The Lazy Devs present:

-- CARPG --

Technical Design Document

Christian Sears
Lucas Josefsson
Kevin Wong
Duarte Maia
Zach Myers
Leon Wong

VFS 2016
GD43

Table of Contents

Click an item to go to its section

Revision History

Version	Date	Details
0.1	23 July 2016	First draft of Overview, Table of Contents, Revision History, and Requirements Specification
0.2	4 Aug 2016	Included a UI flowchart, Network Implementation section, and UI section
0.3	12 Aug 2016	Added Network Chart, edited UI requirements.
0.4	12 Aug 2016	Added Explanatory Diagrams, Technical Implementation: Movement & Spells
0.5	13 Aug 2016	Added Main Menu UI mockups, and Hud mockups
0.6	14 Aug 2016	Added UML Diagrams, Milestone Planning, Table of Contents

Project Overview

High Concept

Title: Carpg
Genre: Multiplayer online car battle arena
Platform: PC
Camera: 3rd Person
Character: Wizard Cars
Core Verbs: Drive, Drift, Cast Spells

CARPG is a MOBA in which players take control of cars in 2v2 or 3v3 battles in which the players face off in an arena where they can drive around, and drift to cast spells.

There are 2 cars available to choose from, and they differ in that they cast spells from different schools of magic when the player drifts. One car may cast Fire and Water spells, while the other may cast Air and Earth spells. Each school has two available spells, one of which is cast after drifting, and the other when the player completes a Donut.

Spells of different schools may interact with each other, creating powerful combos: Fire spells combine with Air spells, while Water spells combine with Earth ones.

Gameplay Overview

Game Modes

CARPG features a single game mode: King of the Hill. In it, the players compete to see who can remain inside a certain radius of the point of interest (henceforth called **Hill**) for the most time. There are several points of interest throughout the map which are activated or deactivated without notice, thus creating points of convergence for the players but also keeping them on the move throughout the arena.

Car Classes

There are 2 classes of cars from which the player can choose his war rig. These classes differ in the spells that are available for them to cast. Each of them is able to cast spells from two **Schools of Magic**:

- **Fire**
  - Primarily for damaging, causes opponents to burn for tick damage (damage over time)
- **Water**
  - Used for controlling or disrupting other player's movements,
- **Air**
  - Primarily for damaging, the air spells also disrupt player's maneuverability by stunning or lifting opponents
- **Earth**
  - Primarily for sectioning off the arena, earth spells are a powerful way of changing the environment to your advantage

The first class is able to cast Fire and Water spells, while the second is able to cast Air and Earth spells. This means that each class has an offensive and a support school at their disposal.

Spellcasting

As we've covered, spells are cast by drifting. There are two spells available to cast for each school of magic – the Drift Spell, and the Donut spell. Players charge the power of their spells by drifting. So a very fast and "successful" drift results in a more damaging, long-lasting, spell with a larger range (we'll get into details on how Drift Quality is calculated in the Spellcasting Feature specification section).

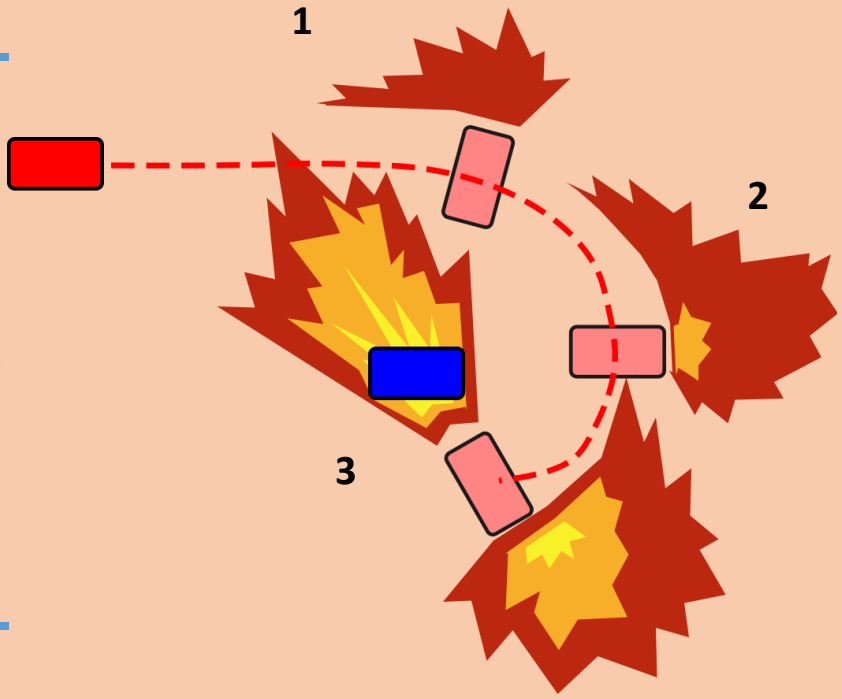
These spells are cast by pressing the Shoot button while charging a drift. If the drift comes full circle, the spell becomes a Donut Spell.

GAMEPLAY EXAMPLE - Spellcasting

In the diagram to the right, the red car casts a fire spell at the blue car (1).

The red car enters a drift. As the drift progresses, the spell build-up increases. The drift creates a growing damaging area behind the casting car (2).

After the drift is done, the red car casts a flamethrower spell directly in front of it, towards the blue car (3).



Controls

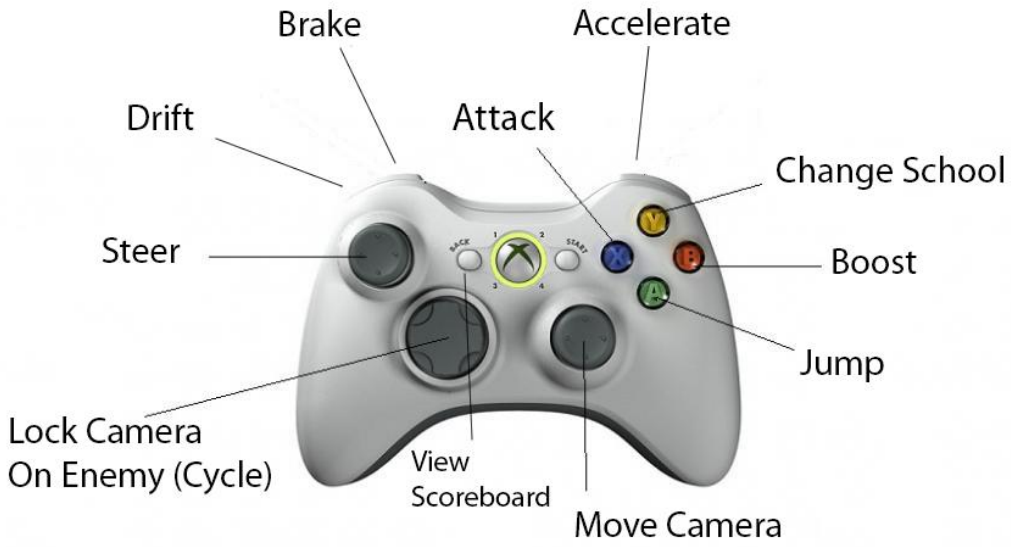
The game is best played with a controller. The following is a mapping of the controls to an Xbox 360 controller.

The attack button is used to cast all spells, and distinguishes which should be cast between the available ones by checking if the player is drifting, and if the player has closed a full donut.

The change school button cycles through the schools of magic available to the player's chosen car.

The lock camera button uses the horizontal buttons of the D-Pad to cycle through enemies. The down button of the D-Pad disables lock-on altogether without needing to cycle.

The Move Camera thumbstick can be pressed to temporarily disable the lock-on camera and go into free view, for as long as the player holds the thumbstick.



Requirements

High-level Gameplay Requirements

Movement

- **Steering & Throttle:**
  - Players can move a car semi-realistically in a 3D space by steering, and using throttle and footbrakes.

- **Jumping:**
  - Players can jump for a short while, and is able to rotate the car in the air by steering (the car must rotate one 2 axes only, without rolling)

- **Boost:**
  - Players can boost for a short period of time, increasing not only their acceleration but also max speed.

- **Drifting:**
  - Players can use the handbrake to drift sideways, losing some speed. Drifting ties in to spellcasting. Refer to the spellcasting section for more information
  - Players must be able to drift a full donut, given that they attempt to do so at a given reasonable speed.

Systems

- **Health:**
  - Cars have a limited amount of health. When their health is depleted, they die and the respawn screen pops up. After a given timer, they may respawn at a determined location

- **Combat:**
  - Cars are not able to damage their team-members, but donut spells affect friendlies and teammates alike.

Spellcasting

- **Drift Spell:**
    - By drifting, players charge up their Drift Spell. The charge of the spell depends on distance travelled while in drift.
    - While drifting and for a short period of time afterwards, they may cast their spell by using the same button as the base spell. The damage and/or area of the spell is dependent on the drift charge. The drift charge gradually decreases after the player finishes drifting without casting

- **Donut Spell**
    - Working much in the same fashion as the drift spell, the donut spell relies on drift charge to cast.
    - If the player drifts a full donut, he will cast an alternate spell, often more defensive or area-based.

The Arena / Environment

- **King of the Hill**
    - The arena consists of 3 hills, of which one is activated at a time. The activated hill can be captured by drifting within a certain radius of it. The hill is neutral when it is first activated. Once a team captures the hill, it starts generating points – the team with the most points at the end wins. The team defending their point must try to prevent the opposing team from drifting within the radius. The attacking team must drift consecutively to build up the capture – the capture buildup gradually decreases while no one is drifting within the radius

- **Health Pickups**
    - The arena will have placed health pickups that fix the players' cars.

