CS 492 Senior Design
Low-Level Design Report

DriveSim

Group Members

Bora Ecer

Oğuz Kaan Ağaç

Alp Ege Baştürk

Alptuğ Albayrak

Alperen Erkek

Supervisor:

Prof. Uğur Güdükbay

Jury:

Prof. İbrahim Körpeoğlu

Prof. Özgür Ulusoy

Innovation Expert:

Cem Çimenbiçer (TaleWorlds Entertainment)

February 18, 2018

1.0 Introduction

Our project will be a machine learning based simulation game where player will design a vehicle and place sensors that will feed the environmental data to our training system. Depending on vehicle design, placement and type of sensors our training system will teach itself how to drive without crashing into walls, pedestrians, cars and road blocking objects. With the trained vehicle, user will be able to play one of the four game mod. The game mods will allow the user to see how well the designed car works. Depending on the score the player gets from the game mods, the player can evaluate the design of the car, and change the design if he/she chooses to do so.

This report will provide detailed information about the DriveSim system. We will provide our design goals, trade-offs and detailed subsystem decomposition of our project.

1.1 Innovation / Creativity

Even though there are countless innovative ideas that are being used in games, there are not many games that uses machine learning as a gameplay mechanic. Our aim is by introducing this new concept in our game to create a unique gameplay experience for the player in which the players can see how different designs try to overcome the challenges by evolving with respect to them.

1.2 Design Goals & Trade-Offs

By analyzing the design goals and the trade-offs of our system, we define critical aspects of our project. Then by these definitions, we can prioritize and improve some aspects of the project more than others in return this makes the end product more on par with the expectations.

1.2.1 Design Goals

1.2.1.1 Ease of Usability

Ease of usability is one of the most important design goals. A game must be easily understable and playable in order to be be enjoyable because if a player cannot understand the game, s/he cannot meet the main requirements of the game and therefore, get bored. Since the main goal of creating game is to entertain the players this would make all the efforts we put in the game meaningless. The ease of usability can be achieved by designing a clean user-interface, making the controls easy to use and carefully explaining the game mechanics that might need explaining.

1.2.1.2 Optimization

By optimizing our game we are not only making our game reachable to the users with low system specs but also making sure that our game runs smoothly in all systems. This would allow the players to enjoy the game without having performance issues.

1.2.1.3 Privacy and Security

Our game should not have access to anything that can be viewed as intrusion to privacy of the users. It can only make changes inside the folder that it is allowed to do so (i.e. save folder). We also plan to include an online scoring feature where users can see the some of the highlighted scores of other players. We have to make sure that no unnecessary data is stored online, and the data that is stored is safe from unwanted accesses unauthorized people.

1.2.1.4 Readability of the Software

The importance of readability is apparent in projects that are being worked with groups. The readability ensures that the different team members can easily understand the code and its purpose without wasting time. It is also important that if in future a new member comes, s/he can get up to speed with the project much more easily and quickly.

1.2.1.5 Understanding How ML Works

The purpose of our game is not just for it to be fun but also give insight about machine learning to its players. We believe that the best way to understand how something works is to observe it while it is working, so we plan to show the interactions between the neurons of the neural network in real time to the users. This way the users can see why their design acts that way.

1.2.1.6 Efficiency

The main design goal of the system is efficiency, to achieve that, the system must be able to work in high performance. Since, a smooth gameplay is one of the most important feature which increases the player’s urge to play the game, we are going to minimize the memory, CPU and GPU usage. In order to achieve that, first, we are going to implement the code in the most efficient way possible. Also, we are going to design the system so that the workload of the objects is going to be nearly balanced.

1.2.1.7 Reliability

Our system will be reliable in terms of being consistent with the boundary conditions. The system should not respond with any unexpected results -like bugs, crashes- which are not specified in the boundary conditions. In order to provide that, we are going to test the system in all possible ways during and after the development stage. Also, the boundary conditions will be selected carefully and with caution so that there won’t be a case with which puts the system in an unexpected situation. This will provide the system to foresee the possible fatal failures which will be dealt with.

1.2.2 Design Trade-Offs

1.2.2.1 Optimization vs Portability

We can choose to deploy our product on multiple different operating systems but this would cost us the time we can optimize our game to run smoothly. We decided that we are going to deploy only on Windows systems. This will allow us to focus on one system only and in return a more optimized end product.

1.2.2.2 Cost vs Reusability

Currently, we do not have any plans to improve our project so that we can build it on another system. Our plan is to improve our project with the existing system.

Therefore, in the future stages of development, we can aim to create a maintainable, low-cost product and sacrifice reusability while doing so.

1.3 Engineering Standards

In this report and previous reports, IEEE standards are used due to its wide acceptance in engineering domain. Moreover, for the technical representation of the system, UML diagrams were used. In addition, logic of ECS was also shown in UML This is for the clearance and wide range of understandability.

