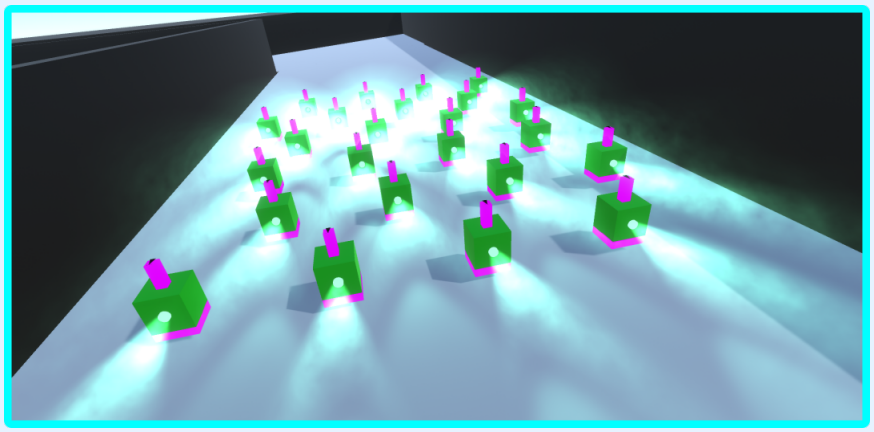

Free-Range, Autonomous Driving Agents for Real-Time Game AI

Team

The Radiant Flux

Estefano Palacios
Israel Pasaca
Vielka Villavicencio

January 19, 2021

"In academia, some AI researchers are motivated by philosophical questions: understanding the nature of thought and the nature of intelligence and building software to model how thinking might work. Others are motivated by psychology: understanding the mechanisms of the human brain and mental processes. And yet others are motivated by engineering: building algorithms to perform human-like tasks. This threefold distinction is at the heart of academic AI, and the different concerns are responsible for the different subfields of the subject.

As game developers, we are practical folks; interested only in the engineering side. We build algorithms that make game characters appear human or animal-like. Developers have always drawn from academic research, where that research helps them get the job done, ***and ignore the rest.****"*

—Ian Millington, *IA for Games*

"A good toy is an object that is fun to play with. Play is manipulation that indulges curiosity. ***Fun is pleasure with surprises.****"*

—Jesse Schell, *The Art of Game Design*

"That extended moment when tic-tac-toe ceased to interest [children] was a moment of great fascination to me. Why, I asked myself, did mastery and understanding come so suddenly? The kids weren't able to tell me that tic-tac-toe is a limited game with optimal strategy. They saw the pattern, but they did not ***understand*** *it as we think of things.*

The isn't unfamiliar to most people. I do many things without fully understanding them, even things I feel I have mastered. I don't need a degree in automotive engineering to drive my car. I don't even need to understand torque, wheels and how the brakes work. I don't need to remember the ins and outs of the rules of grammar to speak grammatically in everyday conversation. I don't need to know whether tic-tac-toe is NP-hard or NP-complete to know it's a dumb game.

The brain is good at cutting out the irrelevant. The brain notices a lot more than we think it does. The brain is actively hiding the real world from us.*"*

—Raph Koster, *A Theory of Fun*

Contents

Contents

1 The Problem

1.1 The Magic Circle

A good game is a digital talisman capable of transporting players to enlightening experiences.

The very *nature* of a video-game is to provide a magic circle, a rich and enchanting problem-space proposed as a playful activity for us players to experience and enjoy.[3] As the game's problems are presented to us, our brains are able to extract *patterns* out of these different problems, and thus our brains train themselves to become amazingly adept at handling the observed patterns with expertise.[4]. As our brains grow and adapt to these challenges, they require increasing levels of difficulty to stay on the level of flow.[6]. Without further surprises from the engagement of gameplay, the game stales into, at best, a pleasurable activity and, at worst, a boring one. After all, in the words of Carnegie-Mellon game designer Jesse Schell, *"Fun in pleasure with surprises."*[3]

Figure 1: Games provide with a magic-circle where disbelief is suspended, but this suspension of disbelief is brittle, and the illusion of otherworldly play will vanish at the slightest provocation.[7]

What, exactly, does this imply about game artificial intelligence? On the outset, if we review the literature, game AI techniques have been based on emulating—through various engineering mechanisms—the behavior of intelligent organisms by *hard-coding* their behavioral patterns. Thus, this emulation

has been, so far, deterministic,[1] using techniques such as behavior trees, decision trees, fuzzy logic, finite state machines, and path-finding algorithms such as $A*$ to provide the illusion of intelligence. In simpler terms, the behavior of the agent has been deterministically pre-*designed*. *Yes*, a game's artificial intelligence can have random behavior built into it and, yes, this can further create the illusion of intelligence by presenting a veil of uncertainty to the player (because, after all, isn't uncertainty in part the essence of all living organisms?[8]). But even in such circumstances, the player's brain will learn to see the patterns,[4] it will learn to master the probability distributions underlying the behavior of artificial agents and, in so doing, the illusion of facing an intelligent opponent vanishes, leaving in its wake an experience of pattern-based problem-solving, a rather much more mundane experience than that of facing an agent that behaves intelligently, that adapts, that improvises, that explores new ways of doing things and isn't afraid to fail. In other words, that *learns*.[2]

The inclusion of randomness into an agent does not preclude the deterministic nature of the *experience of game play*. Put it in another way, the agent's behavior might have been programmed to behave randomly, but the overall experience it unveils to the player is still deterministic in nature, because the *space of possibilities* have been hand-crafted, hard-coded into the very fabric of the agent.

Can we, as game designers and game developers, weave together a more intelligent space of possibilities?

1.2 Learning and Modeling Game AI

Quite probably the first thing that comes into our heads when we think about the nature of learning is the learning that takes place when we interact with our environments. Human beings, in their first two years of living, develop a sensor-motor connection with their surroundings which marks how they relate to their environment through their physical perceptions. This produces an onslaught of information about the nature of cause and effect, the consequences of our actions, and what to do to reach our goals.[8][2]

How does this relate to artificial intelligence in video-games? All across the board, the field of game AI is marred with the technological challenge of designing intelligent behavior which is *learned* early on by humans, but only mimicked by artificial agents in games. For example, in sports games and driving games, we would like to dynamically calculate the shortest way through a racetrack; whilst in role-playing games (RPGs) we seek for players to engage in sophisticated conversations with artificial agents.

In the vast majority of games, artificial intelligence must approach three basic requirements:[1]

1. The ability to *move* characters.
2. The ability to make decisions about *what to do* (e.g., *where* to move those characters).

3. The ability to *think tactically or strategically.*

There are different approaches, but most of the practice of AI today seeks to fulfill these three basic requirements for creating compelling artificial agents.

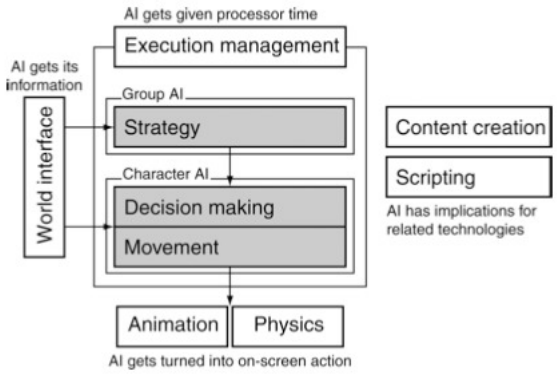

Figure 2: Basic model of a game AI architecture.

In theory, we would like for our agents to select the best actions possible over time. However, to be able to *anticipate* the results of a good action turns out to be a complex problem, for this depends directly on the way the player behaves or on the structure of random distributions that cannot be designed for.

Our agents ought to have the freedom to pick any action in any possible circumstance, and determine which actions are best in any given specific situation. Something to take into account is that the *quality* of the action is usually unclear at the moment of its realization. Hence, it follows that giving feedback *only when something significant has happened* is useful, and that the agent must learn all the actions prior to that event, even though it might not receive any feedback while these actions are being executed.

The basic idea of the problem of reinforcement learning is to capture the more important aspects that a learning agent faces when it interacts over time with its environment in its quest to attain a goal, all the while being able to take actions that affect the game's state.

1.3 Hovercraft, Flying Fairies, Swimming Monsters, and Other Free-Range Driving Gameplay Mechanics

Driving is a quintessential game mechanic.[5]. Specifically, driving in the context of free-range agents—that is, agents that can change their direction of movement *at any moment* and *at will*—are an interesting aspect of game design. We can envision a myriad of game elements, or ludemes[4], that feature this kind of movement mechanic. Flying, supernatural bestiary are an example. Hovercraft

in the sci-fi genre is another[1]. Even swimming fish can be conceptualized as free-range driving agents.

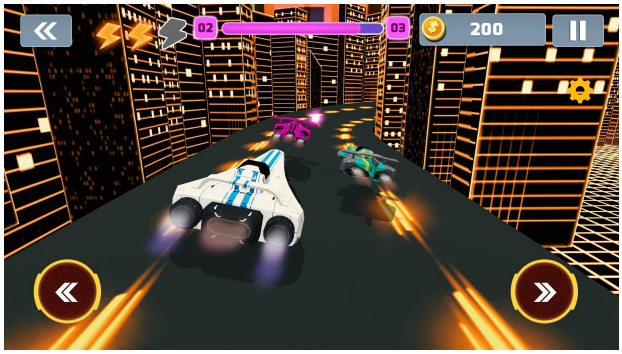

Figure 3: Hovercraft are en example of free-range driving agents.

Traditional cars are *not* an example of free-range driving agents, because their physical properties have to take into account the vagaries of the steering wheel and torque, and they are limited in that they may only *accelerate* forward, and may only move backwards by first breaking into standstill and using a *reverse gear* to shift their direction of movement.

As with other areas of game AI, free-range driving agents haven been usually programmed using traditional (i.e., what we've been so far calling "deterministic") game AI. We seek to correct this trend.

The problem we seek to solve is hence twofold:

1. *Given that creating these agents by hard-coding their behaviors is incredibly exhaustive, time-consuming, and imperfect (at best),* ***is there a way to automatize the processes of learning a function that, given a state*** s_t ***of the agent, can procure a best action*** a_t***, which follows a pre-defined reward path for the agents?*** *Can this function be learned through response to stimuli in the agent's environment?* [a]

2. *Given that the suspension of disbelief is so brittle, so feeble,* ***can we create free-range driving agents which can still surprise players, even after long contact with them?***[b]

[a] As opposed to, of course, what has been engraved in their core implementation,

[b] Can we defy the industry-established notion that modern machine-learning techniques have no place in game AI building? Can this defiance lead to games that continue engage players, by providing opponents that are smarter than what can be achieved with traditional game AI techniques?

[1] I can't help but point out to the *Sentinels* in the movie *The Matrix* as an example.

2 The Goal

Our interest is thus in:

1. ***The development of autonomous, free-range driving agents for real-time game applications.*** *Specifically, we seek to design agents that adapt intelligently to* ***avoid any obstacles in their path*** *when they are presented with a target destination, all the while pursuing such a goal* ***greedily*** *to defeat the player and other agents. Finally, we want our agents to learn how to* ***navigate through mazes*** *of arbitrary complexity.*

2. *To create game AI that surprises players long after their exposure to the game, and, from that* ***to create a game with capitalizes on the strengths of the system for an interesting experience.***

Our problem statement allows the underlying game that uses our technology to take many forms. For example, we could place hundreds of agents which share the same *linear driving goals*, inevitably leading to the concept of a *race*. Agents which share *scattering goals* can be viewed as *stampedes*.