UI Requirements

Splash Screens

- VFS screen,
- Lazy Devs logo, and
- Unity

Main Menu Screens

- Main Menu
- Lobby
    - Create Room
    - Join Room
    - Loadout Screen
- Options Screen

In-Game Screens

- HUD
    - Health bar
    - Drift Meter
    - Minimap (undetermined)
    - Score
    - Timer
    - Capture Progress
    - Spell Active
    - Boost Meter
    - Off-screen Pointers (objectives + players)
    - Kill Feed

- Scoreboard Overlay
- Start / Death / Respawn / Loadout Screen
    - Choose teams
    - Car Type Selection

- Win Screen/ Lose Screen
- Gameplay UI
    - Other player health and names
    - Damage Dealt (undetermined)
    - Lock-on target

Functional Requirements

Gameplay Managers / Static Systems

- Cross-controller Input Manager
- Team Score System
- Respawn Loadout Menu
- Camera Controller
    - Free Camera
    - Lock-on Camera
    - Drifting Camera controls
- Hill-capture system / Game mode manager
- Helper Functions
- Extension Methods
- Screen Manager

Gameplay Individual Car Controllers

- Player Input Controller
- Player Movement / Steering Controls
- Drift Controls
- Spell-casting Controller
- Jump Controls
- Boost Controls
- Combat Controller
    - Friendly Fire Handler
    - Team handling
    - Health system
- Spells
    - Damage
    - Effects
    - Particle Systems
    - Spawning and De-Spawning
        - Flamethrower
        - Wildfire
        - Tidal Wave
        - Blizzard
        - Lightning
        - Tornado
        - Wall
        - Quake

- Spells Combinations
    - Wall of Wildfire
    - Thunderstorm
    - Fire Tornado
    - Geysers
- Spell power to particle systems' attributes controller

Networking Requirements

Local Server (LAN):

- Initial Setup
- Update Car Positions
- Update Capture Point Ownership
- Update Car States
- Update Game State (score, damage dealt)
- Error Handling
- Receive data from other players
- Broadcast data to other players

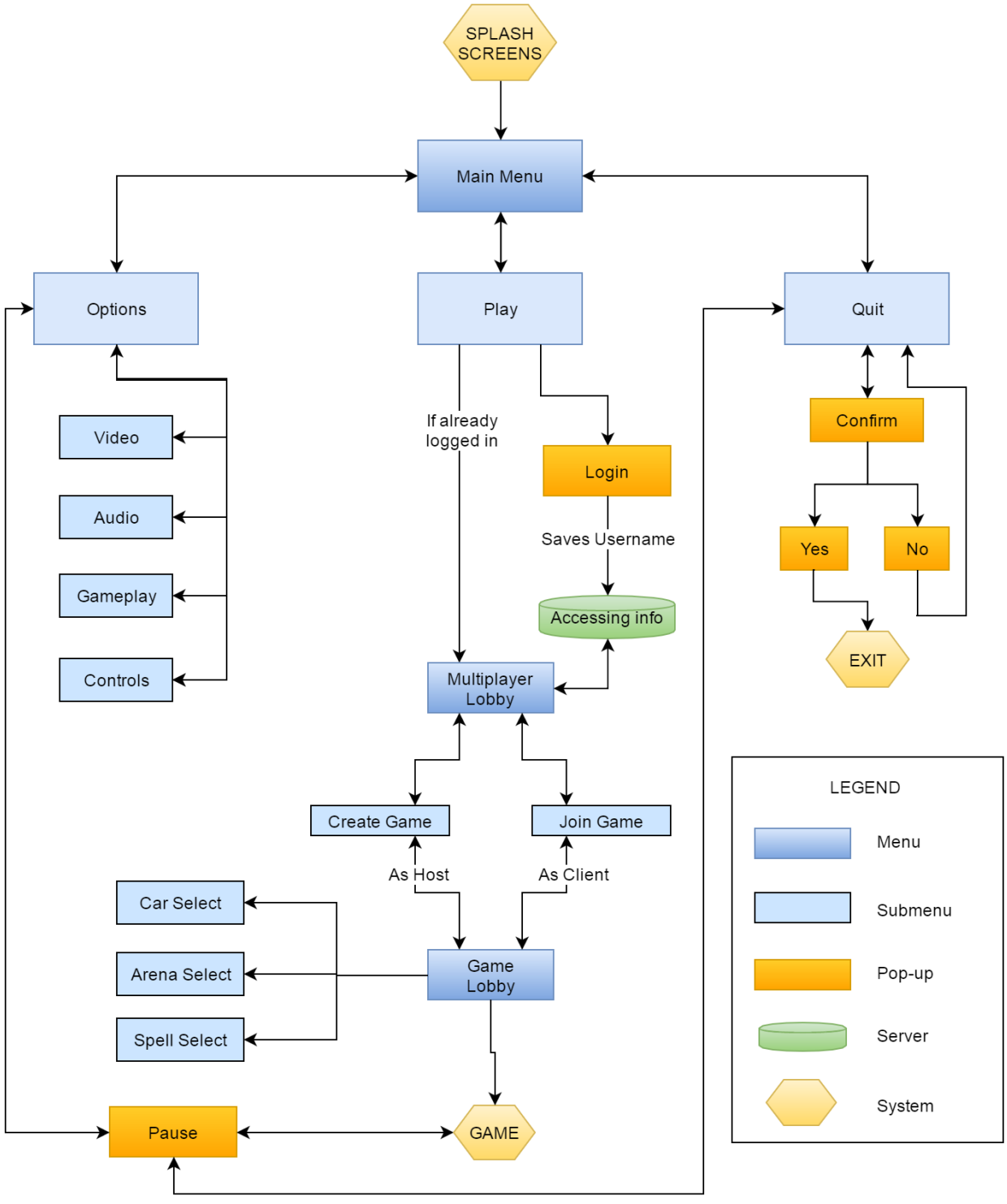
Client Side:

- Interpolation
- Extrapolation
- Client Updates

Server Side:

- Hosting
- Sever Manager
- Arena Manager
- Client Updates
- Register Players
- Anaylitics

UI Screenflow



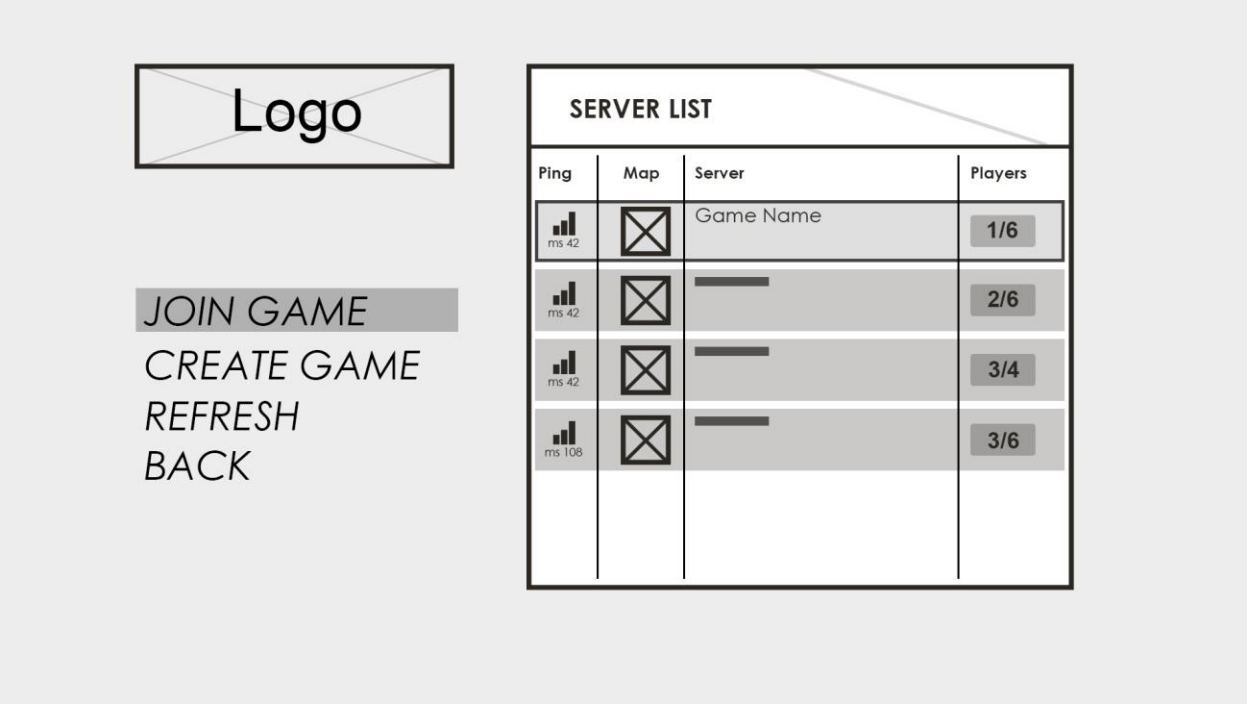
Screen Mockups

Main Menu

Menu selection on the left. The background is either an image or in-game scene of the cars.

Server List

Upon clicking PLAY from the main menu, players are brought to a screen which displays a list of games to join. This list contains each game's information such as player count and ping.