1.3.1 Contemporary Issues

The project is not expected to cause professional or ethical issues. All of the assets that will be used in the game will checked for copyright issues. Therefore, it will not contain anything that will violate copyright issues. This will be free to play game and the source code of the game will be provided publicly for educational purposes. Additionally, if we have time to implement online - multiplayer features, to make the game fair for all players the player’s vehicle data will not be publicly visible, instead we will provide a “ghost” vehicle that follows the path of the original one. So that the other players cannot copy the original vehicle design to get the highest score.

1.3.2 Lifelong Learning

DriveSim is meant to be a informative, entertaining and challenging game with a unique gameplay mechanic to simulate driverless cars. The core feature of the game is making a neural network to learn how to drive while not hitting static and dynamic objects such as walls, cars, pedestrians and meet the goal of different game modes. To do that, the players will design a vehicle model, which includes different body parts with different shapes and various types of sensors. These sensors will be used to detect objects and make a decision which will make the vehicle adjust its behavior.

Our intention to make the game modes such that they will have different objectives to achieve. By doing so, we allow the users to experiment with different vehicle designs for different gameplay modes and observe that a vehicle design that may be successful for a specific game mode may not be successful in other game modes. Therefore, the player should consider all the aspects of the game mode while designing the car.

1.4 Interface Documentation Guideline

Interface Documentation will be explained in below schema. Schema is self explanatory and example is given below.

Class Name (extends Class … / includes Class)
… (Explanation of the usage of the class)
Properties
…
Methods
…

1.5 Definitions, Acronyms and Abbreviations

Unreal Engine: Unreal Engine 4 is a complete suite of development tools made for anyone working with real-time technology. From enterprise applications and cinematic experiences to high-quality games across PC, console, mobile, VR and AR [1].

Entity Component System (ECS): ECS is an architectural pattern that is mostly used in video game development. ECS allows a greater flexibility in defining entities where each object in a game's scene is an entity (e.g. enemies, npcs, bullets, vehicles, etc.). Each entity is consists of components which adds behaviour or functionality. This makes it easier to change the behaviour of an entity at runtime by just adding or removing components. This is especially very useful in deep and wide inheritance hierarchies because it eliminates the ambiguity problems in just systems. [2]

Source Control Management (SCM): Source control (or version control) is a system that records to changes to files so that the user can recall to the specific versions of that file(s) later [3]. This management system also makes it very easy for developers to work on same project simultaneously since they can easily upload and merge their work with these tools (e.g. Git, Mercurial).

