UNIVERSITAT JAUME·I

Final Degree Project Work Report

Development of Visual Effects For a Racing Game In Unity

Edward Alejandro Mena Ordoñez

Final Degree Project

Videogame Design and Development Degree

Jaume I University

June 27, 2025

Project made by: Edward Alejandro Mena Ordoñez

For The Moon

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my final degree project supervisor, José Ribelles Miguel, for his invaluable guidance and collaboration throughout this entire process.

I wish to thank my mother and brother for their lifelong motivation, unconditional support, and for instilling in me an unbreakable will. To my friends, for proposing the wildest ideas and best opportunities, for standing by me through both good and challenging times, and for all the adventures yet to come. To the University for giving me the extraordinary opportunity to pursue my passion through education. And to life itself, for teaching me what it truly means to live.

Finally, to you - for accompanying me during one of the most significant chapters of my life, for everything I learned by your side, and for helping me become a better person who fights to keep promises. Thank you, wherever you are.

Abstract

This Final Degree Project focuses on the development and implementation of advanced visual effects for a 3D racing game, aiming to merge a cartoon aesthetic with realistic graphical elements. The objective is to create an appealing and dynamic visual style that enhances player immersion without compromising performance.

A wide range of effects is designed—including motion blur, dynamic lighting, particle systems, and custom shaders—applied to both the car and the environment. These are implemented in Unity and approached iteratively, evaluating multiple alternatives for each effect before choosing the final method based on visual results, performance balance, and stylistic coherence. In addition to the visual effects, core game features such as arcade-style gameplay, user interface, and audio are developed to support a functional demonstration.

While performance comparison between different approaches when developing the effects will be taken into account, the decisions made throughout the project are guided by visual goals and efficiency considerations. This work represents a blend of artistic direction and technical implementation to achieve an immersive and visually cohesive racing experience.

Contents

CHAPTER 1

Technical Proposal

Index

This chapter details the technical foundation of the project, including its motivation, objectives, scope, working environment, and planning. It also presents the context in which the project was conceived and the expected results of its development.

Project Title

Development and implementation of advanced visual effects, combining a cartoon style with realistic elements to be applied in a 3D racing game and enhance its appearance

Abstract

This Final Degree Project focuses on the development and implementation of advanced visual effects for a 3D racing game, aiming to merge a cartoon aesthetic with realistic graphical elements. The objective is to create an appealing

and dynamic visual style that enhances player immersion without compromising performance. A wide range of effects is designed—including motion blur, dynamic lighting, particle systems, and custom shaders—applied to both the car and the environment. These are implemented in Unity and approached iteratively, evaluating multiple alternatives for each effect before choosing the final method based on visual results, performance balance, and stylistic coherence.

In addition to the visual effects, core game features such as arcade-style gameplay, user interface, and audio are developed to support a functional demonstration. While performance comparison between different approaches when devel- oping the effects will be taken into account, the decisions made throughout the project are guided by visual goals and efficiency considerations. This work represents a blend of artistic direction and technical implementation to achieve an immersive and visually cohesive racing experience.

Keywords

3D Video Games, Arcade Driving, Unity Tool, Visual Effects, Lighting, Shaders

1.1 Motivation

This project arises from the desire to explore the impact of visual effects in the video game development process, particularly within the racing genre. Visual effects play a decisive role in immersion, feedback, and the visual identity of a game. In racing games, where speed and environmental interaction are central, visual effects enhance the sensation of movement, surface feedback, and impact during collisions or maneuvers.

Additionally, the techniques studied and applied in this project are highly relevant to the professional field, where visual effect optimization is essential in commercial production. The combination of a cartoon aesthetic with realistic elements allows for expressive and engaging visuals without the high performance cost of full photorealism.

1.2 Objectives

The project aims to design and implement advanced visual effects that improve the game's graphical quality and immersion while maintaining real-time performance. The specific objectives include:

- Implement visual effects on the car:
  - Dust and smoke particles when skidding or braking.
  - Sparks and light bursts during collisions or when using special mechanics.
  - Progressive visual changes on the car body (e.g., dirt, wear).

  - Neon and dynamic lights to emphasize arcade style.
- Develop environmental effects:
  - Dynamic weather effects (rain, snow, lighting changes).
  - Cartoon-style post-processing to enhance colors and stylization.
  - Volumetric fog and lighting effects for atmospheric depth.
  - Animated textures on the track to simulate wear or tire marks.
  - Artificial lighting to enhance environmental realism.
  - Impact effects to highlight key gameplay moments.
- Develop a cartoon post-processing pipeline to achieve a balance between stylized and realistic visuals.
- Optimize performance to ensure the visual effects do not compromise gameplay experience.
- Develop the base systems for gameplay, UI, and sound.
- Integrate all visual effects into a functional 3D racing game.

1.3 Initial Context and Working Environment

The project is being developed individually, under the supervision of an academic tutor. It will be implemented using the Unity game engine and supported by a suite of development tools.

Development Tools

- Unity (URP pipeline) [17]
- Visual Studio Code [19]
- Blender [2]
- Photoshop [9]
- GitHub Desktop [5]
- Google Docs [6]
- Overleaf [8]
- Jira or Trello [12]

1.4 Task Planning and Timeline

This is the management and planning of the entire project. Although the development of effects and a game is something relative, I make an approximation of what I consider that I will need at least for each section, although that is more time in total than stipulated in the subject, I consider that I must dedicate the time that is necessary to be completely satisfied with all, or the great majority of the objectives, and thus to be able to present a good work.

Task	Estimated Duration
Preparation of initial documentation	20 hours
Research on visual effects and Unity tools	20 hours
Game Design Document (GDD) creation	25 hours
Development of the planned effects	30 hours
Implementation of car-related effects	20 hours
Implementation of environmental effects	30 hours
Cartoon post-processing development	30 hours
Optimization of effects	30 hours
Integration with gameplay, UI, and sound	40 hours
Final testing and adjustments	20 hours
Writing of final report and biweekly updates	50 hours
Preparation of presentation	10 hours

Table 1.1: Project schedule and workload estimate

1.5 Expected Results

- A set of advanced visual effects to enhance vehicle and environmental aesthetics.
- A cartoon-style post-processing layer integrated with realistic elements.
- An optimized system of lighting and particles that boosts immersion without compromising performance.
- All effects integrated into a functional arcade racing game.

1.6 Related Courses

- **VJ1208 - Programming II**
  This course was essential for learning the C# programming language, which is crucial for developing scripts and gameplay logic within Unity.
- **VJ1215 - Algorithms and Data Structures**
  Although this Final Degree Project focuses primarily on visual effects, some elements must be programmed in order to achieve the desired results. To accomplish this, it is necessary to consider code efficiency and optimization to preserve good performance in the final product.
- **VJ1221 - Computer Graphics**
  Knowledge from this subject will be applied to the use of shaders and rendering techniques for the creation of advanced visual effects such as dynamic lighting, reflections, distortions, and post-processing. The course also covered optimization of these effects within the game engine to maintain real-time performance.
- **Triple Project (VJ1222, VJ1223, VJ1224)**
  Including the courses: *Game Concept Design*, *Game Art*, and *Software*

Engineering. This combined project demonstrated the full process of designing, organizing, and developing a real video game. It laid the foundation for understanding what it means to create a game properly through good development practices.

- **VJ1227 - Game Engines**
  The visual effects in this project will be implemented within the game engine, ensuring integration with the rest of the systems and proper interaction with gameplay and the game environment. This course focused on understanding the inner workings of game engines (Unity in this case), making it a key subject for this project.
- **VJ1230 - Audiovisual Production: Theory and Practice**
  Post-processing techniques will be applied to improve the game's visual quality, including effects like motion blur, bloom, and chromatic aberration. Additionally, cinematic approaches will be explored to enhance the presentation of action and improve the player experience.
- **VJ1216 - 3D Design**
  The project includes the creation and optimization of 3D models used in visual effects, including environmental elements and volumetric particles. Appropriate textures and materials will also be applied to achieve the desired visual style.
- **VJ1201 - Physics**
  Although the project's focus is not strictly physical simulation, basic physical principles will be used to ensure that certain effects behave as expected under logical physical laws.

1.7 References

- Unity Technologies. Shader Graph Documentation. `https://docs.unity3d.com/Manual/ShaderGraph.html`
- Unity Technologies. Visual Effect Graph Documentation. `https://docs.unity3d.com/Packages/com.unity.visualeffectgraph@latest`
- Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., & Worley, S. (2002). *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann.
- Bruneton, E., & Neyret, F. (2008). *Precomputed Atmospheric Scattering*. ACM Transactions on Graphics (SIGGRAPH), 27(3), 1–10.
- Technical articles and academic research on visual effects in racing games, including lighting, post-processing, and particle simulation.
- Visual and technical analysis of *Trackmania*, *Asphalt 9: Legends*, and *Need for Speed: Unbound*.

CHAPTER 2

Design Document

Index

This chapter details the visual and technical design of the project. It outlines the goals of the visual effects, the artistic references, the implementation strategy, and the tools used for development and optimization.

2.1 Project Description

2.1.1 General Overview

This project focuses on the development and implementation of visual effects for a 3D racing game with a stylized aesthetic that blends cartoon elements with realistic details. The combination of these styles aims to offer an attractive and dynamic visual experience without compromising performance.

Unlike racing games with purely realistic graphics, this proposal explores an artistic approach that allows greater expressiveness in visual effects and reinforces the game's identity. This stylistic direction also allows for more flexible optimization, as advanced graphical techniques can be used without the demands of full photorealism.

To achieve these effects, technologies such as custom shaders, particle systems, and post-processing effects will be implemented within the Unity engine. Optimization will be a key aspect, ensuring that visual effects enhance the sensation of speed and immersion without compromising frame rate stability.

2.1.2 Objectives

The goal of visual effect development is to improve graphical quality and immersion. Specific objectives include:

- **Speed simulation:** Applying motion blur, visual distortions, and post-processing to intensify the sense of movement.
- **Environmental interaction:** Implementing dynamic particles simulating dust, water, and sparks in response to the vehicle's movement.
- **Advanced reflections and shading:** Using optimized techniques to represent lighting and materials with realistic finishes while maintaining the game's stylized artistic direction.
- **Optimization and performance:** Evaluating and adjusting each effect to ensure real-time viability without compromising frame rate stability.

2.1.3 Scope

The project focuses exclusively on the development and optimization of visual effects for the video game. It does not address gameplay mechanics, AI, or narrative elements in depth. The implementation of these effects includes both the vehicle and the racetrack environment to ensure coherent integration with the game's overall design.

2.1.4 Target Audience

This work is intended for developers, technical designers, and graphic artists interested in creating and optimizing visual effects for racing games. It may also serve as a reference for graphics engine programmers seeking to implement impactful visual solutions in stylized games.

2.2 State-of-the-Art Review

2.2.1 Visual Effects in Racing Games

Visual effects play a crucial role in racing games by conveying speed, impact, and dynamism. Recent advances in graphics techniques have enabled the use of increasingly realistic effects without significantly affecting performance. Commonly used effects in the genre include:

- **Motion blur:** Enhances the sensation of speed by softening moving objects and surroundings.

- **Particle effects:** Simulate elements like dust from tires, displaced water on wet surfaces, or sparks generated in collisions.
- **Advanced lighting:** Includes effects such as reflections, bloom to highlight bright lights, and dynamic shadows that add depth to the scene.

These effects are widely adopted in the genre to reinforce immersion and enhance the visual experience.

2.2.2 Reference Games

Several titles known for their strong visual presentation in the racing genre serve as reference for this project.

Trackmania

Trackmania employs a stylized visual style that optimizes lighting and speed effects to maintain a fluid experience. Key effects include:

- Motion blur and vehicle trails to enhance the feeling of speed.
- Preprocessed lighting to ensure visual consistency without sacrificing performance.
- Use of vibrant colors and strong contrast to reinforce the game's arcade aesthetic.

Figure 2.1: Trackmania Image (2020)

Asphalt 9: Legends

This title features a striking aesthetic with intense post-processing effects that enhance visual impact. Its main visual features include:

- Dynamic reflections on vehicles and asphalt for a realistic feel.
- Particle effects such as smoke, sparks, and dust during drifts or collisions.
- Bloom and selective blur to highlight lights and particle explosions.

Figure 2.2: Asphalt 9: Legends Image (2023).

Need for Speed: Unbound

This game combines a hybrid visual style mixing realism and cartoon elements, emphasizing visual effects through a stylized lens. Highlights include:

- Aggressive motion blur and speed lines to heighten the sense of speed.
- Stylized particles and exaggerated visual effects during drifts or nitro boosts.
- Use of hand-drawn outlines and animations to strengthen its unique visual identity.

Figure 2.3: Need for Speed: Unbound image (2022)

These references serve to define the artistic direction and visual techniques used in this project.

2.3 Development and Implementation

2.3.1 Implementation Techniques

Visual effects in modern games are created using a combination of advanced graphics techniques. Relevant approaches include:

- **Custom shaders:** Modify material appearance in real time to create reflections, advanced shading, and distortions.
- **Post-processing:** Apply effects after rendering, such as bloom, motion blur, and color grading.
- **Particle systems:** Simulate dust, smoke, sparks, and other environmental elements.
- **Dynamic lighting:** Improve the perception of depth and realism in the scene.

2.3.2 Challenges and Optimization

One of the main challenges of advanced visual effects is maintaining performance, especially in fast-paced racing games. Strategies include:

- **Dynamic Level of Detail (LOD):** Reduce effect complexity as their relevance on screen decreases.
- **Efficient GPU usage:** Use optimized graphics calculations to minimize frame rate impact.
- **Selective rendering techniques:** Apply effects only to key elements to avoid unnecessary overhead.

2.3.3 Development Approach

The development process follows an iterative approach where each technique is researched, implemented, and optimized in distinct phases:

1. **Research and analysis:** Study the visual effect techniques used in reference games and assess their feasibility.
2. **Prototyping and experimentation:** Create small prototypes to evaluate visual impact and performance.
3. **Implementation:** Integrate selected effects into a test environment within the game engine.
4. **Optimization:** Adjust parameters, reduce performance cost, and test under different conditions.
5. **Validation and testing:** Evaluate visual impact and measure performance across different devices.

