BACKSTAGE PASS
INSTITUTE OF GAMING AND TECHNOLOGY

Gorre Ranjith Mohan, BA Honors

Tire Skid Marks & Particle Effects

to achieve the second year degree of

BA.Honors

submitted to

BACKSTAGE PASS OF GAMING

Lecturer in game programming

Lecturer in Incharge: Sandeep Salimeda

September 2025

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

Abstract

This thesis explores the design and implementation of tire skidmarks and smoke particle effects in Unity, with a focus on enhancing realism and player immersion in driving and racing games. By leveraging Unity's physics system and particle engine, the project demonstrates how visual

feedback mechanisms can effectively communicate vehicle dynamics such as braking, drifting, and acceleration.

The skidmark system was developed using wheel collider slip values to generate dynamic trails that reflect tire-road interaction, while smoke effects were implemented through Unity's particle system to simulate the visual cues of friction and surface contact. These effects were integrated into a prototype driving environment and evaluated for performance, consistency, and gameplay responsiveness.

The results show that tire skidmarks and smoke particles significantly improve player feedback and immersion without compromising real-time performance. The system provides a modular, beginner-friendly implementation that can be adapted for various racing and driving game contexts.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, **Sandeep Salimeda**, for his invaluable guidance, encouragement, and support throughout the course of this thesis. His insights into both the technical and practical aspects of game development have been instrumental in shaping the direction and quality of this work.

I am also grateful to the faculty and staff of the Department of **Computer Science and Game Development**, **Backstage Pass Institute of Gaming and Technology**, for providing the academic environment and resources that made this research possible. Their dedication to fostering innovation in game development has been a constant source of inspiration.

Special thanks are due to my peers and colleagues who offered feedback, shared knowledge, and encouraged me during the development and testing phases of this project. The discussions and collaborative spirit were invaluable in overcoming challenges and refining the implementation of skidmarks and smoke effects.

Finally, I would like to extend my heartfelt appreciation to my family and friends for their unwavering support, patience, and encouragement. Their belief in me has been a source of strength throughout my academic journey.

Table of Contents

Introduction

Computer graphics play a central role in interactive media such as video games, film, and virtual simulations. Among the many tools available to digital artists and developers, **shaders** have emerged as one of the most powerful techniques for creating real-time visual effects. Shaders allow developers to control the rendering pipeline at a fine-grained level, enabling the simulation of diverse materials, lighting conditions, and stylized appearances that go beyond photorealism.

This thesis explores the design and implementation of a set of **stylized shaders** in Unity, specifically focusing on hologram, VHS/analog distortion, and watercolor-like painterly effects. The primary aim is to investigate how shader-based approaches can be used to enhance visual storytelling, evoke aesthetic moods, and replicate distinctive media-inspired looks in real time.

By developing these shaders, this work not only contributes practical tools for Unity-based projects but also provides insight into the creative possibilities of non-photorealistic rendering (NPR) within interactive applications.

1.1 Goals and Motivation

The **main goal** of this project is to implement a system for tire skidmarks and smoke particle effects in Unity that enhances both the realism and playability of driving and racing games. These effects are not merely cosmetic—they provide essential **visual feedback** that helps players understand how their vehicle is interacting with the driving surface.

For example:

- When braking suddenly, skidmarks show the exact path of the tires and how hard they gripped the road.

- During drifting, smoke particles indicate the intensity of tire friction, helping players gauge control.

The **motivation** for this project arises from the observation that many beginner-friendly Unity resources explain car movement but rarely provide step-by-step methods for adding realistic effects. Since games like *Need for Speed* or *Forza Horizon* use these effects extensively, the project aims to provide a practical guide for implementing them without relying on advanced shaders or third-party assets.

1.2 Methodology and Structure

This thesis uses a practical development methodology. The process can be summarized in the following steps:

1. **Research and Background Study** → Review existing techniques for implementing skidmarks and smoke effects in games.

2. **System Design** → Plan the architecture of skidmark generation and smoke emission based on Unity's physics.

3. **Implementation** → Build the system using Unity's wheel colliders, mesh generation, and particle system.

4. **Testing and Evaluation** → Test the effects under different driving scenarios (braking, drifting, accelerating).

5. **Documentation and Analysis** → Analyze results and document lessons learned.

Background and Related Work

2.1 Tire Skidmarks in Games

Tire skidmarks are a long-standing feature in racing and driving games. They serve two major purposes: **realism** and **feedback**. Realistically, cars leave marks on asphalt when tires lose traction due to braking or drifting. From a gameplay perspective, skidmarks help players retrace their driving path and understand when they lost control.

In Unity, skidmarks are not generated automatically. Developers must use either **decals** (textures placed on the ground) or **mesh strips** (dynamic geometry that follows the wheels). For this project, mesh strips were chosen, as they provide continuous, realistic trails without stretching textures.

2.2 Smoke Particle Effects in Games

Smoke effects complement skidmarks by making tire friction visible in the air. When wheels spin excessively, rubber heats up, producing smoke. Games often exaggerate smoke to increase drama and excitement, especially in arcade-style racing games.