Game Creation

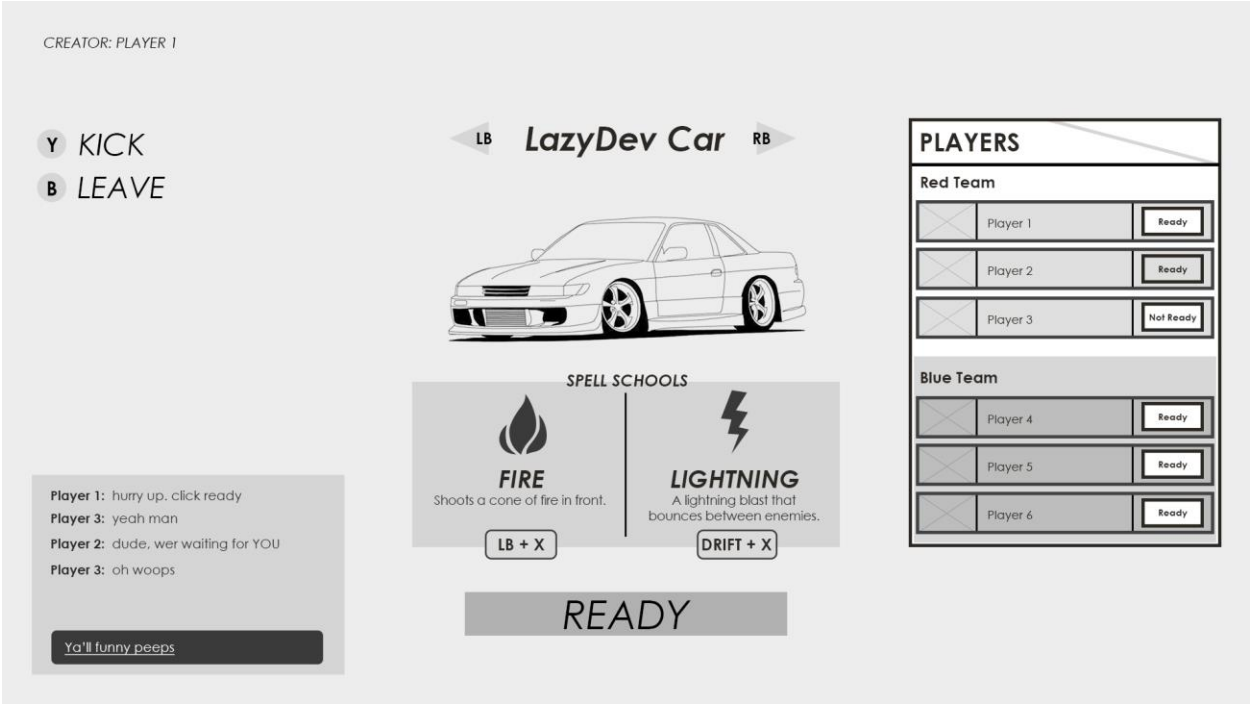
From the Server List menu, if the player chose CREATE GAME, they will be prompted to enter the game name.

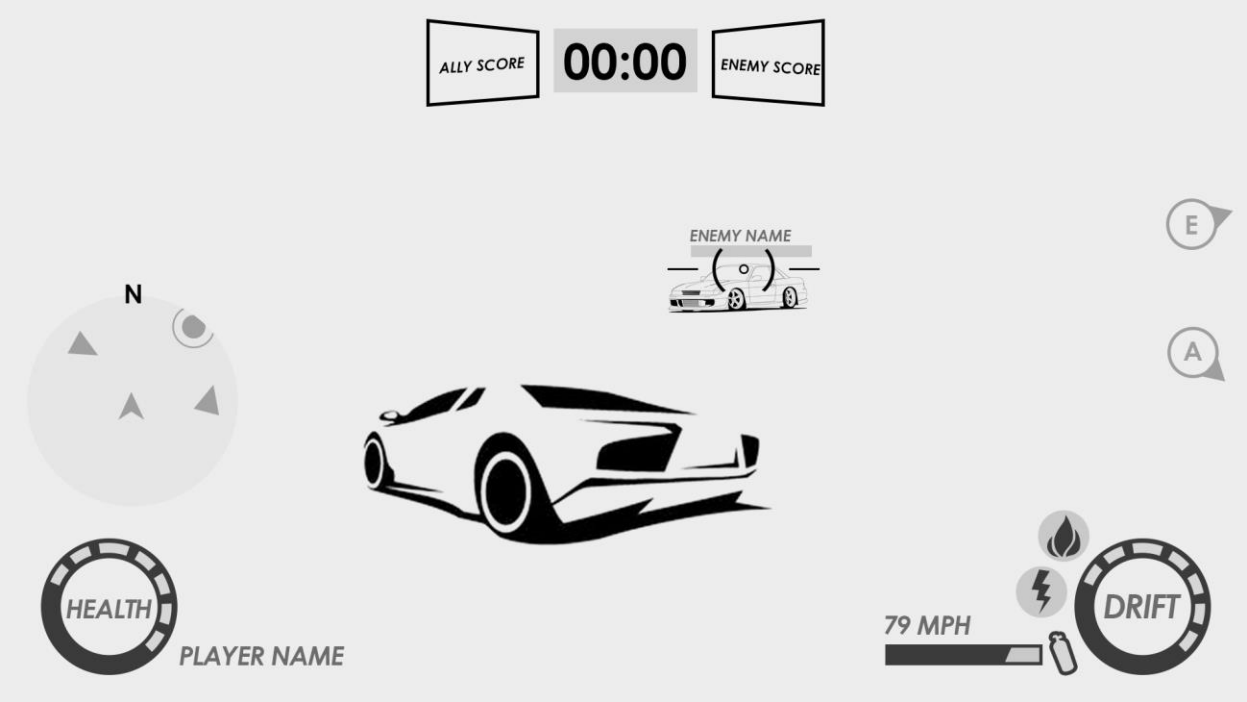
Lobby

Below image is the game lobby! This is where the fun is about to begin.

On the right side is a list of players currently in this game, they are all waiting to start the game. The left bottom is the chat panel. In the middle is where players can choose the car to deploy for the game, along with the description of the spell schools of the car. As indicated, players can push Right or Left Bumper buttons on the controller to switch between cars.

Once ready to begin, players can push the READY button at the middle bottom. Once all players are ready, a countdown will start. 5, 4, 3, 2, 1 – and the game begins.

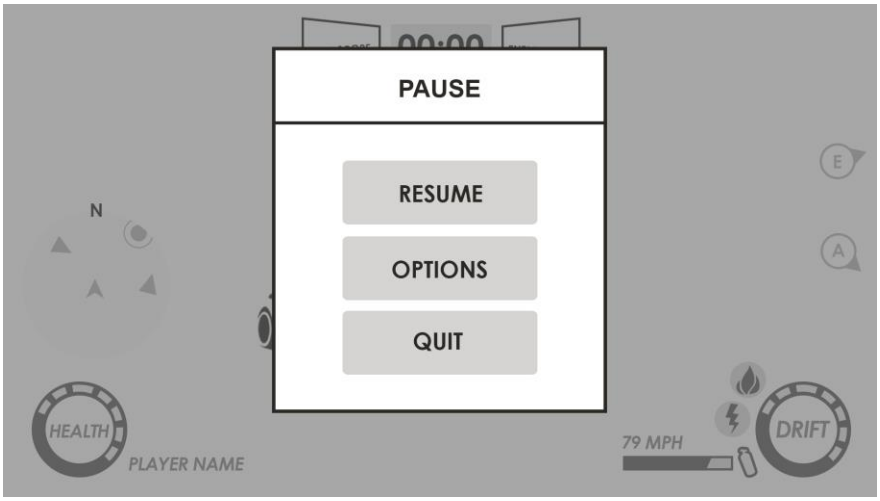




Heads Up Display

Above image is theHUD. There are multiple components to this -

Top: A timer for the current game, as well as the score of both teams.
Left: A minimap and the player's health bar
In-game: A lock-on indicator on enemies. The camera will also follow the locked on target.
Bottom Right: Drift Meter that fills up as the player drifts. Spell icons along with cooldown timer. And Nitrous meter that depletes upon boosting the car, along with a speed indicator.
Right: Off-screen indicators that tell the player about objects they cannot see on the screen.



Pause Menu

This pause menu contain is accessed by pressing “start” on the controller.

OPTIONS and QUIT have the exact same functionality as the ones in the Main Menu.

RESUME will just unpause the game.

Project Conventions & Organization

Overview

CARPG is a PC game made with the Unity3D engine, scripted in the C# language. For player input, it supports mouse + keyboard, though it is best played with a controller – it supports the Xbox 360 controller, and the Playstation 4 controller.

Project Segmentation & Ownership

There are a total of 6 people in-engine implementing the game:

- Two artists, implementing art and set dressing
- One level designer, white-boxing the arena
- Two programmers, of which:
    - One gameplay programmer, handling the implementation over players' controls and actions in the game, as well as implementing audio and visual feedback
    - One network and UI programmer, handling client-server communication, client-side prediction, extrapolation, and implementing UI art and code
- One balancing designer / particle systems implementation

No members should work areas over which they claim no responsibility, without prior clearance from the team members who do have a stake in them. For example, artists do not touch networking or gameplay code without asking the programmers for consent, nor should the programmers touch the implementation of the art and Shaders without first clearing with the artists for art direction.

The balance designer will link the particle systems of the spells with their actual gameplay effects. We develop the game's architecture so that this design can be handled in editor, but it is reasonable to assume that the balance designer will need to access and code for the spells, and so it is considered that he also has responsibility over it, along with the gameplay programmer.

Version Control

The project will use the Perforce software for version control. The folders "Library" and "Temp" should NEVER be pushed onto perforce.

Project Organization

CARPG will be separated into 2 folders, one of which will hold the **Project,** and the other the **Assets in development**. The Project folder will hold the unity project which will build to the final game, and the Assets in Development folder will hold all art & audio that is in development and not yet implemented in the game, along with every asset's progressive iterations. This is so that pushing / pulling and merging through Perforce does not imply downloading / uploading myriad non-vital assets. The Assets in Development folder will live in the VFS Student Shared Storage, while the Project will be synced through Perforce.