In the end, we seek to create an intelligent mechanism for the *short-term* decisions made by the agents on their way to their goal. This expands the applicability of our technology by enabling its use in a layered fashion with other, higher-order technologies. For example, long-term movement goals can be planned by algorithms such as $A*$, but the short-term, frame-by-frame adaptations in the way to those long-term goals will be made by our technology, where agents seek to satisfy those short-term goals *greedily*, that is, seeking to arrive there first or, failing that, as fast as possible.

3 Preliminaries of a Solution

As suggested by our introduction to the problem, it was decided that the best course of action would be to use mechanisms that would allow for our agents to learn from exposure to stimuli. As such, we seek to leverage *reinforcement learning* for the development of our project.[2]

> *"Reinforcement learning is a variation of machine learning where the agent learns what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is never told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial and error search and delayed reward—are the*

two most important distinguishing features of reinforcement learning."

—Sutton et al, *Reinforcement Learning: An Introduction*

The novel aspect of our approach is that machine-learning and reinforcement-learning have been used successfully to teach machines how to play games, but they *haven't* been deployed so successfully in the creation of interesting *opponents* for the player.[1][12][13]

3.1 Q-Learning

To achieve our goal, we will use a customized, online version of *Q-Learning*, based off the suggestions provided by [1] and [2], and based off our own innovations attained in using the algorithm in this particular context. These will be explained in the Sub-Section 3.3.

3.1.1 Why Q-Learning?

Q-Learning is an *off-policy* **temporal difference (TD)** control algorithm for reinforcement learning.

"If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap). The relationship between TD, DP, and Monte Carlo methods is a recurring theme in the theory of reinforcement learning[.]"

—Sutton et al, *Reinforcement Learning: An Introduction*

So we see that Q-Learning draws its strength from two families of algorithms.

To understand why Q-Learning is called *off-policy* we have to take a detour to understand the jargon of reinforcement learning.

A *policy* is a mapping from states to probabilities of selecting each possible action within that state. If the agent is following policy π at time t, then $\pi(a_t|s_t)$ is the probability that action a_t will be taken given that we're in state s_t. In some algorithms for reinforcement learning, we use policy-convergence mechanisms in which we *evaluate* a policy based on how good the series of rewards it gives us, then *improve* over that policy by changing the probability distributions of policy π in order to procure better rewards, and we continue with these two steps, leading to *policy iteration*.[2]

Q-Learning is called *off-policy* because it forgoes the idea of a policy to select an adequate action a, and simply improves what is called an action-value

function Q, and it is through this function's values that we choose the optimal action a to take next. Of course, Q-Learning is still reinforcement learning, so the function Q evolves through each iteration of the algorithm, and it can be proven that this leads to the same behavior as an optimal probabilistic policy $\pi*$.[2] Which means that even if we adopt more complex models, Q-Learning still provably gets to those states with enough time.

Why would we want *off-policy* methods? Because the underlying implementation of *on-policy* methods is technically more complex to materialize from the ground up. And yet their effectiveness differs by a constant, *regardless* of the problem. Yes, Q-Learning is not more effective than SARSA, but, after enough learning, their degree of effectiveness lies always within a constant from each other. Which means Q-Learning gets pretty far in learning the optimal strategy, at a fraction of the cost of getting SARSA to work. In our case, the very definition of the problem *prohibited* us from using SARSA, for our version for reinforcement learning would have to be sample-based. SARSA requires a complete make up of the underlying model to get it to run, including the ability to look into future states, which we can't, for our real-time constraint.[2].

> *"One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning."*
>
> —Sutton et al, *Reinforcement Learning: An Introduction*

Q-Learning is a simple yet powerful algorithm which has low conceptual requirements and can be used in a variety of situations.[1] It's off-policy *and* temporal-difference method, which makes it into a bleeding edge algorithm. That leaves out TD(0) and SARSA which are *on-policy* temporal-difference methods of the same category. Monte Carlo and Dynamic Programming methods are also *on-policy*, and for this reason they weren't considered for the problem at hand. Additionally Monte Carlo is episodic while our problem is not episodic in nature, but intra-episodic. On the other hand, Dynamic Programming techniques require a perfect model of the enviornment as a Markov decision process (MDP), something unavailable to us give the nature of our problem.[2]

3.2 The Algorithm

We've already come with a first draft of our customized algorithm, and presenting it here will help us understand the underlying functionality behind our problem analysis, meta-design, and design, all of which will be presented in the next sections.

Algorithm 1: Online, Sample-Based Q-Learning

Data: $Q(s, a)$ is a dictionary that saves the Q-values per-state, per-action; ρ is the exploration parameter; α is the learning rate; γ is the discount rate.

Input : s_{t-1} the previous state, a_{t-1} the previous action, s_t the current *sampled* state.

Output: a_t the next suggested action.

```
1  Function Learn(s_{t-1}, a_{t-1}, s_t): a_t is
2      if Random(0, 1) < ρ then
3          a_t ← GetRandomAction(s_t)
4      end
5      else
6          a_t ← GetBestAction(Q, s_t)
7      end
8      r ← GetReward(s_t)
9      Q(s_{t-1}, a_{t-1}) = (1 − α) · Q(s_{t-1}, a_{t-1}) + α · (r + γ · Q(s_t, a_t))
10     return a_t
11 end
```

Let's make sense of this algorithm. The first thing to understand is that Q-Learning works through a state-space S that contains *all* possible states of the agent in question. The state space S must be discrete and countably finite.[2] For each state $s \in S$, there's a set $A(s)$ which contains all actions $a \in A(s)$ that are available and can be made through the state s. The algorithm is initialized (usually[1]) so that $Q(s, a) = 0$, $\forall s \in S, \forall a \in A(s)$.

Lines $2 \rightarrow 7$ are *vital*. What they do is either select a *greedy* action or an *exploratory* action, depending on the value of ρ. If, for example, $\rho = {}^1/_{10}$, then *one out of ten times* Q-Learning will select an action—not based on the best available action—but totally at random. This is crucial so that the algorithm continues exploration of new strategies and does not fall into a rut of behavior. `GetRandomAction()` picks *any* of the available actions $a \in A(s_t)$ to the state s_t.

On the other hand—if again we assume that $\rho = {}^1/_{10}$—*nine out of ten times* Q-Learning will select the action a for which $Q(s_t, a)$ is maximized. That is, we select the action $a \in A(s_t)$ with the best Q-Value given our current state s_t[3]. This is the role of the function `GetBestAction()`.

[2] This is one of the major obstacles in using Q-Learning in the context of our application: free-range driving agents have continuously infinite states, and the trick is in creating a good quantization of these states into a countably finite space

[3] With ties broken arbitrarily, a fact that is important in the behavior of the algorithm but which [1] forgot to mention in its presentation

Line 8 is also fundamental to the behavior of the algorithm, and that's because the function `GetReward()` must be carefully designed to reward or punish the agent for having arrived at the particular state s_t. If the previous action a_{t-1} lead the agent to a state which we *don't want to reinforce*, then we punish the agent by having `GetReward()` return a negative value. If, on the other hand, the state s_t is one we *wish to reinforce*, then `GetReward()` can return a positive value.

Line 9 is where all the learning takes place. How do we interpret this assignment? First, notice that it is an exponential average between the Q-value of the *previous* state/action pair against the reward plus the Q-value of the *current* state/action pair. The whole idea is that if the previous action a_{t-1} lead me from state s_{t-1} to state s_t, we want to leave a trace of how beneficial or how prejudicial that state transition was. The parameter α, also called the learning rate, provides a trade-off on how incisive will this learning be. High values of α mean that Q-Learning will quickly forget what is has learned previously in favor for what it is learning *right now*. Low values of α allow the learning to be more cautious. The parameter γ, or discount rate, controls how much an action's Q-value depends on the Q-value state (or states) it leads to. All parameters—ρ, α, γ—belong to $[0, 1]$.

Finally, line 10 returns the suggested action a_t.

3.3 The Differences of our Q-Learning *(optional)*

Our Q-Learning algorithm will be customized for the context of our problem. It will differ from traditional Q-Learning in three respects:

1. It will be *online*, which means that learning will happen *as the game is executing*. Our agents will be constantly learning and adapting. This is different from offline learning, in which the agents learn in a simulated space. In many ways preliminary results show that some pre-training is needed, so calling it online might be an overstep in boundary. However, note that *offline-learning* refers to learning through a simulation in which the player does not take part. This is not our concept.

2. It will be *sample-based*, meaning it doesn't use a formal model of the underlying problem. Traditional Q-Learning uses a formal model of the problem so that, once the algorithm has decided on a course of action a_t, it can immediately evaluate the formal model to procure the next state s_{t+1}. Since our problem is a real-time application in which we don't have an underlying formal model, we're going to *sample* the different states in S from the information in the game's environment. This has engendered interesting challenges. Also, agents do not sample at every moment of the `FixedUpdate()`. They have a `SamplingRate`, which was a necessary introduction because between two subsequent frames there was too little difference between states for Q-Learning to learn meaningful representations. So we "regularized", so to speak, the learning system by taking

samples at near-level human speed. So they are trained with that capability of making 5 decisions per second. Later, once training converges to a minimum, we artificially augment the `SamplingRate` to 20 decisions per second, and their performance increases. So our training method is based on multiple layers of training. Doing this regularization and subsequent super-sampling was our own innovation and we couldn't find literature to support it. Not in books at least.

3. One of our innovations was in the use of a meta-designed greedy-AI that *defines the reward line* for the agents. And so agents follow this overlay of game AI on top of them and within that framework, Q-Learning does its work.

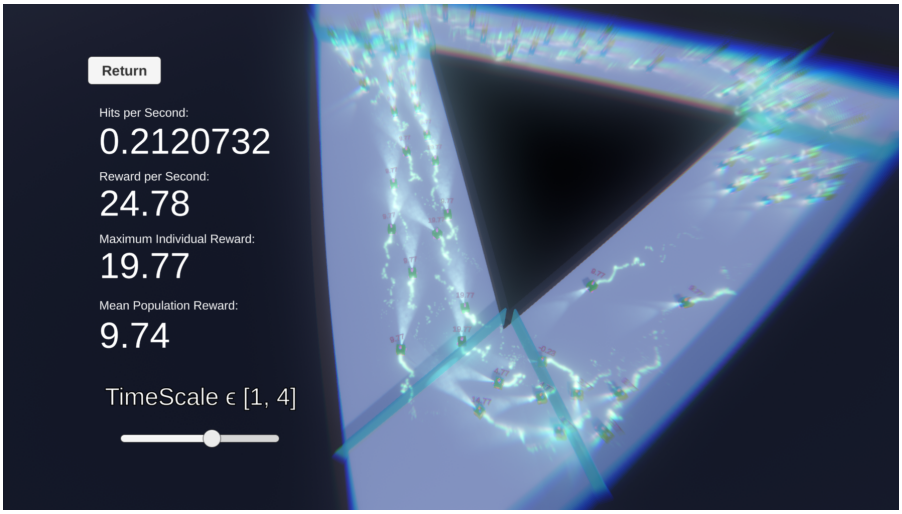

Figure 4: A preliminary view of the behavior of our agents.

The recommendations of [1] is in the process of crafting the AI, and especially on how to craft the state space and the action space for the agents, and how to design the reward curve. These will be explained at appropriate sections below. An additional suggestion is in the values of α, γ, and ρ for each of the cases in which we use Q-Learning.

3.4 The *Unity Engine*

We've decided to use the *Unity Engine* for the development of this project. We've made this decision because the focus of the project is AI, and *not* computer graphics, game linear algebra, or game physics.

To have attempted the project in raw *C++* and *Vulkan* would've been a waste of time. By leveraging the use of *Unity Engine*'s capabilities to handle

physics and rendering, we can focus on iterating over the design and learing process of our agents: we can spend this additional time fine-tuning the algorithms hyperparameters, the reward functions, and other aspects of the solution that are more relevant to the goals of the project.

