PolyStang - Car Controller

Version 1.1

by Faber Brizio

April 5th, 2024

SUMMARY

INTRODUCTION.....3
BASIC SETTING UP.....4
MOBILE SETTING UP.....10
POLYSTANG INSPECTOR.....12
MATERIALS AND SUBMESHES.....19
POST PROCESSING.....21
SOUND DESIGN AND MIXER.....22
SCRIPTS.....23

INTRODUCTION

Welcome to PolyStang - Car Controller 1.0!

This is a car controller containing several models, sounds, effects for a racing game or any kind of game with cars. There are:

- a car model and 3 wheel-models.
- 4 street modules and various props (streetlights, a cone, a cargo container).
- built-in materials of all the models.
- engine and skid sounds, with an audio mixer.
- a demo scene with post-processing, an example scene for customization and a scene setup with mobile controls.
- 4 scripts (CarController, CameraSwitcher, CarLights, CarSounds).

All these elements are organized in folders, as in fig. 1.

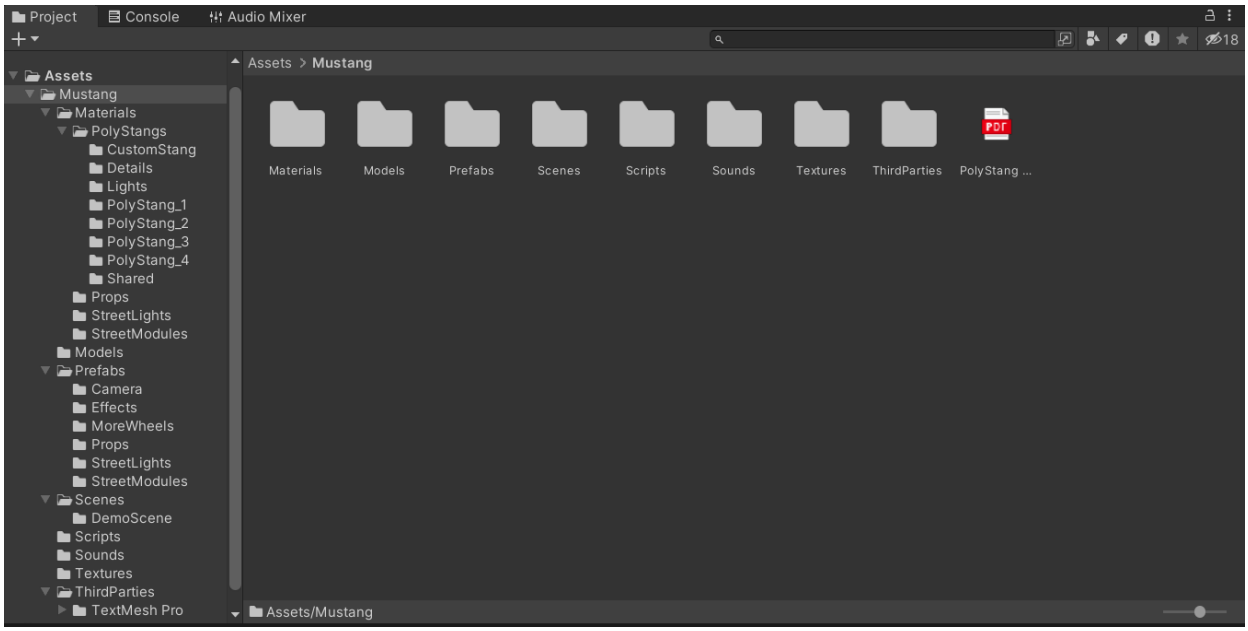

Fig. 1: Organization of the folders

BASIC SETTING UP

To add a PolyStang to a pre-existing scene, follow these 5 steps:

1. Add a PolyStang prefab to your scene, by dragging one of the prefabs into the Scene or Hierarchy (fig. 2).

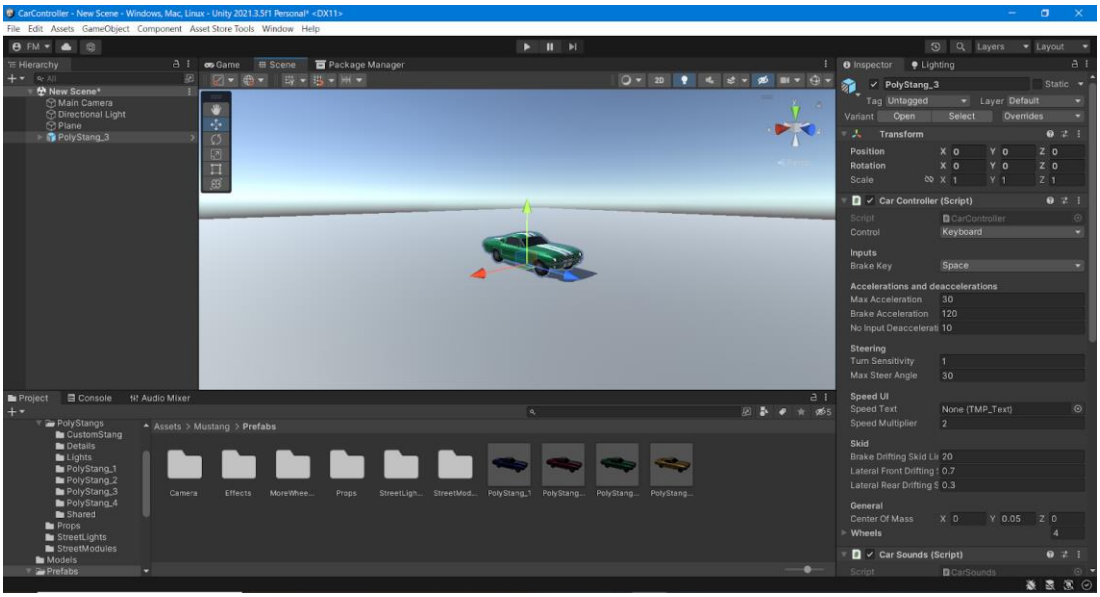

Fig. 2: PolyStang_3 prefab added to the scene.

2. Add the camera prefab (use the “CameraMobile” for mobile games, otherwise the “CameraPC”), the front virtual camera and the rear virtual camera (which require the cinemachine package) and delete any other default camera, as shown in fig. 3.

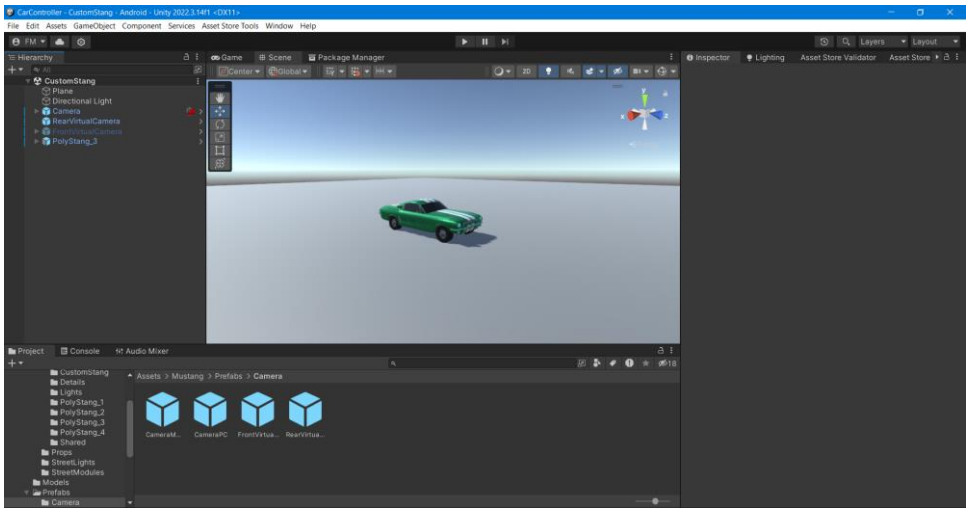

Fig. 3: camera and virtual camera prefabs added to the scene.

3. Set the “Speed Text” in the inspector, so drag “SpeedText” (a TextMeshPro text) as child of the camera prefab into the required field of the Car Controller script attached to the Polystang prefab (fig. 4). The “SpeedText” element is the UI text showing the car current speed.