The Assets in development folder will be broken down into the following structure:

- Assets in Development
    - Graphic
        - Gameplay
            - Cars
                - Car 1 (or Car [name])
                    - Model
                    - Textures
                - Car 2
                - Particles
                    - Per particle effect folder structure (e.g., "Smoke")
            - Environment
                - Props
                - Hero Pieces
                - Terrains
            - Spells
                - Particles
                    - Per school folder structure (e.g., "Fire / Flamethrower")
                - Meshes
                    - Models
                    - Textures
        - UI
            - UI Screens
                - Per screen folder (e.g., Main Menu)
                    - Per UI element folder structure (e.g., "Join Room Button")
    - Audio
        - Soundtrack
            - Per screen folder structure (e.g., "lobby", "gameplay")
        - SFX
            - Gameplay SFX
                - Movement
                    - Per procedural sound folder structure (e.g., "Engine Rev")
                - Spells
                - HUD feedback
            - UI SFX
                - Per screen folder structure (e.g., "Lobby / Play button feedback")

The Project folder will be broken down into the following structure:

- Project
    - Builds
    - Library
    - Project Settings
    - Temp
    - Assets
        - Art
            - Cars
            - Environment
            - Particle Effects
            - UI
        - Materials
            - Physics Materials
            - Cars
            - Environment
        - Packages
        - Resources
        - StreamingAssets
        - Prefabs
            - Game Managers
                - Camera
                - Game Manager
                - Networking
            - Player Objects
                - Cars
                    - Models
                    - Particle Effects
                - Spells
                    - Per school folder structure (e.g., “Fire School”)
                        - Particle Effects
                        - Spell Objects
            - Environment Objects
                - Game Mode vital objects
                - Props
            - Terrains
        - Scenes
        - Scripts
            - Camera
            - Player
                - Movement
                - Spell Casting
                    - Per school folder structure (e.g., “Fire School”)
                        - Per spell folder structure (e.g., “Flamethrower”)
            - Networking
            - UI
                - UI Screens
                - Gameplay UI
                    - Audio / Visual Feedback
                    - HUD
            - Static Classes / Helpers

Naming Convention

The Assets in the Assets in Development folder follow the naming convention:

Models:

AssetName_Owner_001f.mb

Textures:

AssetName_AoMAP_Owner_001.targa
AssetName_AlbedoMAP_Owner_005.targa
AssetName_NormalMAP_Owner_003f.targa
AssetName_MetalnessMAP_Owner_002.targa
AssetName_RoughnessMAP_Owner_011f.targa

Soundtracks:

AssetName_System_Owner_001f.wav

SFX:

AssetName_System_Color_Owner_001f.wav

Where:

- **AssetName** depicts which asset is being worked on (e.g., StonePillar). AssetName does not contain spaces – they are instead implied by capitalizing the first letter of each word.
- **Owner** is the first name of the author of the asset
- **AoMAP** refers to Ambient Occlusion maps, etc.
- **System** depicts which system, if any, it pertains to (e.g., “Flamethrower” for a particle system or “CarRev” for procedure sound)
- **Color** is optional for assets that have multiple versions with different “hues”, such as a particle or slightly altered SFX (e.g., distinguished green fire from red fire, or slow car rev from fast car rev)
- The **Three digit number** counts the number of iterations on the piece
- The **f** that follows the digits determines whether the author believes the piece is finished

Code Conventions & Best Practices

The Scripts in the Project folder should be named such that it is apparent what the functionality it is they implement. Classes named Manager are assumed to be either static or singletons. Classes named Controller are assumed to implement in-game physical actions.

Scripts that implements spells should have the same name as the spell. If they require any other scripts to function, those scripts should be prefaced with the spell they pertain to. So, if we're implementing a script on the individual collision spheres cast by a flamethrower, the script name should be "**FlamethrowerCollisionSpheres"**

Naming Clarity - Variables

The number 1 rule for variable naming is that a variable name cannot be too long. It should be as long as it needs to convey what functionality it implements. This could mean we have a variable called m_DriftActiveTorqueInitialImpulseMagnitude. It is long and unwieldy, but extremely clear and direct. This will help not only balancing but code readability, as well as reduce the need for commenting.

The accessibility of the variables is not dictated by the need to expose them in the editor, but by the architecture of the code. Not all exposed variables are public! We can use Unity's [SerializeField] to expose variables that shouldn't be accessed by other scripts

Naming Clarity – Methods Best Practices

The number 1 rule for Method name is, you guessed it, a method's name cannot be too long. It is as long as it needs to be to convey what its functionality is. If the method returns a value, the method name should be clear about what the returning value is, and not ambiguous.

If we were developing a method for checking whether the player is drifting, and we want to return a bool, a bad example would be "**CheckPlayerDrift( )**". It does not imply a return value. A better name would be "**IsDrifting( )"** because it implies a return bool and we already know what a returning false or true mean (IsDrifting() = false is obvious in implying that the player is not drifting)

It is preferable to implement a summary for a method than it is to comment on its code. The summary should clearly state what the method does, what it returns, how it returns that value, and whether it may have unpredicted results for any other developer aside from the one who implemented it (e.g., if the method is costly, if it changes other values in the class)

Scripts Implementation Rules

- **Member Variables** are prefixed with **m_**
- **Public Variables** start with a **capital case**
- **Private Variables** start with **lower case**
- **Constant Variables** are all uppercase (“const int m_MAXSPEED”)
- **Methods** imply their return value
- **Methods that return bools** assert an statement, and begin with a verb (e.g., “IsOnFire( )”, “CanSpellcast( )”, DoesHideMember( )”
- **No method is longer than 50 lines of code**
- **Scripts that grow past 300 lines of code** have to be separated into **regions**
- **Event Scripts** should make clear which event they fire on: (e.g., OnPlayerDrift( ))

Scripts Organization

Scripts’ variables, properties and methods are arranged in the following manner, from top to bottom:

Library Includes
Required Components
Class declaration
    Public Member Variables
        gameObject’s other components
        other game object references
    Public Member Properties
    Serialized private Variables
        Organized by functionality
    Private Member Properties
    Initialization Methods
        Methods use to initialize the object
    Update Methods
        Methods that are run every frame
    Per functionality regions
        Other Methods
    Event Methods (OnTrigger, OnCollision)

Technical Implementation Specification

Movement Implementation

Overview

Car Movement is implemented using Unity's Wheel Colliders Physics. This section will provide some information about how they work and how we can tweak their values to our advantage. This will be useful reference for future balancing of the controls.

Wheel Colliders simulate real-life physics for grounded vehicles to an astonishing degree. A Car with a single Rigidbody is placed in the scene, with four attached WheelHubs as children. Each WheelHub, aside from its mesh, has a WheelCollider component.

The car moves by computing the force generated by the friction between the tires (simulated by the colliders) and the ground. To compute this force, there are several factors that are taken into consideration.

In the diagram to the right, we see a representation of the WheelCollider:

It computes the force generated by the friction between the tires and the ground, as the motor adds a "fake" torque on the wheel.

The WheelCollider casts a downwards ray to simulate suspension, adjusting the force accordingly.

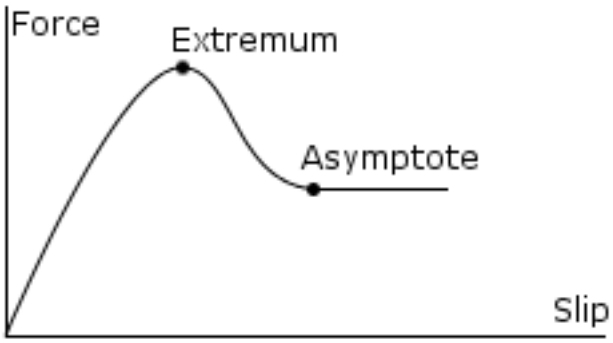
The Car absolutely must have a single Rigidbody attached to it.

Wheel Torque

The car applies a torque to the wheels, making them rotate and generate the friction needed for movement. The car implements a motorTorque exposed variable. However, increasing the motorTorque by itself only makes the tires slip without actually increasing acceleration of the car. A careful balance between all of the parameters must be reached

Wheel Friction Values

Wheel Friction values can be altered to change the car's adherence to the floor. We can change the values on forward friction of the tire, and the sideways friction. Changing stiffness seems to produce unpredictable results. Changing the friction makes the car slide easier. Having different friction values for forward and rear tires seems to be a bad idea because even a slight change produces a big loss of control.



The resulting force is at its maximum when the slip value reaches the Extremum, and tends to the Asymptote value as it further slips.

Center of Mass Offset

Lowering the center of mass of the car makes it less likely to roll into a spin

Downforce

There is a downwards force being applied to the car whenever it is grounded, amplifying the car's ability to steer. Increasing this value makes the car handle better but also more difficult to build and lose momentum

Steer Helper

The steer helper is a factor (0 to 1) which determines how much the car adheres to its forward-facing direction. 0 is raw physics, 1 is full grip. Changing this value on drift is crucial

Traction Control

The traction control is a factor (ranged 0 to 1) that determines how much the car should auto-adjust the motor torque on the wheel when they slip. A value of 1 means that the car fully adjusts the acceleration input to optimal torque, making it impossible for the tires to slip

Breaking the Realism

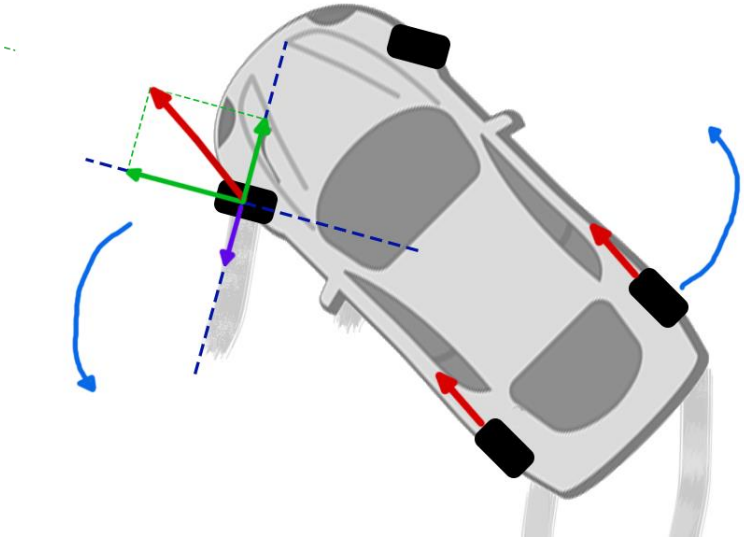
After having achieved a faithful physics simulation to which players can relate, we begin gamifying the controls:

- We clamp the steering angle while the player is not drifting so he cannot go into a drift accidentally

And while drifting:

- We apply a forward-facing force on the back wheels while drifting that is dependent on the acceleration input (diagram below)
- We apply a Torque directly on the car (bypassing the wheel collider physics) that is dependent on the steering input
- We apply an Impulse Torque for the first steering input, so the car snaps its rotation to a slight sideways trajectory
- We apply a counter-steer amplification factor to the steering Torque, controlling how easy or difficult it is to counter-steer a drift

*By applying a **forward-facing force** on the back wheels that is dependent on throttle, the acceleration input results in a boost to speed as well as a **torque** on the car due to the front wheel's **Sideways Friction** force counteracting the **force**'s **perpendicular contribution**.*



Jumping

Players can jump anytime their vehicle is grounded. Jumping simply applies an Impulse Force on the car's Rigidbody that lifts it off the floor to a certain height.