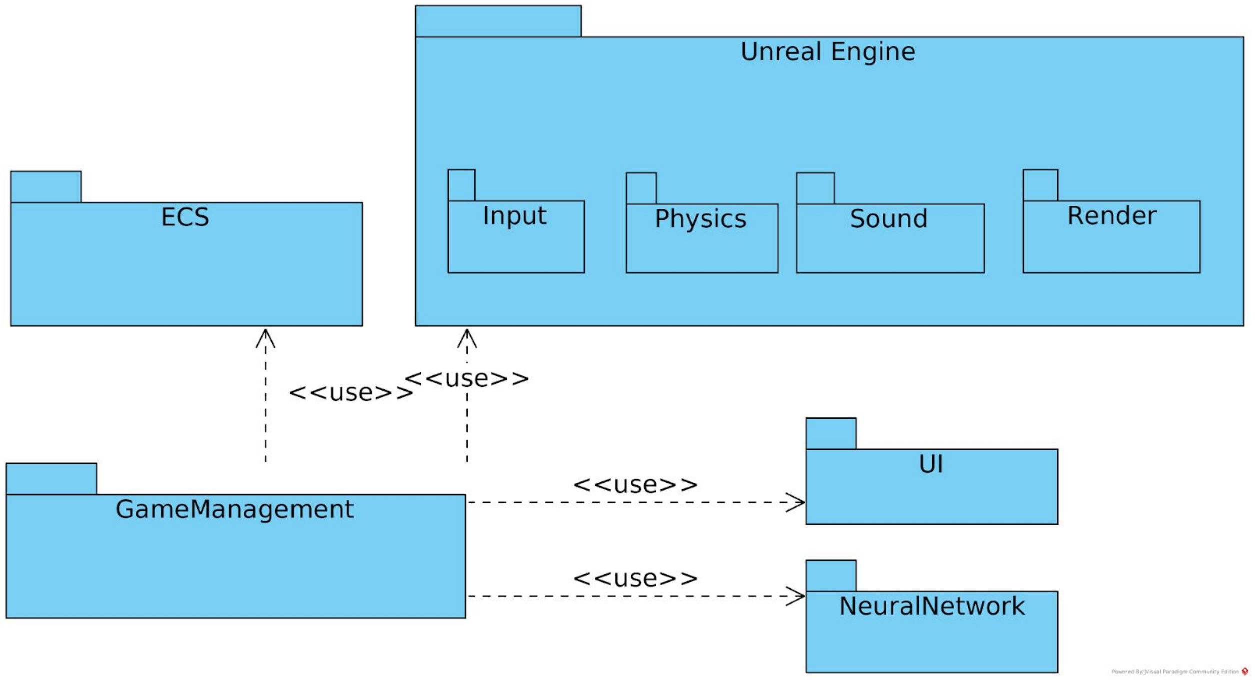
2.0 Packages

2.1 Introduction

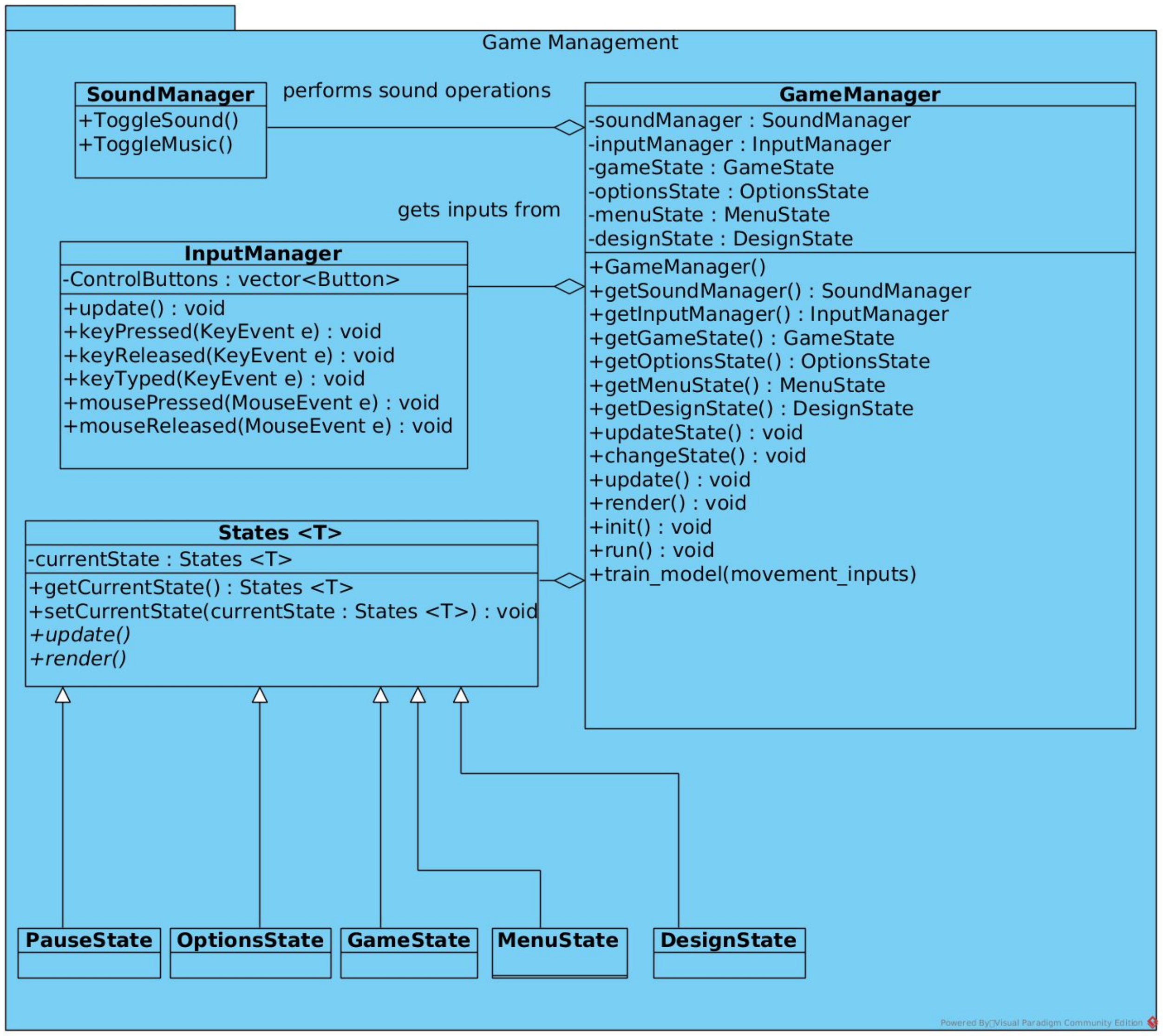
Our high level design includes 5 different subsystems, such as the engine, the game manager, neural network functions, UI, entity component system. For our game to function nicely all these 5 subsystems should work without causing any significant drawback and they should communicate with each other efficiently. Our main approach to structure the project is using the data oriented design patterns. We believe that creating too many abstraction layers where one sits on top of the other distances us from the actual hardware which in return causes in performance penalties, such as frequent cache misses caused by virtual function calls. To avoid such issues, we will try to mostly use flattened data structures such as aligned vectors or arrays and create functions that operate on batches of data rather than a single unit. In fact the ECS or the entity component system design pattern just does this. Separating components from the entities and bundling them together so that we can process them in batches.