Additionally, *Unity Engine* is one of the engines with fastest growing popularity in recent years, specially in the area of indie game development. Interest on the engine is at its peak, with more *Unity* users than ever. When we're using *Unity*, we're using a unified game design language.

Essentially, the only other option we had for crafting this game was the *Unreal Engine*, which *is* probably the most popular game engine in the market. However, *Unreal Engine* cannot export its work to all *UNIX* systems, nor can it export to mobile devices, and it cannot run in *Linux*. It also uses a proprietary version of *C++* (which doesn't allow us to publish the code with an open source license). *Unreal Engine* also works under the paradigm of *inheritance* as opposed to *object composition*. This creates brittle, rigid, hard-to-change class hierarchies that suffer from all sorts of problems like the *diamond inheritance problem.* Additionally, the *Unreal Engine* is geared towards graphics and specific types of games (like shooters, for example). The *Unity Engine* is a much more general *game design* engine and allows for a richer variety of games to be created within it. Plus C# has become open-source, and thus our code can also be published in the same license. Finally, our high-level of expertise with *Unity* meant that we could get right into prototyping our solution quickly, without sacrificing our *capability for expression:* there are no apparent limitations towards what can be created with *Unity*, specially in relation with the *Unreal Engine*.

Perhaps one viable option would've been *Panda3D*, Carnegie-Mellon's engine which can run on both *Python* or *C++*. By having *Python* at our disposal, upgrading this project to use Deep Reinforcement Learning would've been easier. The problem is that *Panda3D* is in its infancy, and it lacks the plethora of features available to Unity right off the box.

The *Unity Engine* will be the project's *only* dependency. Specifically, we will be using *Unity 2019.4.14f1*. The programming language of choice will be C# and *no ML libraries will be used, everything will be custom built from the ground up for this project.* Our goal being to leave the code open-source so that others can learn how to leverage Q-Learning in their own real-time applications.

`MonoBehavior` is the standard class which gives *behaviors* to all *Unity* `GameObjects`. We call classes that inherit from `MonoBehavior` *components*, and in *Unity* a particular gameplay mechanic is implemented as a collection of components through the Composite design pattern.[10] Figure 5 shows `MonoBehavior`'s execution cycle, which will be important to understand the design of our system.[4] In particular, it's important that you review the `Awake()`, `Start()`, `Update()`, `FixedUpdate()`, `OnCollisionEnter(Collision)`, and `OnTriggerEnter(Collider)`. callbacks.

[4] It's an `EPS` image, so you can zoom all you want to study that graph.

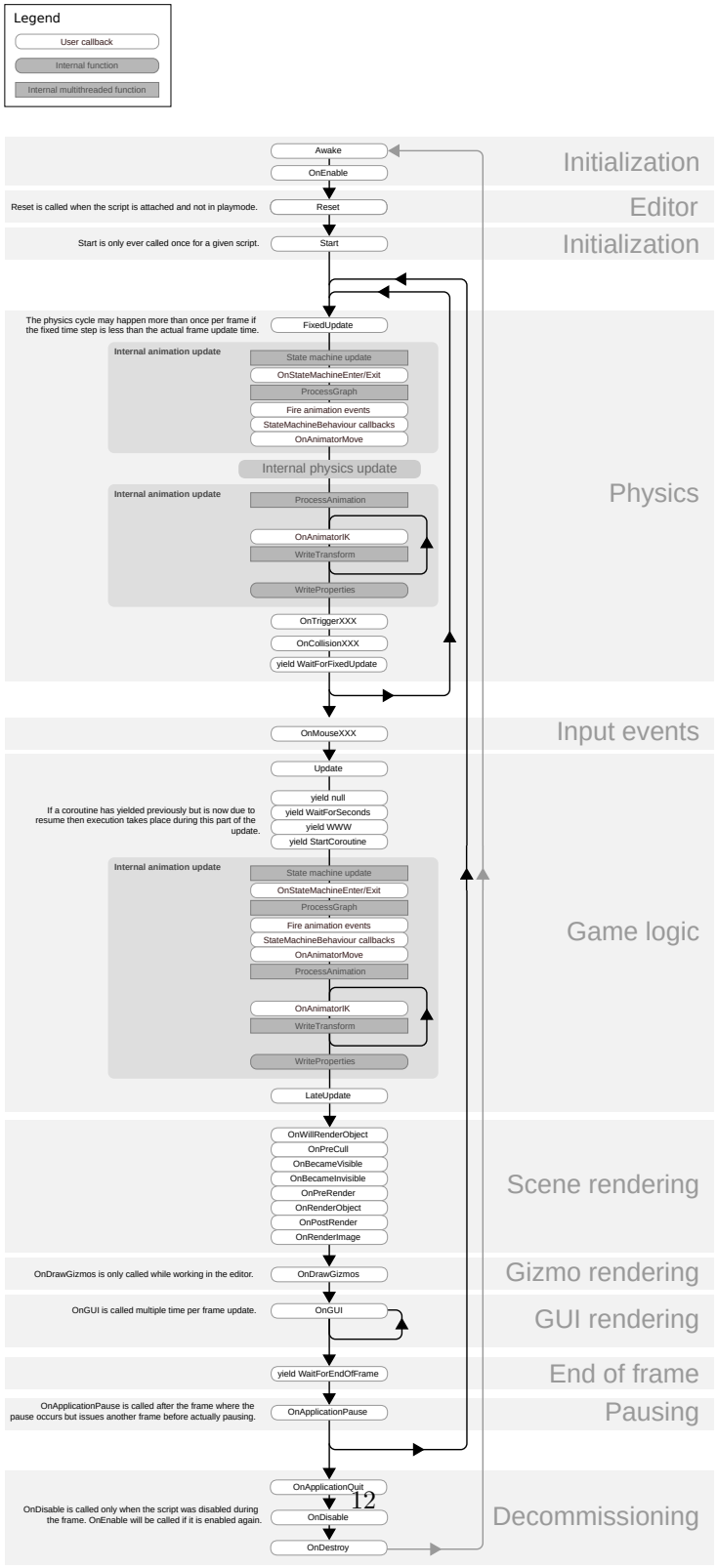

Figure 5: A preliminary view of the behavior of our agents.

4 Our Analysis

Our analysis is comprised of two parts: the use cases and the most comon scenario for each case, and the modeling of the problem of building an intelligent artificial agent.

4.1 Use Cases

There are several *use cases* provided by our system. While our project is designed to be *tinkered* with within *Unity*, we *will* provide a build that allows users to execute the game natively, without the use of *Unity*. The use cases for these builds are (see Figure 6):

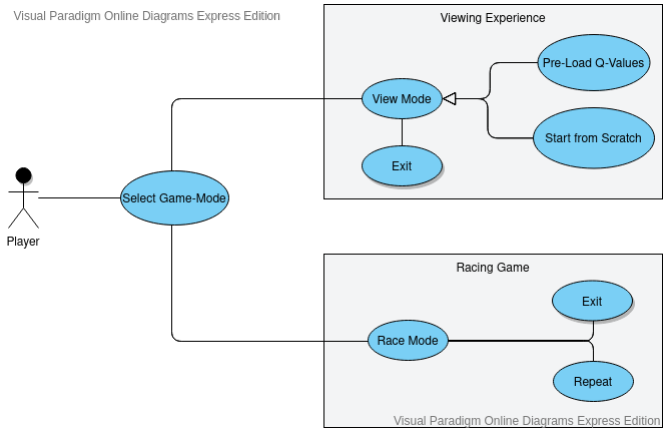

Figure 6: UML: Use cases offered by our system.

1. `Select Game-Mode.` This is accessed through the main-menu, and it gives three choices: view mode with pre-loaded Q-Values, view mode starting from scratch, and play the racing game.

2. `View-Mode::Pre-Load Q-Values.` Puts the player on a prototypical scene to view the agents moving. The agents are loaded with pre-trained Q-Values. There's no other interaction from the part of the player.

3. `View-Mode::Start from Scratch.` Puts the player in a prototypical scene to view the agents moving. The agents aren't pre-trained, so we watch them train from scratch.

4. `Race-Mode.` Puts the player into a race game against pre-trained agents.

5. `Race-Mode::Repeat.` Replay the game once it has ended.

6. `Exit:` In both view-mode and race-mode use cases, exit to the main menu.

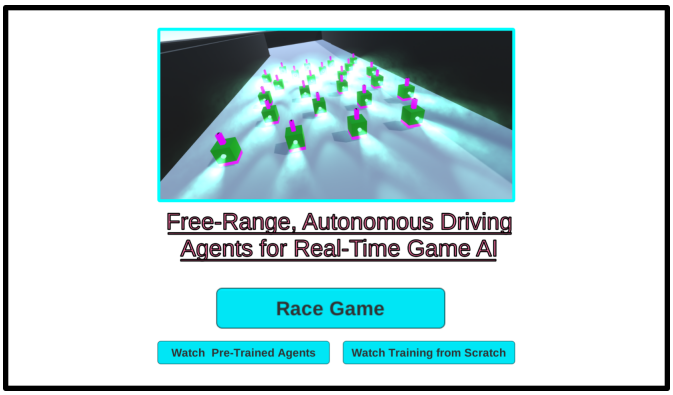

Figure 7: The main menu for the program, showcasing the three main use-cases of the system.

4.1.1 Scenario: `View Mode::Start from Scratch`

This scenario involves selecting the option in the main menu, waiting for a timer as the user reads the instructions, and watching the agents train for an amount of time defined by the user.

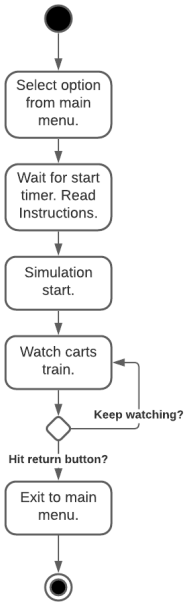

Figure 8: Activity diagram for `View Mode::Start from Scratch`.

4.1.2 Scenario: `View Mode::Pre-Load Q-Values`

This scenario is almost identical to the previous scenario, except that the Q-Values of pre-trained agents are loaded during the start-up timer.

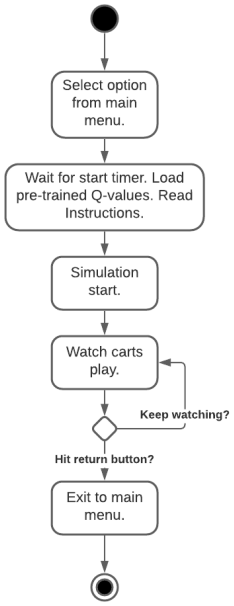

Figure 9: Activity diagram for `View Mode::Pre-Load Q-Values`.

4.1.3 Scenario: `Race-Game` to Completion

This is the main attraction of the project, the `Race-Game`. The player starts by waiting for a *start timer* to finish. As this happens, the player is meant to read the game's instructions, and the pre-trained Q-Values are loaded. Once the game starts, a timer is set to ring in three minutes. In that time, the player must *avoid* all obstacles or she is punished by removing two points from her score. If she reaches a lap (what we'll call a *reference node* below) her score is increased by three points. A small trickle of points is removed on every frame update. Once the timer is finished, if the player has beaten the score of all the artificial agents, she wins. Otherwise she looses.

The player plays with a joystick, though the game can also be configured at build-time to accept input through a mouse interface: wherever the player points and clicks, the player's avatar will move in that direction, as long as she keeps holding the left click down. If playing with a joystick, the player can break with the right trigger. If playing with a mouse, the player can break with the right click.

See Figure 10 for the activity diagram of this scenario.