2.3.4 Developed Visual Effects

Motion Blur **Purpose:** Simulate the sensation of speed by blurring moving elements.

Techniques:

- Shader-based implementation through post-processing.
- Use of velocity vectors to determine blur direction and magnitude.

Optimization:

- Apply selectively to the most relevant objects on screen.
- Reduce samples on low-end devices.

Particle Effects (Smoke, Sparks, Dust) **Purpose:** Improve the perception of impact, drifting, and interaction with the environment.

Techniques:

- Unity's Particle System (for simple effects) or Visual Effect Graph (for advanced, GPU-optimized effects).
- Collision configuration for interaction with ground and objects.

Optimization:

- Reduce particle count based on camera distance.
- Use optimized textures instead of complex models.

Dynamic Reflections **Purpose:** Enhance the visual perception of the environment reflected on vehicle surfaces.

Techniques:

- Use of cubemaps and real-time reflections.
- Screen Space Reflections (SSR) on specific surfaces.

Optimization:
- Render reflections at lower resolution.
- Combine pre-rendered maps with real-time reflections (hybrid system).

Lighting **Purpose:** Improve light perception in the scene, including atmospheric effects.

Techniques:
- Use of lights for fog, dust, or neon elements in urban environments.
- Dynamic shadows to better integrate objects into the scene.

Optimization:
- Bake lighting for static elements.
- Use lights only in key areas of the track.

Custom Materials and Special Effects **Purpose:** Create specific materials and shaders for unique visual effects like holograms, impacts, and screen-space distortions.

Techniques:
- Custom shaders with animated noise and transparency for hologram effects.
- Dynamic decals to project cracks or sparks.
- Screen-space distortion shaders for shockwave effects.
- Emissive materials with animated gradients to simulate energy buildup.

2.4 Tools and Technologies Used

2.4.1 Programming Languages

- **HLSL (High-Level Shader Language):** Used to develop custom shaders for dynamic reflections, shading, and distortion effects.
- **C#:** Used to control shader parameters, particle systems, and post-processing behavior in Unity.

2.4.2 Shader and Particle Tools

- **Shader Graph:** A visual tool for creating stylized, parameter-driven shaders.
- **Visual Effect Graph (VFX Graph):** Used for GPU-accelerated particles, such as smoke, sparks, and collision-based effects.

2.5 Validation Strategy

To ensure the effectiveness and feasibility of the implemented visual effects, a validation process will be carried out based on systematic testing. These tests

focus on ensuring that the effects meet visual quality and performance expectations, maintaining game stability across different hardware configurations.

2.5.1 Visual Quality Impact

The purpose of this evaluation is to verify that the visual effects enhance the game's aesthetics without compromising visual coherence or scene readability.

- **Comparison with visual references:** Graphic references will be established based on visual effects used in similar racing games or titles with a hybrid (cartoon-realistic) aesthetic.
- **User perception evaluation:** Surveys or user tests will be conducted to analyze subjective reactions to the implemented effects, assessing their impact on immersion and gameplay.
- **Stylistic consistency check:** Each effect's integration with the overall scene will be reviewed to avoid visual inconsistencies (e.g., color or lighting mismatches).

2.5.2 Performance Analysis

To ensure that the visual effects do not negatively affect gameplay, performance metrics will be measured under different game scenarios.

- **Resource usage measurement:**
  - GPU load (shader performance, rendering pipeline cost).
  - CPU load (particle processing, post-processing calculations).
  - RAM and VRAM usage, ensuring that effects remain within the memory budget for mid- and low-range hardware.
- **Frame rate (FPS):** Tests will be conducted under various gameplay conditions (open tracks, effect-dense areas, multiple collisions) to evaluate frame rate stability.
- **Bottleneck analysis:** Unity profiling tools (Profiler and Frame Debugger) will be used to detect performance issues and optimize when necessary.

2.6 Compatibility and Scalability

Since the game must maintain stable performance across different hardware, basic quality settings will be implemented to adapt visual effects according to device capacity.

- **Quality modes:**
  - **High quality:** Enables all visual effects with optimal configuration.
  - **Reduced quality:** Lowers particle density and shader complexity to improve performance on low-end hardware.

- **Dynamic adjustments:** Effects such as particle count or screen resolution will be adjusted based on system load.
- **Cross-device testing:** The game will be tested on hardware of varying performance levels to ensure stability.
- **User configuration options:** Players will have the option to enable/disable certain visual effects such as motion blur or lens flares, which may cause discomfort or motion sickness in some cases.

2.6.1 Gameplay Context Validation

It is essential that the visual effects not only look attractive but also contribute positively to gameplay without interfering with the experience.

- **Readability at high speed:** Motion blur and particle effects will be evaluated to ensure they do not reduce visibility of the track or essential elements.
- **Impact and feedback effects:** Effects related to collisions, drifting, and interaction with the environment will be analyzed to verify that they communicate the correct feedback without being overly distracting.
- **Input response time:** The impact of visual effects on input latency and game fluidity will be measured.
- **Visual assists:** It will be ensured that visual effects enhance and support gameplay rather than obstruct it.

2.7 Evaluation and Results

2.7.1 Evaluation Methodology

To assess the effectiveness and viability of the implemented visual effects, tests will be carried out focusing on the following aspects:

- **Visual quality:** Comparison with visual references and subjective user feedback.
- **Performance:** Measuring the impact on frame rate (FPS), GPU and CPU usage.
- **Compatibility:** Testing across different hardware configurations to evaluate stability and scalability.

The tests will be performed under various conditions in order to identify potential bottlenecks and optimize the visual effects accordingly.

CHAPTER 3

Development and Implementation

Index

3.1 Introduction to Visual Effects Development in Unity

The visual effects presented in this project are based on Unity's GPU-powered rendering tools, primarily the Visual Effect Graph (VFX Graph) and Shader Graph, both under the Universal Render Pipeline (URP). These systems enable high-performance simulations, stylized particle rendering, and efficient integration with gameplay elements.

Rather than applying pre-built assets, the development process involved exploring Unity's VFX architecture from a technical perspective. This included investigating how data flows through the VFX Graph contexts, integrating Shader Graph-based materials, and controlling particle behavior through script-triggered properties.

For a complete description of these systems, their internal architecture, and the decision-making process behind their use, see Annex A.

3.2 Approach to Visual Effects in Unity

Unity offers multiple systems for developing visual effects, each with distinct advantages depending on the project's goals and technical constraints. This project evaluated the main options—Shuriken (CPU-based), VFX Graph (GPU-based), Shader Graph, and post-processing—based on their scalability, flexibility, and suitability for stylized effects.

After a comparative analysis and extensive testing, the VFX Graph was chosen as the primary system, due to its performance on modern hardware and its compatibility with URP. Shader Graph was used to create custom materials and define advanced visual behaviors, often bound directly to simulation parameters.

An extended discussion on the differences between Shuriken and VFX Graph, as well as Shader Graph integration, is available in Annex A.

3.3 Visual Effects Implementation

This section presents the visual effects developed throughout the project in a structured and detailed manner. Each effect is analyzed independently, highlighting the purpose it serves within the gameplay experience, the technical considerations behind its implementation, and the decision-making process that shaped the final result.

The project adopts an iterative and experimental approach to visual effects, aiming not only for aesthetic appeal but also for technical soundness, performance awareness, and educational value. Wherever applicable, multiple implementation paths are explored and discussed—from traditional C# scripting to modern GPU-based systems such as the Visual Effect Graph (VFX Graph) and Shader Graph.

Before delving into the detailed implementation of each effect, Table 3.1 provides an overview of all the visual effects included in the project. It summarizes the technology stack used in each case and the intended purpose or gameplay function of the effect. This serves both as a guide for the reader and a reference point for the technical diversity of the implementations that follow.

Effect Name	Technology Used	Purpose
Skidmarks	C# + Mesh Generation	Visual feedback for tire slip and drifting
Skid Smoke	VFX Graph + Shader Graph	Stylized smoke from drifting or wheelspin
Exhaust Nitro Flame	VFX Graph + Flipbook Texture	Boost effect indicating nitro activation
Brake Trail	VFX Graph (Particle Strip)	Luminous trail synced with nitro
Checkpoint Hologram	Shader Graph + Script	Animated hologram to mark active checkpoint
Finish Line Shader	Shader Graph + Custom Texture	Stylized goal indicator with scrolling pattern and holographic effect
Respawn Sphere Shader	Shader Graph (Alpha Clip + Hexagonal Dissolve)	Dynamic spawn/respawn effect with geometric disintegration
Motion Blur	URP Post-Processing	Enhanced speed perception during fast movement
Stylized Post-Processing	URP Volume + Shader Graph	Cartoon look (bloom, outline, color grading)
Dynamic Camera Effects	Cinemachine + Script	FOV shift, lens distortion, screen shake during nitro
Lens Flare	URP Post-Processing	Stylized light reflections synced with environment
Chromatic Aberration	URP Post-Processing (Script-controlled)	Color distortion effect during high-speed events
Collision Sparks	VFX Graph + Script	Impact feedback on hard collisions
Neon Lighting	Material Processing	Glow under the car

Table 3.1: Updated summary of implemented visual effects.

The structure of each subsection follows a consistent format: the visual goal is first introduced, followed by a breakdown of the conceptual approach, implementation details, integration with game logic, performance considerations, and a final evaluation of the results obtained. In specific cases, discarded or alternative approaches are also included, providing insight into the problem-solving and learning process behind each solution.

3.3.1 Skidmarks

1. Objective and Purpose

This effect was designed to simulate the visual trace left on the ground when a vehicle experiences tire slippage, especially during drifting or heavy braking.

It reinforces the feeling of physical contact between the car and the terrain and contributes to a more grounded and responsive driving experience in an arcade-style context.

2. Conceptual Exploration

Unlike other effects in the project that rely on GPU-based simulations via Visual Effect Graph (VFX Graph), this effect is implemented entirely using custom C# scripts and mesh generation. After researching various options—including projected decals, screen-space trails, and GPU-based particles—a decision was made to develop a mesh-based trail system manually. This approach ensured high accuracy, persistence, and full control over the geometry.

While a similar result could have been approximated using VFX Graph with a particle trail output, this method was ultimately discarded for this specific case due to several limitations:

- VFX Graph trails require GPU-based particle emitters and are less suitable for precise ground alignment.
- Accurate collision handling with terrain or roads would require complex sampling (e.g., heightmaps or depth textures).
- Mesh-based generation allowed for direct control over fading, width, and performance management via code.

This makes skidmarks a unique exception within the project's otherwise GPU-based visual effect framework.

3. Implementation Process

The system is implemented using two key scripts:

- `WheelSkid.cs`: Attached to each wheel, it detects slip based on lateral velocity and the difference between vehicle speed and wheel RPM. If thresholds are exceeded, it calculates a world-space position for the mark.
- `Skidmarks.cs`: Responsible for maintaining a dynamic mesh by adding quad segments as the car skids. Each segment stores position, normal, tangent, UVs, and alpha.

The mesh is updated per frame and capped at 2048 segments. A simple transparent shader is used to fade marks naturally into the road surface. Lighting and shadows are disabled to reduce rendering cost.

Shader snippet:

```
Blend SrcAlpha OneMinusSrcAlpha
ZWrite Off
Alphatest Greater 0
```

Technical features:

- Follows terrain normal accurately using raycast info.
- Marks blend smoothly using alpha.
- Very lightweight and scalable.

4. VFX Graph Alternative (Discarded)

A visual prototype using VFX Graph was built as a test, simulating the skid trail using continuous particle emission and trails. However, this method was not adopted due to the following reasons:

- Lack of precise control over ground positioning.
- Complexity in maintaining consistent width and alignment.
- Overhead from simulating particles when a mesh-based approach could be simpler and cheaper.

5. Performance Considerations

- Mesh is preallocated and reused, avoiding allocations.
- Mesh updates only occur when skid state is active.
- No post-processing or lighting overhead.
- Shadow casting and lighting are disabled in the shader.

6. Result and Evaluation

The final implementation is extremely efficient and highly accurate. It visually supports the player's actions and adds a tangible layer of realism to the driving feel. Despite not leveraging VFX Graph, the effect still benefits from careful design and performance optimization.

It stands out as an instructive case where GPU-based simulation was not the most suitable tool, and where a traditional code-driven approach provided more direct control and better results for the intended use case.

3.3.2 Skid Smoke

1. Objective and Purpose

This effect simulates the smoke generated by the tires during high lateral slippage or excessive wheelspin. It serves both as an artistic element that reinforces the arcade aesthetic and as a gameplay feedback mechanism, visually communicating tire loss of grip to the player.

2. Conceptual Exploration

Unlike the physical skidmarks, which are based on mesh generation, this effect was implemented entirely using GPU-based systems: **Visual Effect Graph** for particle simulation and **Shader Graph** for custom stylized rendering.

From the beginning, the intention was to produce a dynamic and stylized look—more expressive than realistic—using animated shaders, volumetric turbulence, and smooth emission control. The smoke needed to feel light, cartoonish, and responsive to gameplay physics.

To achieve this, a procedural material with Voronoi-based distortion, Fresnel falloff, and alpha control was developed using Shader Graph and then integrated with a VFX Graph setup that exposes key parameters such as color, spawn rate, and vehicle speed.

3. Implementation Process

The effect consists of three main components:

Visual Effect Graph:

- **Spawn Context**: Uses a constant spawn rate controlled by a float property exposed to scripts.
- **Initialize Particle**:
  - Lifetime is randomly chosen between 0.8s and 1.2s.
  - Velocity is based on the car's movement direction.
  - Random rotation is applied across all axes.
- **Update Particle**:
  - Applies turbulence using relative noise to simulate swirling motion.
  - Alpha and size evolve over time using curves and gradients.
- **Output Particle**:
  - Uses a custom Shader Graph-based unlit material called `SmokeShader`.
  - Particle size and color are modulated over life using exposed parameters.

Figure 3.1 shows the VFX structure for the stylized smoke material.