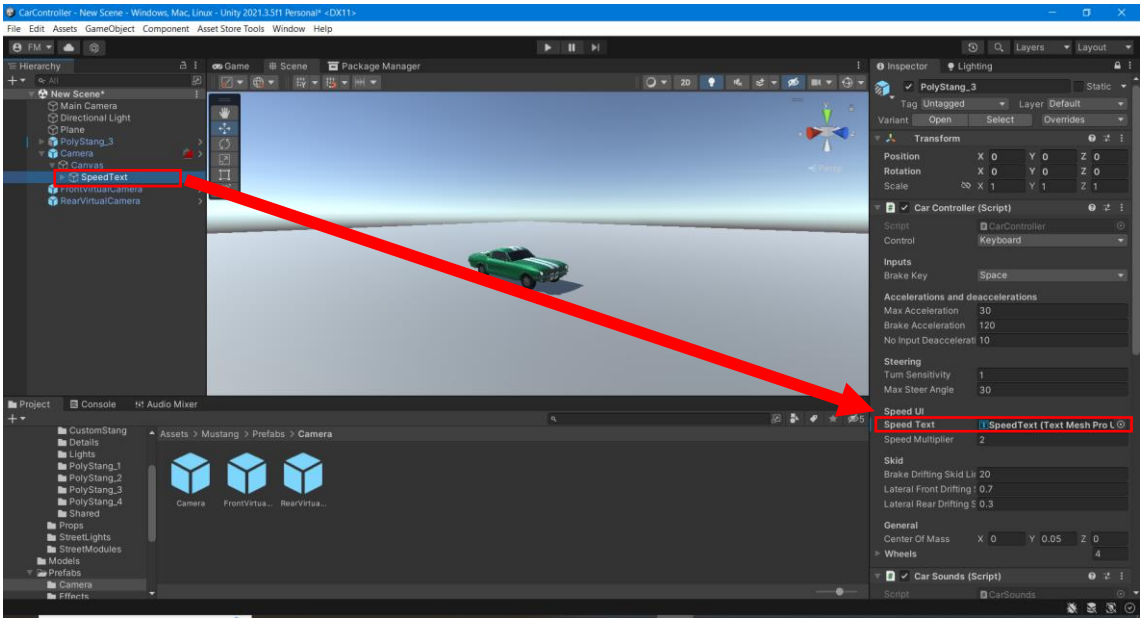

Fig. 4: SpeedText setup.

4. Drag the Front and Rear Virtual Camera into the required fields of the Camera Switcher script (fig. 5). Pay attention to these virtual cameras in the proper fields!

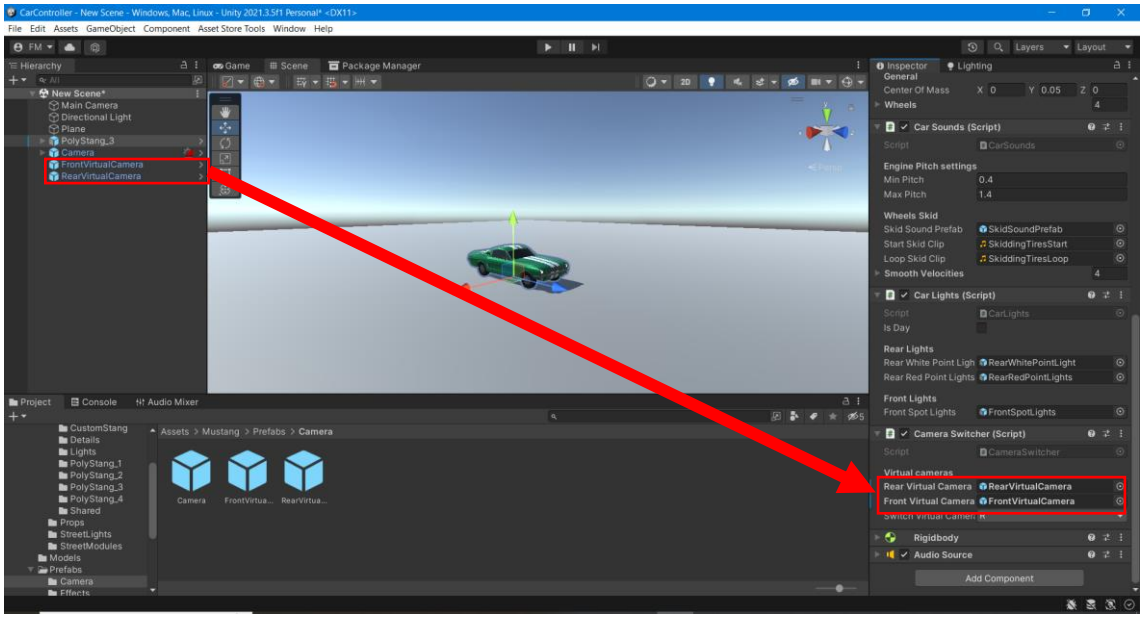

Fig. 5: Virtual cameras setup.

5. Set the “follow” and “look at” settings of the Cinemachine Virtual Camera attached to the Front Virtual Camera prefab (Fig. 6) and to the Rear Virtual Camera prefab (Fig. 7).

   The “CameraFrontPos” transform (child of the PolyStang prefab) must be dragged onto the “follow” field of the Front Virtual Camera prefab, while the “CameraFrontAim” must be dragged onto the “look at” field of the Front Virtual Camera prefab.

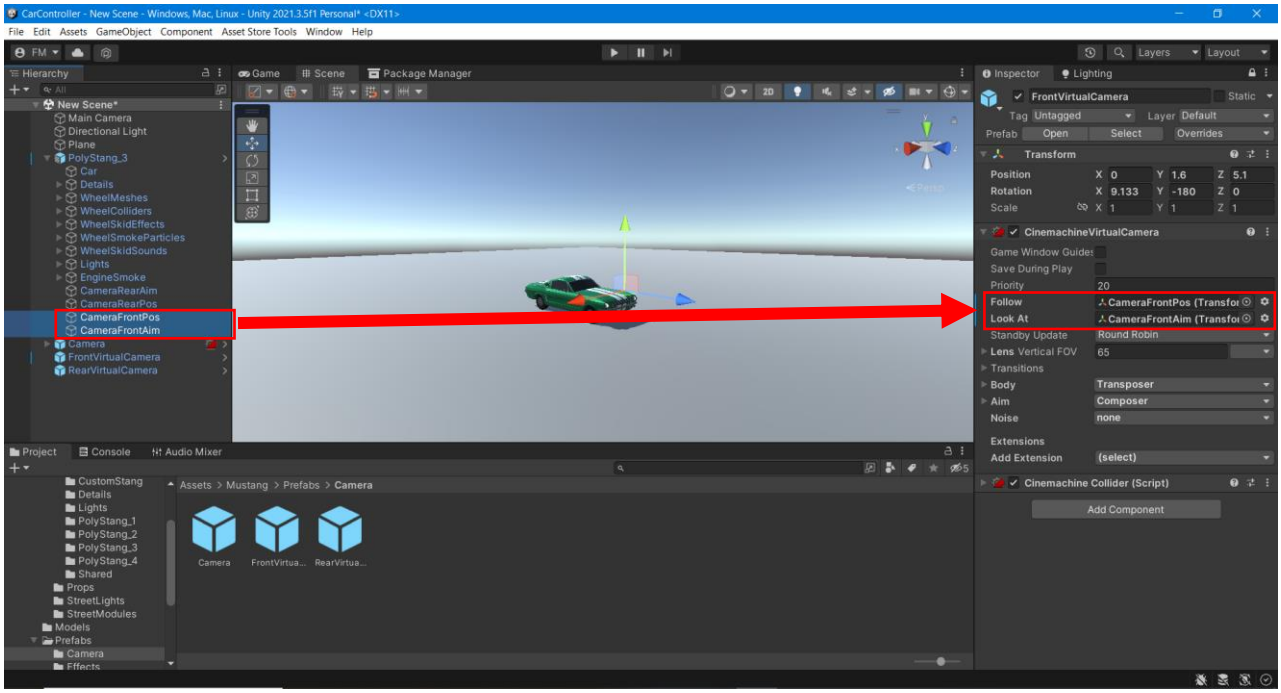

Fig. 6: Front Virtual Camera prefab setup.

The “CameraRearPos” transform (child of the PolyStang prefab) must be dragged onto the “follow” field of the Rear Virtual Camera prefab, while the “CameraRearAim” must be dragged onto the “look at” field of the Rear Virtual Camera prefab.