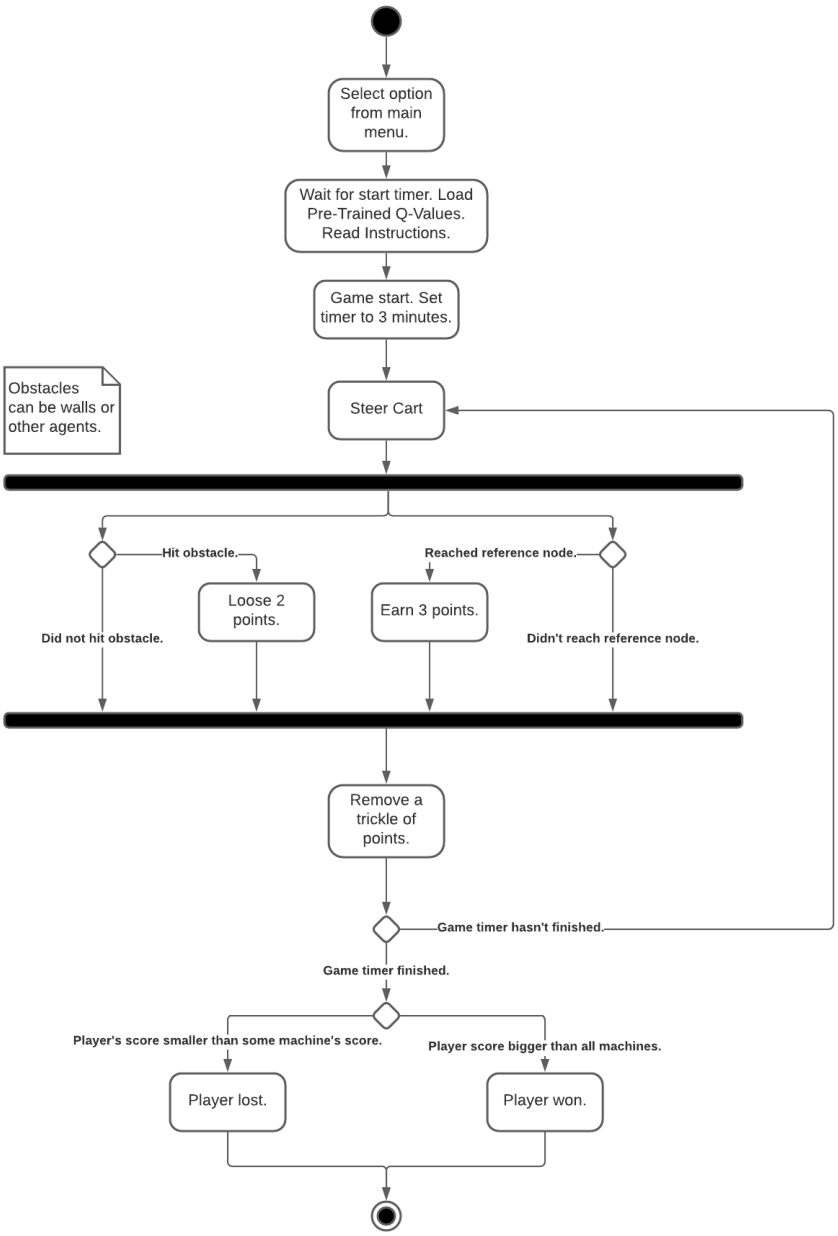

Figure 10: Activity diagram for `Race Mode` to Completion.

4.2 Modeling the Agents

For our problem *modeling* we divided the essence of our problem into two sub-problems. These are shown as roots in Figure 11

The first root is the *exploration of the road*, in which our agent reads a list of reference nodes and must determine what's the structure of the road based on these, which might, in themselves, contain information to aid the agent find the correct route. So that's an important element and we will call it the **reference node**.

The second root is the *avoidance of collisions*, whose purpose is to avoid the collision between agents against agents and between agents against other obstacles. The agent must detect the incoming obstacle. If it can, then it takes an appropriate action, we adjust levels of speed and direction to improve efficacy. If it cannot detect the object, then it will probably collide with the other object, in which case we must figure out how to detect that pattern in the future. We call the frame of information regarding the incoming obstacle as **pattern of observation**.

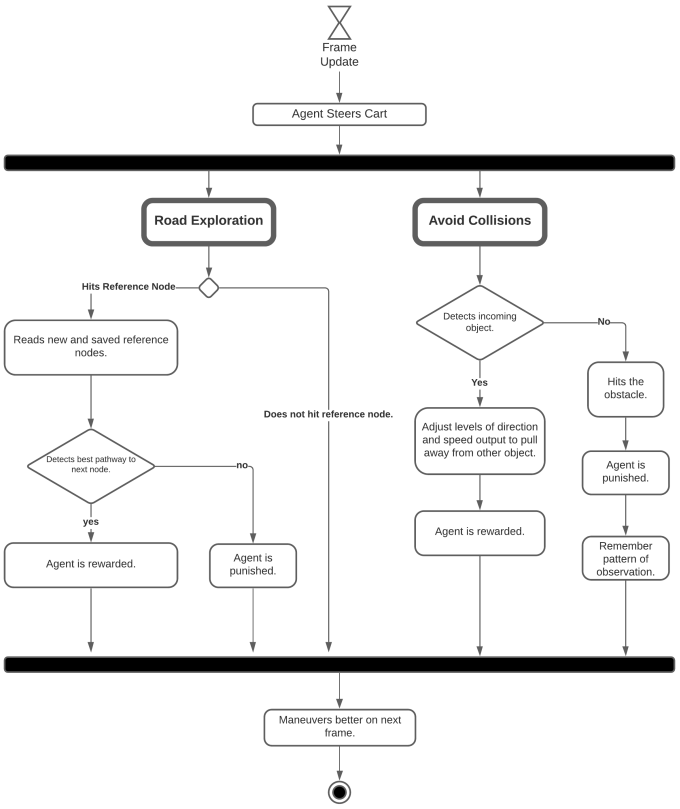

Figure 11: The problem's model.

5 Our Meta-Design

So we have two problems: road exploration and collision avoidance. But there are additional issues surrounding reinforcement learning in the context of our deployment. Yes, reinforcement learning is a common perspective and offers general solutions applicable to the learning of any game. Indeed, the goal of reinforcement learning is to maximize the agent's reward long-term, be it in competitive or cooperative games.

However, learning in scenarios with multiple agents is one of the fundamental problems in current AI research. These independent learning-agents can result in an absence of stationary in the stochastic process: that is, the agent's don't converge to an optimal strategy at any time. Additionally, the presence of adversarial agents can make difficult the exploration for an individual agent during the learning process. Even in restricted environments, the challenges of learning in the presence of other agents are far from being resolved. Some of the actual solutions scale it empirically with specific, restricted tasks, limited to environments of relatively small size.

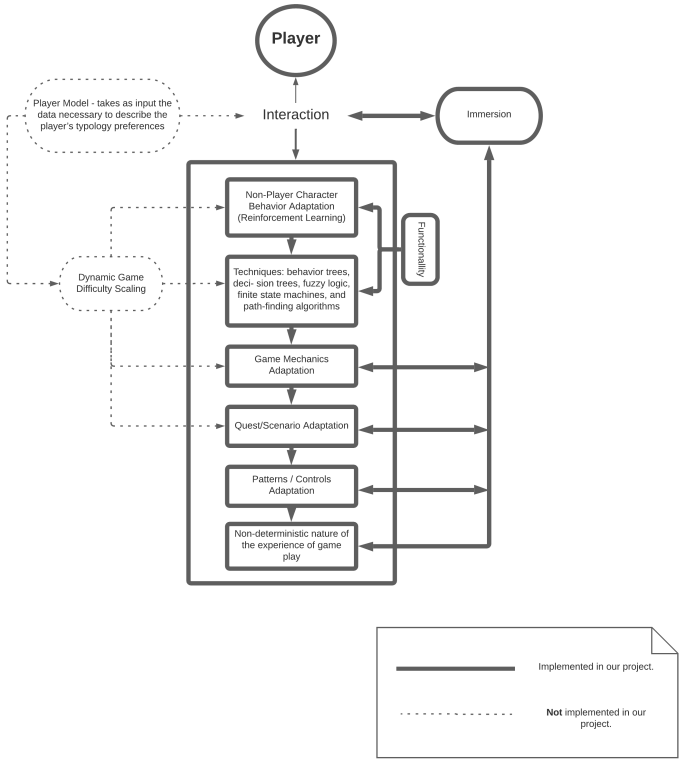

Figure 12: A description of the game's flux and the game/player interaction.[15]

To advance the research of this learning could lead to new developments. In the process of reinforcement learning of a specific agent, there's a risk of over-fitting to individual tasks and about specific opponents. The problem can be approached in a progressive manner: deal with each task of the agent individually. A big part of the current empirical work in reinforcement learning of multiple agents focuses its attention in a few individual's tasks with only one learning agent.

The generalization—beyond individual tasks and different kinds of opponents—is an area that has great potential for future work. In applying reinforcement learning when a new problem faces us, we must be careful with the model, the optimization strategy, and their parameters, for we must learn a behavior that's adequate given the limited set of available samples. In the case of reinforcement-learning, we might see the state space S being so large as to overcome the capabilities of physical memory. To trace learning through this enormous state space would be virtually impossible. Thus, reinforcement learning requires a delicate balance of hyperparameters.

Figure 12 displays the layering process that we will use to create the game. The underlying mechanic will be controlled by Q-Learning and, on top of it, other deterministic game AI techniques will be used. Elements of even higher-order will be iterated through the metric of *immersion*. Q-Learning and the deterministic layer will be created guided by *functionality*. That is, the first two layers will make sure the agent doesn't collide and learns the route of the road, our two sub-problems as stated above. The rest of the elements will be created on top of these devices to produce a game players can enjoy. Our aim is non-determinism.

5.1 Tracer Bullet Process

Our process for transforming the analysis into a design was guided by the *tracer bullet* methodology, where we would try methods on the fly and discard that design, until we arrived at a design that could be set in stone for the purposes of the project.

The **first part** of our analysis was to observe evidence of intelligent behavior. In our analysis of how humans drive, for example, we took a good look at what kinds of information influences a driver. He seems to be rewarded for following the current road direction, and there seems to be a higher-order of thinking for choosing the current road direction. He also seems to use the lots of information on its environment to make decisions about where to move next, such as the speed and angle at which another car is coming towards or away from him. This is what we described above as **patterns of observation**. This information we derived as "important" for a driver will be further explained in Section 6.7. We also looked at The Flying of the Starlings, which was of inspiration for our method.

In the **second part** of our analysis we went into studying the state-space sizes required for an effective learning function. Depending on the pattern of observation, this state space can vary significantly in size. And, as we toyed and

twisted with the model, we saw that memory was a severe limitation. Our version of `CartState` (our patterns of observation) and `Destination` (our reference nodes) would've required hundreds of mega-bytes per Q-Learning dictionary. So having one Q-Learning per agent was not a feasible direction. We then decided our Q-Learning would be a *collective*. All the carts are feeding this one, centralized Q-Learning algorithm, in this way learning speed is maximized, and we lower the memory requirements by several orders of magnitude.

Our **third part** of our analysis went into the studying of the sampling process. We stumbled upon the realization that having agents sample *at every frame* was disruptive for training. So, as explained earlier, we added a sampling rate to the carts, where they don't take a driving decision until a certain amount has passed. Carts were trained with 5 decisions per second, give or take a effects of a small—but totally customizable—deviation. This provided better results than per-frame learning. The hypothesis being that not enough time had elapsed between frames so that states had not changed by enough a measure, meaning that Q-Learning was learning that everything was invalid (every action it tried kept bringing me to the same state).

