Vehicle Physics Simulation NEA - Marker's Guide

The table below provides the locations of some technical skill demonstrations in the project. This is not an exhaustive list.

Technical skill	**Example locations**
Advanced matrix and vector operations	• **Wheel::updateTyreForce_world** - line 58 of Wheel.cpp • **GameCarModel::update** - line 105 of GameCarModel.cpp
Complex user-defined algorithms	• Found throughout Track.cpp • Found in Terrain.cpp, a specific example is **Terrain::generateNormalData**
Dynamic generation of objects based on complex user-defined use of OOP model	• **VisualShell::load** - line 83 of VisualShell.cpp (occurences of **std::make_unique<>()**)
Complex user-defined use of object oriented programming (OOP)	Examples of inheritance: • **ICarModel** base class on line 20 of ICarModel.h • **VehicleSimulation** subclass of **Framework::Application** on line 20 of VehicleSimulation.h An example of composition is the **Car** owning instances of **WheelSystem** and **ControlSystem** on lines 22 and 23 of Car.h Examples of aggregation: • **WheelInterface** storing reference to an **Axle** on line 23 of WheelInterface.h • **ControlSystem** storing pointers to **Wheel**, **TorqueGenerator** and **Brake** instances on lines 22 to 27 of ControlSystem.h Examples of interfaces (abstract classes in C++ with only pure virtual methods): • **TerrainGenLayer** class on line 21 of TerrainGenLayers.hpp • **ICarModel** class on line 20 of ICarModel.h contains 2 pure virtual functions

Technical skill	Example locations
List operations	• **DebugVectorGroup::addVector** on line 5 of DebugVectorGroup.cpp

Coding style and development considerations

Throughout the implementation of the solution, I have focused on following the SOLID design principles.

- S - Single responsibility
- O - Open for extension, closed for modification
- L - Liskov substitution principle
- I - Interface segregation principle
- D - Dependency inversion principle

Before implementing any functionality, I thought about where it should logically sit a) in the hierarchy structure and b) in relation to what was already implemented in that area, thinking carefully about the object relationships in real cars. From here, I focused on adding the feature with appropriate levels of coupling between those already implemented - reducing coupling often meant adding levels of indirection and abstraction.

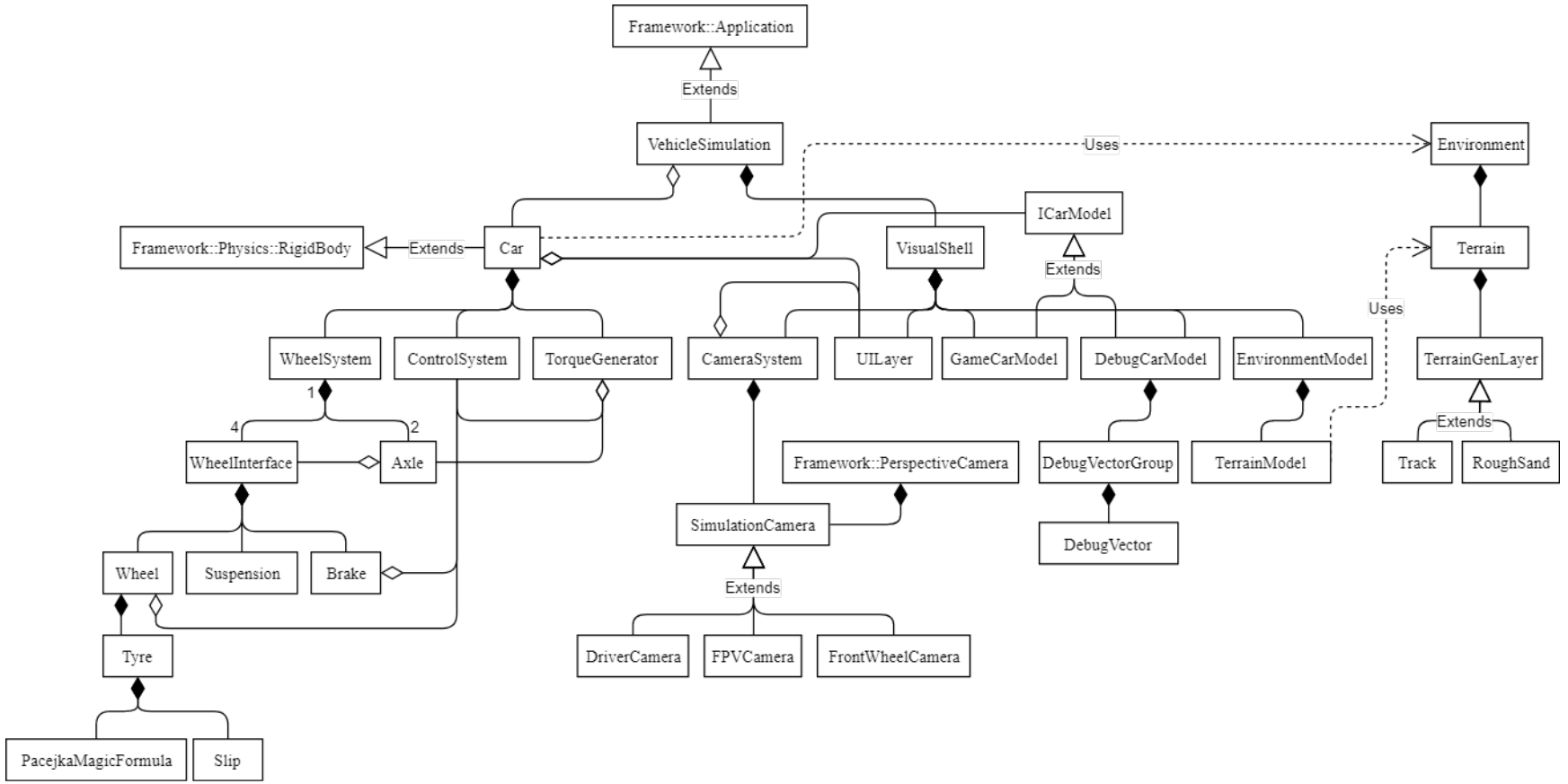
For example, after working on the **Tyre** class, it needed to be linked to the **Car**. Logically, the driver does not *directly* change the deflection angle of the tyres, they rotate a steering wheel, and similarly the tyres themselves are not attached to the steering wheel. As a result, I chose not to have the **Car** class directly encapsulate 4 **Tyre** instances. Instead, the **Wheel** class was added. It could be argued that wheels do more logically 'own' tyres. This design pattern was continued to connect the **Tyre** and **Car** classes in the following way: **Tyre** ⇒ **Wheel** ⇒ **WheelInterface** ⇒ **WheelSystem** ⇒ **Car**.