Secondly we will focus on separating the UI, input and the game logic. The game manager will deal with the game logic and the game loop while communicating with the UI and the input layer through an abstraction layer we will write to make our game logic engine and platform agnostic. This will allow us to break the dependency between these three layers so that we can implement, improve and update these three layers to our liking whenever we want. We will not strictly use MVC (model view controller) approach since it sometimes be too limiting especially if we decide to include IMGUI (immediate mode gui) to our game.

Our current high level subsystem decomposition is given below:

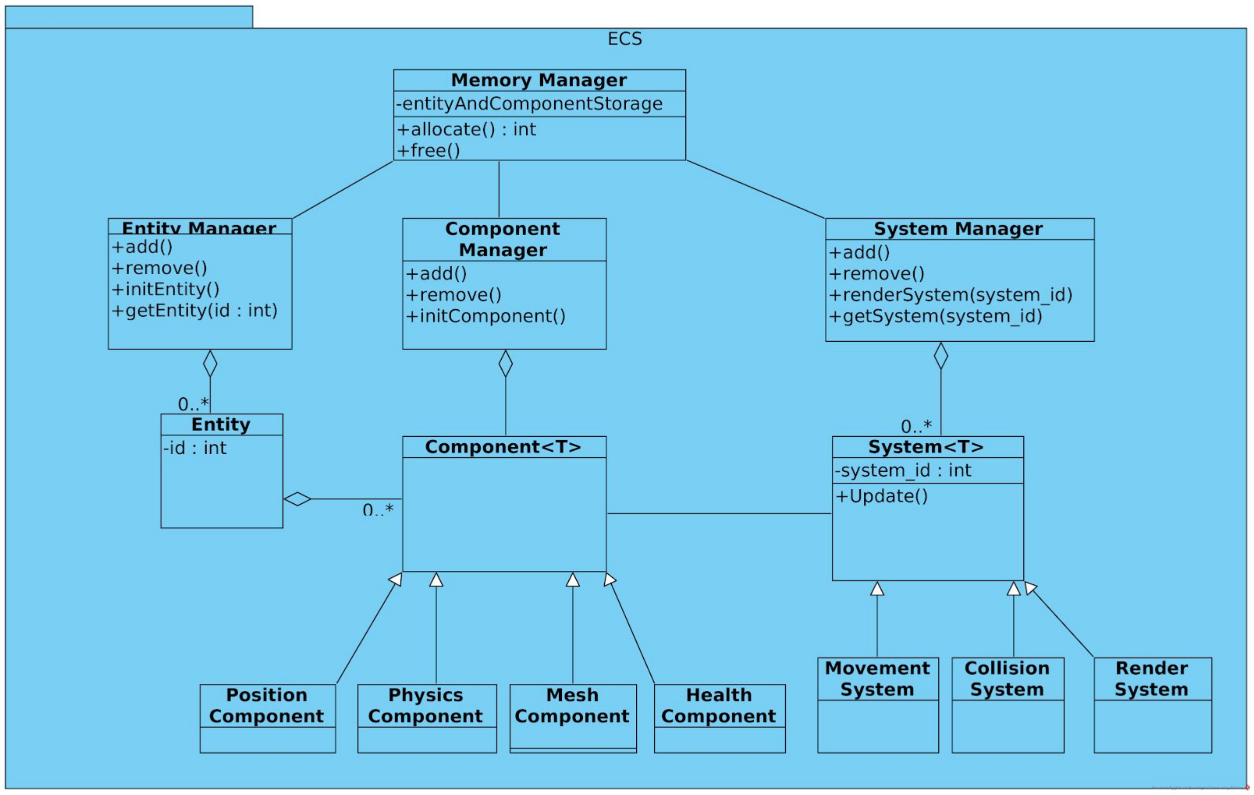

Unreal Engine
Input
Physics
Sound
Render
ECS
<<use>>
<<use>>
GameManagement
<<use>>
UI
<<use>>
NeuralNetwork
Powered By Visual Paradigm Community Edition