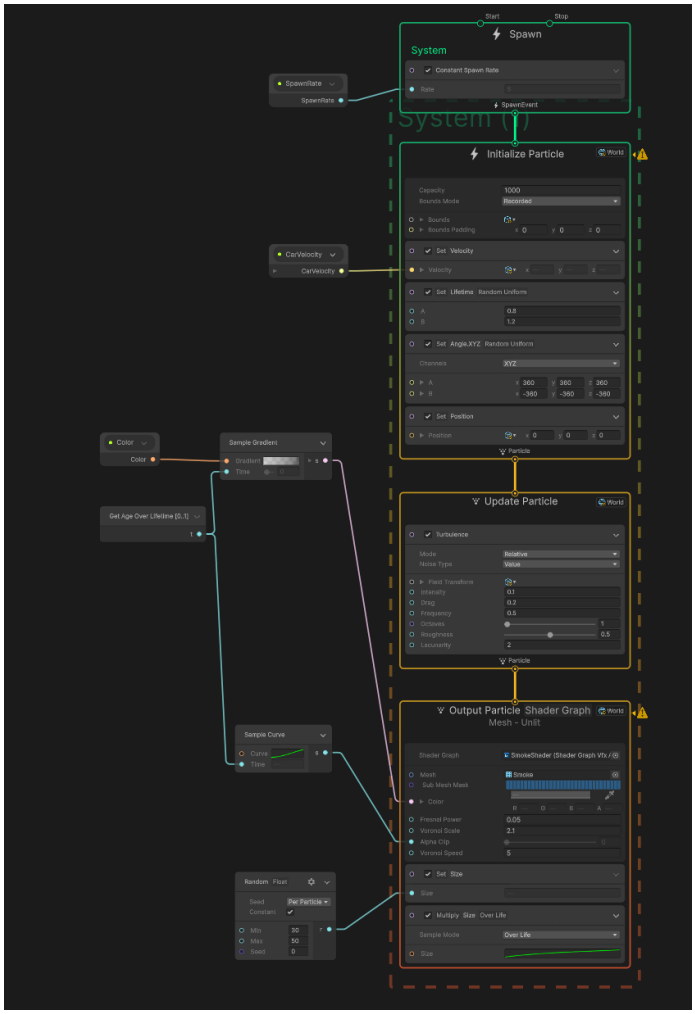


Figure 3.1: Skid Smoke Visual Effect Graph setup.

Shader Graph: Figure 3.2 shows the node structure for the stylized smoke material.

- Uses a **Fresnel Effect** to enhance edge fade.
- Includes animated **Voronoi noise** to simulate smoke dispersion.
- Alpha clipping and base color are driven by VFX Graph via properties.
- Exposed parameters: `FresnelPower`, `VoronoiScale`, `AlphaClip`, `VoronoiSpeed`.

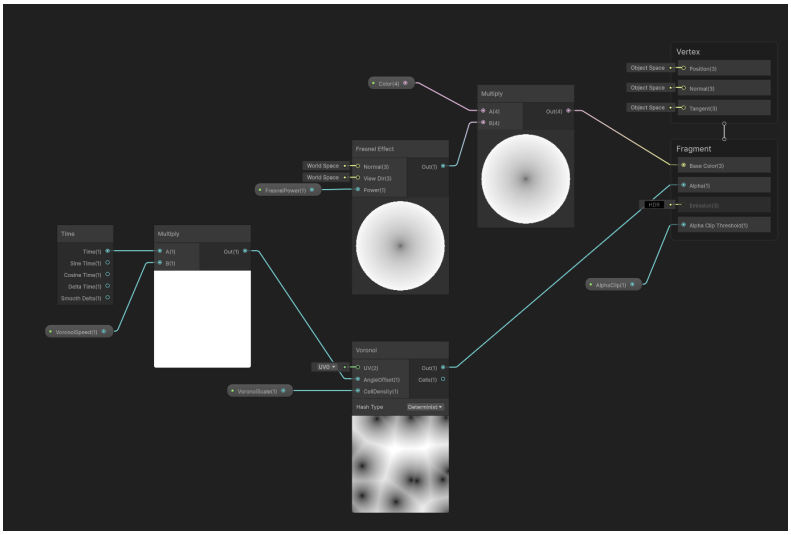


Figure 3.2: Custom Shader Graph material for stylized smoke (SmokeShader).

C# Script Integration: The `WheelSmoke.cs` script is attached to each wheel and evaluates real-time physics data to determine whether smoke should be emitted. The script:

- Computes lateral slip and wheelspin intensity.
- Calculates a spawn rate proportional to slip intensity.
- Positions the emitter slightly above the contact point.
- Starts and stops the VFX Graph based on thresholds.

The script ensures synchronization between the physical behavior of the car and the visual feedback provided by the smoke particles.

4. Parameter Control and Gameplay Integration

The following parameters are exposed and dynamically updated by the script:

- `SpawnRate`: Controls emission frequency based on real-time intensity.
- `CarVelocity` and `CarSpeed`: Drive the directional behavior and lifetime effects.
- `Color` (optional): Allows future tuning based on terrain or state.

Smoke is only emitted when slip intensity exceeds a defined threshold. This prevents visual overload and ensures that the effect only appears when relevant.

5. Performance Considerations

- Particle count is capped and controlled through spawn rate.
- Noise functions use GPU-friendly values with minimal cost.
- Shader Graph materials avoid lighting and shadows to reduce rendering load.

- Turbulence uses low-frequency relative noise to simulate swirling without overdraw.

6. Result and Evaluation

As shown in Figure 3.3, the effect is visually dynamic and communicates slippage clearly to the player without overwhelming the scene. The integration of Shader Graph enables a stylized, cartoon-like aesthetic consistent with the rest of the project.

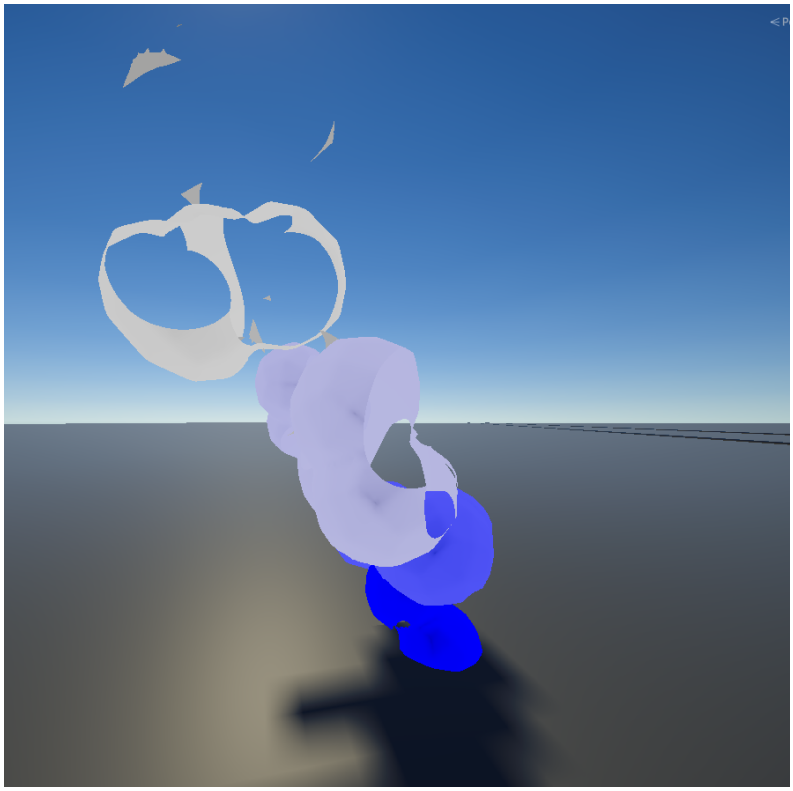


Figure 3.3: Final look of the smoke VFX.

Potential future improvements include:

- Color adaptation based on ground type or tire state.
- More advanced noise layering for added smoke realism.
- LOD control for reducing spawn rate at distance.

3.3.3 Exhaust Nitro Flame

1. Objective and Purpose

This visual effect represents the short bursts of flame and smoke emitted from the vehicle's exhaust during nitro activation. It serves both as a stylistic signa-

ture element and as immediate visual feedback for the use of boost mechanics, enhancing the sense of power and speed.

The effect was designed with modularity in mind: it can be reused and recolored to fit different vehicles or gameplay conditions, and supports future integration with stylized visual shading.

2. Conceptual Exploration

The effect was conceptualized as a layered composition of four distinct particle systems, each contributing a unique visual component: a central flame beam, stylized flame textures, layered smoke, and dynamic sparks.

Instead of combining all visuals into a single output, each layer was implemented as an independent system within the same VFX Graph, allowing for fine-tuned control over behavior, appearance, and blending modes. This modular architecture improves scalability and aesthetic flexibility, while maintaining performance by sharing properties like emission rate and simulation space.

3. Implementation Process

All layers use a **constant spawn rate of 32 particles per second** and a **capacity of 256 particles**. Simulation is done in `Local` space, with each system sharing the same core structure:

- **Initialize Particle**:
  - Set Position uses a local target and updates in world space to maintain physical consistency.
  - Velocity is randomly distributed per component.
  - Lifetime is randomized within a small range.
- **Update Particle**:
  - Velocity is modulated over life using a curve.
- **Output Particle (URP Lit)**:
  - Most systems orient to the camera or velocity.
  - Size is randomized and scaled over life.
  - Color evolves over life using a gradient.

Attributes used across systems:

- `SoftValue` – Controls particle fade distance (soft particles).
- `SmokeColor` – Gradient for smoke layers.
- `FlameColor` – Gradient for flame visuals.
- `FlameText` – 2D texture flipbook with 4 frames used for animated flames.

Each of the four systems serves a dedicated visual function:

Flame Beam This is the core visual of the effect, producing a bright central flame trail using Unity's default particle texture. It uses:

- Additive blending mode for intense glow.
- Orientation facing the camera.
- No texture animation.

Flame Tex A stylized flame layer using a 4-frame flipbook:

- Rotation is randomized with ‘Set Angle XYZ Uniform‘ to introduce visual variation.
- UV Mode is set to `Flipbook` with blending enabled.
- Alpha blend mode is used to layer with other effects.
- ‘Set Tex Index‘ is animated over particle life for flipbook cycling.

Smoke Tex Similar to Flame Tex, but using different velocity and color (from `SmokeColor`) to simulate trailing smoke with offset movement. This provides visual separation from the main flame and adds volume.

Sparks This layer simulates fast, short-lived sparks:

- Uses a circular spawn shape to emit sparks radially.
- Particles are elongated via ‘Set Scale XYZ‘ to resemble streaks.
- Velocity and lifetime are faster and shorter than other layers.
- Orientation is aligned with velocity.

Figure 3.4 shows the node structure for the effect.

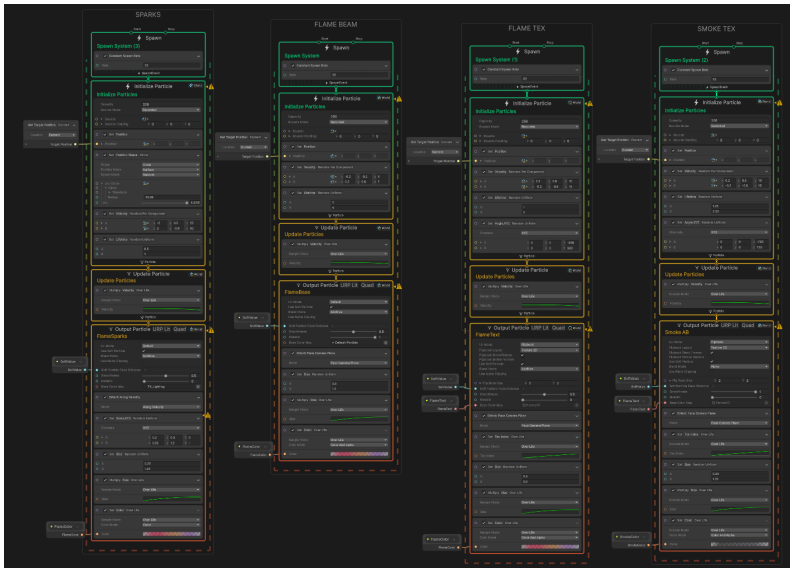


Figure 3.4: Flame Visual Effect Graph setup.

4. Parameter Control and Gameplay Integration

The effect is configured to respond to in-game boost state via script, though the current implementation is visually self-contained. The following parame-

ters can be adjusted externally:

- Color gradients for both flame and smoke.
- Flipbook texture index (optional script-driven animation).
- Emission rate, duration, and position offset.

Its modularity allows for easy reuse and tuning depending on the desired nitro duration or intensity.

5. Performance Considerations

- Shared capacity limits across systems prevent uncontrolled particle generation.
- Particle size and lifetime are optimized for short bursts.
- Flipbook animation uses only 4 frames to minimize texture bandwidth.
- Optionally, Shader Graph can be introduced later for GPU-side effects like Fresnel or Voronoi distortion, as done with the skid smoke effect.

6. Result and Evaluation

As shown in Figure 3.5, the final result is a stylized, multi-layered flame burst that matches the visual identity of the project. Despite being implemented entirely within the Visual Effect Graph, its layered architecture gives it the flexibility of more complex systems.

Figure 3.5: Final Flame VFX.

Future improvements may include:

- Dynamic variation in color or behavior based on nitro duration or engine temperature.
- Integration with custom Shader Graph materials for animated stylization or toon shading.
- Linking particle timing to audio or engine revs.

3.3.4 Brake Trail on Nitro Activation

1. Objective and Purpose

This effect enhances the visual impact of nitro activation by generating a luminous trail behind the brake lights of the vehicle. The intention is to reinforce the arcade-style identity of the project and to evoke references to stylized racing games such as *Need for Speed: Unbound*, where exaggerated visual elements are used to amplify gameplay feedback.

The trail effect gives the impression that the brake lights momentarily leave a glowing residue behind them, suggesting a burst of energy and velocity.

2. Conceptual Exploration

The brake trail was designed as a stylized accent to the previously developed exhaust flame effect. Both effects are visually synchronized to form a coherent boost experience.

To achieve the desired visual, a particle strip system was chosen. The effect consists of stretched quads aligned behind the lights, with a gradient-based color transition and additive blending. The particle strip behaves as a light ribbon, fading over time and matching the shape and movement of the vehicle.