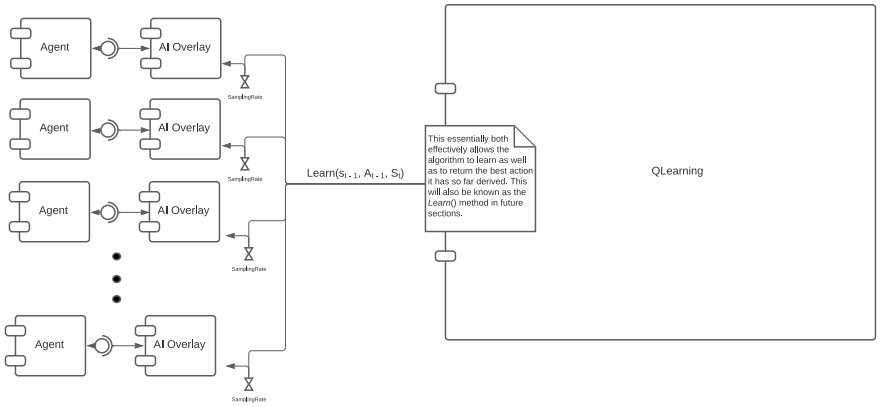

Figure 13: The preliminary analysis for our project for movement mechanics.

As the **fourth part**, we mounted on top of this bare-bones system several layers of complexity. We added an **AI Overlay** which makes deterministic, but greedy decisions on the agent's behalf. It chooses what it considers to be the most "viable" direction for the agent based on the **reference nodes** described earlier. The agent is then rewarded for following this AI Overlay at the Q-Learning level. This AI Overlay, when operating off Q-Learning, is already procuring a high net reward [5]. Adding Q-Learning then, as a next layer of behavior, improves the performance by 333.33%.[6]We also mounted, as part

[5]See Section 8 for empirical information on the degree of success of each layer of this solution

[6]Evidence of this improvement can be watched in this series of videos.

of the AI Overlay, another Q-Learning-driven piece of machinery to guide the agents through mazes. All of this will be explained in detail at later sections.

This was the last step in the conceptualization of the problem. All of these considerations will be explained as they surface during the discussion of *Our Design* in the Section 6 below.

6 Our Design

The following explains in detail the design of the main classes that do all the hard-work on our proposed solution. These *are* set in stone, and may no longer evolve as the project progresses. Also, this presentation is *not* exhaustive either: we've chosen to present the classes that perform the most work in order for you to gauge how we'll implement our solution, and also so that's easier for you to get into the code and tinker with it.

In the discussion that follows, our agents will be known as *carts*. Yes, the technologies implemented in this project can be used for a wide variety of game mechanics, but we chose to implement a cart race game in order to showcase the technologies partially because carts are easy to build from *Unity*'s primitives, but mainly because the concept of a cart is readily understood and grounds the metaphor of the agent into something concrete. Note, however, that these are futuristic, free-range hovercraft carts (i.e., flying carts, carts without wheels).

We will use Q-Learning for two purposes:

1. For creating the *movement behavior* of our agents. That is, we wish to use Q-Learning in such a way that through exposure to stimuli our agents are able to learn how to maneuver their way to a destination successfully, which means they avoid any and all obstacles and get there as fast as possible. This means that this part of Q-Learning is influenced by the patterns of observation we talked about earlier. These are called `CartState` in our model. And this part of Q-Learning is also influenced by the AI Overlay, since Q-Learning rewards the agent for following what the AI Overlay calls the most viable direction.

2. For creating the *learning of the structure of a maze*. Which means we will use Q-Learning so that agents form a *collective* understanding of the structure of a maze and develop a Q-Value matrix that, given any starting point within the maze, the agent is able to leave the maze optimally by following a set of suggested waypoints.

One of the major challenges before us is *quantizing* the state-space of the agents into a countably finite state-space. Remember that these are driving, free-range agents, meaning that their true underlying state is continuous and infinite. How we do this quantization in order to summarize enough properties of the agent so that the algorithm can learn optimal strategies will be a crucial aspect in the design of our solution.

Another major challenge is in the design of *reward mechanisms* for our agents. The wrong reward function can generate agents that fall into ruts of behavior, applying strategies that work locally, on the short-term, but that fail on the long-term.

In the end, it will be the balance between machine-learning through reinforcement-learning and traditional AI techniques which will bring home the results we seek in our project. Not everything will be solved by Q-Learning, for there is no silver bullet in engineering. Part of the challenge is hence how to balance traditional game AI techniques with modern machine learning strategies. The AI Overaly explained above is the vector of this aspect of the solution.

In the discussion that follows, whilst our focus has been on the *design* of the system, we've taken the liberty of discussing some implementation issues as well, but only where those issues are relevant in the discussion of the overall design of the solution.

6.1 `QLearning`

We've decided to encapsulate the Q-Learning algorithm into a generic class called `QLearning<State, Action>`. See Figure 14.

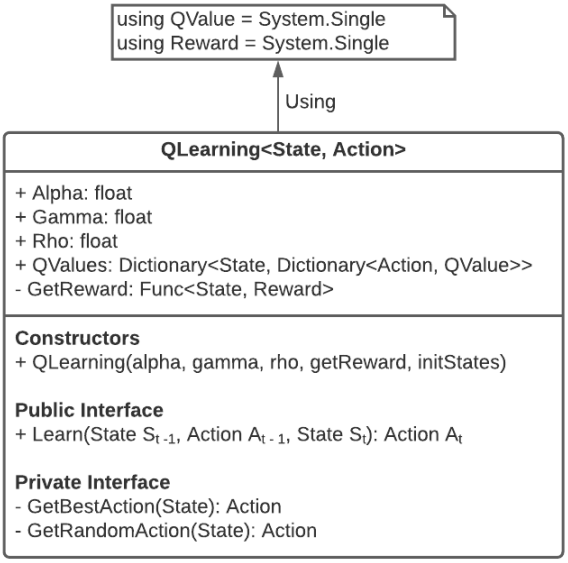

Figure 14: The design of QLearning

It is evident that in the `QLearning` class definition the dictionary `QValues` stands-in for the function $Q(s, a)$ in Algorithm 14.

This class requires that a C# *delegate* `GetReward` be assigned to it (to calculate the reward for a given state s_t), and that a delegate `initStates` be passed

to it during construction to set $Q(s, a) = 0$, $\forall s \in S, a \in A(s)$. We made this decision because only clients of our class can know in the end what the state-space S looks like. *Yes*, `State` is a generic argument of the class and, through reflection, in theory we could've initialized the $Q(s, a)$ dictionary to all zeroes by developing a mechanism to discover all the different combinations of values that the class `State` can take. The problem with this decision is that, depending on how `State` is defined, is can lead to an *enormous* state-space, one that can take gigabytes or even terabytes of data to store. Additionally, there's no way to predict what actions $a \in A(s)$ are available for each state $s \in S$. That is, we cannot assume that all actions are available through a specific state s. Thus, we left it to the clients to initialize the Q-Value store using the state-space and the corresponding actions as they see fit.

Notice how the `Learn` method has basically the same contract as our Algorithm 14 and is, in fact, implemented just like so, except that in the C# programming language and not in pseudo-code.

Finally, realize that `System.Single` is equivalent to `float` in C#.

6.2 `MovementQLearning`

`MovementQLearning` is a class whose reponsibility is to serve as a realization of the Q-Learing algorithm for short-term *movement* of the free-range driving agents. It is a `MonoBehavior`, and it *does not* inherit from `QLearning`, but instead *contains* one.

See Figure 15 for a summary of `MovementQLearning`'s definition.

We chose to make `MovementQLearning` a `MonoBehavior` for two reasons:

1. We wanted to make it easy to append `MovementQLearning` as a reference to the cart objects within *Unity*, so that the cart's components can make use of the algorithm from within. And we wanted a one-to-many relationship, where all the carts share the same underlying algorithm, so that the algorithm learns faster by taking in the stimuli of all the available carts in the scene.

2. We wanted to add the capability of *saving* and *loading* the Q-Values into disk. But we didn't want the game's main loop to halt during this process, so we sought to use multithreading for these operations in order to have the loading and saving of the Q-Values to continue in another thread. For example, a thread is launched to load the Q-Values (a binary file that can weight hundreds of megabytes). Once that thread has furnished a result, we use a double buffer[11] to replace the old Q-Values with the newly-loaded Q-Values in the frame's `Update()` of *Unity*'s main loop.

Notice that this class has the concrete `GetReward` function which is fed into `QLearning`'s `GetReward` delegate to calculate the reward for having arrived at a state s_t. As you can see in the UML, there are a variety of so-called **reward-modifying attributes** which affect the final reward function for movement. For now, though, note that carts are punished for colliding against obstacles

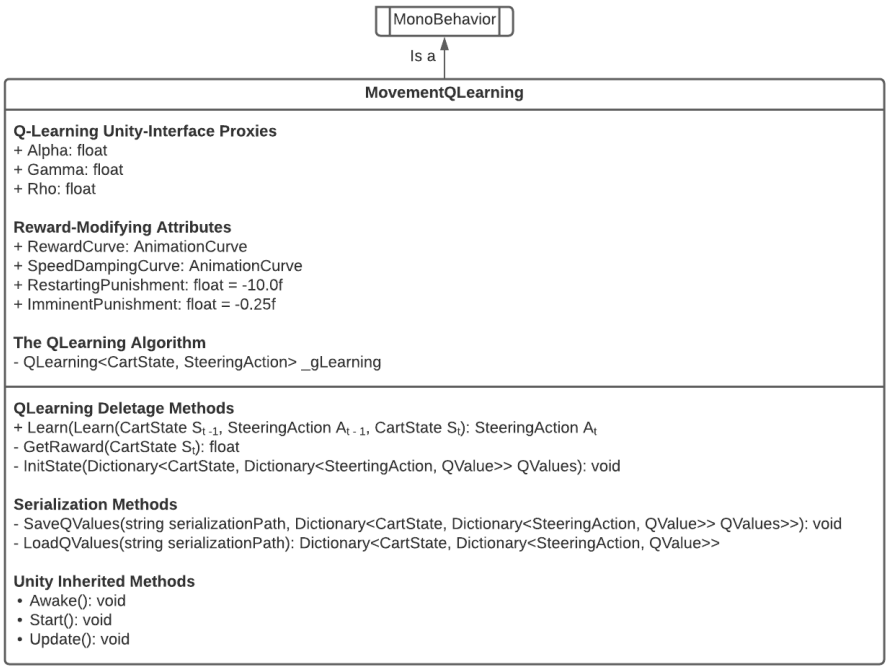

Figure 15: The design of MovementQLearning

or against each other and are also punished for being too close to one another. Carts, instead, are *rewarded* for driving aligned to the *optimal direction*, which is specially defined as the best direction vector towards their next `Destination`. This optimal direction is known in the jargon of our technology as the `CurrentRoadDirection` (`CRD`), which is the direction suggested by the AI Overlay.

In essence, all this class does is *suggest* an action a_t to a cart once the cart is available to make a new decision.[7] This suggested action comes bundled as a `SteeringAction`, a simple `struct` which tells the cart whether to break or whether to move in one of N different steering directions.[8] As you can also see, the `State` parameter for the `QLearning` generic is of type `CartState`. The design of this `struct` required some serious thought, and will be explained in a section further below.

[7] When is a cart able to make a new decision? This will be explained below.

[8] The parameter N governs many aspects of our solution, and is customizable through the script `Configuration.cs`

6.3 `Destination`

A `Destination` is our realization of the referece node. It's purpose is to serve as simple nodes to the `TrackQLearning` class (and algorithm) described next. See Figure 16 for the UML of `Destination`.

First, a `Destination` is always paired with a `BoxCollider`, a relationship which isn't isn't visible in the UML. But the game *prefab* contains it.

Then this class contains information about:

1. A local AABB.
2. The links to another `Destinations`.

The AABB information `Max` and `Min` are two `Transforms` that must be children of `Destination`; these define the local AABB through which we define what's considered the maximal point or minimal point in the volume.[9] Internally, since these are `Transform` objects, they can be then later sampled for world-coordinates of those same local points configured by a level designers. Which is an advantage.