When the players' vehicles are airborne, they may apply Torque and consequently rotate the car to their liking. Steering left and right rotates the car around the global Y axis, while steering up and down rotates the car around the car's local X axis.

Boosting

Players can boost if they have enough BoostCharge. BoostCharge is depleted while the player holds the boost button, and is recovered automatically if the player is not boosting. After having boosted, the recovery is delayed by a tunable amount of time.

Camera Implementation

Basic Rig

CARPG features a third person camera that is positioned behind and above the player's car on a pivot when they begin a match in the arena. As the player turns the camera follows and leads ahead of their movement so that they have a better view of what lies in their intended direction.

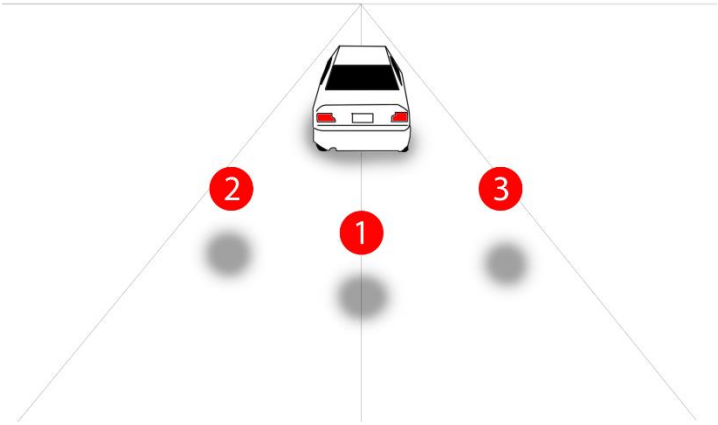
This means the rig is tri-pivotal, blending between positions as the player steers. The camera rig is physics-based, apparently being pulled by a spring: it pulls back as the player accelerates, and comes closer when he loses speed. The player may also move the camera in between the pivots with the right thumbstick.

Boosting

When players use their boost mechanic the camera pulls back away from the car and a visual effect of wind is played around the bounds of the viewport. Should the player turn or drift in this mode the camera will remain in its previous pulled back state until the added velocity of the boost declines from the effect ending or outside forces.

In the diagram, the red spheres represent the camera pivots. They follow along the car's tail, blending between their positions. If the player curves right, the camera is pulled towards 2, with the car framed to the left and the path ahead in the center.

In lock-on mode, the pivots move farther back, increasing the distance to the car and the offset between them

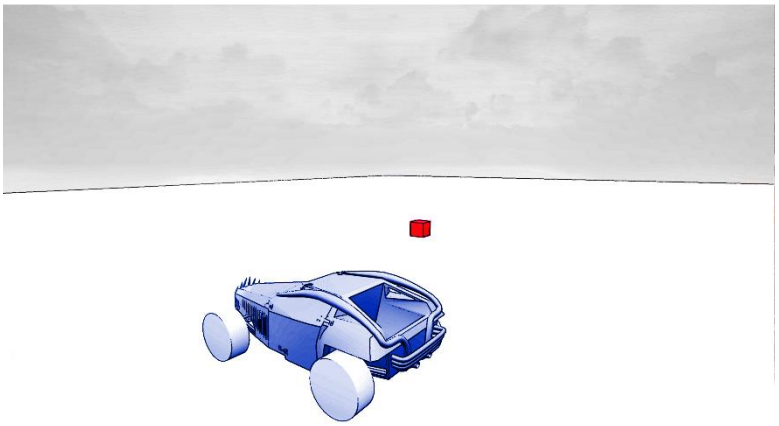


Lock On Rig

Players can lock their camera onto enemies. By pressing the Lock-On keys (see controls section), he can rapidly cycle through enemies and also the free camera. The Locked On camera is deemed very important for combat, since the drifting is by nature chaotic in orientation.

The camera will no longer lead ahead of the player's movement but instead keep the targeted enemy in the center of the viewport. The player's car is still visible at all times, in the foreground with the enemy in the background. In the locked-on mode, when the player drifts, the camera will shift to the outer side of the drift circle to allow for a clear visual of the center of the drift, where the player will eventually cast the spell. The enemy car is still in the center.

The tri-pivotal rig widens the distance between its pivots, and now instead of being placed directly behind the car they are now placed with the player's car in between the pivots and the enemy.



Example of the camera's Lock-On Rig. The player controls the blue car while keeping the dreadful enemy red cube in view.

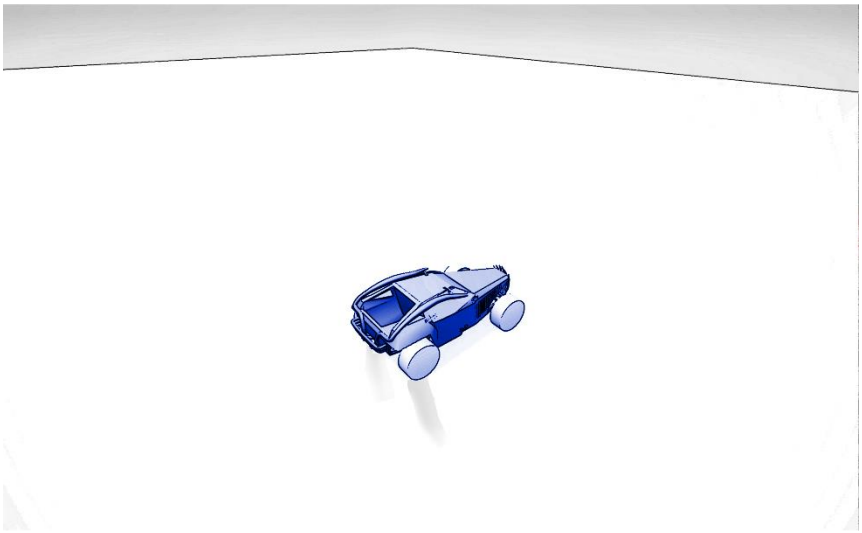
The controls are not adjusted, in that pressing left causes the car to steer left just as it would in the standard rig

The illusion of Speed - VFX

In order to minimize chaotic propagation of collisions through the network, we minimize the speed at which the game plays out. This means that speed is an artifact of camera special effects that convey a sense of speed.

When players begin moving very fast the camera tracks in slightly and a **vignette** circles around the edge of the camera's viewport. Since stretches are not commonplace in the game, terminal velocity is seldom hit. We place a **motion blur** and a slight **camera shake** on the camera at high speeds that does not impact the gameplay a lot since most of it is done at relatively low speeds. The camera resists the lead it gives the player when they turn to simulate a harder control over the car when traveling at terminal velocity.

Drifting



Example of the camera's drifting rig.

The camera shifts sideways, pulling back and up to offer a broader view of the terrain. The player can then see his car's tail, as well as the drift rear-wheel particles, but he can also see targets ahead.

This rig becomes active when the player drifts without being locked-on, which means he can adjust the camera angle with the right thumbstick

Object Culling

Because the terrain plays on elevation and obstacles, the lock-on camera never lacks feedback for what the player is targeting. The targeted enemies are rendered over obstacles that stand between them and the camera, with a special effect depicting the color of their team.

The player's car is always in view, since the camera moves towards the player's car any time there's an obstruction in the way, just enough to be in front of the blocking object.

Spells Implementation & Specification

Drift Quality - Translating Movement to SpellPower

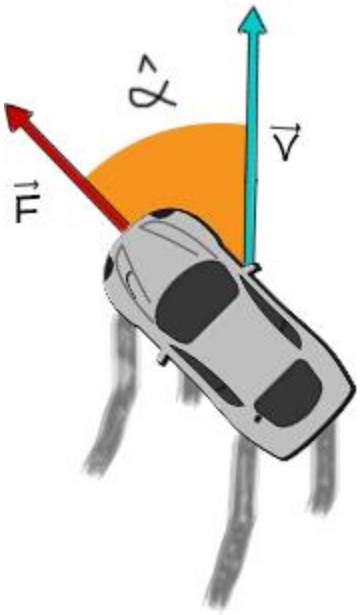
Each car has two available spell-schools to toggle between. The spellcasting logic of the car is implemented in a SpellCasting script, which will compute the drift power, and do all other movement-related logic such as the jump-cast.

Spell power increases with distance travelled while drifting, as well as with the angle between the car's forward-facing direction and the car's velocity heading. This means that the more sideways and long a drift is, the more powerful the spell becomes.