3. Implementation Process

The effect was implemented using a single VFX Graph configured as follows (Shown in Figure 3.6):

- **Spawn Context:**
  - Constant rate: 32 particles/second.
- **Initialize Particle Strip:**
  - Strip count: 1.
  - Capacity: 256 particles per strip.
  - Lifetime: fixed at 1 second.
  - Position: `Get Current Position (Local)`, since the effect is placed in World space.
- **Output Particle Strip (URP Unlit):**
  - Size: initial value of 0.1.
  - Multiply Size over Life.
  - Set Scale XYZ with Y = 2.05 to match brake light dimensions.
  - Set Color over Life using a red gradient.
  - Tiling mode: `Stretch`.
  - Color mapping: `Gradient Mapped`.
  - UV mode: `Scale and Bias`, scale set to (1, 1).
  - Blend mode: `Additive`.
  - Main texture: uniform white trail image.

4. Control and Synchronization

The brake trail effect is activated and deactivated via script, in sync with the nitro system. The same script used for the exhaust flame effect also manages the state of this trail effect, ensuring that both begin and end at the same time. This ensures visual consistency and a unified feedback system.

5. Performance Considerations

- Particle count is limited and lifetime is short.
- No forces or collisions are applied.
- Uses unlit shader with additive blending for minimal GPU load.
- Texture and gradient are lightweight and reused.

6. Result and Evaluation

The result shown in Figure 3.7 is a bright, elegant trail that visually connects the brake lights to the direction of motion, enhancing the sense of momentum. The style pays homage to high-energy visuals from modern arcade racers, such as *Need for Speed: Unbound*, and contributes to the exaggerated, cartoon-inspired aesthetic of the project.
A possible enhancement in future iterations includes:

- Integration of stylized Shader Graph materials with glow, Fresnel or distortion.
- Dynamic trail length based on car speed or nitro duration.
- Color shifting based on energy level or vehicle type.

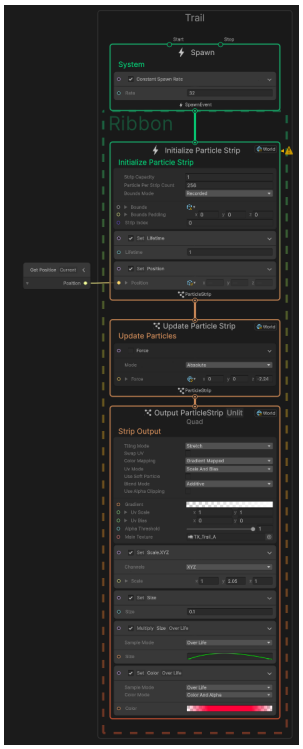


Figure 3.6: The brake trail VFX Graph configuration visualized in the Unity Editor.

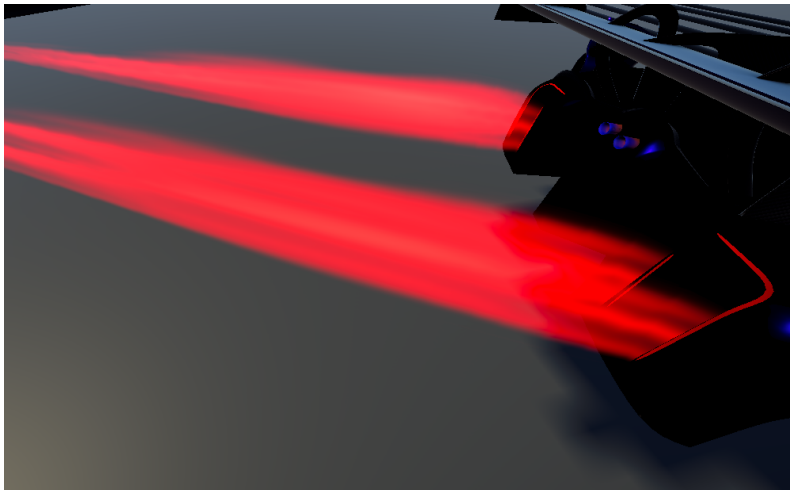


Figure 3.7: Final Brake Trail VFX.

3.3.5 Checkpoint Hologram Shader

1. Objective and Purpose

This shader creates dynamic patterns for checkpoint markers, generating visual interest through interfering animated stripes. Designed for clear player recognition for checkpoints, it serves as both a gameplay indicator and stylistic element consistent with the project's technical aesthetic.

2. Technical Basis

The effect leverages optical interference principles through:

- Two layered stripe patterns with independent parameters
- Real-time animation via shader graph time manipulation
- Minimal texture dependencies (fully procedural)

3. Implementation Process

The Shader Graph implementation (Figure 3.8) consists of:

- **Dual Stripe Systems**:
  - Primary stripes (`Stripe1Frequency/Thickness`)
  - Secondary crossing stripes (`Stripe2Frequency/Thickness`)
  - Combined via Multiply node for interference effect
- **Pattern Animation**:
  - Horizontal movement controlled by `LineSpeed`
  - Rotation modulated via Sine node and `LineRotationSpeed`
  - Time-based offset calculation:

```
offset = Time * LineSpeed * (1 + sin(Time * 0.5))
```

- **Visual Integration**:
  - Fresnel edge blend with `FresnelStrength` control
  - Alpha channel modulation for smooth transitions
  - Emissive output for HDR visibility

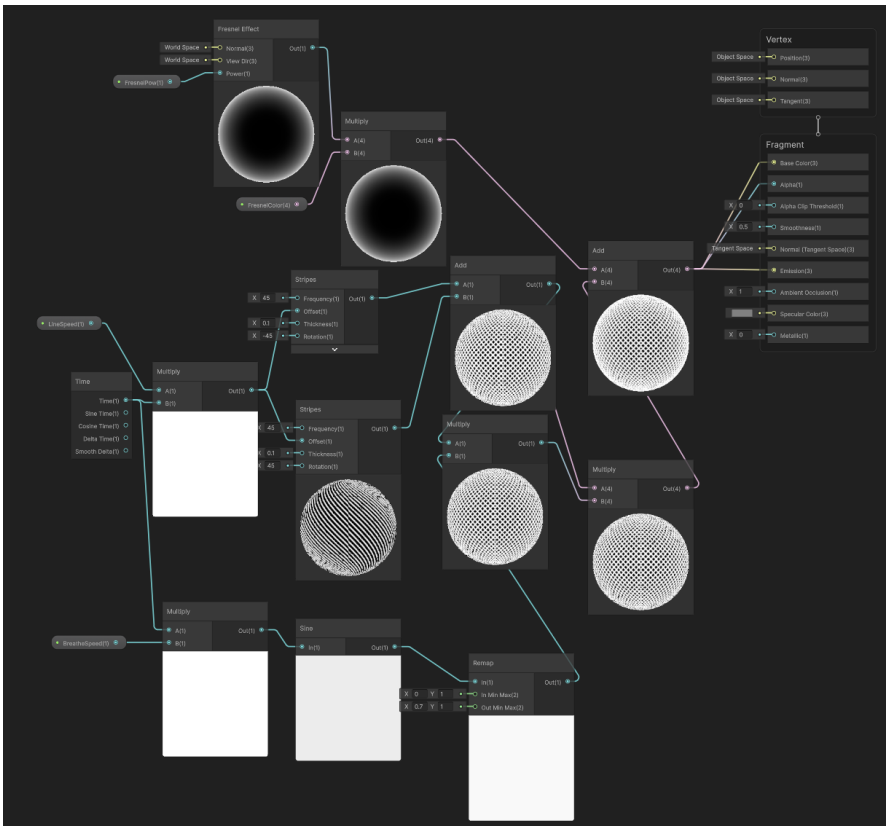


Figure 3.8: Shader Graph implementation showing: (A) Dual stripe generators, (B) Animation control system, (C) Output composition nodes.

4. Parameter Specifications

Parameter	Range	Function
`LineSpeed`	0-5	Base scroll velocity
`LineRotationSpeed`	0-2	Pattern rotation rate
`FresnelStrength`	0-1	Edge glow intensity
`Stripe1/2Frequency`	0.1-10	Lines per unit
`Stripe1/2Thickness`	0.01-0.5	Line weight

5. Performance Characteristics

- 100% procedural (no texture sampling)

6. Visual Output

The resulting effect (Figure 3.9) demonstrates:

- Clear moiré pattern from stripe interference
- Smooth edge blending via Fresnel
- Consistent visibility across viewing angles

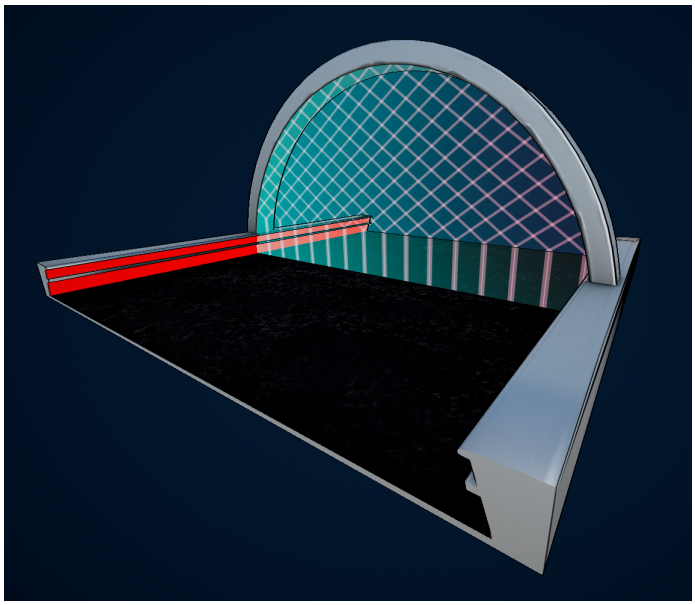


Figure 3.9: Final checkpoint appearance showing animated stripped pattern and holographic effect.

Technical Notes:

- All parameters are clamped to prevent artifact generation
- Uses world-space coordinates for consistent scaling
- Alpha clipping disabled for transparency blending

3.3.6 Finish Line Shader

1. Objective and Purpose

The finish line shader creates a dynamic, animated pattern that visually distinguishes race completion points. Designed for high visibility and stylistic coherence with the project's arcade aesthetic, it combines stripes with texture-based patterns to produce a distinctive marker that responds to gameplay events.

2. Conceptual Exploration

The implementation merges two complementary approaches:

- Stripe generation for consistent, performance-efficient patterns

- Texture-based details for artistic control

Reference influences include:

- Racing game finish line effects
- Sci-fi holographic displays
- Motion graphics principles

3. Implementation Process

The Shader Graph implementation (Figure 3.10) utilizes these key systems:

- **Animated Stripes**:
  - Custom stripe pattern with controllable frequency (`LineFrequency`)
  - Thickness modulation via `LineThickness` parameter
  - Real-time animation using `Time` node and `ScrollSpeed`
- **Texture Layering**:
  - Checkerboard texture with independent `FinishSpeed` control
  - Linear Dodge blending for additive effects
  - UV distortion based on player proximity
- **Edge Enhancement**:
  - Fresnel effect scaled by `FresnelStrength`
  - Normal-based rim lighting
  - Camera-angle responsive opacity

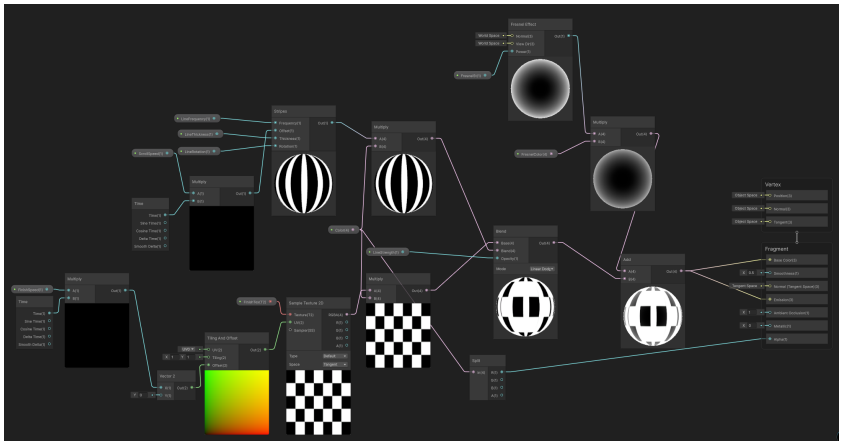


Figure 3.10: Shader Graph architecture showing: (A) Stripe generation system, (B) Texture blending network, (C) Fresnel enhancement setup.

4. Parameter Control

The effect is modulated through these exposed properties:

Parameter	Type	Function
`LineFrequency`	Float (0.1-5)	Stripe density
`LineThickness`	Float (0.01-0.5)	Pattern weight
`ScrollSpeed`	Float (0-10)	Base animation rate
`FinishSpeed`	Float (0-5)	Checkerboard motion
`FresnelStrength`	Float (0-2)	Edge glow intensity

5. Performance Optimization

- Single texture sample (checkerboard) with additional effects enhancement

6. Visual Analysis

As demonstrated in Figure 3.11, the shader provides clear visual feedback to the user that make a difference between a standard checkpoint and the finish line:

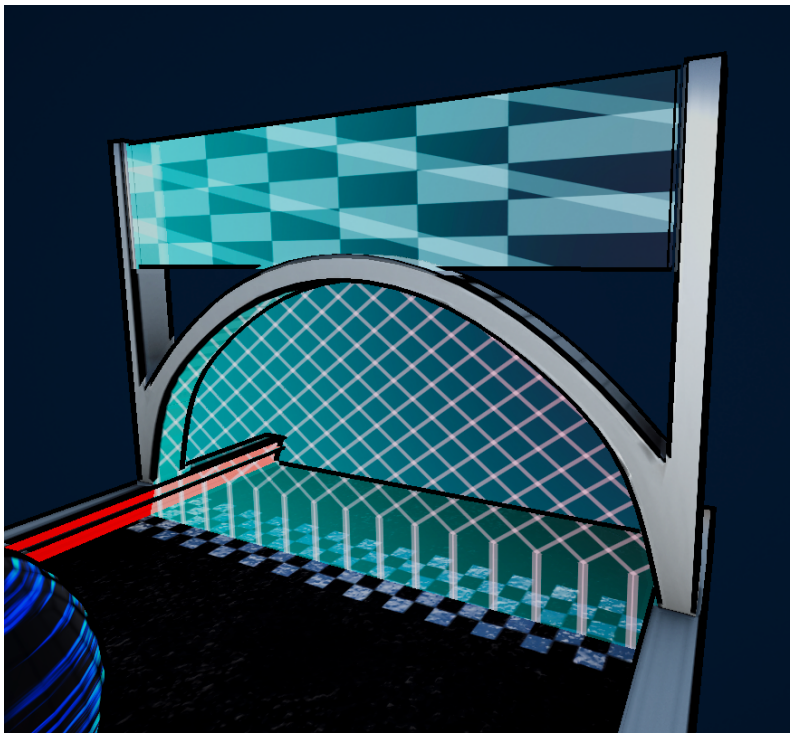


Figure 3.11: Final VFX.