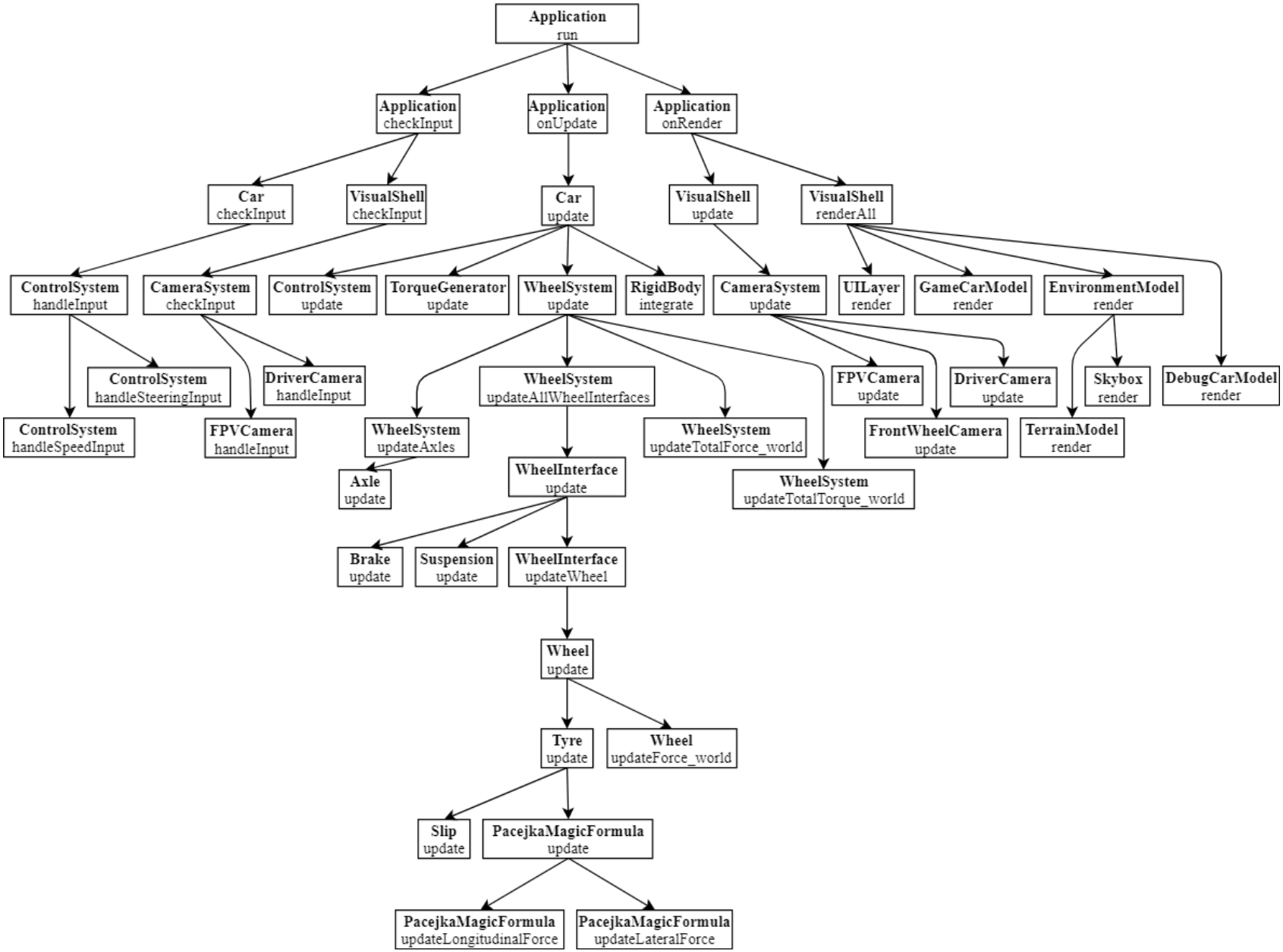
I also considered the future development and maintainability of the project (including beyond A level) when implementing functionality. Some of the much smaller classes, such as **Brake** or **TorqueGenerator**, are small to satisfy the single responsibility principle elsewhere in the project (better to have a separate small class, than have another section of extra responsibility for a big class), but also to **allow for extension**.