F *-> Normalized Forward-Facing Vector*
V *-> Normalized Velocity Bearing*
α *-> Angle between F and V*

The drift quality is computed by factoring the angle between the velocity and forward, and also the total distance travelled (ΔX). Frame by frame, this is the equation that gives us the drift quality, if we assume a magnitude M.

$$\Delta DQ = |DOT(F,V)| * \Delta X * M$$
$$DQ = SUM(\Delta DQ)$$



The SpellCasting script will then call the active spell school's script (also a component of the car), and ask to cast the appropriate spell (drift or donut)

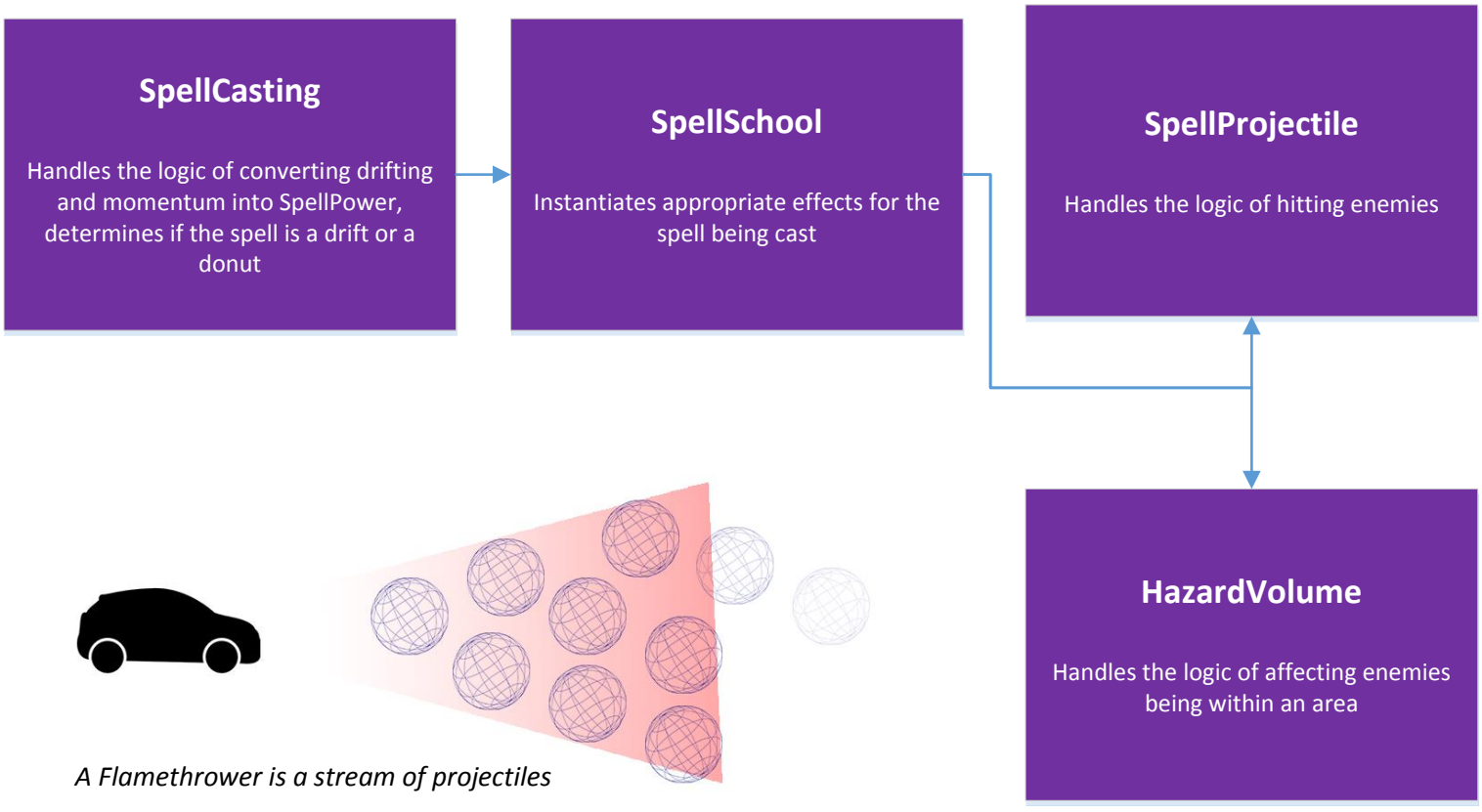
The FireSchool script will then instantiate the effects that the spell requires.

Spell Types

Spells vary a lot in terms of functionality. But at a high enough level, they can be broken down into two distinct types: Projectile-based, and Area-based.

Projectile-based spells fire streams of collision spheres that handle all the spell effects themselves. The spell only hits insofar as its fired projectiles do, and is thus able to deal half damage depending on the player accuracy. An example of a projectile spell is the Flamethrower.

Area-based spells instantiate hazard volumes in the world that place a given effect on players that enter its bounding volumes. An example of an area-based spell is the Lightning, which fires instantly, hitting all enemies within a given radius.

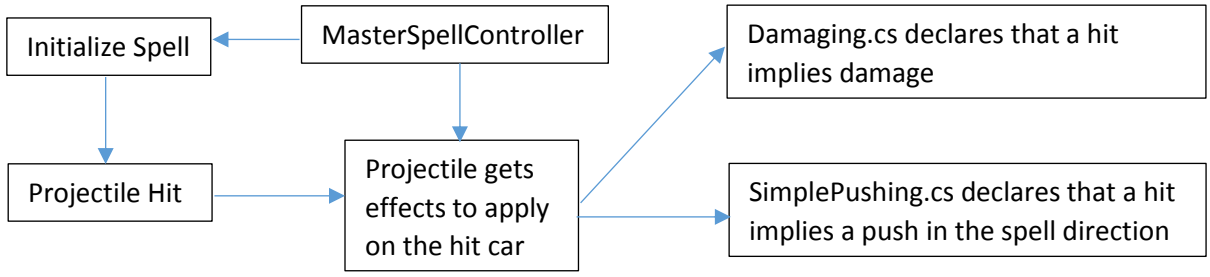


A Flamethrower is a stream of projectiles

Building Modular Spells

Spells, as given by their names, are not scripts but prefabs. Spells are, in fact, collections of scripts that describe their effects.

For instance, there is no Lightning.cs script that implements that particular spell's stun effect. Rather, a MasterSpellController.cs holds a reference to the spell type (projectile or aoe), and to a list of effects that the spell inflicts, including but not limited to the Stun. And so, spells themselves become prefab collections of scripts of the effects that compose them.



EXAMPLE – Flamethrower Components

The MasterSpellController of the Flamethrower prefab declares that it is a Projectile-based spell, and that its effects are: Damaging, and SimplePushing.

For each projectile instantiated, the MasterSpellController attaches a pushing and damaging effect with the parameters defined in the Damaging.cs and SimplePushing.cs scripts.

If our design led us to conclude that a flamethrower should stun as well, we could add this functionality simply by adding a Stun.cs script to the spell prefab.

Architecting the spells as collections of components allows us to assemble them as Lego, speeding up iteration and the response time to new Design conclusions gained from playtesting. If we, say for balance, wanted a flamethrower that also stuns opponents, we could simply attach a Stun.cs spell effect onto the prefab, and the MasterSpellController would handle passing it onto the projectiles.

Spells Specification

This section details how each spell works, and which variables we need exposed in editor to balance them.

The Flamethrower

Instantiates a cone of fire that damages and pushes enemies. It fires a stream of invisible projectiles in front of the car, for a duration and damage given by the DriftPower

Wildfire

Instantiates a small flame that spreads clones of itself to nearby areas or cars. If a car comes within a given radius of the flame or one of its clones, the wildfire spreads to the car, dealing damage over time. Cars that are lit with wildfire also spread it to other cars and nearby areas. The effect has a total duration after which all of its flames die down.

Tidal

A wave of water shoots out of the front of the car, pushing all cars that it comes into contact with. The size of the wave is dependent on DriftPower.

Blizzard

Instantiates a cloud that slows down cars that travel inside it, while also pushing them in random directions so as to simulate strong chaotic winds.

Lightning

Cars within a given radius of the casting player are stunned and unable to control their cars for a certain duration. The spell is instantly cast, meaning that the opponents have to be within the radius on the same frame that the spell is cast.

Tornado

Instantiates a Tornado in the arena that lifts and rotates cars around it. It deals a little damage over time.

Wall of Rocks

Instantiates a mesh in the terrain depicting a wall of spiked rocks. If the wall hits a player while it is forming, it applies an upward impulse that pushes them off. The wall remains in the arena for a duration that is independent to the actual spell duration.

Quake

Kicks up a cloud of dust. Players that drive through the area are lifted consecutively by small upward impulses, rendering drifting impossible. Cars within the quake have a shake applied to their camera.

Spells Components

This section details all the spell effects / components that we expect to have, and gives an overview of which components each spell implements. To see all of the possible spell effects, please refer to the SpellCasting UML Diagram.

With these parameters, we are able to construct all the spells with little to no custom code. All spells must declare a MasterSpellController that will handle passing the effects onto the hit car.