Future enhancements could incorporate:

- Lap count visualization
- Audio-reactive parameter modulation

3.3.7 Respawn Sphere Shader

1. Objective and Purpose

The respawn sphere effect creates a dynamic geometric disintegration pattern during player spawn/respawn sequences, using a stylized hexagonal dissolve effect with glowing edges. This serves as both a visual transition mask and a signature sci-fi aesthetic element, reinforcing the project's arcade-style visual identity.

2. Conceptual Exploration

The implementation combines holographic de/re-materialization techniques with game-optimized shader effects, drawing inspiration from sci-fi references while maintaining real-time performance. Key visual goals included:

- Clear communication of spawn/respawn state changes
- Seamless integration with the vehicle model

3. Implementation Process

Developed in Shader Graph with URP, the effect uses these technical components (Figure 3.12):

- **Fresnel Edge Glow**:
  - Calculates view-dependent surface angles
  - Applies exponential falloff controlled by `FresnelPower`
  - Multiplies with emissive color for atmospheric rim lighting
- **Procedural Surface Animation**:
  - Animated Simple Noise texture with `NoiseScale` and `NoiseSpeed` control
  - World-space UV mapping for consistent movement
  - Vertex displacement for pulsation effect
- **Hexagonal Dissolve**:
  - Alpha clipping with branching logic for clean edges
  - Radial progression from center to edges
  - Normal-from-height simulation for light interaction

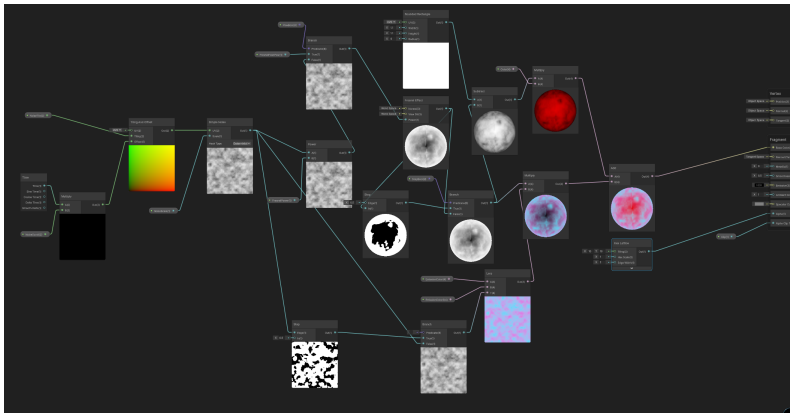


Figure 3.12: Shader Graph implementation showing: (A) Fresnel effect setup, (B) Noise animation system, (C) Alpha clip masking logic.

4. Parameter Control

The effect is controlled through these exposed parameters:

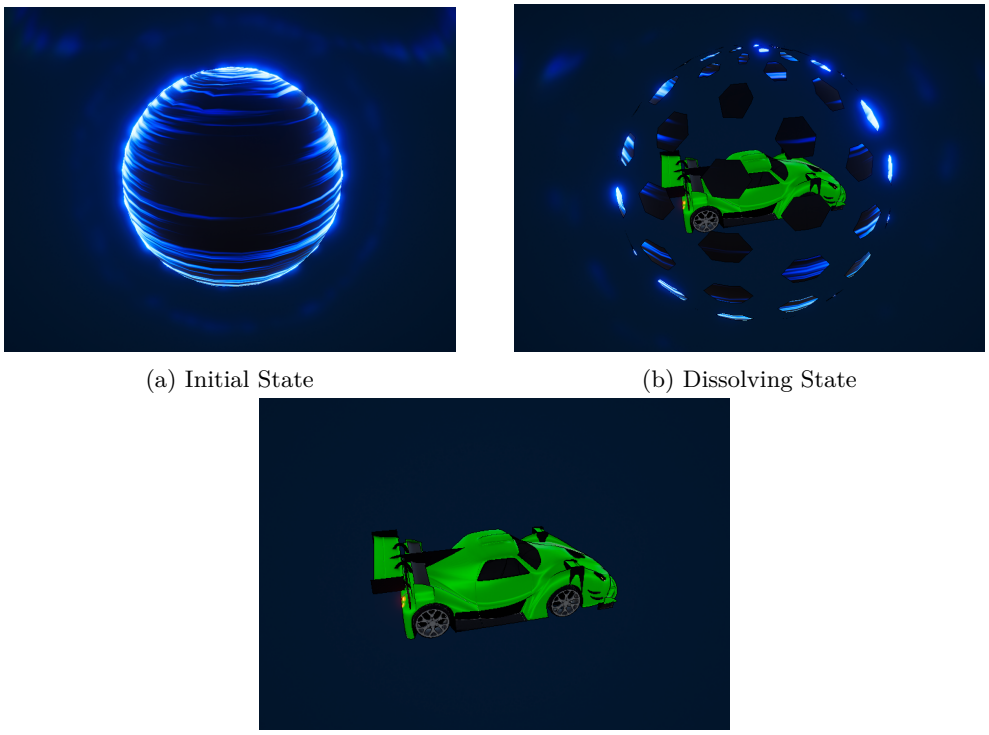
- `FresnelPower`: Controls edge glow intensity (range 1-10)
- `NoiseScale`/Speed: Adjusts procedural surface details
- `EmissionControl`: Manages emissive intensity during state changes
- `AlphaClip`: Threshold for dissolve effect progression

5. Performance Considerations

- Uses inexpensive alpha clipping instead of transparent blending
- All calculations in vertex/fragment shader (no geometry shader)
- Shared material instances across all spawn events
- LOD-based complexity reduction for distant spawns

6. Result and Evaluation

The final implementation (Figure 3.13) successfully balances visual impact with performance requirements. The hexagonal dissolve pattern provides clear visual feedback while the Fresnel glow maintains visibility across all camera angles.



(a) Initial State (b) Dissolving State

(c) Final state

Figure 3.13: Respawn sequence showing: (1) Initial sphere formation with Fresnel glow, (2) Hexagonal dissolution mid-process, (3) Final vehicle reveal with residual emission.

Future improvements could include:

- Audio-reactive parameter modulation
- Damage-state variants using color shifts
- Multiplayer differentiation through unique grid patterns

3.3.8 Motion Blur

1. Objective and Purpose

Motion blur is a post-processing effect used to simulate the natural blur of fast-moving objects or camera movement. In racing games, it enhances the sensation of speed and momentum, making rapid changes in direction or acceleration feel more dramatic and immersive. In this project, it serves as a subtle but effective layer that reinforces the arcade aesthetic and visual feedback associated with high-speed driving.

2. Conceptual Exploration

The effect was implemented using Unity's native support for motion blur within the Universal Render Pipeline (URP). From version 2022.2 onward, URP includes built-in motion blur as part of its post-processing stack, eliminating the need for custom shaders or external plugins.

Given the arcade nature of the game and the focus on dynamic gameplay, the effect was tuned to highlight velocity bursts—particularly during nitro activation or fast directional changes—without compromising clarity or introducing excessive blur artifacts.

3. Implementation Process

The effect is applied as a post-processing volume in the scene:

- A **Global Volume** is created and assigned a `Motion Blur` override.
- **Mode**: Set to `Camera Only`, to simulate motion based on the camera's transform.
- **Intensity**: Adjusted to a value between 0.4 and 0.8 for a noticeable but non-intrusive blur.
- **Clamp**: Used to prevent excessive smearing during very rapid movement.

The volume is either always active or enabled dynamically via script when nitro is triggered, allowing the effect to serve as an additional boost feedback.

The URP Asset settings were updated to ensure that post-processing is enabled globally. If the effect were to use object motion vectors (e.g., for individual cars or objects), the appropriate render features would need to be enabled as well, but this was deemed unnecessary for the desired result.

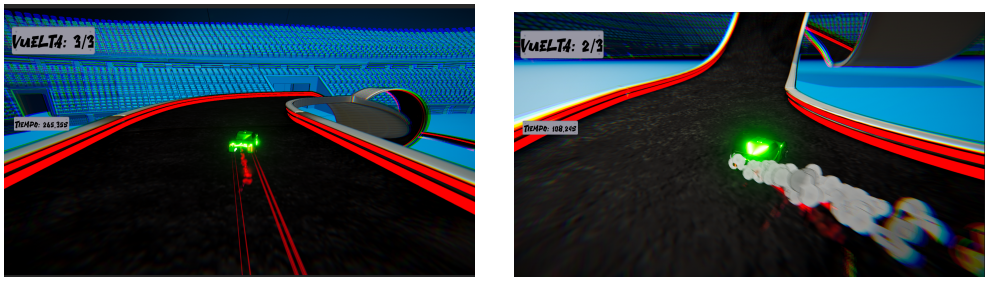
4. Performance Considerations

- The `Camera Only` mode is computationally lighter than `Object` or `Both`, making it suitable for mid-range hardware.
- The effect is only active during high-speed gameplay states, such as nitro usage.
- No additional GPU simulation or shader complexity is introduced.

5. Result and Evaluation

The resulting visual (Figure 3.14) enhances the perception of speed without overpowering the scene. It complements other effects such as the nitro flame and brake trail, and reinforces the energetic, stylized identity of the game.

Motion blur is especially effective in third-person driving games, where the relationship between the player's perspective and the car's motion is visually expressive. It plays a supporting but essential role in the visual experience.



(a) Without Motion Blur (b) With Motion Blur

Figure 3.14: Comparison of the visual impact of the Motion Blur effect. Notice the increased sense of speed and smoothness when the effect is enabled.

6. References

Unity Technologies. *Motion Blur in Universal Render Pipeline*. Unity Manual. `https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@latest/index.html`

3.3.9 Stylized Post-Processing

1. Objective and Purpose

The post-processing layer in this project serves a dual purpose: to enhance the visual clarity and polish of the scene, and to reinforce the cartoon-inspired artistic identity of the game. Rather than relying solely on realistic lighting or physically accurate shaders, the post-processing pipeline was tuned to produce vibrant colors, smooth gradients, and accentuated contrasts, supporting the arcade look and feel.

This system is applied globally via a URP `Volume Profile`, and is composed of multiple standard effects, alongside a custom fullscreen edge-detection shader used to generate outlines, giving the scene a stylized, illustrative quality.

2. General Post-Processing Stack

The game makes use of Unity's built-in URP post-processing framework, applied through a global volume. The following components are active and configured to achieve a saturated, cartoon-style look:

- **Bloom**: Adds glow to bright areas, enhancing neon effects and light trails.
- **Vignette**: Focuses the viewer's attention towards the center of the screen subtly.
- **Chromatic Aberration**: Slight RGB offset at screen edges, reinforcing the digital arcade atmosphere.

- **Color Adjustments**: Increased saturation, adjusted contrast, and gamma correction.
- **Motion Blur**: Applied globally to enhance perceived velocity (see Section 3.3.8).

These effects were chosen for their visual coherence and minimal performance cost, and were tuned in combination to complement the more complex GPU-based VFX elements described throughout this chapter.

3. Edge Outlines with Fullscreen Renderer

To further stylize the scene, a custom fullscreen outline shader was developed using Shader Graph, rendered via a Fullscreen Renderer Feature in URP. This effect outlines the depth discontinuities between objects in the scene, simulating the black contour lines typically found in hand-drawn animation.

Implementation Process The outline effect was implemented using Shader Graph and applied in a fullscreen pass. The effect is based on depth difference sampling and is applied four times in separate directions to detect edges from all sides.

- **Input**: Scene Depth texture sampled via screen-space UV.
- **Edge Detection**: For each direction (up, down, left, right), a sample offset is applied and compared to the central pixel's depth using a subtract operation.
- **Aggregation**: All directional differences are combined and modified by thickness parameters.
- **Alpha Composition**: The final edge value is exponentiated, biased, and multiplied with a color input to produce clean outline shapes.
- **Output**: Fragment Shader outputs color only where the depth difference exceeds a given threshold, resulting in edge outlines.

Shader Parameters The shader exposes several tunable properties:

- `Color` – Outline color.
- `Thickness` – Controls the sampling offset distance.
- `Thickness Bias` – Post-processing factor to control fade-off.

The node setup was duplicated and applied across four sampling directions using Vector2 offsets. The result is a full 2D sobel-like effect, applied directly to the camera output, creating outlines along any object with significant depth variation.

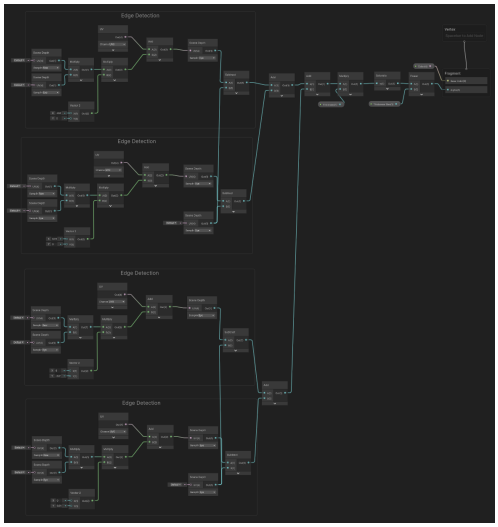
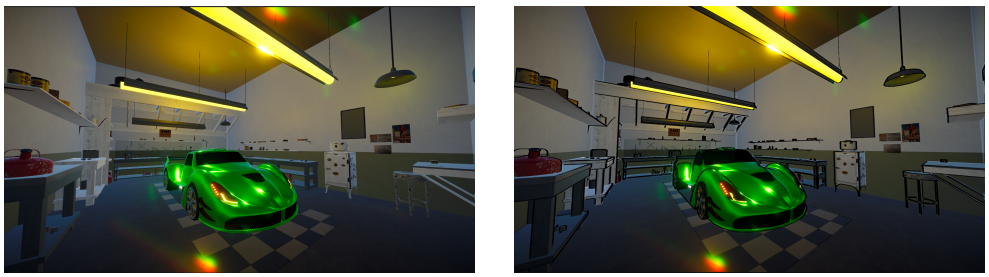


Figure 3.15: Outline Edge Detection Setup.