Then it contains the list of other `Destinations` linked to it. Again, this is because `TrackQLearning` requires a multi-link graph. This will be explained in the next section.

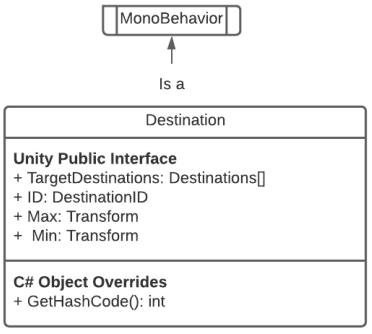

Figure 16: The design of Destination

6.4 `TrackQLearning`

`TrackQLearning` is another realization of Q-Learning but its purpose is for carts to *learn their way through a maze*. Mazes are directed graphs of `Destination` objects. As you can see from Figure 17, this realization for Q-Learning follows

[9] For the keen reader, this is *not* possible through code by sampling the `BoxCollider`. What you do is, most of the time, align the `Min` and `Max` `Transforms` according to the Trigger2D. Such are the limitation of working which such a high-level engine.

the same pattern as `MovementQLearning`, but it has `Destination` objects both as states *and* actions. That is, once we reach a `Destination` as a state, the algorithm suggests another `Destination` as an action.

`TrackQLearning` is part of the AI Overlay.

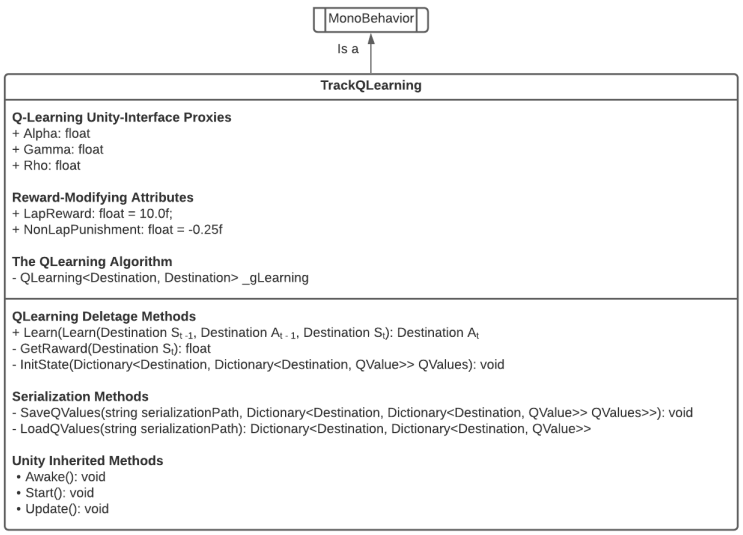

Figure 17: The design of TrackQLearning

In our project, all scenes will be considered mazes, even tracks for a race. The only difference is that a racetrack is a maze that's laid out as a graph that appears like a unidirectional linked list of `Destinations`, while mazes have more complex graphs. In the case of racetracks, `TrackQLearning` is *still* working but its job is impercetible, since each `Destination` only has one outgoing link, and that is to its one and only next-`Destination`. So, in this case, when `TrackQLearning` is asked for a suggestion on the next `Destination`, it will invariably output the same and only thing. Things get more interesting with more complex mazes, however.

Look at Figure 18. In our project, the graph for this maze is built *procedurally* through an algorithm that systematically launches rays whose origin lies at the center of each of the small rooms you can spot in Figure 18. The `Destination` objects are the cyan-colored triggers visible in the figure and serve as bounds for each of the rooms. The algorithm launches a ray in each of the four cardinal directions from the center of each the room, and creates a graph where each node (`Destination`) has an outgoing link to all `Destinations` that can be visited from that node. Our demo will include this maze, though it takes carts a while to find the exit, and even more so to create a matrix of Q-Values such that, from any starting point, the cart knows how to exit the maze optimally.

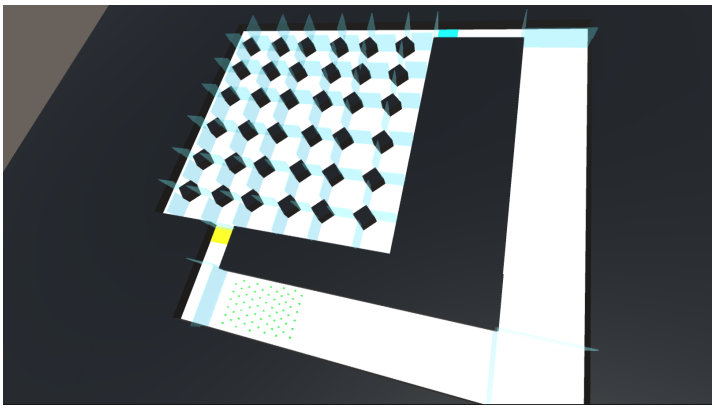

Figure 18: An intricate maze (from a conceptual point of view).

Mazes of arbitrary complexity can be built in this fashion.

Our favorite maze is the *Triangle* track (see Figure 19), which will serve as a testbed for our race game. It's a simple track, but thanks to it's simplicity we've been able to populate it with over 90 carts, and the richness of their interactions quickly train the Q-Learning matrix, converging to pleasing agent behavior.

The design of the reward function for `TrackQLearning` is simpler than that for `MovementQLearning`. If the `Destination` to which we've arrived is considered the `LapDestination`, we reward the agent. Otherwise we punish the agent. *Yes*, this does mean that we're punishing the agent every time they cross a `Destination` which isn't the `LapDestination`. But this is good because it quickly creates the avoidance of pointless loops as Q-Learning leaves a trail of negative Q-Values in its matrix.

`TrackQLearning` is part of the AI Overlay discussed during the analysis of the problem.

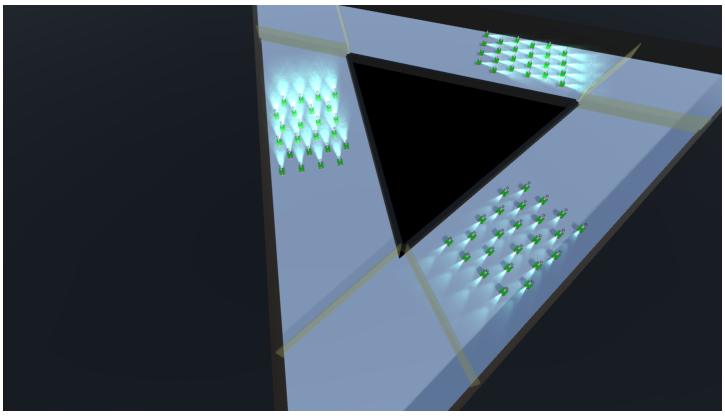

Figure 19: Our Triangle racetrack, which will be featured in the racing game.

6.5 Carts

Carts in our game are *Unity* `GameObjects` made-up of several different components which, as a whole, give the carts their behavior. There are two types of carts:

Player Cart: the cart that's maneuvered by the player.

Automatic Cart: the carts that use AI to maneuver their way through mazes.

These two types of carts share some components. For now, however, we'll stick to automatic carts and their components. Once we've got that under our belt, the design of the player's cart will segue easily from our previous discussion.

Automatic carts have the following components attached to them:

- `Rigidbody`: *Unity*'s component for modeling physical mechanics.
- `SphereCollider`: *Unity*'s component for sampling collisions based on a sphere.
- `Cart`: the most essential script for a cart. Provides an interface to control breaking and maneuverability.
- `AmbientStateSampler`: the most intricate script in our game. It samples the cart state and quantizes it, leaving it ready for consumption by `QLearnedManipulator`. Part of the AI Overlay lies in this component.
- `QLearnedManipulator`: A component which manipulates a cart based on the suggestions of the two Q-Learning algorithms `MovementQLearning` and `TrackQLearning`. Part of the AI Overlay *also* lies in this component.

The last three components where implemented by us, and they shall be explained in the sections that follow.

6.6 `Cart`

The `Cart` `MonoBehavior` is the interface through which we manipulate our carts. It provides methods such as `PushBreak()` and `LetGoBreak()` to manage breaking. It also has a C# property `Steering` which sets the direction that the cart is currently moving towards (when it is not breaking). The speed at which the cart moves depends on the `Acceleration` attribute. See Figure 20 for the UML diagram for our `Cart` component.

Note that `Cart` is mean to be *manipulated* by other components. That is, by itself it does nothing. It is the responsibility of other components to brake the cart or to set a steering direction for it. Without a so-called *manipulator* that controls `Cart`, it does essentially nothing. For example, during initialization, `Cart`'s `Steering` property is set to `Vector3:zero`, so it will remain static until a manipulator changes the steering direction of the cart.

A `Cart` can have two states: `Moving` or `Restarting`. When the cart collides with an object, it goes into `Restarting`, and in this state it cannot move. Once in `Restarting`, a timer is triggered. When the timer is greater than the attribute `RestartingTime`, then the `Cart` goes back into `Moving` state.

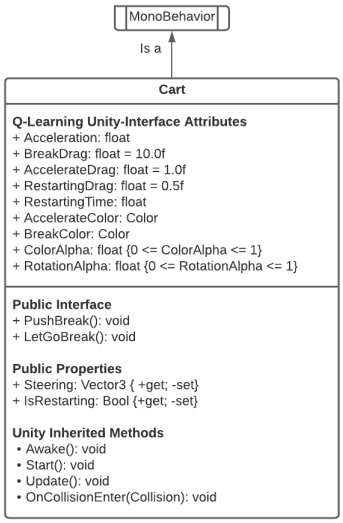

Figure 20: UML: `Cart`

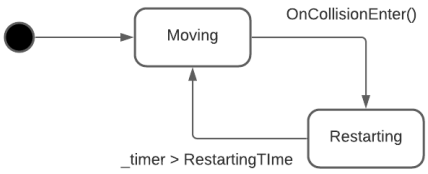

Figure 21: UML: `Cart` State Machine

6.7 `CartState`

Before introducing `AmbientStateSampler`, which forges the state of the cart through sampling, we must first analize what have designed as being part of the "state of the cart". The so-called patterns of observation.

This struct represents a state of the cart according to the model needed by our Q-Learning implementation. As you'll note, most of the fields are `sbytes` in order to save memory, because the Q-Learning data-structure (a hash dictionary) can grow unwildly large if we're not careful, crashing *Unity* in the

process. That's why we don't use `Int32` or `Int64`: because they occupy more memory, and the expanse of values afforded by these types is not needed in our implementation.

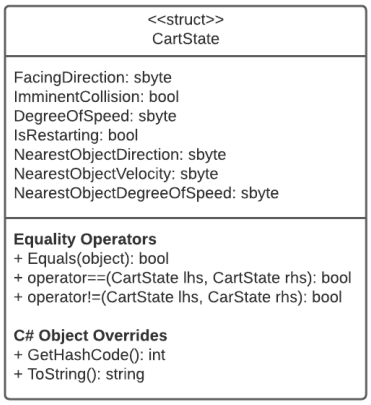

Figure 22: UML: `Cart` State Machine

What follows is a description of what each field represents in the context of `CartState`:

1. `FacingDirection:` Given the `CurrentRoadDirection` (CRD) (see `AmbientStateSampler`), which of the N different directions are we facing? With 0 meaning that we're facing the CRD, $N/2$ meaning we're facing *opposite* of CRD, and N being the variable of the same name found in `Configuration.cs`.

2. `ImminentCollision`: Are we in imminent collision with another object?

3. `DegreeOfSpeed`: What's our degree of speed (where the degrees of speed M are defined in `Configuration.cs`).

4. `IsRestarting`: Is the cart in `Restarting` state?