2.2 Game Logic

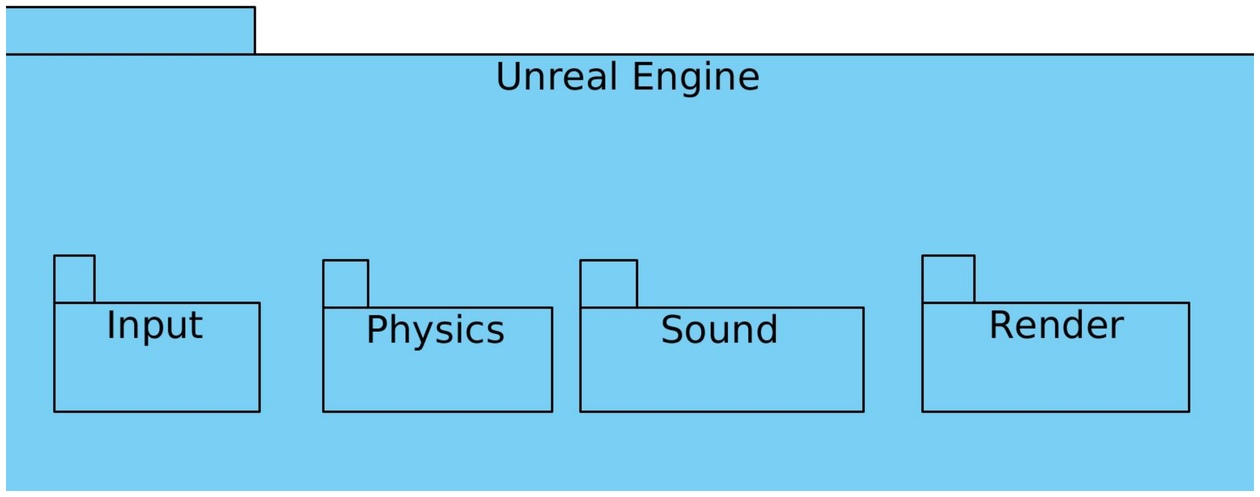


Game Management
SoundManager
+ToggleSound()
+ToggleMusic()
performs sound operations
GameManager
-soundManager : SoundManager
-inputManager : InputManager
-gameState : GameState
-optionsState : OptionsState
-menuState : MenuState
-designState : DesignState
+GameManager()
+getSoundManager() : SoundManager
+getInputManager() : InputManager
+getGameState() : GameState
+getOptionsState() : OptionsState
+getMenuState() : MenuState
+getDesignState() : DesignState
+updateState() : void
+changeState() : void
+update() : void
+render() : void
+init() : void
+run() : void
+train_model(movement_inputs)
gets inputs from
InputManager
-ControlButtons : vector<Button>
+update() : void
+keyPressed(KeyEvent e) : void
+keyReleased(KeyEvent e) : void
+keyTyped(KeyEvent e) : void
+mousePressed(MouseEvent e) : void
+mouseReleased(MouseEvent e) : void
States <T>
-currentState : States <T>
+getCurrentState() : States <T>
+setCurrentState(currentState : States <T>) : void
+update()
+render()
PauseState
OptionsState
GameState
MenuState
DesignState
Powered By Visual Paradigm Community Edition

Entity Component System (ECS) will deal with, as the name suggest, entities and the components related to the game entite. It will include a job system and a component system. Component system's main function will be attaching or detaching components from or to entities. The job system will include what is called "job functions", type of "batch" functions that we have mentioned above. These job functions will work on a couple of components and update them in batches. This system will increase the overall performance of our system and will create a nice decoupling between the entities and their components.

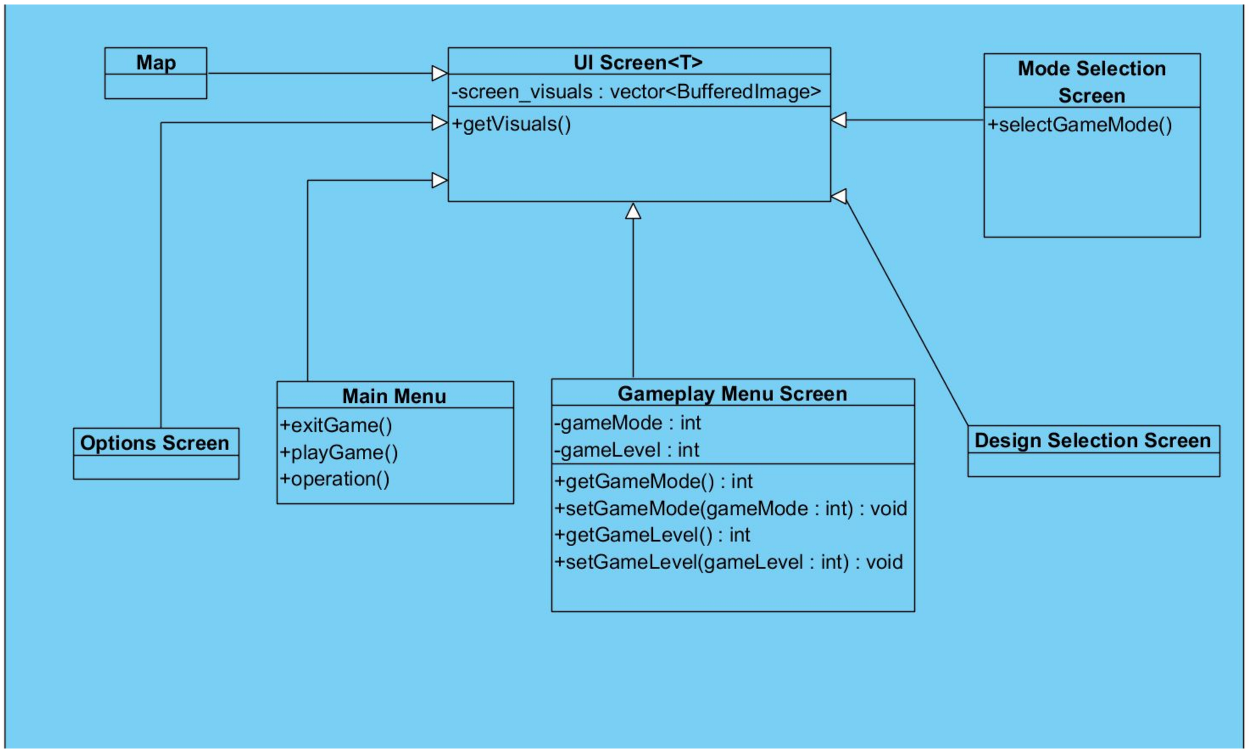


We will use UE4 as our game engine. It will deal with the rendering, physics, mathematics, sound and input. We will create some coroutines around these systems so that we can easily make the engine talk with our game logic.