4. Result and Evaluation

The outline effect (Figure 3.16) adds a stylized, cartoon-like edge to all visible geometry, unifying the visual identity and reinforcing the graphic contrast. It particularly enhances the readability of silhouettes and vehicle details, even in motion.

Used in combination with bloom and saturation adjustments, the full post-processing pipeline effectively simulates a hybrid cartoon/realistic rendering style. This effect is also lightweight and runs well on mid-range hardware due to its efficient fullscreen implementation.



(a) Without Outlines (b) With Outlines

Figure 3.16: Comparison of the visual impact of the Outline effect. Notice the visual difference and highlight of each element in scene.

5. Future Improvements

Possible future upgrades include:

- Adding normal-based outline detection to complement depth-based lines.
- Using stencil buffers to exclude UI or HUD elements.

- Allowing dynamic outline thickness based on camera distance.

3.4 Secondary Visual Effects

In addition to the core visual effects systems previously described, the project includes a series of minor or supporting VFX that contribute to enhancing moment-to-moment gameplay feedback. These effects, while technically simpler, serve to reinforce the arcade feel, visual polish, and responsiveness of the overall experience.

3.4.1 Sparks on Collision

Sparks are used to emphasize moments of impact—whether from collisions with the environment, objects, or during high-intensity driving events such as rough landings or hard braking.

Implementation: The effect is implemented using a GPU-based VFX Graph with the following setup:

- **Spawn**: Burst emission on trigger (e.g., collision event).
- **Initialize**: Random lifetime (0.3–0.6s), small velocity cone, random rotation.
- **Update**: Gravity applied, size and opacity over life.
- **Output**: Additive blending, small glowing quads stretched in motion direction.

This effect is triggered through collision scripts that check for relative velocity between the player vehicle and the hit surface. When above a certain threshold, a burst of sparks is instantiated at the point of contact.

3.4.2 Underglow Neon Lights

Two simple neon strips are positioned beneath the vehicle chassis, serving as a stylistic accent.

Implementation:

- Basic emissive material applied to rectangular meshes
- Color property exposed for runtime modification
- Constant emission intensity

Integration: The neon color can be modified through a simple MaterialPropertyBlock reference, though in the current implementation it remains static as a purely cosmetic element.

3.4.3 Boost Camera Effects

When activating nitro, the camera system applies multiple distortion effects to enhance the sensation of speed:

Implementation Components:

- **FOV Expansion**:
  - Smooth transition to a wider field of view
  - Implemented via script that overrides post-processing
- **Lens Distortion**:
  - Barrel distortion effect (Intensity: -0.3 to -0.5)
  - URP post-processing override during boost
- **Screen Shake**:
  - Cinemachine Impulse Source with 0.2s burst
  - Low-frequency noise pattern (0.5-2Hz)
- **Chromatic Aberration**:
  - Intensity increases from 0.1 to 0.8
  - Direction aligned with movement vector

Technical Control: All effects are managed through a single script that:

- Triggers on nitro activation/deactivation
- Handles interpolation between states
- Coordinates effect timing

3.5 Core Gameplay and System Implementation

1. Introduction

Beyond the visual effects systems developed throughout this project, a fully functional gameplay base was implemented to support and demonstrate their integration in a working racing environment. This section details the technical development of the core game logic, physics system, user interface, and camera management—all of which were built with a focus on supporting arcade-style racing mechanics.

The implementation covers the following key areas:

- **Vehicle physics**: A custom driving controller was developed to manage car behavior using Unity's physics system, with particular attention to arcade responsiveness and support for drift-based driving.

- **Camera system**: A third-person dynamic camera was implemented using Unity's *Cinemachine*, with follow and look-at behaviors attached to the vehicle.

- **Gameplay loop**: The level progression system is based on checkpoint detection, lap tracking, and completion logic, allowing the player to complete a defined race loop.
- **User interface**: A functional HUD displays live data such as elapsed time and current lap, and manages end-of-race feedback with final score, restart, and return-to-menu options.

Each of these systems was developed with modularity and readability in mind, with the goal of serving as a stable base for gameplay testing and integration of all VFX systems.

In the following sections, the technical structure, logic, and implementation details of each area will be explained, including code excerpts, diagrams, and runtime behavior.

3.5.1 Arcade Car Physics System

1. Objective and Design Principles

The vehicle physics system was designed to deliver responsive and stylized driving behavior, aligned with the arcade nature of the game. Rather than pursuing physical accuracy, the focus was on control, fun, and compatibility with the visual and gameplay feedback provided by other systems. Features such as drifting, nitro boosts, and aerial jumps were prioritized to support dynamic and exaggerated gameplay moments.

2. Technical Architecture

The system is built upon Unity's standard physics engine [15]. It employs a `Rigidbody` component combined with `WheelColliders` to simulate traction, acceleration, and steering. User input is managed through Unity's Input System (`InputActionAsset`) [14], and all physics interactions are executed within the `FixedUpdate()` loop for consistency and stability.

The controller includes:

- Acceleration and braking logic using torque curves and conditional force application.
- Speed-sensitive steering mechanics.
- Drift handling with dynamic friction and chassis tilt.
- Nitro boost system with temporary velocity increase and synchronized VFX/audio.
- Jump system triggered by collision with tagged surfaces (e.g., ramps).
- Engine audio modulation according to speed and torque output.

3. Acceleration and Braking

Vehicle propulsion is applied only to drive wheels and scales with speed using a customizable torque curve. This curve reduces power delivery at higher speeds to simulate traction loss and limit acceleration naturally. When braking, a fixed braking torque is applied. During idle states, a moderate braking force is used to avoid unwanted rolling.

4. Steering System

The steering angle is dynamically interpolated based on the current speed of the vehicle. This ensures high responsiveness at low speeds and increased stability at high speeds. Steering is applied only to the designated steering wheels and uses a time-based interpolation factor to smooth transitions.

5. Drift Mechanics

Drifting is activated under two conditions: a minimum velocity threshold and a steering input beyond a defined margin. Once activated, the wheel friction is reduced dynamically, and an additional corrective torque is applied to simulate lateral force and oversteer behavior.

Visually, the car's chassis is tilted according to the steering direction to reinforce the effect. When the drift ends, friction is restored to its normal level, and the chassis returns to its neutral orientation.

6. Nitro Boost Integration

The nitro system allows for a temporary increase in acceleration and maximum speed. When activated by user input, the system:

- Multiplies the torque output and top speed.
- Triggers visual effects such as exhaust flames and brake trails.
- Plays an accompanying sound effect.

After a fixed duration, the values are reset to their defaults and the associated effects are stopped. This mechanic adds a burst mechanic that visually and mechanically alters gameplay for a limited time.

8. Additional Features

- **Downforce simulation**: A downward force proportional to speed is applied to improve traction at high velocities.
- **Audio feedback**: Engine sound pitch and volume dynamically reflect current speed and acceleration.

- **Stability**: The center of mass is lowered to reduce rollover risk and improve cornering.

- **Visual effects integration**: Boost-related VFX (nitro flames and brake trails) are triggered directly by the controller to maintain synchronization.

9. Evaluation

The arcade car physics system achieves a balance between simplicity, responsiveness, and stylized control. It forms the foundation of the game's driving mechanics while enabling coherent integration with visual and audio feedback systems. This controller supports gameplay scenarios ranging from high-speed racing to stunt-like maneuvers, ensuring that the driving experience remains engaging and expressive.

3.5.2 Third-Person Camera System with Cinemachine

1. Objective and Rationale

To accompany the arcade driving model and enhance spatial awareness during gameplay, a third-person dynamic camera system was implemented. The camera is responsible for framing the vehicle appropriately, communicating speed and movement, and supporting visual readability of the environment without distracting the player.

2. Technology Used

The system is built using Unity's **Cinemachine**, a modular camera framework that provides real-time procedural camera control [13]. In particular, the implementation uses the `CinemachineCamera` component, configured in *Follow* and *Look At* mode to track the player's vehicle.

3. Configuration

The camera rig is configured as follows:

- **Follow Target**: The virtual camera is set to follow a dedicated target object positioned on the vehicle (usually an empty GameObject located slightly above and behind the car's center).

- **Look At Target**: The same target object is used to maintain focus and ensure camera movement remains centered on the vehicle's movement direction.

- **Transposer settings**: A soft follow is configured using damping parameters to create a smooth, lagged motion when the car changes speed or direction.
- **Composer settings**: Screen position offsets and dead zones are adjusted to keep the vehicle within an ideal viewing region while allowing for minor camera shifts.

4. Behavior and Tuning

The camera automatically adjusts its position and rotation based on the car's trajectory. The smoothing behavior absorbs sudden directional changes while maintaining a sense of motion. Parameters such as vertical offset, distance, damping speed, and rotation lag are calibrated to create a balance between visibility and visual dynamics.

This setup avoids abrupt cuts or rigid tracking, which would otherwise reduce the fluidity of the driving experience.

5. Integration with Gameplay

No manual control over the camera is given to the player. Instead, the system ensures that the camera naturally reacts to gameplay events (e.g., drifting, jumping, boosting) through its dynamic response model. The camera is also compatible with other visual effects such as motion blur and FOV distortion triggered during boost events, as discussed in previous sections.

6. Evaluation

The use of Cinemachine greatly simplifies camera implementation while providing high-quality dynamic framing. The resulting camera behavior supports both usability (clear visual feedback) and stylization (fluid motion), making it a key component in the overall presentation of the game.

3.5.3 Lap and Checkpoint System

1. Objective

To provide a complete and functional gameplay loop for the racing experience, a checkpoint-based lap system was developed. This system tracks the player's progression through the circuit, validates the completion of laps in the correct sequence, and coordinates with the user interface to display timing and lap information.

2. Structure and Components

The system is composed of three main elements:

- **Checkpoints**: Represented as colliders placed along the track in sequential order. Each checkpoint is responsible for notifying the central system when crossed.
- **CheckpointSystem**: Manages the checkpoint list, monitors progression, records sector times, and triggers lap completions.
- **GameManager**: Oversees the overall race flow, including lap count, timing, and finalization logic. It also manages car respawning and user interface updates.

3. Checkpoint Progression Logic

Each checkpoint is linked to the `CheckpointSystem`, which maintains a list of checkpoints in the correct order. When the player passes through a checkpoint, the system verifies if it corresponds to the expected index. If so, the sector time is recorded, and the system advances to the next checkpoint.

Upon crossing the final checkpoint, the system:

- Registers the lap completion.
- Resets the internal checkpoint index.
- Notifies the `GameManager` via event delegation.

This structure ensures that laps are only considered valid if all checkpoints are passed in order, preventing sequence skipping or lap abuse.

4. Lap Control and Race Flow

The `GameManager` tracks the total number of laps and current progression. When a lap is completed, it increments the counter and checks whether the race has finished. If the final lap is completed:

- The race timer is stopped.
- Car input and movement are disabled.
- The finish panel is shown with final time and options.

The player can also be respawned manually or programmatically, restoring the car to the last valid checkpoint position or the start.

5. Time Tracking and Feedback

Sector times and total lap times are continuously recorded during gameplay. This data can be displayed via the user interface for feedback, debug purposes, or to enable features such as ghost comparisons or lap time leaderboards in the future.

6. Evaluation

This system provides a robust and modular way to define race structure within a level. It ensures accurate player tracking, offers a foundation for competitive features, and integrates tightly with both the user interface and core game logic. Its use of event-driven communication and sequential validation makes it scalable for more complex race logic if needed.

3.5.4 Race User Interface (HUD and End Screen)

1. Purpose and Scope

The User Interface (UI) system provides real-time visual feedback to the player during the race, contributing both to gameplay clarity and immersion. It displays key information such as lap progression, elapsed time, and final results upon race completion.

2. HUD Overview

During gameplay, a heads-up display (HUD) is shown on screen, including:

- **Lap Counter**: Displays the current lap and total number of laps (e.g., "Lap 2 / 3").
- **Race Timer**: Shows the total elapsed race time, updated every frame.

These elements are updated by the `GameManager`, which tracks race progression and calls UI update methods as needed. The system ensures that all HUD data remains synchronized with actual gameplay state.

3. End-of-Race Screen

Upon completing the final lap, the interface switches to an end screen, displaying:

- The **final race time**.
- A **"Restart"** button, which reloads the race scene.
- A **"Main Menu"** button, which returns the player to the main interface.

These options allow the player to seamlessly restart the challenge or exit, providing a complete and uninterrupted game loop experience.

4. Integration with Game Systems

All UI updates are managed through a dedicated `UIManager` class, which exposes public methods called by the `GameManager`:

- `UpdateRaceUI(lap, totalLaps, time)` – Updates the HUD with current data.
- `ShowFinishPanel(time)` – Displays the final screen with the total race time.

This separation ensures modularity and clean communication between gameplay systems and the interface.

5. Evaluation

The UI system is functional, minimalistic, and tightly integrated with the race controller logic. It provides essential feedback without distracting the player, and supports a complete loop from race start to race end with intuitive controls for restarting or exiting.

3.5.5 Audio System and Sound Integration

1. Overview

Sound design plays an essential role in reinforcing player feedback and enhancing the perceived intensity of gameplay. The current implementation provides a complete audio system covering all gameplay states, from menu navigation to intense racing moments. Each scene features distinctive background music - energetic tracks during races and spatialized radio-style playback in garage/-menu scenes where sound physically emanates from environment objects.