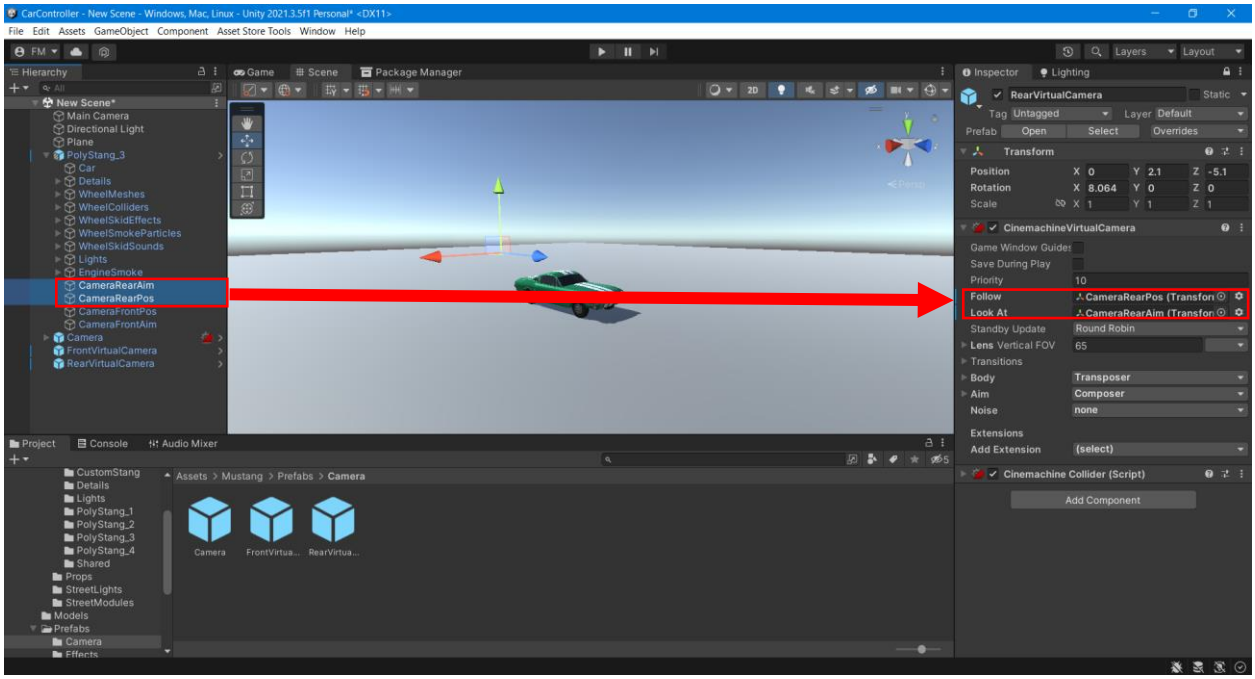

Fig. 7: Rear Virtual Camera prefab setup.

Additional optional steps:

- Before starting the game, be sure to delete any other camera in the scene (for example, as in the previous figures 2-7 is shown, there is a default camera object in the scene called “Main Camera”, which should be deleted).
- You can set the front camera as false in the hierarchy. This is done automatically at the start of the game, but it is recommended to have the camera prefab positioned as default.
- You might not be using the post-process layer used in the demo scene, so you can easily delete it from the camera prefab (Fig. 8).

Fig. 8: deleting the Post-process Layer from the camera prefab.

- If you want to use the post-process layer in the demo scene in your scene, you need to create an empty object in your scene (and, for example, call it “PostProcessVolume”).

Add a post-process volume component to it and check the “is Global” field, drag the “PP_Demo” post-process profile (that you find in the “Scenes” folder) onto the “Profile” field of the Post-process Volume component (Fig. 9).

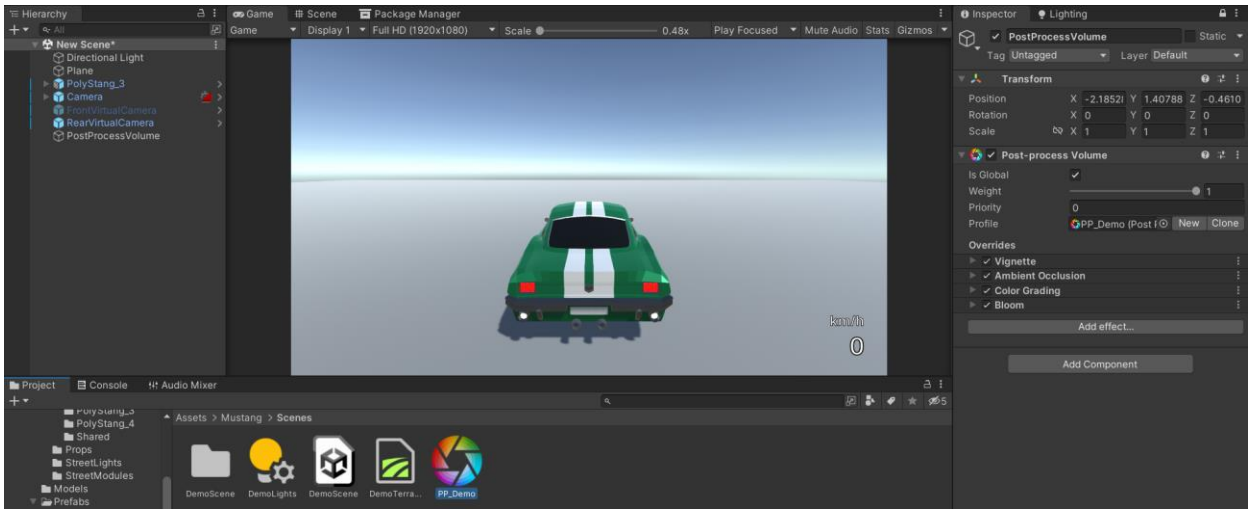

Fig. 9: Post-process Volume setup.

Don’t forget to set the Layer of the “PostProcessVolume” object to “PostProcessing” (Fig. 10). If you don't have a layer with such a name, add one and call it like that. Remember this field is key-sensitive, so the spelling of the layer is relevant.

Fig. 10: PostProcessing layer setup.

Also check out that the post-process layer in the camera has the field “Layer” set to the post-processing layer that you have used for the post-process volume element (Fig. 11).

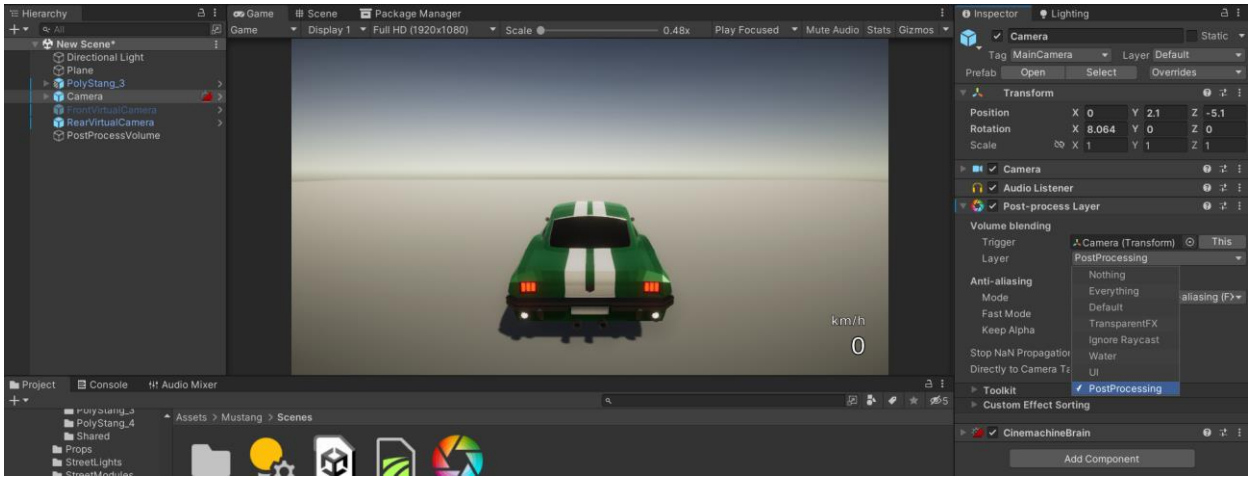

Fig. 11: Post-process Layer check.

MOBILE SETTING UP

In this section all the basic mobile setup is shown. The steps to follow to set up the PolyStang Car Controller are the same as shown for the basic setting up, but keep in mind to:

- Use the “CameraMobile” prefab for the camera instead of the “CameraPC”. The “CameraMobile” camera already has mobile buttons set up properly.
- Pay attention to configure the “Control” field in the CarController script as “Buttons”, instead of “Keyboard” (Fig. 12). This is crucial for the script to work. Otherwise, the inputs will just work for the keyboard.

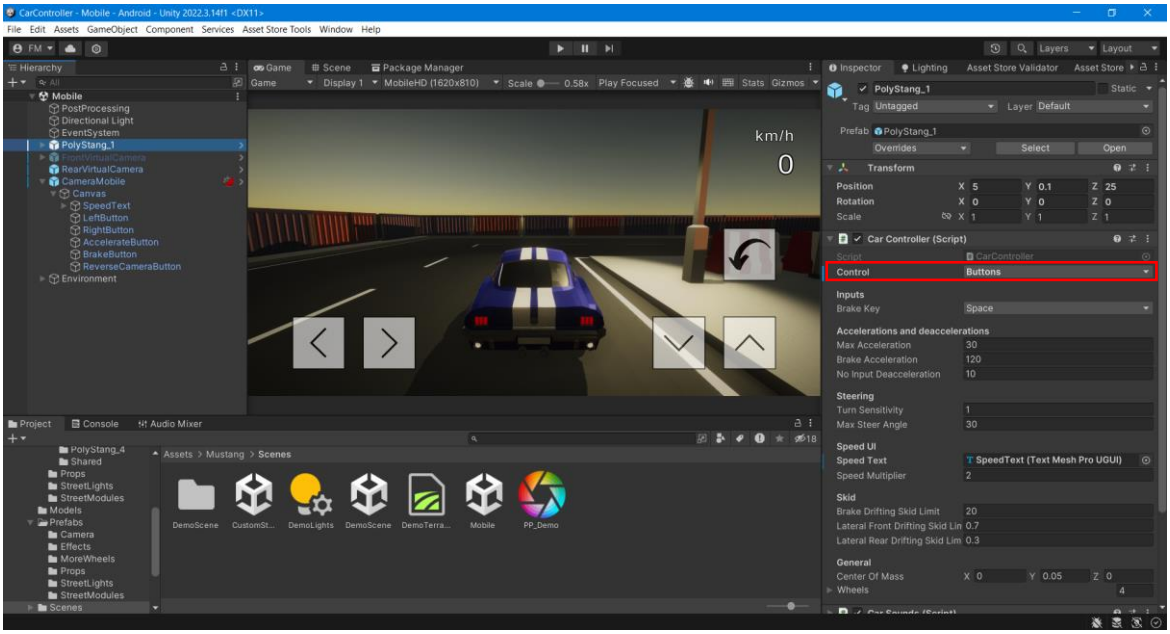

Fig. 12: Configuring the CarController script.