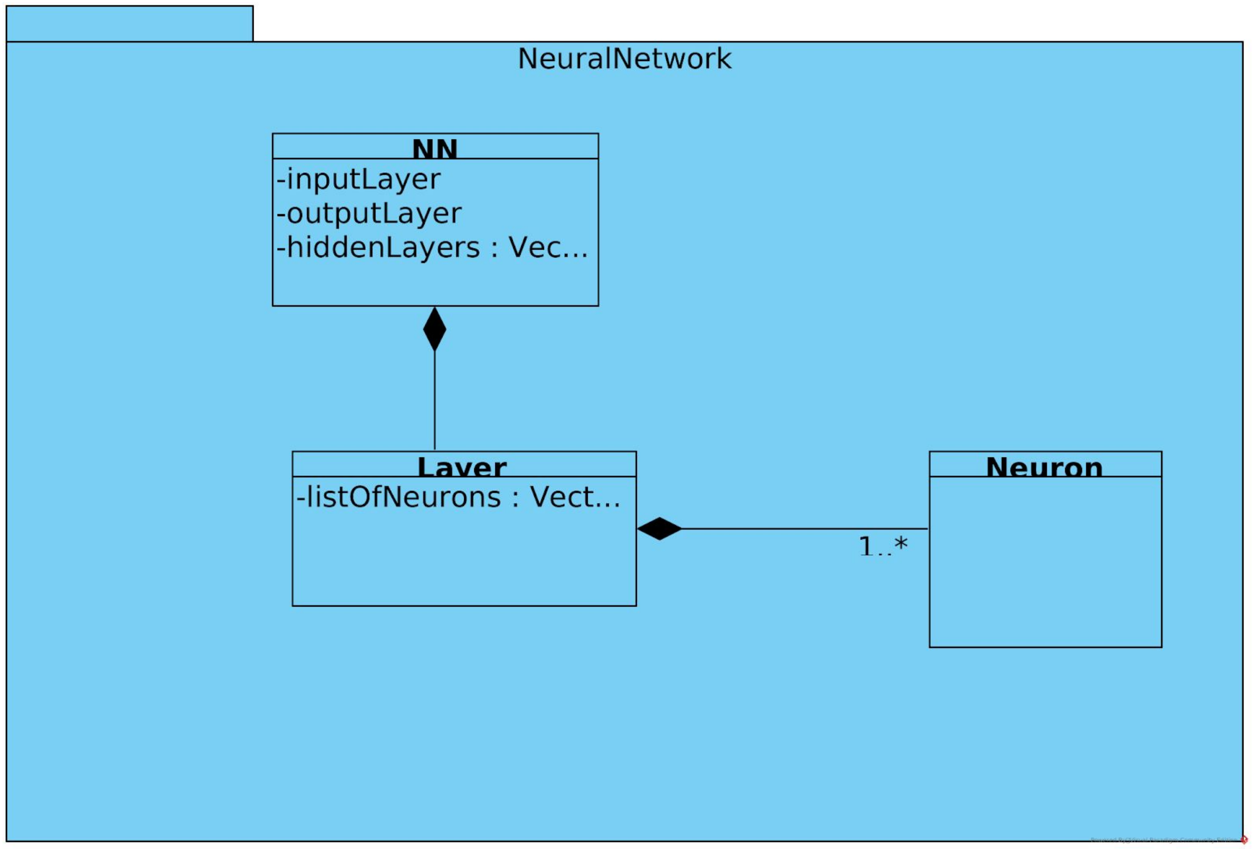
2.3 User Interface

As mentioned above, we will try to separate the UI from the game logic as much as we can. UI subsystem will deal with the user input, create appropriate data structures and pass appropriate messages to the other systems that need to act on.