2. Engine and Skid Sounds

The vehicle controller includes dynamic audio modulation for the engine, adjusting both pitch and volume based on current speed. This provides real-time acoustic feedback that helps convey acceleration and deceleration.

Skid sounds are also implemented to activate when drifting is triggered. These sounds are synchronized with the drift mechanics and are stopped automatically when traction is restored.

3. Boost Activation Feedback

The nitro boost includes a dedicated sound cue triggered when the system is activated. This sound is played simultaneously with the visual exhaust flames and camera distortion, creating a cohesive and immersive moment of increased velocity.

4. Crash sounds

Whenever the player crashes against a wall or the car's bottom touches the ground (this generating sparks), a crash sound is triggered, varying its volume acording to the current speed.

5. Game Event Sounds

Key gameplay moments have distinct audio feedback:

- Checkpoints play a confirming "ping" sound
- The finish line triggers a celebratory musical stinger
- Track editor actions produce mechanical placement sounds

UI elements maintain consistent auditory feedback:

- Standard buttons use standard click sounds
- Vehicle customization features context-sensitive audio (paint sprays, wrench sounds)

4. Sound Component Structure

All sound effects are managed through Unity's `AudioSource` components attached to the its correspondant GameObject. Triggering logic is encapsulated within a script, ensuring that audio is always in sync with gameplay mechanics.

6. Evaluation

The audio system provides comprehensive coverage across all game scenes and interactions, forming a complete sensory feedback loop. Implementation ensures synchronization with visual effects and gameplay mechanics, particularly noticeable during boost sequences where audio, visuals and camera effects combine for maximum impact. The modular architecture maintains excellent performance while allowing for straightforward expansion, with all core systems fully implemented and tested.

3.5.6 Project Scope and Implementation

1. Complete System Overview

The project encompasses all key systems required for a fully functional racing experience, including:

- Core racing gameplay with complete visual/audio feedback systems
- Three main game modes:
  - Race mode (primary gameplay circuit)
  - Vehicle customization workshop
  - Track editor with placement/editing tools
- Supporting systems:

  - Main menu and UI navigation
  - Settings and configuration management

2. Collaborative Development

Developed in conjunction with another student's parallel work, this implementation focuses specifically on:

- Visual effects pipeline and rendering systems
- Core vehicle physics and gameplay mechanics
- Camera and audio feedback systems
- Main menu setup, interfaces and navigation flows

The integrated project combines these elements with complementary systems developed by my collaborator, including:

- Vehicle customization functionality
- Track editing tools and asset management

All systems have been successfully merged into a cohesive final product, with consistent art direction and technical implementation across all components. The modular architecture allowed for efficient parallel development while ensuring seamless system integration.

3. Chapter Overview

This chapter has presented the complete technical implementation of all visual effects and core gameplay systems, establishing the foundation for the racing experience. The developed systems work in harmony to create a cohesive audiovisual feedback loop that supports the game's arcade-style mechanics.

Key components include the VFX pipeline built with Unity's Visual Effect Graph and Shader Graph, dynamic camera systems, and fully integrated audio feedback. All systems have been successfully implemented and tested, demonstrating stable performance across different gameplay scenarios.

The following Results chapter will provide detailed analysis and evaluation of these systems, examining their visual impact and effectiveness in enhancing the player experience.

CHAPTER 4

Results

Index

4.1 General Overview

The Final Degree Project has reached its completion stage with a solid implementation of both technical and artistic objectives. A fully functional 3D arcade-style racing game was successfully developed, integrating custom gameplay systems, stylized visual effects, and a responsive user interface.

Rather than following a strictly utilitarian approach, the project embraced experimentation with Unity's rendering tools—Visual Effect Graph, Shader Graph, and URP post-processing—to create a distinctive, stylized visual identity. Each system was implemented with performance in mind and integrated tightly with gameplay logic.

4.2 Implemented Visual Effects

The full list of visual effects developed throughout the project includes:

- Skidmarks (C# mesh generation)

- Drift smoke (VFX Graph + Shader Graph)
- Nitro exhaust flame (layered GPU particle systems)
- Brake trail (particle strip)
- Checkpoint holograms (animated shader)
- Finish line shader (striped animated material)
- Spawn/respawn dissolve sphere (hexagonal alpha clip)
- Collision sparks (GPU burst particles)
- Neon lighting (script-driven emissive materials)
- FOV distortion (dynamic camera modification)
- Post-processing: motion blur, bloom, vignette, chromatic aberration
- Outline effect (fullscreen edge-detection shader)

Each of these effects was detailed and technically justified in Chapter 3. Their integration was driven by gameplay events, real-time physics, and feedback requirements to ensure both functionality and stylistic consistency.

4.3 Visual Outcome in Context

To showcase the results of the implementation work, this section presents selected frames captured during real gameplay. Each image highlights one or more of the systems working together to create a coherent and visually engaging racing experience.

4.3.1 Race Start and Checkpoints

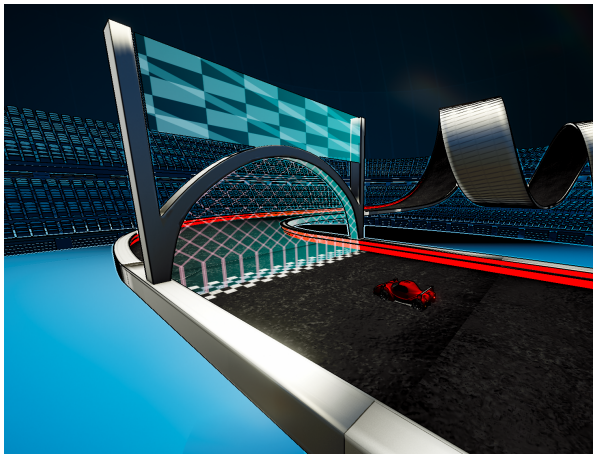


Figure 4.1: Race start with stylized checkpoint holograms and lighting. The procedural shader communicates active targets clearly and dynamically.

4.3.2 Nitro Boost in Action

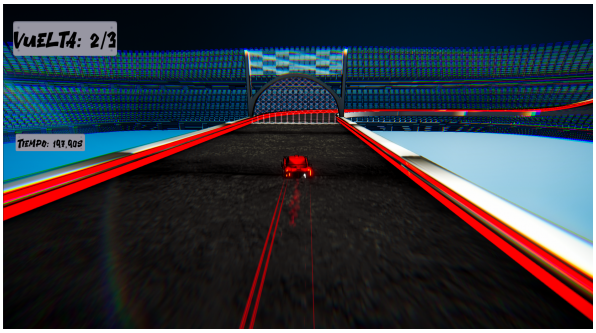


Figure 4.2: Combined nitro feedback: exhaust flame, brake trail, motion blur, and FOV distortion. Effects are synchronized to enhance boost activation.

4.3.3 Drifting and Visual Feedback

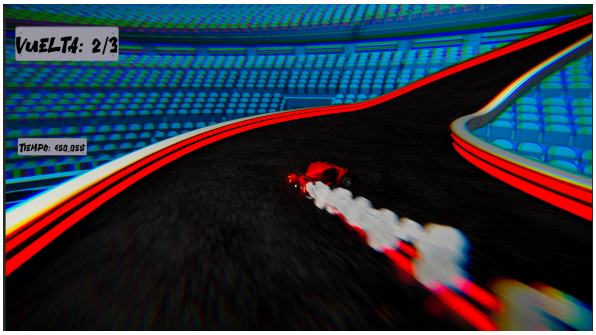


Figure 4.3: Drifting scene showing skidmarks, stylized smoke with Voronoi animation, and car tilt. Visual effects reinforce the physics-driven drift logic.

4.3.4 Collision and Recovery

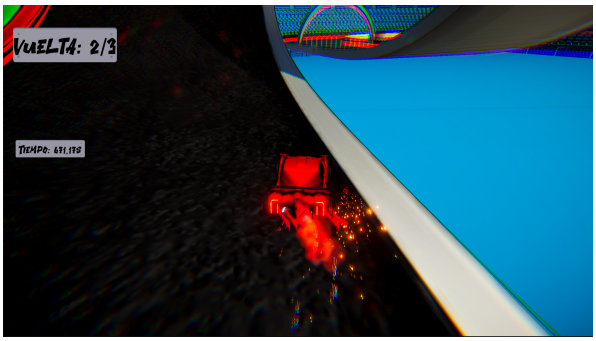


Figure 4.4: Collision moment with spark burst and environment response. This provides impactful feedback.

4.3.5 Respawn and Game Loop Clarity

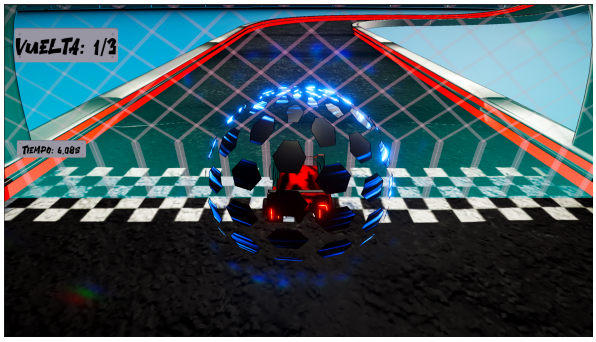
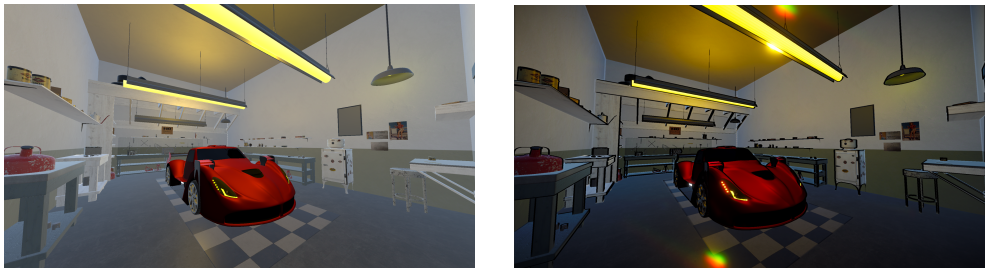


Figure 4.5: Spawn/respawn dissolve shader in action. A hexagonal alpha clip transition highlights the re-entry into the race.

4.3.6 Stylized Rendering and Final Aesthetic



(a) Without Post Processing (b) With Post Processing

Figure 4.6: Final game aesthetic combining outline shader, bloom, color grading and neon lighting. The result maintains clarity and style

4.4 Visual Summary of Implemented Effects

A collection of annotated captures is presented to showcase the final appearance of the most relevant visual effects implemented throughout the project. These images provide an individual overview of how the visual systems behave, reinforcing their purpose and contribution to the overall experience.

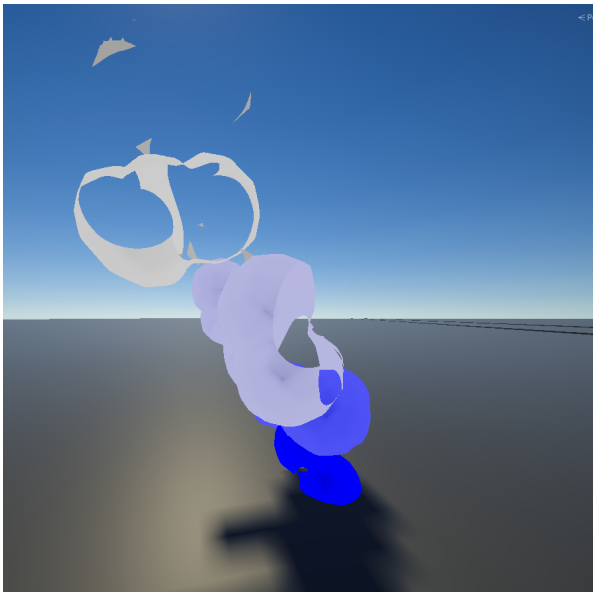


Figure 4.7: Stylized smoke emission.

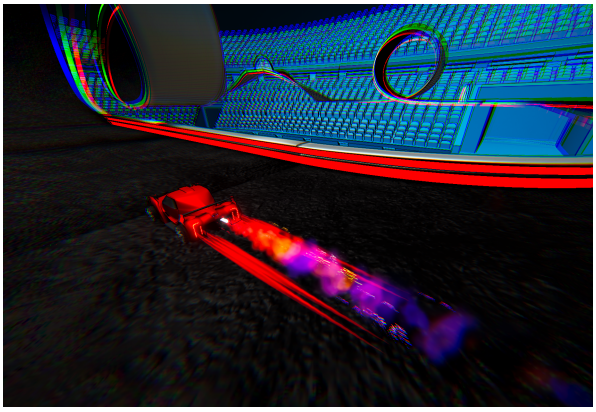


Figure 4.8: Nitro boost moment: exhaust flame, brake trails, motion blur and FOV distortion combined.

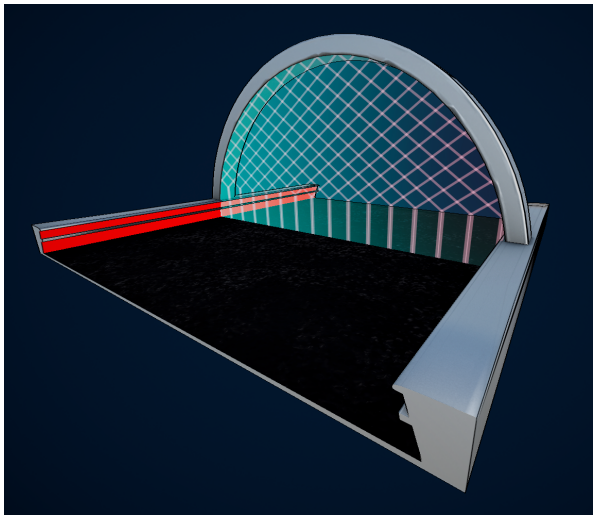


Figure 4.9: Stylized checkpoint shader with animated hologram effect.