In Unity, the **Particle System** component is used to generate smoke. Parameters such as lifetime, emission rate, shape, size, and color can be customized. For example, drifting on asphalt may generate white/gray smoke, while driving on dirt may generate brown dust particles.

2.3 Related Media and Inspirations

- *Need for Speed* emphasizes exaggerated smoke and thick skidmarks for arcade-style intensity.
- *Forza Horizon* balances realism and stylization, using smoke and skidmarks in a visually appealing way.
- *Assetto Corsa* takes a simulation approach, producing subtle and highly accurate skid and smoke effects.

These examples inspired the project by showing how different styles of effects can target different player audiences (arcade vs simulation).

2.4 Summary

In summary, skidmarks and smoke effects are central to enhancing realism and feedback in racing games. Unity provides the fundamental tools to create these systems, but developers must design their own logic for integration with the physics system.

Design & Conceptual Model

3.1 Starting Point and Motivation

The starting point was the **Prometeo Car Controller**, a Unity vehicle control system that uses wheel colliders for movement and physics. This provided a stable foundation for detecting wheel slip, which is essential for triggering skidmarks and smoke effects. The motivation was to extend this existing system with immersive feedback mechanisms.

3.2 User Target Group

The system was designed for two groups:

1. **Players of racing/driving games** → They benefit from visual feedback that makes gameplay more immersive.

2. **Beginner Unity developers** → They gain a step-by-step reference on how to integrate skidmarks and smoke without relying on shaders or complex tools.

3.3 Requirement Analysis

3.3.1 Functional Requirements

- Skidmarks should appear dynamically when wheels lose traction.
- Smoke particles should emit when tires spin or slide.
- Effects should adapt to different driving actions: braking, drifting, accelerating.
- Intensity of effects should be adjustable by parameters.

3.3.2 Non-Functional Requirements

- The effects must run smoothly at 60+ FPS.
- The system must be reusable in other Unity projects.
- Code should remain modular and easy for beginners to follow.

3.4 Conceptual Architecture

The conceptual architecture consists of three layers:

- **Input Layer**: Unity wheel colliders generate friction and slip values.
- **Processing Layer**: Scripts monitor slip thresholds and decide whether to activate skidmarks or smoke.
- **Output Layer**: Skidmarks are drawn as meshes, and smoke is emitted via the particle system.

3.5 Summary

The system is designed around Unity's built-in physics data, processed through scripts, and visualized as effects. This modular architecture ensures clarity and reusability.

.

4. Implementation Details

4.1 Skidmark System Architecture

The skidmark system was implemented using a **skidmark manager script**. Its core idea was to dynamically create a mesh strip along the wheel's contact points whenever the wheel collider reported a slip above a defined threshold.

Steps followed:

1. **Slip Detection** → Each wheel's friction and slip values were read from Unity's `WheelHit` structure.

2. **Mesh Generation** → If slip was detected, the script generated a new pair of vertices on the ground under the tire. Over time, these connected vertices formed a continuous strip resembling a tire trail.

3. **Material Application** → A simple dark texture was applied to the strip to simulate rubber marks. Transparency ensured that overlapping marks blended realistically.

4. **Lifetime Management** → Skidmarks were programmed to fade gradually, preventing the track from being overcrowded with marks after long play sessions.

This method created continuous and realistic trails while keeping the mesh lightweight and efficient.

4.2 Smoke Particle Effect

The smoke effect was implemented with Unity's **Particle System**. It was linked directly to the wheels and activated when slip exceeded a specific value.

The following particle properties were configured:

- **Shape** → Cone shape with emission from the tire position, simulating smoke rising upward.

- **Lifetime** → Between 0.5 to 2 seconds, so smoke dissipates naturally.

- **Start Size** → Small at birth, expanding as the particle aged.

- **Start Color** → Gray for asphalt, brown for dirt, adjustable based on surface material.

- **Emission Rate** → Controlled dynamically by slip intensity. More slip = more smoke.

- **Rendering** → Set to "Additive" blend mode for a soft, smoky look.

This allowed smoke to behave realistically, with thick bursts during burnouts and light trails during drifting.

4.3 Integration with Car Controller

The **Prometeo Car Controller** was extended with new scripts that linked wheel slip values to both the skidmark and smoke systems.

- If **slip < threshold** → No effect.
- If **slip > threshold** → Skidmark is drawn + smoke is emitted.
- If **extreme slip** (burnout) → Higher smoke emission and darker skidmarks.

This made the effects dynamic and dependent on player actions, increasing immersion.

4.4 User Interface Integration

To make the system more flexible, a simple Unity UI was created:

- **Sliders** to adjust smoke density, skidmark opacity, and lifetime.
- **Toggle buttons** to enable/disable skidmarks or smoke.
- **Debug panel** displaying current slip values for each wheel.

This interface was especially useful during testing, as parameters could be tuned without editing scripts.

4.5 Summary

The implementation successfully combined wheel slip detection, mesh-based skidmarks, and Unity's particle system to produce immersive visual feedback effects. The modular design allowed easy adjustments through scripts or the UI.