5. `NearestObjectPosition`: In which of the N different quantized directions is the nearest object's position, relative to *our* CRD? -1 means there's no nearest object.

6. `NearestObjectDirection`: What's the direction of movement, relative to *our* CRD, of the nearest object? -1 means there's not nearest object.

7. `NearestObjectDegreeOfSpeed`: What's the degree of speed of the nearest object? -1 means there's no nearest object.

Given these definitions, it is clear that the size of the state space is

$$(N) \cdot (2) \cdot (M) \cdot (2) \cdot (N+1) \cdot (N+1) \cdot (M) \tag{1}$$

If $N = 16$ and $M = 4$, as is the current configuration for our project, then we have $295,936$ possible states, which is over two hundred thousand states. Since we have $N+1$ actions per state, that creates $4,734,976$ state transitions, which creates a total of about 243 megabytes[10] of runtime requirements just to hold the `QValues` dictionary defined in the `QLearning` algorithm held by `MovementQLearning`.

6.8 `AmbientStateSampler`

`AmbientStateSampler` is quite possibly the most complex script in the entire system, and all of its details will not be presented here. We'll leave a few surprises for your tinkering. That said, `AmbientStateSampler`'s is part of our AI Overlay (in tandem with `TrackQLearning`) and its main responsibility is to *quantize* the state of the cart in two steps: it *continuously* samples the maximum speed attained by the cart on its `FixedUpdate()` *Unity* callback and it *instantaneously* samples the both the `CurrentRoadDirection` and infomation on nearby objects on its `GetState()` method.

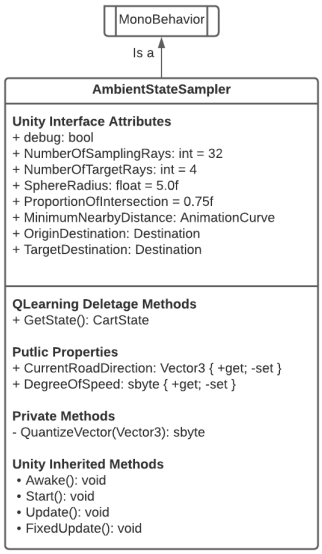

Figure 23: UML: `AmbientStateSampler`

The `FixedUpdate()` callback samples the maximum attained speed of the agent so that `GetState()` can quantize this information in terms of a `DegreeOfSpeed`.

[10]This datum was at first gauged from *Unity's* profiler, but we developed our own custom way to deducing an object's size for the purposes of these results.

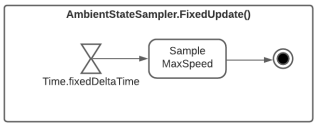

Figure 24: UML: `AmbientStateSampler.FixedUpdate()`

The `GetState()` function calculates the `CurrentRoadDirection` by casting $m =$ `AmbientStateSampler.NumberOfTargetRays` spheres towards different positions in the `TargetDestination`. The spheres are cast along a ray starting from the cart's position, sliding forward through the ray until they reach the target. The sphere that collides with the fewest obstacles is considered to have pointed into the most viable target direction for the cart. This path of least resistance is considered the `CurrentRoadDirection`. The details are more complex, but this is the gist of it.

`GetState()` also launches $k =$ `AmbientStateSampler.NumberOfSamplingRays` rays, where each ray starts at the center of cart, and the ray directions are calculated in a manner such that the rays point outwards out of the cart, forming a circle around it. These rays are meant to collide with obstacles, and the ray that reports the nearest obstacle gives us the nearest obstacle information for `CartState`.

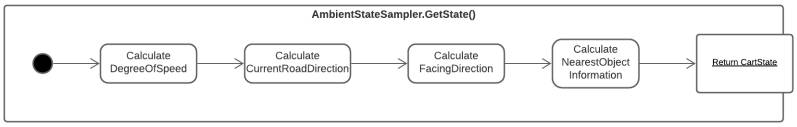

Figure 25: UML: `AmbientStateSampler.GetState()`

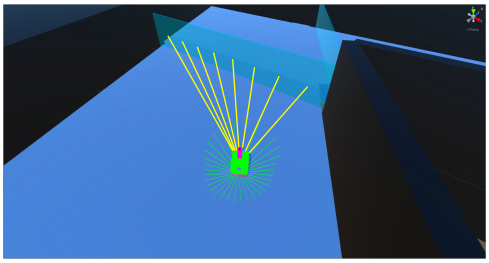

Figure 26: Each cart casts rays that surround it to query whether an obstacle is nearby. These are the green rays in the figure. The yellow rays in the figure are the direction of the spheres cast towards the `TargetDestination`.

6.9 `QLearnedManipulator`

`QLearnedManipulator` is the *Unity* component that brings together all the classes so far discussed and integrates them into one whole algorithm. It's purpose is to control the `Cart` component throught the use of the Q-Learning algorithms so far discussed. In this manner, `QLearnedManipulator` has a reference to both `MovementQLearning` and `TrackQLearning`. See Figure 27 for a reference on its class diagram.

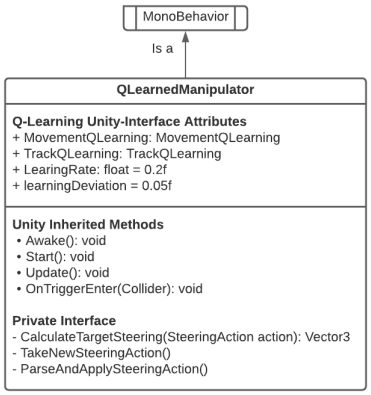

Figure 27: UML: `QLearnedManipulator` Class Diagram

One of the major discoveries in designing this solution was that, when we prototyped for the carts to make decisions *on every frame*, the result was that *not enough time had ellapsed between frames for there to be a significan state change*, so S_t was too similar to S_{t-1}, and learning was hampered, since the algorithm kept learning that, no matter what decision it took, the outcome was the same. So this lead to the innovation of a *learning rate* for the carts. What this means is that carts don't make decisions on every frame, but have to wait some time (in the order of milliseconds), to try to make a new decision. This is represented in the `LearningRate` attribute for `QLearnedManipulator`, and the whole sampling process is summarized in Figure 28. As you can glimpse from this activity diagram, carts also wait some time *before* starting to make decisions. This design decision was made so that carts aren't synchronized in their decision making. Finally, `QLearnedManipulator` calls its own method `TakeNewSteeringDirection()` to query `MovementQLearning` for a new action to take. This is described below.

When `TakeNewSteeringDirection()` is called, `QLearnedManipulator` queries `AmbientStateSampler` for the current (sampled) cart state. It then uses its reference to `MovementQLearning` to procure a suggestion on what action to take next. Finally, it uses its own `ParseAndApplySteertingAction()` to control the `Cart` component. This summarized in Figure 29.

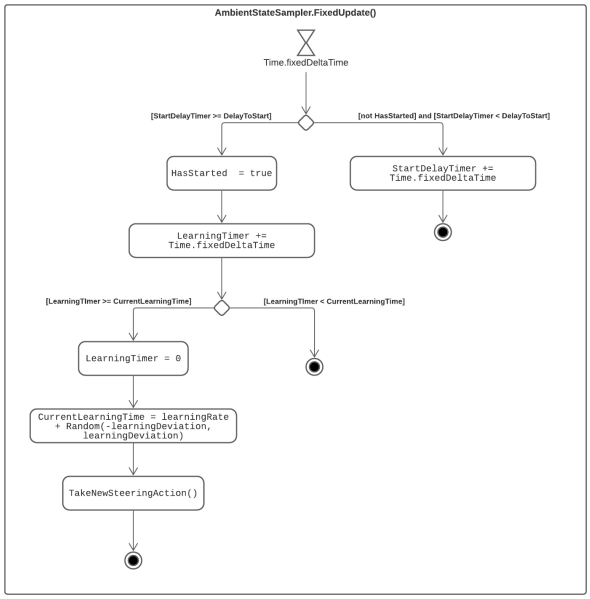

Figure 28: UML: `QLearnedManipulator.FixedUpdate()` Activity Diagram

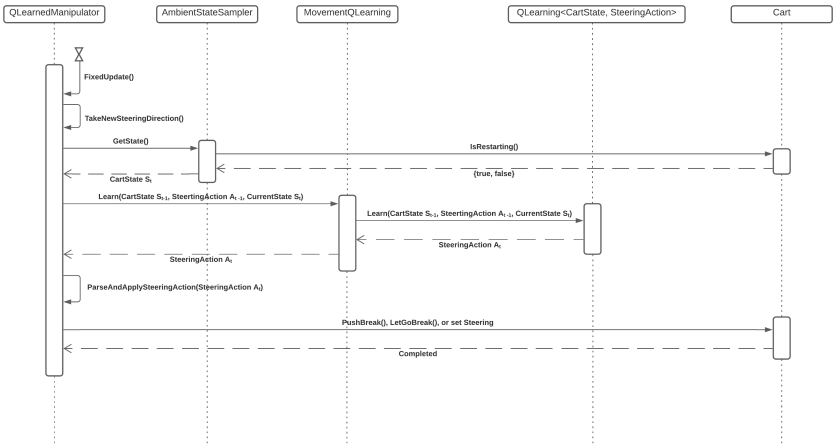

Figure 29: UML: `QLearnedManipulator.TakeNewSteeringDirection()` Sequence Diagram

When the cart triggers a `Destination` object, the `OnTriggerEnter(Collider)` *Unity* callback is raised. In this scenario, the system queries `TrackQLearned` for a suggestion as to which is the next `Destination` to travel to next.

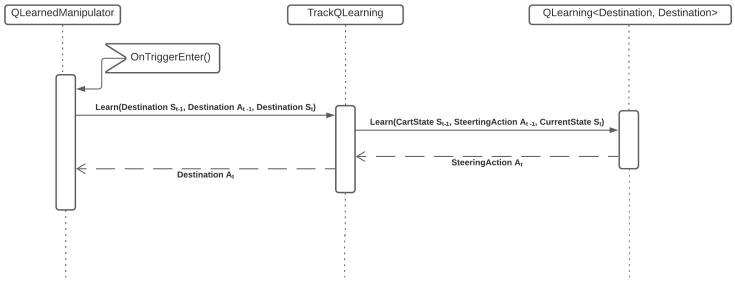

Figure 30: UML: `QLearnedManipulator.OnTriggerEnter(Collider)` Sequence Diagram

7 Implementation Models

Our implementation of Q-Learning has three hyper-parameters with the names *alpha*, *gamma*, and *rho*. They were explained to some degree earlier, but here we describe them with more detail here.

Learning Rate (α) Controls how much tranference occurs between the current Q-Value has over the stored Q-Value. It's in the range of $[0, 1]$. Thus, a value of 0 gives birth to an algorithm with no learning, and a value of 1 gives no credit to earlier experiences. However, this parameter benefits with the passage of time. Initially, it can start with a relatively high value, and then gradually be reduced until it reaches a value below normal. In this manner we let learning quickly change the underlying Q-values, even if we hace little information stored. This helps us to protect the learning earned. In our project, this value starts at 0.3[1], and is decreased to 0 once training has achieved favorable results.

Discout Rate (γ) Controls how much an action's Q-Value depends on the state it leads to. It's in the range of $[0, 1]$. A value of 0 means that every action will earn a Q-Value solely on the reward directly earned for arriving at that new state. A value of 1 evaluates the state of the reward of the curent action, and makes it equal to the Q-Value to the state the action leads to. Lower values stabilize faster, but admit shorter sequences. However, higher values help us procure longer action sequences but learning takes more time. For this project γ is set to a value of 0.75[1].

Exploration Parameter (ρ) Helps us control the frequency with which the algorithm will perform a random action as opposed to a greedy action. It's values are between $[0, 1]$. If we use a value of 1, then the algorithm will never use the acquired learning, and will always take actions at random. In our project we use a value of 0.2 during training and then, once training has reached acceptable values, we switch to a value of 0, turning agents into totally greedy entities.