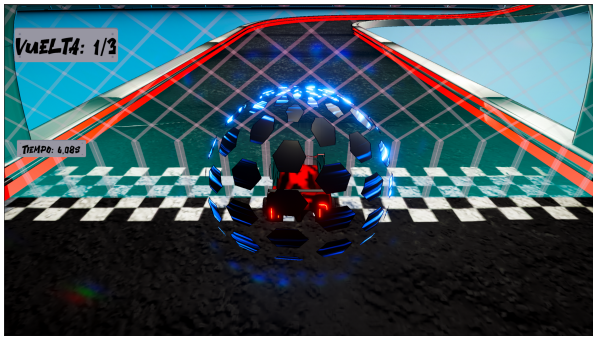


Figure 4.10: Respawn dissolve effect using hex pattern and Fresnel rim lighting.

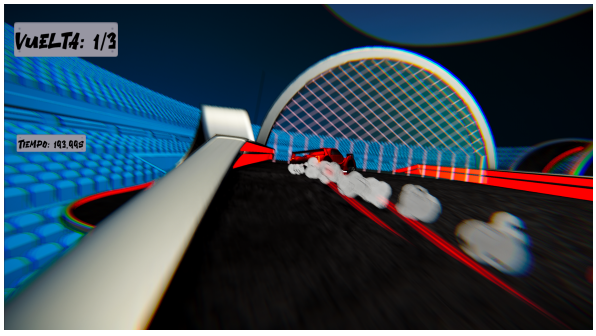


Figure 4.11: Full post-processing pipeline with outline effect, color grading, and bloom.

This visual summary demonstrates the technical and artistic consistency achieved in the final build, showing that the implemented effects are not iso-

lated features, but fully integrated systems supporting a cohesive arcade racing experience.

4.5 Performance Evaluation

Although no formal benchmarking session was conducted, performance was monitored consistently throughout the development process on mid-range to high-end hardware. During gameplay testing and iteration, no noticeable frame drops or visual bottlenecks were observed, even when multiple effects were active simultaneously.

This allowed all visual effects to be developed and integrated without requiring major compromises or redesigns due to performance issues. The results indicate a stable and responsive experience under normal conditions.

Several good practices were followed to ensure visual efficiency:

- Particle counts were capped, and spawn rates were dynamically adjusted via scripts.
- Shader complexity was reduced through the use of unlit materials and additive blending.
- Post-processing profiles were modular and optimized for lightweight rendering.
- Scripts included basic distance-based logic to limit unnecessary updates or emissions.

Overall, the visual effects layer proved to be scalable and performant, with parameter controls exposed for further adjustment if needed during future platform adaptation or optimization phases.

4.6 Result Summary

The implemented effects and shaders not only meet the functional objectives but also contribute significantly to the visual identity and player experience of the game. Their integration into core gameplay loops and responsiveness to player actions demonstrate a technically and artistically cohesive system.

This result serves as both a portfolio-quality prototype and a platform for future iterations, expansions, or stylization explorations.

CHAPTER 5

Conclusions

Index

5.1 General Evaluation

This project has been, above all, a deeply enriching experience on both a technical and personal level. The development of advanced visual effects within Unity, and their integration into a functional arcade racing game, has allowed for an in-depth exploration of tools, workflows, and design principles that are fundamental to modern game development.

Working on visual effects—my personal passion—has not only been enjoyable, but has also proven to be a gateway into the technical depth and creative potential of GPU-based workflows, procedural systems, and real-time rendering pipelines.

5.2 Personal Learning and Growth

The process of designing, implementing, and optimizing each effect has fostered a better understanding of how visual systems interact with gameplay, hardware constraints, and player perception. Beyond the aesthetic dimension,

this project has required careful consideration of performance, rendering cost, and integration across multiple systems.

As a result, I now approach the development of visual effects not only as artistic expression, but also as a discipline of precision, iteration, and engineering.

5.3 Collaboration and Future Potential

This project is part of a joint effort, and it has been structured to allow future expansion beyond academic scope. Together with my collaborator, we have identified the potential of the project as a candidate for public release. Once the academic phase is completed, we intend to continue polishing, scaling, and expanding the game—adding more features, visual content, and gameplay depth.

The modularity of the current systems and the clarity of the visual direction provide a strong foundation for this post-academic development.

5.4 Lines of Future Work

Based on the experience gained and the current state of the project, several lines of future work are already outlined:

- Expanding the set of VFX, especially contextual and environmental effects.
- Refining the UI, audio, level editor and customization systems to support a full-featured player experience.
- Exploring new shader techniques and GPU instancing optimization to enhance style and scalability.
- Conducting user tests and performance profiling across different platforms to ensure accessibility and stability.

5.5 Final Reflection

The project has exceeded expectations in many areas and has validated the potential of combining stylized visuals with gameplay-aware design. It marks a strong starting point in what I hope will be a long trajectory in real-time VFX development. I leave this phase of the project with a sense of pride, motivation, and a clear direction for future growth—both for the game and for myself as a developer.

Bibliography

[1] Gameloft. *Asphalt 9: Legends*. Video Game, 2018. Available at: https://asphaltlegends.com

[2] Blender Foundation. *Blender*. Available at: https://www.blender.org

[3] Bruneton, E., & Neyret, F. (2008). *Precomputed Atmospheric Scattering*. ACM Transactions on Graphics (SIGGRAPH), 27(3), 1–10.

[4] Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., & Worley, S. (2002). *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann.

[5] GitHub Inc. *GitHub Platform*. Available at: https://github.com

[6] Google. *Google Docs*. Available at: https://docs.google.com

[7] Electronic Arts. *Need for Speed: Unbound*. Video Game, 2022. Available at: https://www.ea.com/games/need-for-speed/need-for-speed-unbound

[8] Overleaf Ltd. *Overleaf*. Available at: https://www.overleaf.com

[9] Adobe. *Adobe Photoshop*. Available at: https://www.adobe.com/products/photoshop.html

[10] Unity Technologies. *Particle System (Shuriken)*. Available at: https://docs.unity3d.com/Manual/ParticleSystem.html

[11] Ubisoft Nadeo. *Trackmania*. Video Game, 2003–present. Available at: https://www.trackmania.com

[12] Atlassian. *Trello*. Available at: https://trello.com

[13] Unity Technologies. *Cinemachine Documentation*. Available at: https://docs.unity3d.com/Packages/com.unity.cinemachine

[14] Unity Technologies. *Input System Manual*. Available at: https://docs.unity3d.com/Packages/com.unity.inputsystem

[15] Unity Technologies. *Unity Physics Manual*. Available at: https://docs.unity3d.com/Manual/PhysicsSection.html

[16] Unity Technologies. *Shader Graph Documentation*. Available at: https://docs.unity3d.com/Manual/ShaderGraph.html

[17] Unity Technologies. *Universal Render Pipeline (URP)*. Available at: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal

[18] Unity Technologies. *Visual Effect Graph Documentation*. Available at: https://docs.unity3d.com/Packages/com.unity.visualeffectgraph

[19] Microsoft. *Visual Studio*. Available at: https://visualstudio.microsoft.com

APPENDIX A

Detailed Overview of Unity's Visual Effects Pipeline

A.1 Introduction to Visual Effects Development in Unity

This section outlines the technological and conceptual foundations that support the visual effects developed in this project. Rather than merely applying pre-existing solutions, the project was approached as a technical exploration, prioritizing a deep understanding of Unity's visual effects pipeline and how it integrates within the rendering workflow.

The development process was guided by ongoing research into the various methodologies available within Unity for creating visual effects. The choice to work exclusively with Unity's Visual Effect Graph (VFX Graph) was motivated not only by its capabilities, but also by the opportunity it provided to explore GPU-based visual simulation in depth. This section documents that exploration process, including the functional principles of VFX Graph, how it compares to the traditional Unity Particle System, and its interplay with Shader Graph.

A strong emphasis is placed on understanding the internal logic of these tools and their practical integration into a real-time project. Through this approach, the chapter not only provides insight into implementation details, but also demonstrates a clear learning path through experimentation, analysis, and technical decision-making. Each visual effect presented later in this document is the result of both creative design and technical evaluation informed by the principles discussed here.

A.2 Approach to Visual Effects in Unity

Unity provides several distinct systems for the creation of visual effects, each with its own strengths, limitations, and areas of application. Selecting the appropriate toolset requires not only an understanding of the available technologies, but also a clear definition of the project's visual goals, performance constraints, and target platforms.

Among the main options for VFX development in Unity are:

- **The Particle System (Shuriken)**: a modular, CPU-based system that enables rapid development of particle-based effects and integrates deeply with Unity's physics and scripting environments.
- **The Visual Effect Graph (VFX Graph)**: a GPU-based system designed for complex, high-performance simulations involving large quantities of particles and procedural control.
- **Shader Graph**: allows the creation of custom visual behaviors at the material level and can be used in conjunction with both Shuriken and VFX Graph for advanced rendering.
- **Post-processing**: a stack of camera-level effects including bloom, motion blur, color grading, and screen-space ambient occlusion, which can support the overall visual coherence of the scene.

The project was developed using Unity's **Universal Render Pipeline (URP)**, which supports VFX Graph and Shader Graph natively [17]. This decision encouraged the adoption of GPU-accelerated workflows and modern graphical techniques.

Throughout development, each available system was evaluated in practice. The criteria included:

- **Scalability and performance** on mid-range hardware.
- **Visual flexibility**, especially for stylized or exaggerated effects.
- **Integration with gameplay systems**.
- **Control over simulation parameters**.
- **Learning potential**, aligned with the project's educational purpose.

Ultimately, the Visual Effect Graph was chosen as the main system. Shader Graph complemented it by defining the visual style of the particle outputs. A comparison with Shuriken, as well as Shader Graph integration, is discussed in the following sections.

A.2.1 Unity Visual Effect Graph (VFX Graph)

The Unity Visual Effect Graph (VFX Graph) is a GPU-based system [18] that enables real-time simulation and rendering of complex effects through node-based programming.

1.1 Internal Architecture

VFX Graph is built on GPU instancing and compute shaders. It uses four main *contexts* to define effect behavior:

- **Spawn**: controls emission rate and timing.
- **Initialize**: defines initial state of particles.
- **Update**: applies behaviors like turbulence or drag.
- **Output**: renders particles (billboards, meshes, lines, etc.).

Each particle's attributes are stored in GPU buffers, allowing parallel execution with low CPU usage.

1.2 Node-Based User Interface

VFX Graph uses a node editor where users connect functional blocks like:

- **Blocks**: actions like applying velocity or color.
- **Operators**: math and logic functions.
- **Properties**: script-exposed parameters for control.

Figure A.1 shows an example of a Shader Graph material assigned to a VFX Graph output, illustrating the interconnection between visual style and simulation behavior.

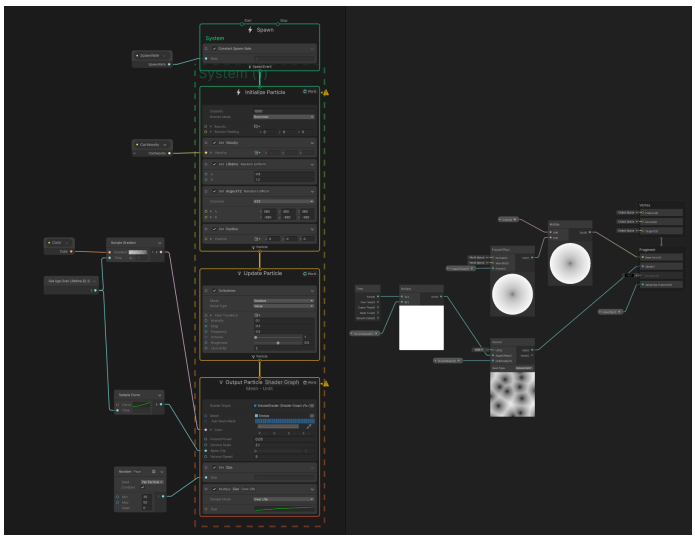


Figure A.1: Example of Shader Graph material assigned to a VFX Graph Particle Lit output.

1.3 GPU Execution and Performance

VFX Graph can simulate hundreds of thousands of particles at once using the GPU. This is ideal for volumetric effects like smoke, fire, or trails, with minimal CPU strain [4].

1.4 Interactivity and Integration

VFX Graph integrates with gameplay through:

- **Custom Events**: triggered from C# scripts.
- **Property Bindings**: link particle behavior to game state.
- **Camera and Light Data**: enhance realism through environmental input.

1.5 Advanced Capabilities

Some of the advanced features explored during development include:

- Curl noise for stylized smoke motion.
- Mesh and ribbon outputs for glowing trails.
- Sampling textures or data maps for procedural variation.

1.6 Practical Workflow

The effects were developed iteratively using this workflow:

1. Sketch or plan desired visual result.
2. Define spawn and initialization conditions.
3. Add Update behaviors (gravity, distortion, fadeout).
4. Select appropriate Output style and configure shader/material.
5. Expose parameters for in-game tuning.
6. Trigger the effect from gameplay logic.

A.2.2 Comparison with Unity's Particle System (Shuriken)

Table A.1 summarizes the technical differences between VFX Graph and Shuriken, highlighting why the former was chosen.

Aspect	**Shuriken (Particle System)**	**VFX Graph**
Execution Model	CPU-based	GPU-based (compute shaders)
Performance	Efficient for low particle counts	Scales well with complex effects
Physics Integration camera data	Full (colliders, rigidbodies)	Limited to depth buffer
Control	Full scripting access	Script bindings and events
Complexity	Simple effects only	Procedural, volumetric, layered
Pipeline Integration	Legacy support	Designed for SRPs (URP/HDRP)

Table A.1: Technical comparison between Shuriken and Visual Effect Graph

Use Case Context

VFX Graph was used because it:

- Handled high-particle-count effects like smoke and fire efficiently.
- Matched the exaggerated and stylized look of the game.
- Encouraged deeper learning about GPU workflows.

Shuriken would have been appropriate for simpler, low-cost effects (e.g., sparks, debris), but was intentionally avoided to prioritize advanced learning.

A.2.3 Shader Graph and Integration

Shader Graph enabled the creation of custom materials used in all VFX outputs. It allowed precise control over:

- Emission and transparency.
- Noise-based distortion and glow.
- Time-based animations and scrolling effects.

Table A.1 earlier illustrates a material used in a VFX system with animated distortion and color cycling.

In combination with VFX Graph, Shader Graph provided a visual layer of depth, stylization, and interactivity not achievable with default shaders. This synergy proved fundamental for the hybrid visual style pursued throughout the project.