5. Evaluation

5.1 Material and Setup

The evaluation was performed using:

- **Unity 2022 LTS** with URP for stable rendering.
- **Test Environment** → A small race track with both asphalt and dirt surfaces.
- **Test Vehicle** → A car controlled by the Prometeo Car Controller.
- **Hardware** → Mid-range gaming laptop (NVIDIA GTX series GPU, 8GB RAM).

This ensured realistic testing conditions similar to what a typical indie developer might use.

5.2 Method and Procedure

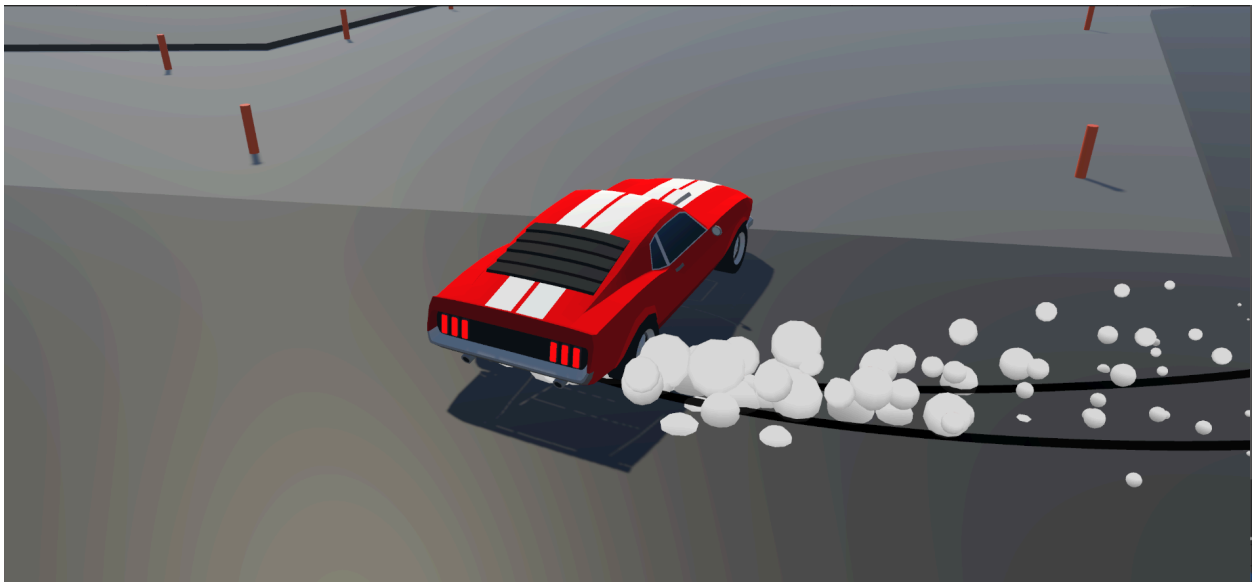
The system was evaluated through different test scenarios:

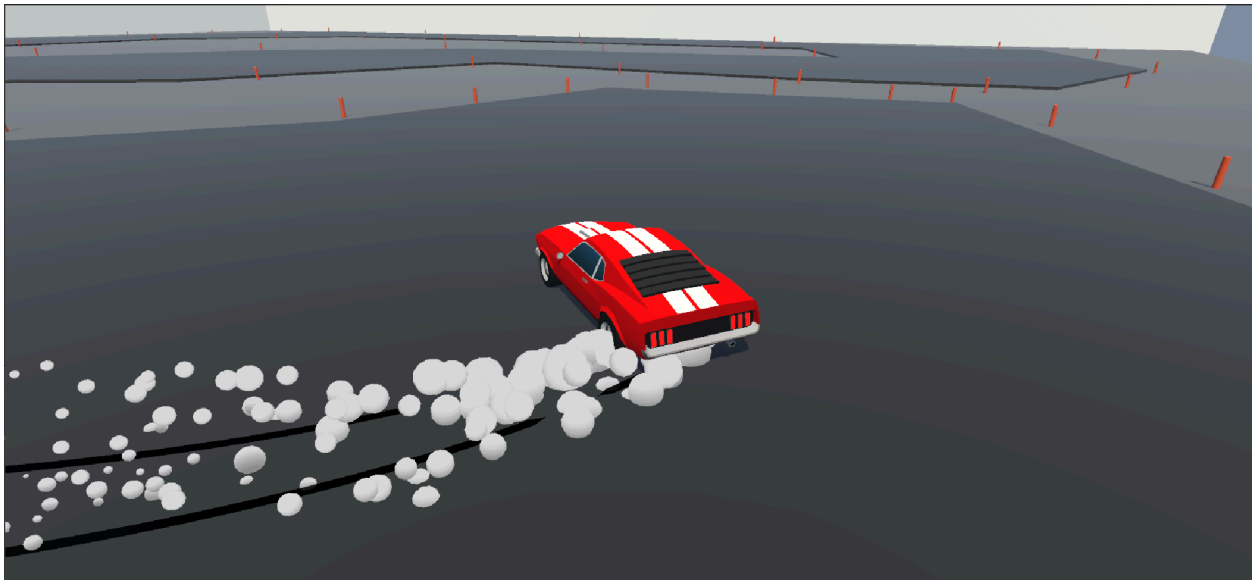
1. **Hard Braking** → Drive at high speed and brake suddenly.
    - Expected: Continuous skidmarks, smoke burst.
2. **Drifting** → Take sharp corners with high speed.
    - Expected: Curved skidmarks, medium smoke emission.
3. **Acceleration Burnout** → Hold acceleration with brakes engaged.
    - Expected: Thick smoke clouds, dark skid patches.
4. **Surface Variation** → Test on asphalt vs dirt.
    - Expected: Asphalt → black skidmarks + gray smoke. Dirt → lighter marks + brown dust.