The controller is ready to be used now. It can be useful to know how the buttons work:

- The “ReverseCameraButton”, is responsible for changing the camera view from front to rear. It simply calls the public function “SwitchCamera()” responsible for the camera switching (find more details about the scripting in the Scripts Section).
- The “RightButton”, “LeftButton”, “AccelerateButton” and the “BrakeButton” change the values “moveInput” and “steerInput” used in the CarController script, respectfully for accelerating forwards and backwards, and steering the front wheels right and left (find more details about the scripting in the Scripts Section). The values are changed using “Event Triggers” attached to the buttons.

POLYSTANG INSPECTOR

In this section all the relevant fields of the scripts that can be changed from the inspector are illustrated.

Car Controller

The first script to be shown is the Car Controller (Fig. 13):

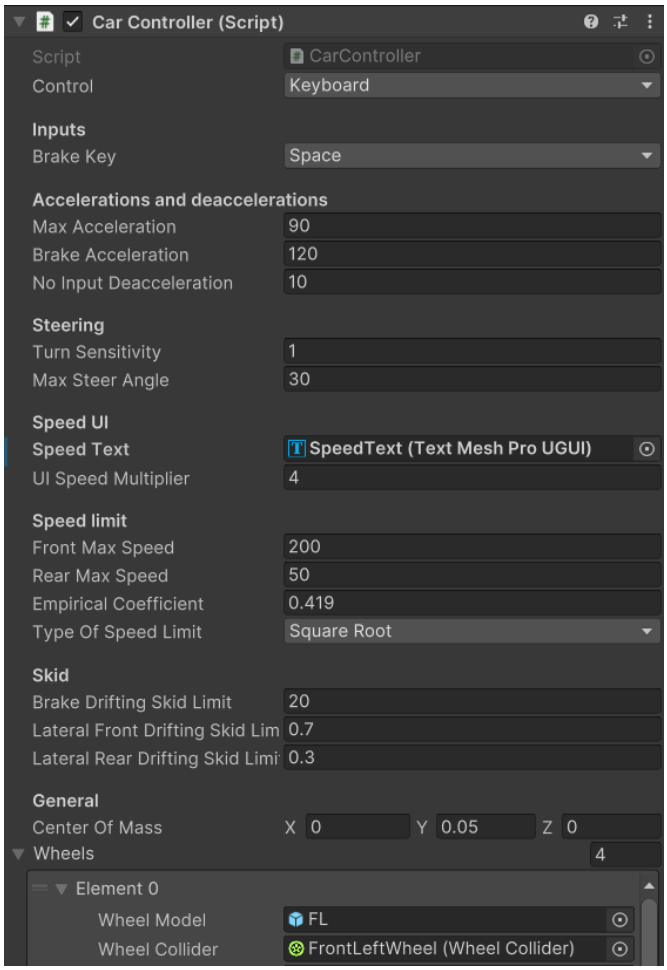

Fig. 13: Car Controller fields in the Inspector.

The first field is “Control” and is thought to make easier the process to use mobile controls. If you need to use this package for mobile, change this field to “Buttons”. You can call the main functions for steering, acceleration and breaking as public classes from UI buttons.

If you are using the PC inputs, the brake key field lets you choose which key is the one the player can use for the break.

In the acceleration and deceleration section you can customize the following parameters:

- Max acceleration: this is the acceleration in the forward direction. To go faster, you can increase it.
- Brake acceleration: this value regulates the strength of the brake. Increase it to stop the car quicker.
- No input deceleration: if no key is pressed, the speed is decreased using this value. It is to be considered as a negative acceleration. This value is used to simulate the engine braking power.

In the steering section, you can change the following parameters:

- Turn sensitivity: this value is used to clamp the rotation of the front wheels, when steering left or right. Decrease this value to rotate the front wheels slower, or vice versa to rotate them faster.
- Max steer angle: it is the maximum angle of steering (i.e. the rotation of the front wheels when steering).

In the speed UI section, you can change the following parameters:

- Speed text: it is the TextMeshPro_Text UI object displaying the velocity of the car in km/h.
- Speed multiplier: to scale the velocity displayed in the UI you can adjust its value with this parameter. You're not actually changing the speed of the car, but the number reported as the speed of the car.

The speed limit section is part of the new update 1.1. The first two parameters, Front Max Speed and Rear Max Speed, respectively control the max speed that the car can reach moving forwards and backwards. The third parameter, named "Empiric Coefficient", is used by this script for transforming the wheel rotational speed to the actual speed of the car. If you change the following parameters, you do *not* need to change its value:

- Parameters of the acceleration and deceleration section.
- Steering parameters.
- Front Max Speed or Rear Max Speed.
- Parameters in the Skid section.

- Wheel size (in particular, the radius which is used to estimate the wheel rotational speed).

If you change the UI Speed Multiplier, you should change the Empiric Coefficient proportionally (for example if you double the value of the first one, you must double the value of the second one too). This is something you could do when you scale down or up the car (so the world feels smaller or bigger). The Empiric Coefficient could be change also if you change the option “Type Of Speed Limit”.

The Type Of Speed Limit has three possible options:

- No Speed Limit: in this case no speed limit is imposed, and the car controller works like in the previous version 1.0 (even though a few more computations are done).
- Simple: the motor power is decreased linearly as the car speed increases, getting to around 0 when the car is reaching the speed limit.
- Square Root: the motor power is decreased parabolically as the car speed increases, getting to around 0 when the car is reaching the speed limit.

If you are using the Simple option, you can adjust the Empiric Coefficient to a slightly higher value than the one you are using with the Square Root option (for example, 0.420 works perfectly for the Polystang). If you are not sure how to change the Empiric Coefficient, change the Front and Rear Speed Limits instead.

In the Skid section (*) you can adjust the skid sensitivity: increase all those values to produce skid marks at higher limits of drifting or braking. The “Brake Drifting Skid Limit” is used only when braking, so it effects the skid marks produced by the rear wheels (which brake). The ratio between the other two parameters (which represent the limit before skid marks are produces due to lateral drifting) is suggested to be 7:3 (default setting) or 9:4.

The centre of mass is used to make the car stable. Set the y value to lower values to make the car more stable.

In the Wheels section you can set up wheels. Add or remove wheels according to the number of actual wheels of your vehicle. Every wheel must have all its field setup to work:

- The wheel model: this model is animated according to the physics of the wheel.
- The wheel collider: this field is the most important one. All the physics are calculated from its variables.

- The wheel effect obj: this is the object corresponding to the skid marks. The child of this object is the skid mark prefab that you can find in Mustang > Prefab > Effects > SkidMark. This object has a trail renderer attached, which is activated or deactivated accordingly to the parameter in the “Skid section” (*) previously mentioned.
- The smoke particle: this is the object corresponding to the particle systems. The child of this object is actually the skid smoke particle system prefab that you can find in Mustang > Prefab > Effects > SkidSmokeParticle. This object has a particle system attached, which is activated or deactivated accordingly to the parameter in the “Skid section” (*) previously mentioned.
- The axel: this field is required to differentiate between front and rear wheels (it is important to properly simulate the steering and skid marks).
- The skid sound: this is the object corresponding to the sounds produced when skid marks are left. The child of this object is the audio source prefab that you can find in Mustang > Prefab > Effects > SkidSoundPrefab. This object has an audio source attached, which is activated or deactivated accordingly to the parameter in the “Skid section” (*) previously mentioned.
- The index: every wheel must be indexed from 0 to higher accordingly to the number of wheels of the vehicle.

