Georgi Tsvetanski
GAME 315 – Programming for Interactive Design
Professor Sujan
12/09/2025

Game Functionality and User Interface Analysis

1. Game Functionality

Gameplay Mechanics

The core gameplay of Drift Immersive focuses on a physics-based driving experience where the player must balance speed and control. Instead of using a pre-made vehicle package, I programmed the car's behavior myself to ensure it felt "heavy" and grounded in VR.

- **Driving & Drifting:** The car is controlled through a custom input script (PlayerController.cs) that feeds data into my movement logic (CarMovementBehavior.cs). The challenge comes from the physics tuning; the car takes time to accelerate and brake, meaning players cannot just hold the gas button down constantly.
- **Drift Mechanics:** I created a specific mechanic for drifting (DriftBehavior.cs) where holding the drift button lowers the car's grip friction. This allows the rear of the car to slide out while I boost the steering sensitivity, giving players the reward of executing a stylish turn if they time their counter-steering correctly.
- **VR Interaction:** In VR mode, the mechanic shifts to "diegetic" interaction. While the player still uses a controller for buttons, they use their physical head movement to look into turns, which mimics real-world performance driving.

Progression Systems

(Note: As planned in the design phase, progression was simplified to focus on VR stability.) Currently, the game does not feature a traditional progression system like unlocking cars or levels. The "progression" is purely skill-based. Because the physics are custom-tuned to be challenging, the player progresses by mastering the drift mechanics and improving their lap times. I had originally planned a collectible "Diamond" system, but I prioritized refining the VR camera and physics stability over adding gamified collection elements.

Modular Behavior

Since there are no AI opponents in this build, the "behavior" of the game comes from the modular structure of the car itself. I designed the code so that different parts of the car act independently, similar to how an NPC might have different states.

- **Visual Behavior:** The wheels (WheelController.cs) and steering wheel (SteeringWheelController.cs) react automatically to the car's physics. For example, the

steering wheel rotates based on the player's input, and the driver's hands follow it manually, creating a believable cockpit environment without complex animation trees.

Other Features (VR Integration)

The standout feature is the VR integration. I had to replace standard camera systems with a custom solution (VRCameraCarFollow.cs) that keeps the player seated in the car and everything goes smoothly. This script handles the alignment of the VR headset (XR Rig) to the driver's seat, allowing for a single-player experience that feels much larger than it is because of the immersion.

2. User Interface (UI)

Design and Aesthetic

For the User Interface, I chose a "diegetic" design philosophy. In VR, pasting 2D text over the player's eyes causes eye strain and breaks immersion. Therefore, I removed most standard HUDs and placed the UI elements directly onto the car's dashboard. The aesthetic is simple and functional, using high-contrast text that mimics a digital racing dashboard.

Clarity and Usability

The UI communicates essential information naturally:

- **Speedometer:** I programmed the speedometer (SpeedometerController.cs) to translate the car's physics velocity (linearVelocity) into a readable format.
- **Visual Feedback:** The dashboard displays the speed in kilometers per hour via text (SpeedAndTimerUI.cs), which is easier to read quickly in VR than an analog needle.
- **Feedback Issues:** There is a known bug where the analog needle briefly jumps to a high value when the level loads before settling. However, because the digital text display remains accurate, the player is never confused about their actual speed.

3. Challenges and Solutions

VR Camera Jitter

Challenge: When I first implemented VR, the camera would shake violently when the car accelerated. This was caused by the camera script trying to update its position at a different frequency than the car's physics engine.

Solution: I wrote a custom script (VRCameraCarFollow.cs) that updates the camera position in LateUpdate(). This forces the camera to wait until all physics calculations are finished for the frame before snapping to the driver's seat, resulting in a smooth experience that prevents motion sickness.

Physics Updates (Unity Versions)

Challenge: During development, I realized that some of my older physics code was using outdated methods for checking speed, which caused warnings in the newer version of Unity I was using.
Solution: I updated my movement scripts (like CarMovementBehavior.cs) to use linearVelocity instead of the older standard. This ensured my math for acceleration and drifting remained accurate without causing errors in the console.

Hand Animation Alignment

Challenge: I wanted the driver's hands to turn with the steering wheel, but setting up a full IK (Inverse Kinematics) system was causing issues with my prefab structure.
Solution: Instead of over-engineering it with external tools, I wrote a simpler solution in SteeringWheelController.cs. I manually rotate the hand and forearm transforms in the code to match the steering input. This was a cleaner solution that achieved the visual goal without needing complex animation assets and aided in submitting my project deliverable on time.