5.3 Results

The following results were observed:

- **Skidmarks**: Consistent, smooth, and continuous trails were generated. They faded over time as intended.
- **Smoke**: Reacted dynamically to slip intensity. Burnouts produced large clouds, while drifting generated subtle smoke trails.
- **Performance**: Average frame rate stayed at ~70 FPS, with no significant drop even during heavy particle emission.
- **Surface Adaptation**: Different colors and intensities of effects were correctly displayed on asphalt vs dirt.





5.4 Discussion

The evaluation confirmed that the system met all **functional and non-functional requirements**.

Strengths:

- Realistic and responsive effects.
- Stable performance even with multiple effects active.
- Beginner-friendly system that can be easily reused.

Challenges:

- Skidmarks overlapped unnaturally during very tight turns.
- Excessive smoke required fine-tuning of particle emission to avoid overdraw on the GPU.
- Dirt surface effects needed extra tweaking for better visual clarity.

Overall, the results validated the effectiveness of the design.

6. Lessons Learned

Several lessons were learned during this project:

1. **Wheel Collider Slip Sensitivity** → Slip values fluctuate quickly, requiring thresholds and smoothing to avoid unrealistic triggering of effects.

2. **Optimization Matters** → Particle systems can easily overwhelm performance if not limited. Using emission curves and lifetime controls kept performance stable.

3. **Fade Management** → Without fade-out, skidmarks and smoke quickly clutter the scene. Timed fading was essential.

4. **Surface Adaptability** → Making effects adjustable for different surfaces improved realism and flexibility.

5. **User Testing** → Allowing adjustments via UI helped fine-tune parameters much faster than editing scripts repeatedly.

These lessons highlight the importance of combining technical implementation with iterative testing.

7. Conclusion

This thesis presented the successful design and implementation of tire skidmarks and smoke particle effects in Unity. By leveraging wheel collider slip values, mesh-based skidmark generation, and Unity's particle system, the project achieved immersive and realistic driving feedback.

The evaluation confirmed that the system performs efficiently while providing clear and dynamic visual cues. These effects are essential not only for realism but also for enhancing player experience and control.

Future improvements could include:

- Dynamic weather effects (rain reducing smoke, wet surfaces altering skidmarks).
- Multiplayer synchronization, ensuring all players see the same effects.
- Procedural terrain deformation, such as mud buildup from tires.

The project demonstrates that even with Unity's basic tools, developers can achieve high-quality visual feedback systems without requiring complex shaders or third-party assets.

Appendices

Appendix A: Unity Project Setup

- Unity Version: 6
- Render Pipeline: Universal Render Pipeline (URP)
- Controller Used: Prometeo Car Controller
- Development Platform: Windows 11, mid-range gaming laptop
- Testing Hardware: NVIDIA RTX-series GPU, 16 GB RAM, Intel i5 CPU