Car Sounds

Let’s have a look at how the Car Sounds script work in the Inspector (Fig. 14):

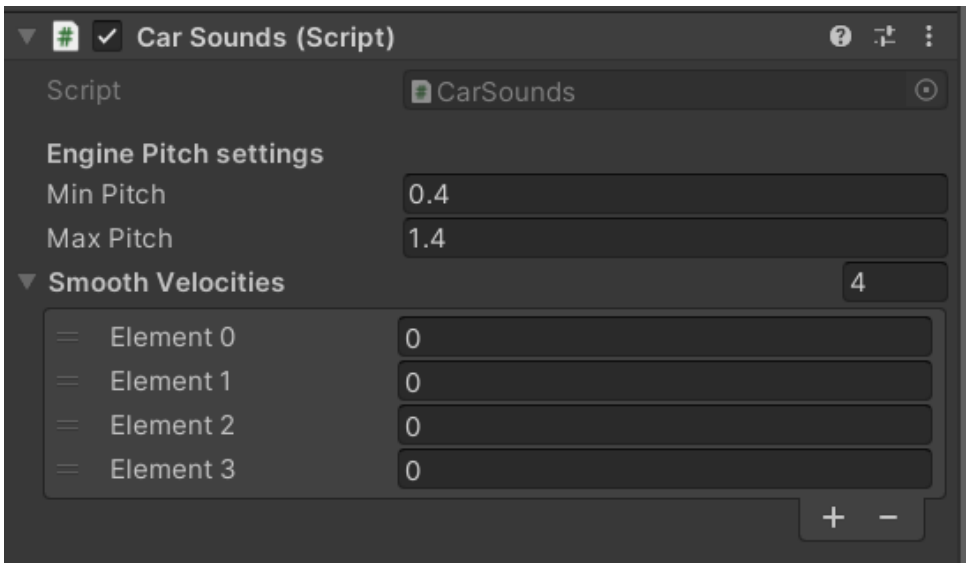

Fig. 14: Car Sounds fields in the Inspector.

The pitch settings are the main parameters meant to be changed for sound designing. The “Smooth Velocities” are just variables used in the script for the function “SmoothClamp()” used in this script: just get sure to have as many smooth velocities as the number of wheels of your vehicle. Otherwise, the script should break, and no sound would be played.

To simulate the noise of the engine when the car is moving, the pitch of the engine sound (which is played on the “Awake()” function) is changed from a minimum value to a maximum value. So, the frequency of the audio clip increases, when the car’s velocity is higher, and it decreases, when the car slows down.

As far as the pitch settings are concerned, the “Min Pitch” represents the pitch of the audio source of the engine when the car is not moving, while the “Max Pitch” is the maximum pitch that can be reached. Over that value, even if the car is moving faster, the pitch remains the same.

Car Lights

Let’s have a look at how the Car Lights script work in the Inspector (Fig. 15):

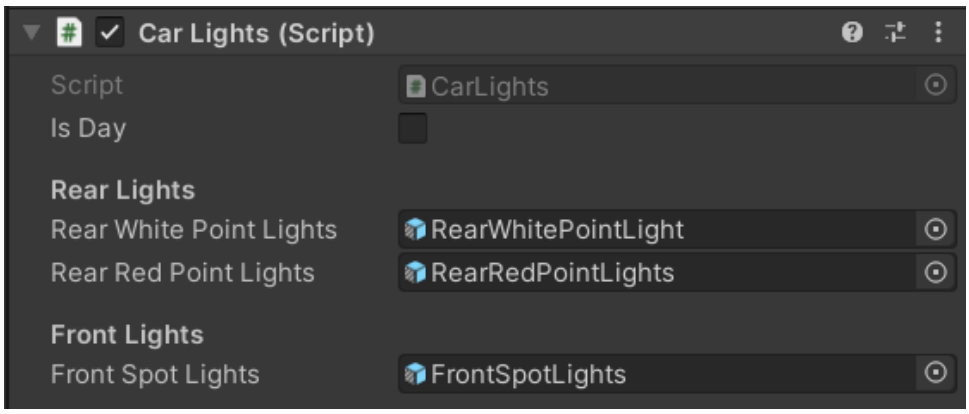

Fig. 15: Car Lights fields in the Inspector.

The only parameter meant to be customized in this script is the first bool “Is Day”. If it’s set to be true, at the beginning of the game, the front spotlights are turned off, otherwise they are turned on (like it is meant to be at night or evening).

As far as the other fields are concerned, they are the red lights on the rear of the car used when the brake is pressed (i.e. the brake key in the Car Controller script), and the white lights on the rear of the car when the reverse gear is used (which is the “S” button or down arrow button with PC controls).

Camera Switcher

Let’s have a look at how the Camera Switcher script work in the Inspector (Fig. 16):

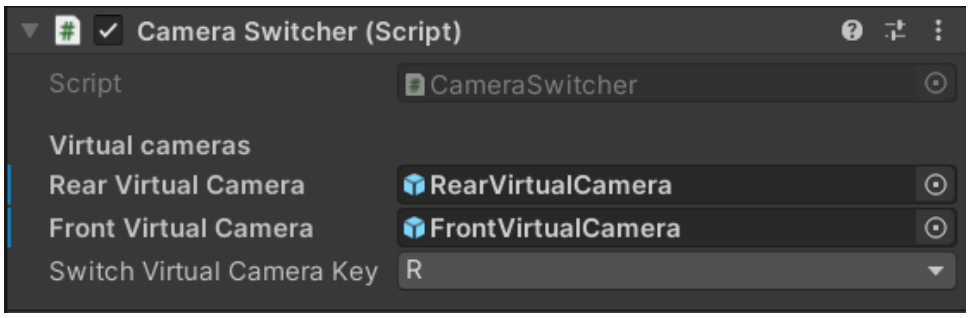

Fig. 16: Camera Switcher fields in the Inspector.

The camera switcher is responsible for switching the view of the camera from the front to the rear of the car, and vice versa. To do so, two different Virtual cameras are used and set as shown in the “Basic Setting Up” section. The only difference between the front and the rear virtual cameras is that the front one has a priority of 20, while the rear one of 10. Which means that the camera prefab will consider the settings of the front virtual camera when it’s active in the Hierarchy, otherwise it will follow the setting of the rear virtual camera.

The two virtual cameras fields must be set (as shown in the “Basic Setting Up” section) otherwise the script will break, and the cameras will not work correctly.

The key to switch cameras can be changed from the Inspector, and that key is exposed as “Switch Virtual Camera Key” (it’s set as R as default).

MATERIALS AND SUBMESHES

All the meshes shared in this asset are subdivided into sub-meshes. This means that all those different parts of the mesh can have a different material applied. Simply, one model, but multiple materials. An example of sub-meshes is shown in Fig. 17:

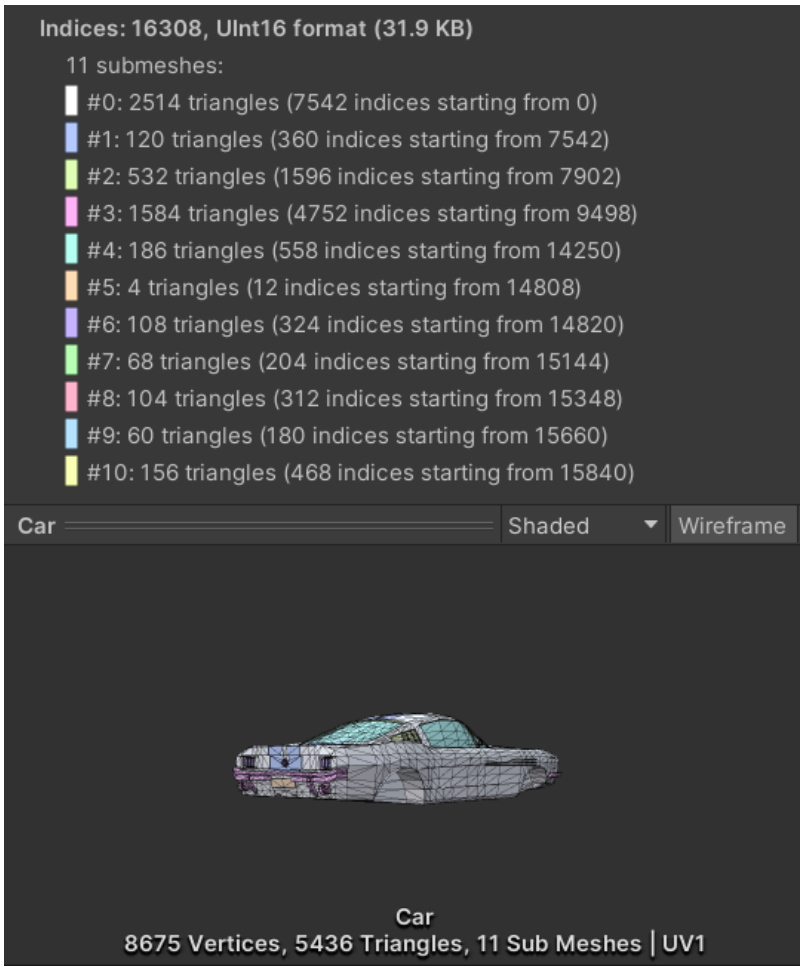

Fig. 17: Sub-meshes of the PolyStang.

An example of how you can customize your PolyStang is shown down below (Fig. 18) using multiple custom materials. It can be interesting to also change the colour of the lights to fit with the colours of the rear and front sub-meshes of the “light bulbs”.

Not only the PolyStang has sub-meshes, but also all the other models, like the street modules, the props or the wheels. So, go ahead and customize the materials and have your unique models as you prefer!

Fig. 18: example of customization of PolyStang with the help of sub-meshes.