Taking the **Brake** class for example, this class was developed to allow for extension, to simulate many more aspects and properties of a brake's behaviour (such as heating, wear, material type, surface area, ABS, mass, moment of inertia etc.) using **the same interface** as the current very simple model. These components can be extended to function as their own subsystems.

Code structure maps

Two structure diagrams have been included on the next pages. They represent all the code in the final version of the program. The first is an entity relationship diagram designed to make navigating between classes easier - it's possible to quickly see where a class is positioned relative to others. The second diagram shows an execution tree of one iteration of the applications main loop (in-order traversal), added to make tracing execution easier.


Framework::Application
Extends
VehicleSimulation
Uses
Environment
ICarModel
Framework::Physics::RigidBody
Extends
Car
VisualShell
Extends
Terrain
Uses
WheelSystem
ControlSystem
TorqueGenerator
CameraSystem
UILayer
GameCarModel
DebugCarModel
EnvironmentModel
TerrainGenLayer
1
4
2
WheelInterface
Axle
Framework::PerspectiveCamera
DebugVectorGroup
TerrainModel
Extends
Track
RoughSand
SimulationCamera
DebugVector
Wheel
Suspension
Brake
Extends
Tyre
DriverCamera
FPVCamera
FrontWheelCamera
PacejkaMagicFormula
Slip


Application
run
Application
checkInput
Application
onUpdate
Application
onRender
Car
checkInput
VisualShell
checkInput
Car
update
VisualShell
update
VisualShell
renderAll
ControlSystem
handleInput
CameraSystem
checkInput
ControlSystem
update
TorqueGenerator
update
WheelSystem
update
RigidBody
integrate
CameraSystem
update
UILayer
render
GameCarModel
render
EnvironmentModel
render
ControlSystem
handleSteeringInput
DriverCamera
handleInput
WheelSystem
updateAllWheelInterfaces
FPVCamera
update
DriverCamera
update
Skybox
render
DebugCarModel
render
ControlSystem
handleSpeedInput
FPVCamera
handleInput
WheelSystem
updateAxles
WheelSystem
updateTotalForce_world
FrontWheelCamera
update
TerrainModel
render
Axle
update
WheelInterface
update
WheelSystem
updateTotalTorque_world
Brake
update
Suspension
update
WheelInterface
updateWheel
Wheel
update
Tyre
update
Wheel
updateForce_world
Slip
update
PacejkaMagicFormula
update
PacejkaMagicFormula
updateLongitudinalForce
PacejkaMagicFormula
updateLateralForce