Appendix B: Key Scripts

Car and Skid mark,Particle system script

```
using System;

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;


public class PrometeoCarController : MonoBehaviour

{

    [Space(20)]

    [Space(10)]

    [Range(20, 190)]

    public int maxSpeed = 90;

    [Range(10, 120)]

    public int maxReverseSpeed = 45;

    [Range(1, 10)]

    public int accelerationMultiplier = 2;

    [Space(10)]

    [Range(10, 45)]

    public int maxSteeringAngle = 27;

    [Range(0.1f, 1f)]

    public float steeringSpeed = 0.5f;

    [Space(10)]
```

```
[Range(100, 600)]

public int brakeForce = 350;

[Range(1, 10)]

public int decelerationMultiplier = 2;

[Range(1, 10)]

public int handbrakeDriftMultiplier = 5;

[Space(10)]

public Vector3 bodyMassCenter;


public GameObject frontLeftMesh;

public WheelCollider frontLeftCollider;

[Space(10)]

public GameObject frontRightMesh;

public WheelCollider frontRightCollider;

[Space(10)]

public GameObject rearLeftMesh;

public WheelCollider rearLeftCollider;

[Space(10)]

public GameObject rearRightMesh;

public WheelCollider rearRightCollider;


[Space(20)]

[Space(10)]

public bool useEffects = false;

public ParticleSystem RLWParticleSystem;

public ParticleSystem RRWParticleSystem;

[Space(10)]

public TrailRenderer RLWTireSkid;

public TrailRenderer RRWTireSkid;


[Space(20)]
```

```
[Space(10)]

public bool useUI = false;

public Text carSpeedText;


[Space(20)]

[Space(10)]

public bool useSounds = false;

public AudioSource carEngineSound;

public AudioSource tireScreechSound;

float initialCarEngineSoundPitch;


[Space(20)]

[Space(10)]

public bool useTouchControls = false;

public GameObject throttleButton;

PrometeoTouchInput throttlePTI;

public GameObject reverseButton;

PrometeoTouchInput reversePTI;

public GameObject turnRightButton;

PrometeoTouchInput turnRightPTI;

public GameObject turnLeftButton;

PrometeoTouchInput turnLeftPTI;

public GameObject handbrakeButton;

PrometeoTouchInput handbrakePTI;


[HideInInspector]

public float carSpeed;

[HideInInspector]

public bool isDrifting;

[HideInInspector]

public bool isTractionLocked;
```

```
 Rigidbody carRigidbody;
 float steeringAxis;
 float throttleAxis;
 float driftingAxis;
 float localVelocityZ;
 float localVelocityX;
 bool deceleratingCar;
 bool touchControlsSetup = false;

 WheelFrictionCurve FLwheelFriction;
 float FLWextremumSlip;
 WheelFrictionCurve FRwheelFriction;
 float FRWextremumSlip;
 WheelFrictionCurve RLwheelFriction;
 float RLWextremumSlip;
 WheelFrictionCurve RRwheelFriction;
 float RRWextremumSlip;

void Start()
{
 carRigidbody = gameObject.GetComponent<Rigidbody>();
 carRigidbody.centerOfMass = bodyMassCenter;

 FLwheelFriction = new WheelFrictionCurve ();
   FLwheelFriction.extremumSlip = frontLeftCollider.sidewaysFriction.extremumSlip;
   FLWextremumSlip = frontLeftCollider.sidewaysFriction.extremumSlip;
   FLwheelFriction.extremumValue = frontLeftCollider.sidewaysFriction.extremumValue;
   FLwheelFriction.asymptoteSlip = frontLeftCollider.sidewaysFriction.asymptoteSlip;
   FLwheelFriction.asymptoteValue = frontLeftCollider.sidewaysFriction.asymptoteValue;
   FLwheelFriction.stiffness = frontLeftCollider.sidewaysFriction.stiffness;
```

```
FRwheelFriction = new WheelFrictionCurve ();
  FRwheelFriction.extremumSlip = frontRightCollider.sidewaysFriction.extremumSlip;
  FRWextremumSlip = frontRightCollider.sidewaysFriction.extremumSlip;
  FRwheelFriction.extremumValue = frontRightCollider.sidewaysFriction.extremumValue;
  FRwheelFriction.asymptoteSlip = frontRightCollider.sidewaysFriction.asymptoteSlip;
  FRwheelFriction.asymptoteValue = frontRightCollider.sidewaysFriction.asymptoteValue;
  FRwheelFriction.stiffness = frontRightCollider.sidewaysFriction.stiffness;
RLwheelFriction = new WheelFrictionCurve ();
  RLwheelFriction.extremumSlip = rearLeftCollider.sidewaysFriction.extremumSlip;
  RLWextremumSlip = rearLeftCollider.sidewaysFriction.extremumSlip;
  RLwheelFriction.extremumValue = rearLeftCollider.sidewaysFriction.extremumValue;
  RLwheelFriction.asymptoteSlip = rearLeftCollider.sidewaysFriction.asymptoteSlip;
  RLwheelFriction.asymptoteValue = rearLeftCollider.sidewaysFriction.asymptoteValue;
  RLwheelFriction.stiffness = rearLeftCollider.sidewaysFriction.stiffness;
RRwheelFriction = new WheelFrictionCurve ();
  RRwheelFriction.extremumSlip = rearRightCollider.sidewaysFriction.extremumSlip;