POST PROCESSING

The post-processing volume used in the demo scene (as well as in the other scene of this asset) is very minimalistic and retro-style. It uses the following effects:

- Vignette: only the intensity and smoothness are changed. It helps to create some shading at the borders of the screen.
- Color Grading: the selected mode is “ACES”. It is a beautiful effect (also used in cinematographic works) that helps to create deepness and simulate a realistic colour grading. The temperature is increased, and the tint is switched to a bluer tone to give the environment a “old-style look”. To have a “horror look” decrease the temperature to –20, for example.
- Ambient Occlusion: this effect improves shadows.
- Bloom: this effect helps simulate volumetric light.

Any change or feature added to the post-processing has an impact on the graphical performance of the game, so it is a good practise to keep it as simple as possible.

SOUND DESIGN AND MIXER

The sounds simulated in this car controller are two: the engine noise and the skid mark sounds.

The engine noise sound's pitch is changed through the "CarSound" script and is proportional to the module of the velocity of the rigidbody of the car. The maximum pitch is the highest pitch that can be reached and over that limit the pitch is constant. To know more about how the script works look at the Script section below.

There is a skid mark sound for every wheel of the car and the respective audio source objects are positioned where the wheels are. The audio sources are 3D. When the game starts, they are played (on the "Awake()" function, actually), but their volume is set to 0. Every time each wheel drifts over the limits set in the car controller, the volume of the audio source corresponding to that wheel is set to 1. At the same time, every time all these audio sources volumes are constantly reduced using the "SmoothClamp" function to reach 0. To help you understand how this effect works, look at Fig. 19:

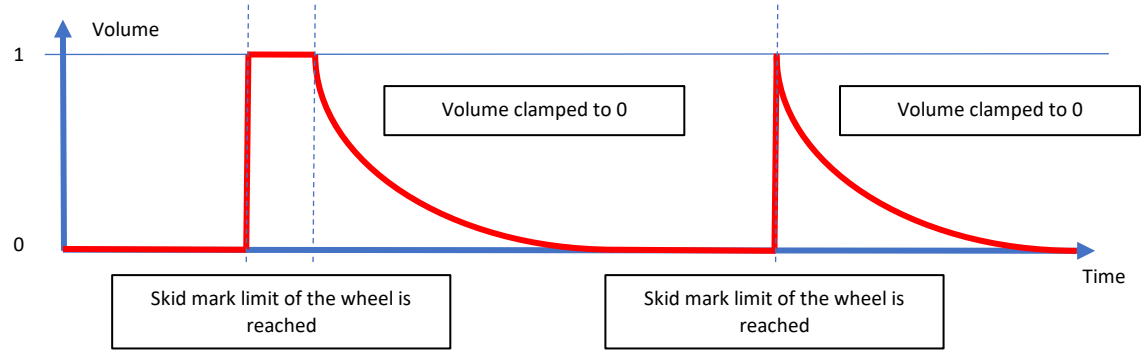

Fig. 19: example of volume trend when drifting or not.

All the audio clips of the above-mentioned sound effects are assigned to the SFX group of the audio mixer. The engine audio clip and skid mark audio clips are assigned to two different groups under the SFX group.

SCRIPTS

The scripts used in this asset are four:

- CarController: this script is meant to deal with the inputs and move the car accordingly. From this script functions in the CarSounds and CarLights scripts are called.
- CarSounds: this script deals with the car engine sound and has two functions (PlaySkidSound() and StopSkidSound()) to be called from the CarController, which change the volume of the wheels' skid mark audio source.
- CarLights: this script has various public functions to turn the lights on and off, when braking and when the reverse gear is used.
- CameraSwitcher: this script is responsible for switching the camera view from rear to front, and vice versa.

CarController

Let's have a look at how the CarController script works from top to bottom.

```
public enum ControlMode // this car controller works for both pc and touch devices. You can switch the control mode from the inspector.
{
    Keyboard,
    Buttons
};
```

Fig. 20: public enum ControlMode.

In this part of the script the enum ControlMode is stated. This statement is used to control from the inspector if the controls are for PC (keyboard) or for mobile (Buttons). Through the variable `public ControlMode control` (line 32) the control mode selected influences the behaviour of the script. This detail is meant to make it easier to use this script for mobile purposes.

```
public enum Axel // used to identify front and rear wheels.
{
    Front,
    Rear
}

[Serializable]
public struct Wheel // wheel bits: all fields must be filled to make the wheel work properly.
{
    public GameObject wheelModel;
    public WheelCollider wheelCollider;
    public GameObject wheelEffectObj;
    public ParticleSystem smokeParticle;
    public Axel axel;
    public GameObject skidSound;
    public int index;
}
```

Fig. 21: public struct Wheel.

In this first part of the script the struct Wheel is defined, so Wheel objects can be created freely in the inspector under the list `public List<Wheel> wheels` (line 58). Every wheel is going to have assigned a specific wheel model, collider, wheelEffectObj (which is the skid mark), smokeParticle (which is the particle system simulating the smoke produced by the skid mark), axel (front or rear, as in the enum Axel), skidSound (the audio source of the skid mark) and an index.

```
[Header("Inputs")]
public KeyCode brakeKey = KeyCode.Space;

[Header("Accelerations and deaccelerations")]
public float maxAcceleration = 30.0f;
public float brakeAcceleration = 50.0f;
public float noInputDeacceleration = 10.0f;

[Header("Steering")]
public float turnSensitivity = 1.0f;
public float maxSteerAngle = 30.0f;

[Header("Speed UI")]
public TMP_Text speedText;
public float UISpeedMultiplier = 4;

[Header("Speed limit")]
public float frontMaxSpeed = 200;
public float rearMaxSpeed = 50;
public float empiricalCoefficient = 0.41f;
8 riferimenti
public enum TypeOfSpeedLimit
{
    noSpeedLimit,
    simple,
    squareRoot
};
public TypeOfSpeedLimit typeOfSpeedLimit = TypeOfSpeedLimit.squareRoot;
private float frontSpeedReducer = 1;
private float rearSpeedReducer = 1;

[Header("Skid")]
public float brakeDriftingSkidLimit = 10f;
public float lateralFrontDriftingSkidLimit = 0.6f;
public float lateralRearDriftingSkidLimit = 0.3f;

[Header("General")]
public Vector3 _centerOfMass;

public List<Wheel> wheels;

float moveInput;
float steerInput;

private Rigidbody carRb;

private CarLights carLights;
private CarSounds carSounds;
```

Fig. 22: other variables.

All the variables in Fig. 22 are used in the various functions that are later explained, so their meaning is going to be defined later.

```
void Start() // called the first frame, when the game starts.
{
    carRb = GetComponent<Rigidbody>();
    carRb.centerOfMass = _centerOfMass;

    carLights = GetComponent<CarLights>();
    carSounds = GetComponent<CarSounds>();
}

void Update() // called every frame.
{
    GetInputs();
    AnimateWheels();
    WheelEffectsCheck();
    CarLightsControl();
}

void LateUpdate() // called after the "Update()" function.
{
    Move();
    Steer();
    BrakeAndDeacceleration();
    UpdateSpeedUI();
}
```

Fig. 23: Start(), Update() and LateUpdate().

In the Start() function, the references to the scripts CarSounds and CarLights are done, as well as the reference to the Rigidbody component. These components are all attached to the car model object. Moreover, the centre of mass of the Rigidbody is changed according to the user's preferences (the vector `_centerOfMass` is exposed in the Inspector).

In the Update() and LateUpdate() all the main functions controlling the car are called. In particular, the inputs and wheel animations are done in the Update(), before the car is moved or steering is applied, as well as braking, deceleration or updating the speed UI. Let's have a look at those functions!

```
public void MoveInput(float input) // used for touch controls.
{
    moveInput = input;
}

public void SteerInput(float input) // used for touch controls.
{
    steerInput = input;
}

void GetInputs() // inputs.
{
    if (control == ControlMode.Keyboard)
    {
        moveInput = Input.GetAxis("Vertical");
        steerInput = Input.GetAxis("Horizontal");
    }
}
```

Fig. 24: Inputs.

As shown in Fig. 24, the inputs can be got in two different manners:

- Using the GetInputs() which is called in the Update() function. It's important to notice that if the control mode is set to "Buttons", the "moveInput" (used to store the input to move forwards or backwards) and the "steerInput" (used to store the input to steer the front wheels left or right) are not updated. In this case, they are updated in the second manner below.
- Using buttons, calling the public functions "MoveInput()" and "SteerInput()": you can set the appropriate buttons to call these functions and to set the "moveInput" and "steerInput" to +1 or –1, giving them as argument +1 or –1.