2.4 Neural Network

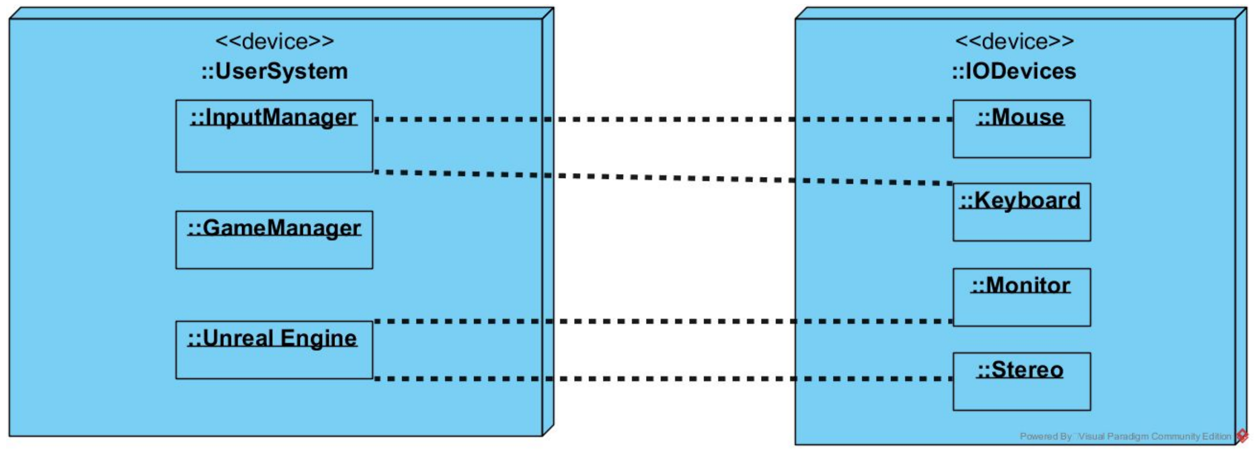
Neural network subsystem will encapsulate all functioning related with the neural network such as training agents, cars, updating node values, node places etc.



2.5 Overview

Unreal engine provides us with powerful and gigantic IO libraries. We will create a layer or a wrapper around the part of the library that we actually need so that we can isolate the parts we need and specifically work on those. This will create simplicity and will help us adjust the input system as we like.

Since our software is a game, we will need hardware accelerated graphics i.e a dedicated graphic processor unit (GPU). Fortunately unreal engine can easily handle the rendering side of our game. It allows us to write configurable shaders, and do gpu calculations. Our game can, potentially, be cross-platform since UE4 will handle all platform specific details, however development main focus will be on Microsoft Windows.



We will have our own Entity-Component System(ECS). This will allow us to control and update entities in a efficient, fast and reliable way. ECS will also provide us nice decoupling between components such that dependency between components will be minimum.

DriveSim's game manager will be "controller" of the game state, settings and inputs. This will be main controller that will control the current state of the game. Lastly, It's important to note that our Neural Network will be independent of the game.

These different system will allow DriveSim to work on different levels such that they won't depend on each other. This will provide fast and elegant system that is needed for heavy tasks. During these tasks, because of the multi-level system, they will work in synchronous way.

Finally, we will separate our UI from our game logic. It's important to have such system that can work without UI and vice verse because during the implementation or later phases one of them may require functionality change and dependency between them would make it cost more.

3.0 Class Interfaces

3.1 Game Logic

3.1.1 Game Management

Class Name SoundManager
Controls the sound assets and their status
Properties
-Player player
Methods

-void PlaySound(int sound,float volume) //Plays the sound fx, if volume given it plays at that volume, otherwise plays the sound at the sound level selected in Options.
-void StopSound(int sound) //Stops the sound fx if it's playing otherwise does nothing
-void ToggleMusic(int music) //Stops/Plays the music at the sound level selected in Options

Class Name InputManager
Controls and handles the inputs taken from user.
Properties
-vector<Button> ControlButtons
Methods

-void update() // Updates the current status and checks if any input given.
-boolean isKeyPressed(Button key) // Checks if key pressed or not.
-boolean isKeyReleased(Button key) // Checks if key released or not.

-boolean isMousePressed(Button key) // Checks if given mouse key pressed or not -boolean isMouseReleased(Button key) // Checks if given mouse key released or not.

Class Name States <T>
General template for the states that will be used in the game. Controls overall states. States can be: *PauseState: This state occurs when user pause the game while in-game. *OptionState: This state occurs when users clicks on "Options" whether from main-menu or from pause state *GameState: This state occurs when users sees his/her simulations *MenuState: This state occurs on main-menu. *DesignState: This state occurs on car editor where user designs his/hers car.
Properties
-State state
Methods

-State<T> getCurrentState() // returns game's current state.
-void setCurrentState(State<T> state) // sets currentState to changed state
-void update() // Updates state based on game logic and ECS. This triggers ECS to update its entities.
-void render() // Renders State elements this includes UI elements as well.

Class Name GameManager
Current game's manager. This is main class that controls current state and its components. It uses ECS, UI, NeuralNetwork and Unreal Engine.
Properties
-SoundManager soundManager -InputManager inputManager -States<T> states
Methods
-SoundManager getSoundManager() // returns soundManager -InputManager getInputManager() // returns inputManager

-States<T> getStates() // Returns states
-void updateState() // This updates sounds, medias as well as triggers currentState's update method.
-void changeState(State<T> state) // This method stops game elements, if needed saves necessary information and changes current state to given state.
-void render() // This method makes renderable objects ready and calls currentState's render method.
-void Init() // This method is called on Initialisation of Game Manager. Mostly during first startup of the game
-void train_model(movement_inputs)

3.1.2 ECS

Class Name MemoryManager
Handles storage that is needed for entities and components
Properties
-uint_64_t total_allocated_size
Methods
-void* allocate(int_64 size) // Allocates memory for given size and return allocated location's pointer. -void free(void* ptr) // Frees allocated memory if available else does nothing.