  RRWextremumSlip = rearRightCollider.sidewaysFriction.extremumSlip;
  RRwheelFriction.extremumValue = rearRightCollider.sidewaysFriction.extremumValue;
  RRwheelFriction.asymptoteSlip = rearRightCollider.sidewaysFriction.asymptoteSlip;
  RRwheelFriction.asymptoteValue = rearRightCollider.sidewaysFriction.asymptoteValue;
  RRwheelFriction.stiffness = rearRightCollider.sidewaysFriction.stiffness;

  if(carEngineSound != null){
    initialCarEngineSoundPitch = carEngineSound.pitch;
  }

  if(useUI){
    InvokeRepeating("CarSpeedUI", 0f, 0.1f);
  }else if(!useUI){
    if(carSpeedText != null){
      carSpeedText.text = "0";
```

```
  }
}

if(useSounds){
  InvokeRepeating("CarSounds", 0f, 0.1f);
}else if(!useSounds){
  if(carEngineSound != null){
    carEngineSound.Stop();
  }
  if(tireScreechSound != null){
    tireScreechSound.Stop();
  }
}

if(!useEffects){
  if(RLWParticleSystem != null){
    RLWParticleSystem.Stop();
  }
  if(RRWParticleSystem != null){
    RRWParticleSystem.Stop();
  }
  if(RLWTireSkid != null){
    RLWTireSkid.emitting = false;
  }
  if(RRWTireSkid != null){
    RRWTireSkid.emitting = false;
  }
}

if(useTouchControls){
  if(throttleButton != null && reverseButton != null &&
```

```
    turnRightButton != null && turnLeftButton != null
    && handbrakeButton != null){

      throttlePTI = throttleButton.GetComponent<PrometeoTouchInput>();
      reversePTI = reverseButton.GetComponent<PrometeoTouchInput>();
      turnLeftPTI = turnLeftButton.GetComponent<PrometeoTouchInput>();
      turnRightPTI = turnRightButton.GetComponent<PrometeoTouchInput>();
      handbrakePTI = handbrakeButton.GetComponent<PrometeoTouchInput>();
      touchControlsSetup = true;

    }else{
      String ex = "Touch controls are not completely set up. You must drag and drop your scene buttons in the" +
      " PrometeoCarController component.";
      Debug.LogWarning(ex);
    }
   }

}

void Update()
{
  carSpeed = (2 * Mathf.PI * frontLeftCollider.radius * frontLeftCollider.rpm * 60) / 1000;
  localVelocityX = transform.InverseTransformDirection(carRigidbody.linearVelocity).x;
  localVelocityZ = transform.InverseTransformDirection(carRigidbody.linearVelocity).z;

  if (useTouchControls && touchControlsSetup){
    if(throttlePTI.buttonPressed){
      CancelInvoke("DecelerateCar");
      deceleratingCar = false;
      GoForward();
    }
```

```
  if(reversePTI.buttonPressed){
   CancelInvoke("DecelerateCar");
   deceleratingCar = false;
   GoReverse();
  }

  if(turnLeftPTI.buttonPressed){
   TurnLeft();
  }
  if(turnRightPTI.buttonPressed){
   TurnRight();
  }
  if(handbrakePTI.buttonPressed){
   CancelInvoke("DecelerateCar");
   deceleratingCar = false;
   Handbrake();
  }
  if(!handbrakePTI.buttonPressed){
   RecoverTraction();
  }
  if((!throttlePTI.buttonPressed && !reversePTI.buttonPressed)){
   ThrottleOff();
  }
  if((!reversePTI.buttonPressed && !throttlePTI.buttonPressed) && !handbrakePTI.buttonPressed && !deceleratingCar){
   InvokeRepeating("DecelerateCar", 0f, 0.1f);
   deceleratingCar = true;
  }
  if(!turnLeftPTI.buttonPressed && !turnRightPTI.buttonPressed && steeringAxis != 0f){
   ResetSteeringAngle();
  }
 }else{
```

```
if(Input.GetKey(KeyCode.W)){
  CancelInvoke("DecelerateCar");
  deceleratingCar = false;
  GoForward();
}
if(Input.GetKey(KeyCode.S)){
  CancelInvoke("DecelerateCar");
  deceleratingCar = false;
  GoReverse();
}

if(Input.GetKey(KeyCode.A)){
  TurnLeft();
}
if(Input.GetKey(KeyCode.D)){
  TurnRight();
}
if(Input.GetKey(KeyCode.Space)){
  CancelInvoke("DecelerateCar");
  deceleratingCar = false;
  Handbrake();
}
if(Input.GetKeyUp(KeyCode.Space)){
  RecoverTraction();
}
if((!Input.GetKey(KeyCode.S) && !Input.GetKey(KeyCode.W))){
  ThrottleOff();
}
if((!Input.GetKey(KeyCode.S) && !Input.GetKey(KeyCode.W)) && !Input.GetKey(KeyCode.Space) && !deceleratingCar){
  InvokeRepeating("DecelerateCar", 0f, 0.1f);
  deceleratingCar = true;
```

```
    }
    if(!Input.GetKey(KeyCode.A) && !Input.GetKey(KeyCode.D) && steeringAxis != 0f){
      ResetSteeringAngle();
    }
  }
  AnimateWheelMeshes();
}

public void CarSounds(){
  if(useSounds){
    try{
      if(carEngineSound != null){
        float engineSoundPitch = initialCarEngineSoundPitch + (Mathf.Abs(carRigidbody.linearVelocity.magnitude) / 25f);
        carEngineSound.pitch = engineSoundPitch;
      }
      if((isDrifting) || (isTractionLocked && Mathf.Abs(carSpeed) > 12f)){