In the next Fig. 25, the basic functions for accelerating forwards and backwards ("Move()") and steering left and right ("Steer()") are shown. Remember that in the Steer() function only the front wheels are steered, using a if() statement. In the "Move()" function for each wheel the rotational speed is calculated. Depending on the fact that the car is currently moving forwards or backwards (or the car is accelerating or decelerating), a "reducer" variable is estimated from the wheel rotational speed (using the Empirical coefficient). This reducer variable varies in a range between 0 (when the car reaches the speed limit) and 1 (when the car is not moving). The function to change its value is different for every "Type Of Speed Limit".

```
void Move() // main vertical acceleration.
{
    foreach (var wheel in wheels)
    {
        // rotational speed is proportional to radius * frequency: the empirical coefficient is around 0.41
        float currentWheelSpeed = empiricalCoefficient * wheel.wheelCollider.radius * wheel.wheelCollider.rpm;

        if (moveInput > 0 || currentWheelSpeed > 0) // when moving forwards
        {
            if(currentWheelSpeed > frontMaxSpeed) // important check: it prevents the car from accelerating indefinetly
            {
                currentWheelSpeed = frontMaxSpeed;
            }

            // cases: different speed reducing technics
            if (typeOfSpeedLimit == TypeOfSpeedLimit.noSpeedLimit)
            {
                frontSpeedReducer = 1;
            }
            else if (typeOfSpeedLimit == TypeOfSpeedLimit.simple)
            {
                frontSpeedReducer = (frontMaxSpeed - currentWheelSpeed ) / frontMaxSpeed;
            }
            else if (typeOfSpeedLimit == TypeOfSpeedLimit.squareRoot)
            {
                frontSpeedReducer = Mathf.Sqrt(Mathf.Abs((frontMaxSpeed - currentWheelSpeed) / frontMaxSpeed));
            }

            // applying reduction
            wheel.wheelCollider.motorTorque = moveInput * 600 * maxAcceleration * frontSpeedReducer * Time.deltaTime;
        }
        else if (moveInput < 0 || currentWheelSpeed < 0) // when moving backwards
        {
            if (currentWheelSpeed < - rearMaxSpeed) // important check: it prevents the car from accelerating indefinetly
            {
                currentWheelSpeed = - rearMaxSpeed;
            }

            // cases: different speed reducing technics
            if (typeOfSpeedLimit == TypeOfSpeedLimit.noSpeedLimit)
            {
                rearSpeedReducer = 1;
            }
            else if (typeOfSpeedLimit == TypeOfSpeedLimit.simple)
            {
                rearSpeedReducer = (rearMaxSpeed + currentWheelSpeed) / rearMaxSpeed;
            }
            else if (typeOfSpeedLimit == TypeOfSpeedLimit.squareRoot)
            {
                rearSpeedReducer = Mathf.Sqrt(Mathf.Abs((rearMaxSpeed + currentWheelSpeed) / rearMaxSpeed));
            }

            // applying reduction
            wheel.wheelCollider.motorTorque = moveInput * 600 * maxAcceleration * rearSpeedReducer * Time.deltaTime;
        }
    }
}

void Steer() // to rotate the front wheels, when steering.
{
    foreach(var wheel in wheels)
    {
        if (wheel.axel == Axel.Front)
        {
            var _steerAngle = steerInput * turnSensitivity * maxSteerAngle;
            wheel.wheelCollider.steerAngle = Mathf.Lerp(wheel.wheelCollider.steerAngle, _steerAngle, 0.6f);
        }
    }
}
```

Fig. 25: basic movement and steering.

The “BrakeAndDeacceleration()” function concerns the management of braking and deceleration (Fig. 25).

First, with the first “if()” statement, when the “brakeKey” is pressed (which is the key that can be customized from the inspector and is assigned to be the input key for braking), for each wheel collider the brakeTorque is applied though time.

Secondly, if this “brakeKey” is not pressed (look at the “else if()” statement) and the “moveInput” is 0 (so the player is not pressing the “w” or “s” key, or the top and down arrow keys), a brakeTorque is applied through the “noInputDeacceleration” value. This value should be quite lower than the “MaxAcceleration” value, which is used in the “Move()” function (Fig. 25). You can choose how low this value must be, but the main goal of this deceleration is to simulate the “braking engine power” (google it!), when the car is moving but no input is entered.

The last “else” statement happens when the player is not braking but is pressing a key to move forwards or backwards (so the “moveInput” is not 0). In this case, we want to get sure that no brakeTorque is applied through the wheel colliders.

```
void BrakeAndDeacceleration()
{
    if (Input.GetKey(brakeKey)) // when pressing space, the brake is used.
    {
        foreach (var wheel in wheels)
        {
            wheel.wheelCollider.brakeTorque = 300 * brakeAcceleration * Time.deltaTime;
        }

    }
    else if(moveInput == 0) // with no vertical input, a slight deacceleration is used to slightly slow down the speed of the car.
    {
        foreach (var wheel in wheels)
        {
            wheel.wheelCollider.brakeTorque = 300 * noInputDeacceleration * Time.deltaTime;
        }
    }
    else // with vertical input, no brake or deacceleration is applied.
    {
        foreach (var wheel in wheels)
        {
            wheel.wheelCollider.brakeTorque = 0;
        }
    }
}
```

Fig. 26: braking and deceleration.

The “AnimateWheels()” function is responsible for animating the wheels. Every time this function is called, a Quaternion (“rot”) and Vector3 (“pos”) are created, and then the position and rotation of the wheel colliders of every wheel is saved through them, using the “GetWorldPos()” function.

Then, that the position and rotation of the wheel models are applied, using the “pos” and “rot” updated in the previous step.

```
void AnimateWheels() // to animate wheels accordingly to the car speed.
{
    foreach(var wheel in wheels)
    {
        Quaternion rot;
        Vector3 pos;
        wheel.wheelCollider.GetWorldPose(out pos, out rot);
        wheel.wheelModel.transform.position = pos;
        wheel.wheelModel.transform.rotation = rot;
    }
}
```

Fig. 27: animating wheels.

The “WheelEffectsCheck()” and the “EffectCreate()” functions (Fig. 28) are respectly responsible for checking if the skid mark effect and the smoke particle system of every single wheel must be activated, and for activating them.

For each wheel, the conditions for lateral drifting are first checked before all the “if()” and “else if()” statements, because they are needed for them. The lateral drifting value is store in a temporary float (“lateralDrift”) and is set equal to the “sidewaysSlip” variable of the “GroundHit” taken from the single wheel collider.

Then, the first “if()” statement is considered true when the car is braking and moving faster than a certain limit (the “brakeDriftingSkidLimit”, which can be changed from the Inspector), and it concerns only the rear wheels (they are the only ones for which the skid mark and particle effects are activated). If this statement is true, then the effects are created calling the “EffectCreate()” function and sending the wheel object for which the statement is true.

The second and third “else if()” statements are quite similar: the second one regards the front wheels, while the third one the rear wheels. They are true when the “lateralDrift” of that single wheel is higher than the “lateral drifting skid limit” of that wheel. There are to different limits, because the front wheels can also rotate when steering so they are “more sensitive” to drifting. This is why the “lateral drifting skid limit” for the front wheels is higher than the one of the rear wheels (the proportion is around 7:3, or 9:4). If one of those statements is true, then the

effects are created calling the “EffectCreate()” function and sending the wheel object for which the statement is true.

If none of the above-mentioned conditions is true (“else”), then all the effects are stopped: the emission of the Trail Renderer responsible for the skid marks on the ground for the specific wheel is stopped, and the volume of the skid mark audio clip of that wheel starts to lower down through the functions “StopSkidSound()”, which is in the “CarSound” script.

The “CreateEffect()” function, which can be called by the “WheelEffectsCheck()”, simply activates the skid mark Trail Renderer and the smoke particle system produced by the skid mark, and calls the function “StartSkidSound()” from the “CarSound” script (which does the “opposite” of the “StopSkidSound()”) setting the volume of the skid sound to 1 for that particular wheel. This is why the wheel object must be sent to the “CreateEffect()” function as argument, so it “knows” for which wheel the effects must be created. To know more about the functions “StartSkidSound()” and “StopSkidSound()”, have a look to the CarSound script section”!

```
void WheelEffectsCheck() // checking for every wheel if it's slipping: if yes, the "EffectCreate()" function is called.
{
    foreach (var wheel in wheels)
    {
        // slipping ---> skid
        WheelHit GroundHit; // variable to store hit data
        wheel.wheelCollider.GetGroundHit(out GroundHit); // store hit data into GroundHit
        float lateralDrift = Mathf.Abs(GroundHit.sidewaysSlip);

        if (Input.GetKey(brakeKey) && wheel.axel == Axel.Rear && wheel.wheelCollider.isGrounded == true && carRb.velocity.magnitude >= brakeDriftingSkidLimit)
        {
            EffectCreate(wheel);
        }
        else if(wheel.wheelCollider.isGrounded == true && wheel.axel == Axel.Front && (lateralDrift > lateralFrontDriftingSkidLimit)) // drifting: front wheels
        {
            EffectCreate(wheel);
        }
        else if(wheel.wheelCollider.isGrounded == true && wheel.axel == Axel.Rear && (lateralDrift > lateralRearDriftingSkidLimit)) // drifting: rear wheels
        {
            EffectCreate(wheel);
        }
        else
        {
            wheel.wheelEffectObj.GetComponentInChildren<TrailRenderer>().emitting = false;
            carSounds.StopSkidSound(wheel.skidSound, wheel.index); // actually decreasing the volume of the skid to 0: see the "CarSound" script.
        }
    }
}

private void EffectCreate(Wheel wheel) // actually creating the effects: 1) trail renderer for the skid, 2) smoke particles, 3) skid sound.
{
    wheel.wheelEffectObj.GetComponentInChildren<TrailRenderer>().emitting = true;
    wheel.smokeParticle.Emit(1);
    carSounds.PlaySkidSound(wheel.skidSound); // actually setting the volume of the skid to 1
}
```

Fig. 28: wheel effects.