Flamethrower

ProjectileBased
ApplyDamage.cs
ApplyForceImpulse.cs

WildFire

AreaBased
ApplyDamageOverTime.cs
ApplySpreadEffect

Tidal

AreaBased
ApplyDamage.cs
InstantiateMovingMesh.cs
ApplyImpulseForce.cs

Blizzard

AreaBased
ApplySlow.cs
ApplyConsecutiveEffects.cs
ApplyImpulseForce.cs

Lightning

AreaBased Instant
ApplyDamage.cs
ApplyStun.cs

Tornado

AreaBased
ApplyDamageOverTime.cs
ApplyLift.cs
ApplyConcentricForce.cs

Earth Wall

AreaBased
InstantiateStaticMesh.cs
ApplyForceImpulse.cs
ApplyDamage.cs

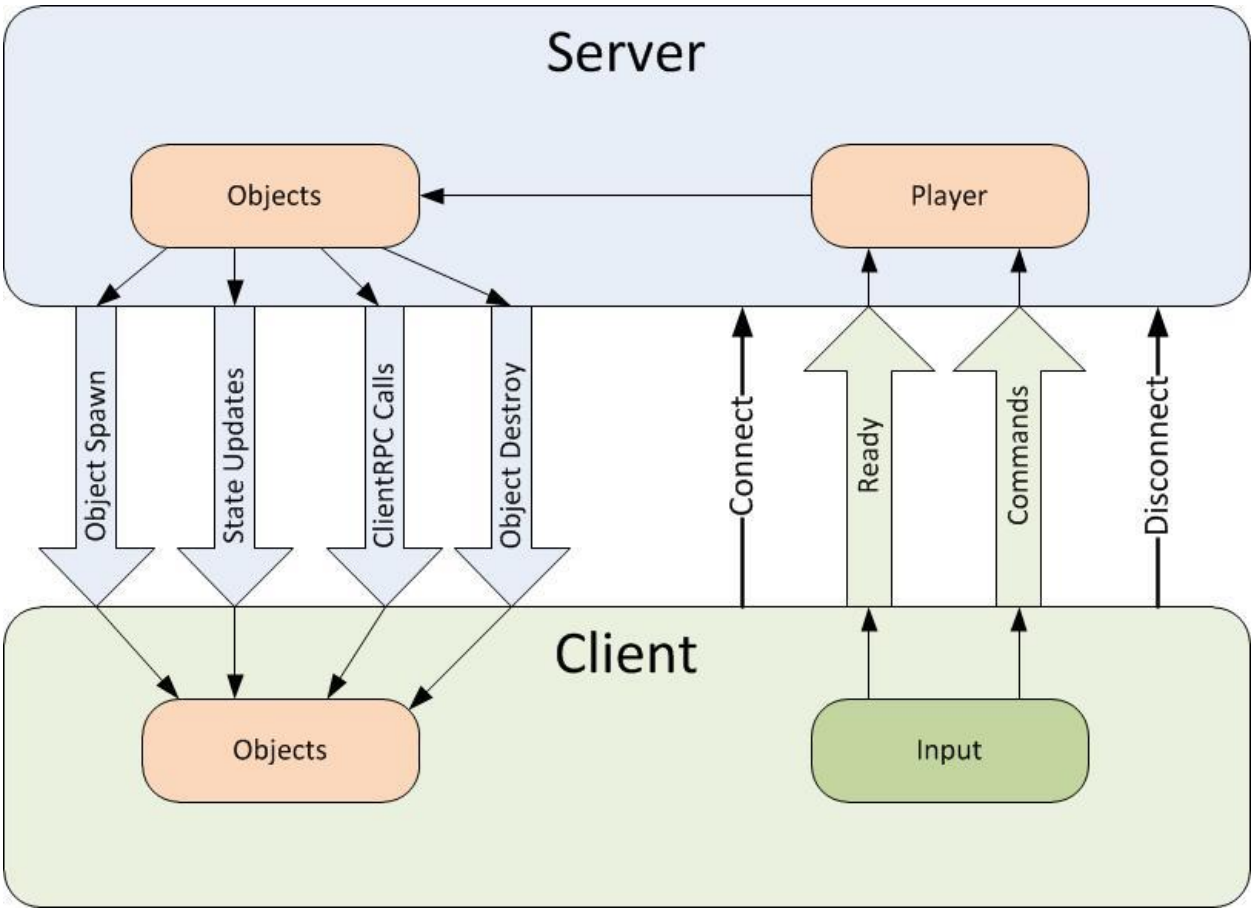
Quake

AreaBased
ApplyConsecutiveEffects.cs
ApplyLiftImpulse.cs
ApplyCameraShake.cs
ApplyDamageOverTime.cs

Networking Implementation

• Client/Server model.

Unity's Unet uses a Client/Server model, however the server is a player as well (AKA the "host"), so it seems like Peer 2 Peer, but it is not.



• Usage of RPC

There are multiple ways to use Remote Procedural Calls in Unet. The main 2 are:

- Commands – runs functions on the server using data on the client.
  - I.e. Player controls and casting spells

- ClientRPC – runs functions on all clients using data from the server
  - I.e. Capture Point state chagnes and score updates

Hardware and Software Requirements

Software	Required For	Licenses
Unity 5 Pro	To make the Game. We will use multiple tools within Unity	6
Unet	To make the game multiplayer	Free
Shader Forge	For creating shaders, without coding from scratch.	2
Autodesk Maya	For Modelling 3D art assets	2
Z-Brush	Additional Modelling 3D art assets	2
xNormals	3D Models Normal Map Generator	Free
Quixel	3D Texturing	2
Photoshop CC	To make UI art and textures for 3D objects	6
Adobe Premiere Pro	To make a trailer for the game	1
Wwise	Unity Audio Implementation	2
Unity Pro Builder	White-boxing level editor plugin for Unity	Free
Perforce	Source Control	4
Microsoft Visual Studio	For programmers to write code	4
Google Drive	Create docs, presentations, task lists, and spreadsheets	Free
Microsoft Office	Create docs, presentations, task lists, and spreadsheets	6
Slack	Communications and file sharing	Free

Hardware	Amount
Windows PC + Mouse + Keyboard	6
Game Controllers	6
Wacom Intuos Tablet	3
Monitors (2 per person)	12

Technical Diagrams

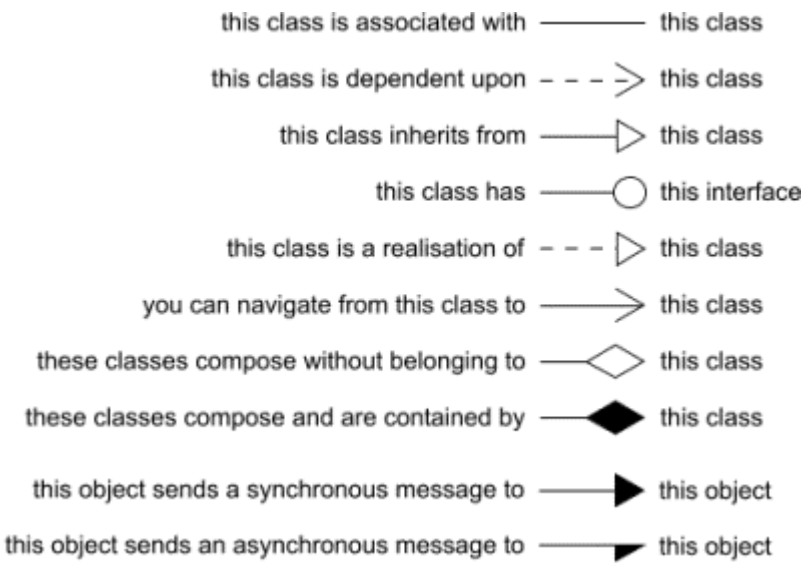
UML Overview

This section gives an overview over the architecture of the project through UML diagrams for the classes implemented. In Unity, relationships between scripts is not always hierarchical (inheritance), but a single GameObject has multiple scripts which interact with eachother.

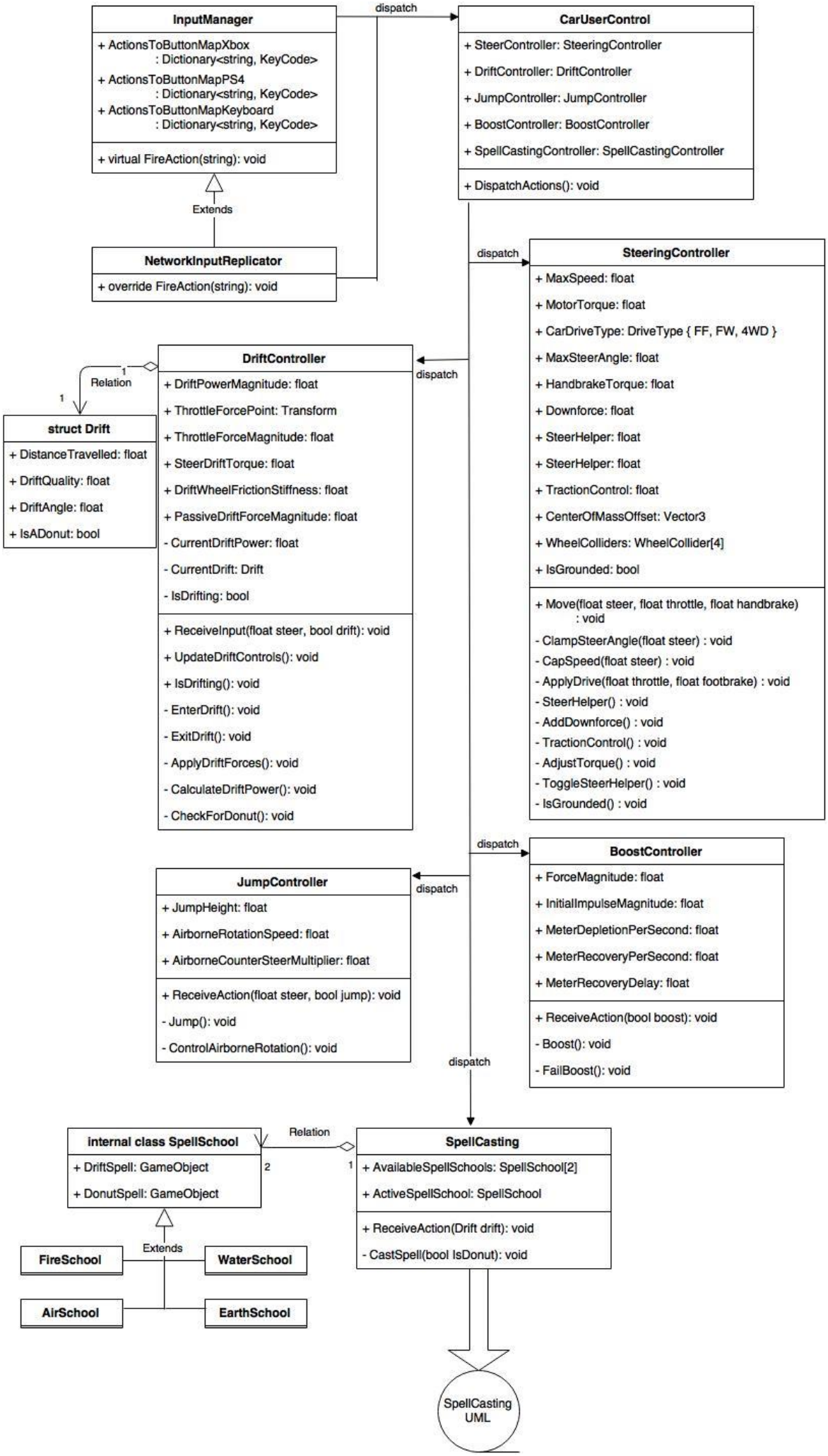
The Car UML Diagram shows all the scripts which will be attached to the car, as well as giving an overview of how they communicate to one another. It implements mostly movement-handling scripts, as well as one SpellCasting script which will communicate with the SpellCasting UML Diagram. They were separated for readability, as their functionality is sectioned enough to warrant separation

The SpellCasting UML Diagram picks up where the Car UML Diagram left off by describing how the instanced spell's scripts behave and relate to eachother.