        if(!tireScreechSound.isPlaying){
          tireScreechSound.Play();
        }
      }else if((!isDrifting) && (!isTractionLocked || Mathf.Abs(carSpeed) < 12f)){
        tireScreechSound.Stop();
      }
    }catch(Exception ex){
      Debug.LogWarning(ex);
    }
  }else if(!useSounds){
    if(carEngineSound != null && carEngineSound.isPlaying){
      carEngineSound.Stop();
    }
    if(tireScreechSound != null && tireScreechSound.isPlaying){
      tireScreechSound.Stop();
```

```
    }
  }
}

public void TurnLeft(){
  steeringAxis = steeringAxis - (Time.deltaTime * 10f * steeringSpeed);
  if(steeringAxis < -1f){
    steeringAxis = -1f;
  }
  var steeringAngle = steeringAxis * maxSteeringAngle;
  frontLeftCollider.steerAngle = Mathf.Lerp(frontLeftCollider.steerAngle, steeringAngle, steeringSpeed);
  frontRightCollider.steerAngle = Mathf.Lerp(frontRightCollider.steerAngle, steeringAngle, steeringSpeed);
}

public void TurnRight(){
  steeringAxis = steeringAxis + (Time.deltaTime * 10f * steeringSpeed);
  if(steeringAxis > 1f){
    steeringAxis = 1f;
  }
  var steeringAngle = steeringAxis * maxSteeringAngle;
  frontLeftCollider.steerAngle = Mathf.Lerp(frontLeftCollider.steerAngle, steeringAngle, steeringSpeed);
  frontRightCollider.steerAngle = Mathf.Lerp(frontRightCollider.steerAngle, steeringAngle, steeringSpeed);
}

public void ResetSteeringAngle(){
  if(steeringAxis < 0f){
    steeringAxis = steeringAxis + (Time.deltaTime * 10f * steeringSpeed);
  }else if(steeringAxis > 0f){
    steeringAxis = steeringAxis - (Time.deltaTime * 10f * steeringSpeed);
  }
  if(Mathf.Abs(frontLeftCollider.steerAngle) < 1f){
```

```
    steeringAxis = 0f;
  }
  var steeringAngle = steeringAxis * maxSteeringAngle;
  frontLeftCollider.steerAngle = Mathf.Lerp(frontLeftCollider.steerAngle, steeringAngle, steeringSpeed);
  frontRightCollider.steerAngle = Mathf.Lerp(frontRightCollider.steerAngle, steeringAngle, steeringSpeed);
}

void AnimateWheelMeshes(){
  try{
    Quaternion FLWRotation;
    Vector3 FLWPosition;
    frontLeftCollider.GetWorldPose(out FLWPosition, out FLWRotation);
    frontLeftMesh.transform.position = FLWPosition;
    frontLeftMesh.transform.rotation = FLWRotation;

    Quaternion FRWRotation;
    Vector3 FRWPosition;
    frontRightCollider.GetWorldPose(out FRWPosition, out FRWRotation);
    frontRightMesh.transform.position = FRWPosition;
    frontRightMesh.transform.rotation = FRWRotation;

    Quaternion RLWRotation;
    Vector3 RLWPosition;
    rearLeftCollider.GetWorldPose(out RLWPosition, out RLWRotation);
    rearLeftMesh.transform.position = RLWPosition;
    rearLeftMesh.transform.rotation = RLWRotation;

    Quaternion RRWRotation;
    Vector3 RRWPosition;
    rearRightCollider.GetWorldPose(out RRWPosition, out RRWRotation);
    rearRightMesh.transform.position = RRWPosition;
```

```
    rearRightMesh.transform.rotation = RRWRotation;
  }catch(Exception ex){
    Debug.LogWarning(ex);
  }
}

public void GoForward(){
  if(Mathf.Abs(localVelocityX) > 2.5f){
    isDrifting = true;
    DriftCarPS();
  }else{
    isDrifting = false;
    DriftCarPS();
  }
  throttleAxis = throttleAxis + (Time.deltaTime * 3f);
  if(throttleAxis > 1f){
    throttleAxis = 1f;
  }
  if(localVelocityZ < -1f){
    Brakes();
  }else{
    if(Mathf.RoundToInt(carSpeed) < maxSpeed){
      frontLeftCollider.brakeTorque = 0;
      frontLeftCollider.motorTorque = (accelerationMultiplier * 50f) * throttleAxis;
      frontRightCollider.brakeTorque = 0;
      frontRightCollider.motorTorque = (accelerationMultiplier * 50f) * throttleAxis;
      rearLeftCollider.brakeTorque = 0;
      rearLeftCollider.motorTorque = (accelerationMultiplier * 50f) * throttleAxis;
      rearRightCollider.brakeTorque = 0;
      rearRightCollider.motorTorque = (accelerationMultiplier * 50f) * throttleAxis;
    }else {
```

```
                                        frontLeftCollider.motorTorque = 0;
                                        frontRightCollider.motorTorque = 0;
    rearLeftCollider.motorTorque = 0;
                                        rearRightCollider.motorTorque = 0;
                                }
  }
}

public void GoReverse(){
  if(Mathf.Abs(localVelocityX) > 2.5f){
    isDrifting = true;
    DriftCarPS();
  }else{
    isDrifting = false;
    DriftCarPS();
  }
  throttleAxis = throttleAxis - (Time.deltaTime * 3f);
  if(throttleAxis < -1f){
    throttleAxis = -1f;
  }
  if(localVelocityZ > 1f){