Class Name EntityManager
Handles all the entities and has close relationship with MemoryManager. Entity manager holds the entities that is used in the ECS system. A Entitiy object holds (int) id which is used for getter function.
Properties
-MemoryManager memoryManager -Entity[] entities
Methods

-void add(Entity entity) // Adds given entity into the manager.
-void remove(Entity entity) // Removes given entity from the manager.
-void InitEntity() // Initiation for the Entity Manager, usually called during first execution of the game.
-Entity getEntityById(int id) // Gets entity by its id.

Class Name ComponentManager
Handles components that are wrappers for the ECS and Unreal Engine components. It holds components that is used in the ECS system. A component holds entity information belongs to that specific component. A component can be: -Position Component: Holds the information about entities' position,velocity etc. -Physics Component: Holds information about intersect,collusion etc. -Mesh Component: Holds information about entities' meshes and their textures -Health Component: Holds information about entities' overall health and its system.
Properties
-MemoryManager memoryManager -Component<T>[] components
Methods

-void add(Component component) // adds given component to the manager.
-void remove(Component component) // removes given component from the manager
-void InitComponent() // Initiation for the Component Manager, usually called during first execution of the game.

Class Name SystemManager
Performs global actions on every Entity that possesses a Component of the same aspect as that System. SystemManager holds zero or more Systems. A system has a int that’s related to its id and used for getter.A system can be: -Movement System: Performs actions on movement of entities and its position components -Collision System: Performs actions on collusion of entities and its physics components -Render System: Performs actions on visualisation of entities and its mesh components
Properties
-MemoryManager memoryManager -System<T>[] systems
Methods

-void add(System<T> system) // adds given system into the manager.
-void remove(System<T> system // removes given system from the manager.
-void renderSystem(int system_id) // performs global actions on given system and calls Update function on given system
-System<T> getSystem(int system_id) // Gets the given system from manager by its given id.

3.2 User Interface

Class Name UIScreen<T>

Draws visuals loaded into its buffer. These visuals can be:
-Map: Holds visuals belonging to each map
-Options Screen: Holds visuals related to options.
-Main Menu: Holds visuals related to main menu. In this UIScreen, there are 3 methods:
*void exitGame() // Triggers default exit pathway
*void playGame() // Loads level and initialise belonging game logic. Loads visuals as well
*void operation() // Loads Neural-Network simulation
-Gameplay Menu Screen: Holds visuals related to game level selection. It has 2 integer variables called gameMode and gameLevel which holds information about current game state. It has getter and setter methods for these variables
-Design Selection: Holds visuals related to vehicle design screen

Properties
vector<BufferedImage> screen_visuals // holds current screen's visuals
Methods
-vector<BufferedImage> getVisuals() // returns current screen's visuals

3.3 Neural Network

Class Name NN
The neural network consists of multiple layers and uses genetic algorithm for training.
Properties
Layer inputLayer // first layer of the neural network Layer outputLayer // last layer of the neural network vector<Layer> hiddenLayers // the layers between first and last layers
Methods

Class Name Layer
A single layer consists of multiple neurons.
Properties
vector<Neuron> listOfNeurons // neurons in this layer
Methods

Class Name Neuron
A single neuron only holds a single weight.
Properties
float weight
Methods

4.0 Conclusion

Throughout this report, technical informations about the DriveSim is explained. High-level, low-level designs, our current and planned system is clarified and our subsystem and their working details are related. These specifications gives detailed answer how and why DriveSim will be different compared to other similar games. DriveSim is aimed to be implemented in correlation with these factors.

DriveSim’s unique and simple UI will allow users to understand the basic concepts of the game easily and clear the confusion that new players face. It will be simple but elegant enough give unique view.

DriveSim will provide unique way to challenge player’s creativity and teach them how neural network thinks based on their choices. This created opportunity to users to challenge each other and create a competition. We believe that this competition will improve driverless car technology. Experiences learned through the game’s life can be transferred real-life situation, if needed.

5.0 References

[1] "Dream Car Builder". *Steam*. 2018. [Online]. Available: https://store.steampowered.com/app/488550/Dream_Car_Builder/. [Accessed: 11 Nov. 2018].

[2] Klande, David. "Cars Incorporated". *Wildduckgames*. 2018. [Online]. Available: http://www.wildduckgames.com/index.php/carsinc. [Accessed: 11 Nov. 2018].

[3] T. Coventry, "Requirements management – planning for success!", *Pmi.org*, 2018. [Online]. Available: https://www.pmi.org/learning/library/requirements-management-planning-for-success-9669. [Accessed: 11 Nov. 2018].