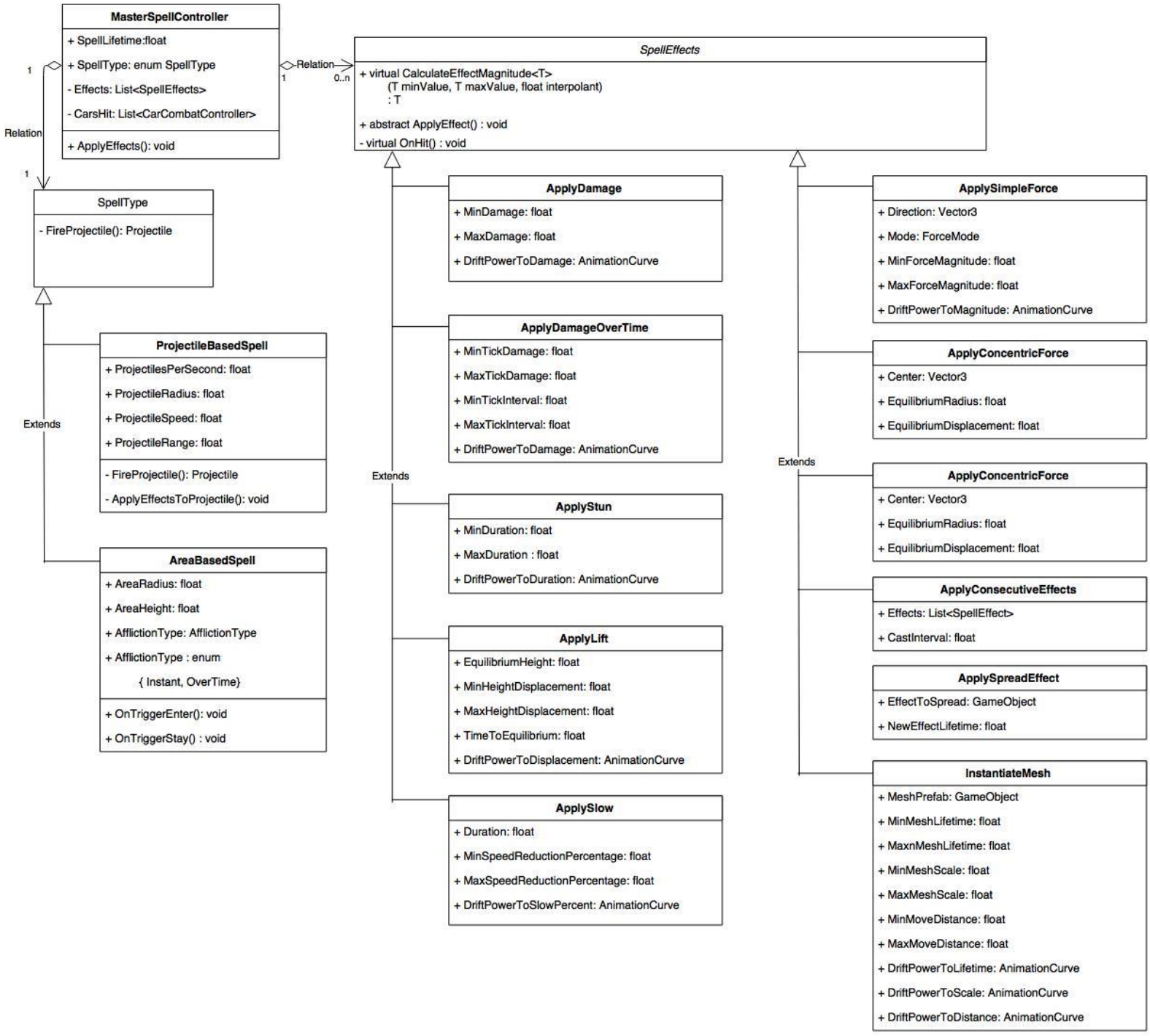
Legend



Car Diagram


InputManager
+ ActionsToButtonMapXbox : Dictionary<string, KeyCode>
+ ActionsToButtonMapPS4 : Dictionary<string, KeyCode>
+ ActionsToButtonMapKeyboard : Dictionary<string, KeyCode>
+ virtual FireAction(string): void
dispatch
CarUserControl
+ SteerController: SteeringController
+ DriftController: DriftController
+ JumpController: JumpController
+ BoostController: BoostController
+ SpellCastingController: SpellCastingController
+ DispatchActions(): void
Extends
NetworkInputReplicator
+ override FireAction(string): void
dispatch
SteeringController
+ MaxSpeed: float
+ MotorTorque: float
+ CarDriveType: DriveType { FF, FW, 4WD }
+ MaxSteerAngle: float
+ HandbrakeTorque: float
+ Downforce: float
+ SteerHelper: float
+ SteerHelper: float
+ TractionControl: float
+ CenterOfMassOffset: Vector3
+ WheelColliders: WheelCollider[4]
+ IsGrounded: bool
+ Move(float steer, float throttle, float handbrake) : void
- ClampSteerAngle(float steer) : void
- CapSpeed(float steer) : void
- ApplyDrive(float throttle, float footbrake) : void
- SteerHelper() : void
- AddDownforce() : void
- TractionControl() : void
- AdjustTorque() : void
- ToggleSteerHelper() : void
- IsGrounded() : void
1
Relation
1
DriftController
+ DriftPowerMagnitude: float
+ ThrottleForcePoint: Transform
+ ThrottleForceMagnitude: float
+ SteerDriftTorque: float
+ DriftWheelFrictionStiffness: float
+ PassiveDriftForceMagnitude: float
- CurrentDriftPower: float
- CurrentDrift: Drift
- IsDrifting: bool
+ ReceiveInput(float steer, bool drift): void
+ UpdateDriftControls(): void
+ IsDrifting(): void
- EnterDrift(): void
- ExitDrift(): void
- ApplyDriftForces(): void
- CalculateDriftPower(): void
- CheckForDonut(): void
dispatch
struct Drift
+ DistanceTravelled: float
+ DriftQuality: float
+ DriftAngle: float
+ IsADonut: bool
dispatch
BoostController
+ ForceMagnitude: float
+ InitialImpulseMagnitude: float
+ MeterDepletionPerSecond: float
+ MeterRecoveryPerSecond: float
+ MeterRecoveryDelay: float
+ ReceiveAction(bool boost): void
- Boost(): void
- FailBoost(): void
dispatch
JumpController
+ JumpHeight: float
+ AirborneRotationSpeed: float
+ AirborneCounterSteerMultiplier: float
+ ReceiveAction(float steer, bool jump): void
- Jump(): void
- ControlAirborneRotation(): void
dispatch
internal class SpellSchool
+ DriftSpell: GameObject
+ DonutSpell: GameObject
2
Relation
1
SpellCasting
+ AvailableSpellSchools: SpellSchool[2]
+ ActiveSpellSchool: SpellSchool
+ ReceiveAction(Drift drift): void
- CastSpell(bool IsDonut): void
Extends
FireSchool
WaterSchool
AirSchool
EarthSchool
SpellCasting UML

Spell Casting Diagram



Milestone Planning

Milestone	Date (dd / mm / yy)	Deliverable

End of Pre-Production	17/08/2016	1 Prototype proving networked collisions
		1 Prototype with:
		• Movement
		  ○ Steering
		  ○ Drifting
		  ○ Jumping / Air Controls
		  ○ Boosting
		• 1 Simple Spell

M1	09/09/2016	Merge both pre-pro prototypes, Clean-up the project's folder structure, Add 1 Donut Spell Whiteboxed Level Basic Game Mode implemented Basic gameplay HUD implemented
M2	22/09/2016	Arena Design / Game Mode Implemented 75% of all spells functional 50% of Audio Implemented 75% of Particle Effects Implemented One Car fully Modelled & Textured One Hero Asset Modelled Screenflow and non-gameplay UI implemented
Alpha	14/10/2016	All Spells Implemented Level Design complete, save Set Dressing Both Cars Complete Terrain Complete All Hero Assets Complete 70% of other environment art done
Beta	11/11/2016	All Audio Implemented All Art Finalized Arena Completely Set Dressed All UI implemented
Final	25/11/2016	Balance Testing Trailer for the game Splash Screens Credits Presentation Rehearsal

Thank you for listening!

We hope you've enjoyed the ride…

And we certainly hope you enjoy playing!

For now, this is it for CARPG!

Please find us in the slack channel or email Duarte (gd43duarte@vfs.com) or Leon (pg07leon@vfs.com) for any clarification you need