    Brakes();
  }else{
    if(Mathf.Abs(Mathf.RoundToInt(carSpeed)) < maxReverseSpeed){
      frontLeftCollider.brakeTorque = 0;
      frontLeftCollider.motorTorque = (accelerationMultiplier * 50f) * throttleAxis;
      frontRightCollider.brakeTorque = 0;
      frontRightCollider.motorTorque = (accelerationMultiplier * 50f) * throttleAxis;
      rearLeftCollider.brakeTorque = 0;
      rearLeftCollider.motorTorque = (accelerationMultiplier * 50f) * throttleAxis;
      rearRightCollider.brakeTorque = 0;
```

```
    rearRightCollider.motorTorque = (accelerationMultiplier * 50f) * throttleAxis;
   }else {
                                        frontLeftCollider.motorTorque = 0;
                                        frontRightCollider.motorTorque = 0;
    rearLeftCollider.motorTorque = 0;
                                        rearRightCollider.motorTorque = 0;
                                }
  }
}

public void ThrottleOff(){
  frontLeftCollider.motorTorque = 0;
  frontRightCollider.motorTorque = 0;
  rearLeftCollider.motorTorque = 0;
  rearRightCollider.motorTorque = 0;
}

public void DecelerateCar(){
  if(Mathf.Abs(localVelocityX) > 2.5f){
    isDrifting = true;
    DriftCarPS();
  }else{
    isDrifting = false;
    DriftCarPS();
  }
  if(throttleAxis != 0f){
    if(throttleAxis > 0f){
      throttleAxis = throttleAxis - (Time.deltaTime * 10f);
    }else if(throttleAxis < 0f){
       throttleAxis = throttleAxis + (Time.deltaTime * 10f);
    }
```

```
    if(Mathf.Abs(throttleAxis) < 0.15f){
      throttleAxis = 0f;
    }
  }
  carRigidbody.linearVelocity = carRigidbody.linearVelocity * (1f / (1f + (0.025f * decelerationMultiplier)));
  frontLeftCollider.motorTorque = 0;
  frontRightCollider.motorTorque = 0;
  rearLeftCollider.motorTorque = 0;
  rearRightCollider.motorTorque = 0;
  if(carRigidbody.linearVelocity.magnitude < 0.25f){
    carRigidbody.linearVelocity = Vector3.zero;
  }
}

public void Brakes(){
  frontLeftCollider.brakeTorque = brakeForce;
  frontRightCollider.brakeTorque = brakeForce;
  rearLeftCollider.brakeTorque = brakeForce;
  rearRightCollider.brakeTorque = brakeForce;
}

public void Handbrake(){
  isTractionLocked = true;
  RLwheelFriction.extremumSlip = 0.4f;
  rearLeftCollider.sidewaysFriction = RLwheelFriction;
  RRwheelFriction.extremumSlip = 0.4f;
  rearRightCollider.sidewaysFriction = RRwheelFriction;

  rearLeftCollider.brakeTorque = brakeForce;
  rearRightCollider.brakeTorque = brakeForce;
```

```
  if(Mathf.Abs(localVelocityX) > 2.5f){
    isDrifting = true;
    DriftCarPS();
  }else{
    isDrifting = false;
    DriftCarPS();
  }
}

public void RecoverTraction(){
  isTractionLocked = false;
  RLwheelFriction.extremumSlip = RLWextremumSlip;
  rearLeftCollider.sidewaysFriction = RLwheelFriction;
  RRwheelFriction.extremumSlip = RRWextremumSlip;
  rearRightCollider.sidewaysFriction = RRwheelFriction;
}

public void DriftCarPS(){
  if(useEffects){
    if(isDrifting){
      if(RLWParticleSystem != null && RRWParticleSystem != null){
        RLWParticleSystem.Play();
        RRWParticleSystem.Play();
      }
      if(RLWTireSkid != null && RRWTireSkid != null){
        RLWTireSkid.emitting = true;
        RRWTireSkid.emitting = true;
      }
    }else{
      if(RLWParticleSystem != null && RRWParticleSystem != null){
        RLWParticleSystem.Stop();
```

```
            RRWParticleSystem.Stop();
          }
          if(RLWTireSkid != null && RRWTireSkid != null){
            RLWTireSkid.emitting = false;
            RRWTireSkid.emitting = false;
          }
        }
      }
    }

    public void CarSpeedUI(){
      if(useUI && carSpeedText != null){
        carSpeedText.text = Mathf.RoundToInt(Mathf.Abs(carSpeed)).ToString();
      }
    }
}
```

Bibliography

- Unity Technologies. *Unity Manual – Wheel Colliders*. Unity Documentation. https://docs.unity.com/
- Unity Technologies. *Particle System Overview*. Unity Documentation. https://docs.unity.com/
- Millington, I. (2019). *Game Physics Engine Development*. CRC Press.
- Eberly, D. (2004). *Game Physics*. Elsevier.
- Gregory, J. (2018). *Game Engine Architecture*. A K Peters/CRC Press.

Ludography

- *Need for Speed: Heat* (2019). Electronic Arts.
- *Forza Horizon 5* (2021). Playground Games, Xbox Game Studios.
- *Assetto Corsa* (2014). Kunos Simulazioni.
- *Gran Turismo 7* (2022). Polyphony Digital.
- *Burnout Paradise* (2008). Criterion Games, Electronic Arts.