Class name	Description
CharacterIdentityControl	Contains identity markers: isPlayer, isBot. Overrides: IsLocalPlayer, IsOwner, OwnerClientId form NetworkBehaviour. Also contains custom character spawn parameters.
RigidbodyNetworkTransform	Network car physics emulation. Allows rigidbody to be non-kinematic and interact with other rigidbodies during network transform synchronization. Contains interpolation and extrapolation (prediction).
CarPhysicsController	Controls all the car physics: acceleration, brakes, steering, gravity etc. Even nitro.
ModifiersControlSystem	Contains active power ups and commands, waiting for processing. More detailed information is provided in the source code.
CommandReceiver	Receives modifiers/commands.
WeaponControlSystem	Main weapon control class. Contains all the weapons and controls to manage them. Every weapon has a Weapon component on it. Weapons spawn bullet ammo objects with Bullet or Mine components on them. More detailed information is provided in the source code.
HealthController	Controls current vitality status. Also runs death event.
NitroController	Controls current nitro amount and its activity status.
PlayerCarBehaviour	Player car inputs handling: movement, steering, nitro, weapons, etc.
AICarBehaviour	Bot car logic: movement, navigation, weapon handling, etc.
RaceStatusController	Controls current place, lap and race status.
RespawnController	Respawns car at the last checkpoint. Reills car’s heath and nitro if car has been destroyed.
CharacterUIController	Controls HUD if it's a local player or status bar if it's bot or other player.
CarColorController	Car’s start color setup.
HealthVFXIndicationController	Controls black smoke intensity according to car's health.
ExhaustPipeVFXController	Controls exhaust pipes smoke and nitro effects.
WheelsVFXController	Manages three main wheel VFX components: WheelMeshController - controls wheel rotation TireSkidMarksController - controls skid marks TireSmokeController - controls smoke from wheels

Other

Class name	Description
ServiceUserController	Session user object. For service usage only. Spawns as separated game object.
PickUpItemController	Handles picking up and ammo refills. Has restore time.
PlayerDataKeeper	Static class. Loads/Saves local data. Works with PlayerPrefs.
SceneLoadManager	Singleton. Works with loading target scenes through LoadingScene. Contains two ways of loading scenes: regular and network oriented. More detailed information is provided in the source code.
GameCameraController	Game camera, targeted on the local player.