In the last part of the CarController script, the car lights and the speed UI are managed (Fig. 29). The lights state is under two conditions:

- First, if the car is braking, the rear red lights are turned on calling the function “RearRedLightsOn()”, which is the CarLights script. If the latter statement is false, the function “RearRedLightsOff()”, which is the CarLights script, is called to turn the rear red lights.
- Similarly, the rear white lights are managed considering the “moveInput” float (which is used to keep the information about the keys “w”, “s” and down arrow and up arrow. When its value is negative, the player is “using” the reverse gear, so the white rear lights must be turned on. Else if its value is positive, then they must be turned off.

Finally, the speed UI is updated: the velocity of the rigidbody attached to the car is rounded to be an integer and then this value is converted to a string and used to update the text of the speed UI.

```
void CarLightsControl() // controlling lights, through the specific script "CarSounds".
{
    if(Input.GetKey(brakeKey)) // the red lights are activated when the brake is pressed
    {
        carLights.RearRedLightsOn();
    }
    else
    {
        carLights.RearRedLightsOff();
    }

    if(moveInput < 0f) // the rear white lights are activated when the player is pressing "S" or down arrow.
    {
        carLights.RearWhiteLightsOn();
    }
    else
    {
        carLights.RearWhiteLightsOff();
    }
}

void UpdateSpeedUI() // UI: speed update.
{
    int roundedSpeed = (int)Mathf.Round(carRb.velocity.magnitude * speedMultiplier);
    speedText.text = roundedSpeed.ToString();
}
```

Fig. 29: car lights management and speed UI update.

CarSounds

Let’s have a look at how the CarSounds script works.

First, some variables are stated. Their function is going to be described when they are used. In the “Start()” function, the carAudio variable is set as the attached audio source component attached to the car, and the rigidbody of the car is set similarly too (Fig. 30). IMPORTANT:

do not delete that audio source or the engine sound will not be simulated. Also get sure that the audio clip of this audio source is not null and is being played “On Awake”.

```
    private float currentSpeed;

    private Rigidbody carRb;
    private AudioSource carAudio;

    [Header("Engine Pitch settings")]
    public float minPitch;
    public float maxPitch;
    private float pitchFromCar;

    public float[] smoothVelocities; // IMPORTANT: add enough floats of this array from the inspector (as many as the wheels).

    void Start() // called the first frame, when the game starts.
    {
        carAudio = GetComponent<AudioSource>();
        carRb = GetComponent<Rigidbody>();
    }
```

Fig. 30: variables and “Start()” function.

Then the engine sound is updated through the “Update()” function at every frame of the game (Fig. 31). The pitch of the engine sound is set as the “minPitch” (the minimum value of the pitch, reached when the car is not moving) plus the “pitchFromCar” (which is proportional to the current velocity magnitude of the rigidbody attached to the car). This value is also clamped between the “minPitch” and the “maxPitch” (the maximum value of the pitch).

```
void Update() // called every frame
{
    EngineSound();
}

void EngineSound() // changing the pitch according to the speed
{
    currentSpeed = carRb.velocity.magnitude;
    pitchFromCar = currentSpeed / 60f;

    carAudio.pitch = Mathf.Clamp(minPitch + pitchFromCar, minPitch, maxPitch); // keeping sure to stay into the pitch bounds.
}
```

Fig. 31: engine sound pitch updating.

The functions “PlaySkidSound()” and “StopSkidSound()” (Fig. 32) are never called in this script, but from the CarController script.

```
// called from Car Controller
public void PlaySkidSound(GameObject skidSound) // setting the volume of the skid sound to 1.
{
    skidSound.GetComponent<AudioSource>().volume = 1;
}

// called from Car Controller
public void StopSkidSound(GameObject skidSound, int index) // changing the volume from the skid sound from the value it currently has to 0.
{
    skidSound.GetComponent<AudioSource>().volume = Mathf.SmoothDamp(skidSound.GetComponent<AudioSource>().volume, 0, ref smoothVelocities[index], 0.1f);
}
```

Fig. 32: controlling the volume of the skid mark sound effects.

The “PlaySkidSound()” function receives the skidSound, whose volume must be set to 1, from the CarController script, which “knows” which wheel is drifting (Fig. 28).

The “StopSkidSound()” function receives the skidSound, whose volume must be lowered using the “SmoothDamp()” function to 0, from the CarController script, which “knows” which wheel is no longer drifting (Fig. 28). In this case also an index must be passed, because, for each wheel, a specific float is used as “ref float velocity” (it is a variable that the “SmoothDamp()” function needs): those floats are the ones in the array of floats named as “smoothVelocities”. Note that the time used for smoothing the volume is the float “0.1f”, as fourth argument of the “SmoothDamp()” function. That is the only parameter in this asset that is not exposed to the Inspector because it is not considered as important as others to be changed. If you want to change it, you must consider that if it is lower, the skid mark sound effect is smoothed faster to 0, and if it is higher, the sound effect is smoothed slower to 0. Consider also that, unless you really need to change it, the default value works fine and seems sufficiently appropriate to add a bit of delay (not too much) for fading of the sound effect.

CarLights

Let’s have a look at how the CarLights script works.

First, some variables are stated. Their function is going to be described when they are used. In the “Start()” function the rear red and white lights are set as false (in the case the user changed their state and forgot to turn them off), and the front spotlights are turned on if the bool “isDay” is false (probably if the scene is set at night), or vice versa, they are turned off (Fig. 33).

```
public bool isDay; // change this bool from the inspector to have the front spot lights on when it's night.

[Header("Rear Lights")]
public GameObject RearWhitePointLights;
public GameObject RearRedPointLights;

[Header("Front Lights")]
public GameObject FrontSpotLights;

void Start() // called the first frame, when the game starts.
{
    RearWhitePointLights.SetActive(false);
    RearRedPointLights.SetActive(false);

    if(isDay == false)
    {
        FrontSpotLights.SetActive(true);
    }
    else
    {
        FrontSpotLights.SetActive(false);
    }
}
```

Fig. 33: variables and “Start()” function.

The other functions in Fig. 34 are never called in this script but in the CarController script that “knows” how the car is moving or if the player is braking.

```
// rear white lights.
public void RearWhiteLightsOn()
{
    RearWhitePointLights.SetActive(true);
}
public void RearWhiteLightsOff()
{
    RearWhitePointLights.SetActive(false);
}

// rear red lights.
public void RearRedLightsOn()
{
    RearRedPointLights.SetActive(true);
}
public void RearRedLightsOff()
{
    RearRedPointLights.SetActive(false);
}
```

Fig. 34: turning the lights on and off.

CameraSwitcher

Let’s have a look at how the CarLights script works (Fig. 35). This is the simplest script of all. First, the two virtual cameras for the view from the rear of the car (“RearVirtualCamera”) and the view from the front (“FrontVirtualCamera”) are exposed to the Inspector (to be set). The key for switching is also exposed to the Inspector and set as default to “R”.

The rear virtual camera has a priority of 10, while the front one of 20 (these values are changed from the cinemachine virtual camera component). When the front virtual camera is active in the Hierarchy, the main camera (which must have a cinemachine “brain” component attached – the camera prefab has it as default) follows the front virtual camera’s “instructions”. Vice versa, the main camera follows the rear virtual camera’s “instructions”.

This is why at the “Start()” function the front virtual camera is set inactive, and the rear virtual camera is set active, as default.

To switch between the two views, during the “Update()” function, the virtual cameras are switched (setting one as active and the other not, and viceversa) using the function “SwitchCamera()”.

```
[Header("Virtual cameras")]
public GameObject RearVirtualCamera;
public GameObject FrontVirtualCamera;

public KeyCode switchVirtualCameraKey = KeyCode.R;

void Start() // called the first frame, when the game starts.
{
    RearVirtualCamera.SetActive(true);
    FrontVirtualCamera.SetActive(false);
}

void Update() // called every frame
{
    if(Input.GetKeyDown(switchVirtualCameraKey))
    {
        SwitchCamera();
    }
}

public void SwitchCamera()
{
    if(RearVirtualCamera.activeInHierarchy) // if the rear virtual camera is active, switch to the front virtual camera
    {
        RearVirtualCamera.SetActive(false);
        FrontVirtualCamera.SetActive(true);
    }
    else // vice versa, switch to the rear virtual camera
    {
        RearVirtualCamera.SetActive(true);
        FrontVirtualCamera.SetActive(false);
    }
}
```

Fig. 35: Camera switcher script.