8 Results

8.1 Empirical Results

One of the goals we sought with this project is to prove empirically that Q-Learning can be adapted under an AI Overlay layer to make fine-grained decisions for a game AI. And we set out to do just that.

In the Figure 31 we can visualize the difference between the driving agents using reinforcement learning. The average collisions per second without learning is around 16, while it decreased drastically to around 5 and 1 en the first 200[*seconds*] of gameplay using Q-Learning with sub-sampling and super-sampling, respectively. In this maner, we show the effectiveness of our technology with modern techniques of reinforcement learning in free-range, autonomous driving agents.

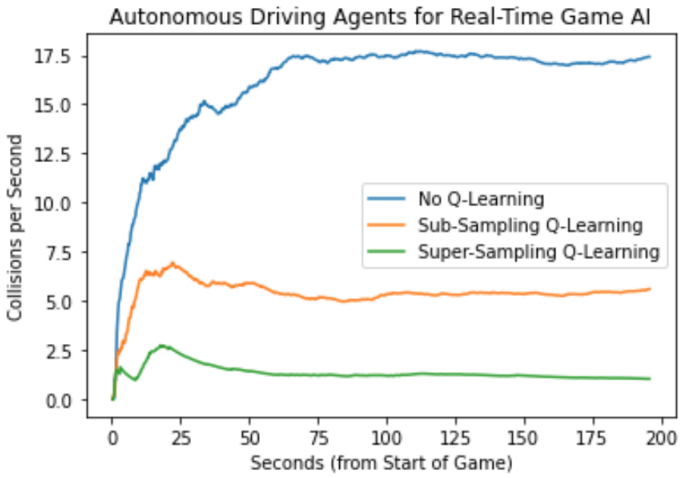

Figure 31: Difference in performance between no Q-Learning, sub-sampling, and super-sampling.

8.2 The Game

There's a handful of possible games available to us through this game mechanic of the moving hive.

8.2.1 A Game for Everyone

If we give the player a limited amount of time to defeat the computer, will he be able to do so? That's a game that everybody can enjoy. So, a goal can be:

> ***Can I beat all the machine's scores in under*** 5 $[min]$***.***

This goal turns out not to be so simple. Further testing is needed to see whether players enjoy this match. I can defeat the computer in a couple of minutes. But I've practiced endlessly. Might not come as easy to others.

Of course, it's all in the art of balancing. Right now we're rewarding players 3 points for reaching a lap, punishing players by 2 points by colliding against an obstacle, and we're always taking away a trickle of reward of every moment that passes. Combinations of these reward and punishment mechanics could lead to games that are much harder in essense. For example, since the machine is so good at dodging obstacles, taking 4 points as a punishment for colliding and giving 2 points of reward for reaching a lap makes this game into an impossible challenge. Again, further testing is needed.

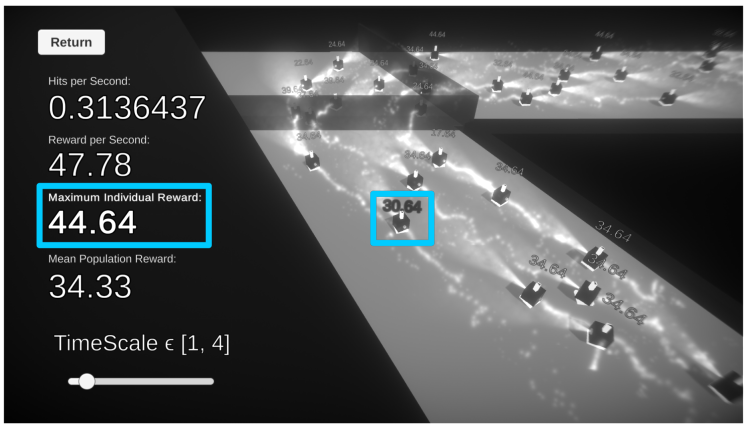

Figure 32: A shot in the middle of the race game. As you can see, we haven't defeated the computer in this run. Yet.

A preliminary play-test shows that the AI is indeed capable of surprising players and pose a formidable challenge.[11] Like all things in game design, there are patterns that can be discerned on how to beat the machine, but these are few and happen only occasionally. In general the player must be at its wits end to constantly outmaneuver the machine in tight scenarios. It's not an easy game to play. The decisions that machine has learned are *not* aggressive. Rather, some agents will gladly make room for the player to pass. But, *as a collective* the entire hive of agents play better, overall, than the player. In the case of this play-test the mean reward procured by all agents was consistently higher than it was for the player. That is, on average, the agents were earning more points than the player, as a whole.

[11] The test was carried as an experiment on *Facebook* and *WhatsApp*. Anyone who wanted to download and play the game was welcome to do so

Current trend shows that very avid players are able to defeat the puzzle in 1 attempt. But most players need anywhere from 3 to 8 attempts (one of them needed 20) to defeat the puzzle. I consider that a success. The surveys that we've done show that players rate the game a 6.2/7, which isn't a bad score at all, for a game made with boxes and a few basic elements. So far 25 players have filled the surveys.

The following big "iterations" were made to the game according to the feedback in the surveys:

- **Added another AI Overlay called *greedy-lazy* to nullify a dominant strategy.**
- **Changed the game feel of the controllers so players have a more raw, direct access to their inputs.**
- **Added an interface through mouse.**

8.2.2 A Game for Game Designers

There are deeper themes emerging than originally expected. For example, let's suppose the player is given more power in current the puzzle: he's faster and larger than most carts. But this leads towards a very bad performance outcome for the player, as it's difficult to beat the machine when they are at an edge. The player will find it impossible to raise the score higher than the machine. This is might be the ultimate purpose of the experience. For the player to realize this truth.

Unless, of course, the player decides to play by her own rules, and seek to accumulate to *least* amount of points: seeking to be to cart with the worst score. She will find this is really easy. Which has morals implications: nobody is punishing you for doing something bad, while we punished the agents to get them towards their current winning behavior.

A question then arises to the player: can you find your ultimate winning behavior in this scenario?

And the answer:

To maximimze reward for all involved.

This could be called the *resonant theme* of this experience.[3].

In the end—while crafting all the appropriate graphical elements to support this experience fully is not possile at the moment—the current elements in so far implemented give a pretty accurate experience of being cast out and, ultimately, isolated. For being *different*. The aim is for the player to find herself inventing some goal to wreathe into game, while in truth no goal is given. Just some scores hovering above the carts and from there she *might* discover the underlying theme. At which point the game becomes boring and the puzzle is solved.

The puzzle being:

What's the theme of this game?

Multiple answers, are of course possible, and the player might be contented with her own answer.

9 Impact

We believe that our contribution to the body of cientific knowledge can be sumarized in a game design technique we designed to develop the game for this project: *The Hive Mind and the Originality Kernel*. The abstract of the emerging paper goes like this:

> *In the literature, the use of reinforcement learning to train artificial agents to play a game is pervasive. However, there's less prevalence on using these techniques to craft engaging experiences for players. Here, we showcase a new game design technique, the Hive-Mind and the Originality Kernel, which can be used to craft the decision-making process of artificial agents in a multi-agent, hive-style environment. The Hive-Mind is the collective decision-making process shared by all agents. It is controlled by a centralized Q-Learning algorithm. The Originality Kernel is a value that represents some variable aspect of agent sub-classes in the hive. By using an ad-hoc AI Overlay to manipulate, in real-time, the Originality Kernel, and by making Q-Learning's hyper-parameters be dependent on this Kernel, all agents can share a degree of learning while providing the means to grant different behavioral flavors to each agent sub-class. The authors implemented a single-player racing mini-game using this technique. It featured autonomous driving agents with two different sub-classes: greedy and lazy. Play-testing sessions were carried where N players participated and were evaluated with the GEQ Core testing module. Favorable scores were attained for all the test's components, validating the approach.*

10 Conclusion

10.1 The Promise of Reinforcement Learning

The promise wielded by reinforcement learning is fascinating.

For an agent to be capable of discovering how the world works simply by observing things and noticing what happens, is a conceptual leap in the process of building AI. After all, this is how we think animals and people do to learn.

The critical idea is to simulate this on a computer. Carrying out this form of learning turns out to be useful in almost any application, and as a side-effect, we can learn about how our minds work.

In reinforcement learning, we focus on the problem at hand while we interact with a world that's in constant change. We don't simply wait for our agents to calculate "good behavior" and then execute it. We instead expect that our agents do things *wrong* at first, and then, slowly but surely, perfect their comprehension about how good or how bad their actions were and, in the case of online learning, this comprehension happens on the go, as the agents are performing their sought-after goals.

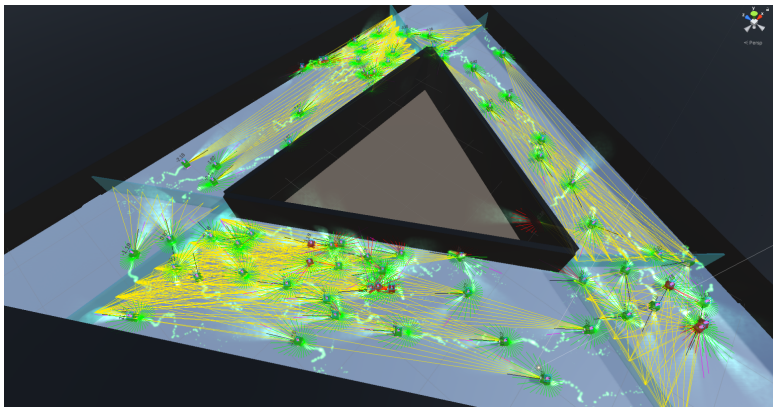

Figure 33: A preview of all the carts sampling the information with their rays.

In this view, the agent's world is suddenly not so static.

This means that our agents find themselves constantly in new situations and their goals, in the short-term, change, and evolve. The agent thus integrates, immediately, their most recent experience to take the most viable course of action on the next faced scenario. The proposal of *online* learning is extremely powerful, and it's one of the featuring characteristics of certain kinds of reinforcement learning. It's challenging to sponsor such a method, it requires a new perspective.

Video-games are excellent testbeds for trying out artificial intelligence methods. However, to be able to test the capabilities of an agent in a video-game, it's necessary to test it in scenarios for which the agent has not been optimized. For this, we need to be able to create new games or mechanics manually or automatically to test the particular capacities of the artificial agent in question. This could be fundamental to evaluate how the algorithm can learn and evolve to comprehend many changing environments.

In video-games—whether they are created for entertainment, simulation, or education—the variety of virtual worlds that we can forge for our artificial agents is limited only by our imagination, which offers us grand opportunities to test problems relevant to automated learning. The presented environment can be constructed with variable, deterministic, and discrete characteristics.

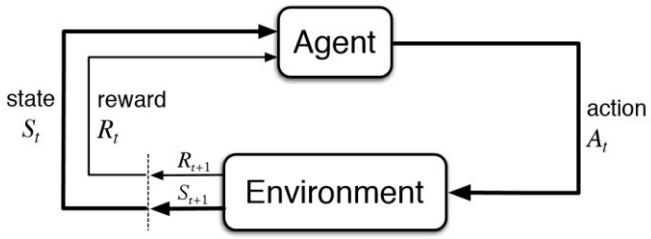

Figure 34: A chart depicting the essential process of reinforcement learning.

One of the issues is in making sure that an agent has enough experience to learn from a high-performance policy without wreaking havoc on its environment, other agents, or itself. This is a less serious issue in video-games than it is in the real world, for the environment is controlled, digital, and impermanent. An interesting question arises with the use of reinforcement learning in MMORPGs, where and agent *could* wreak havoc in the game's virtual world. These considerations are beyond the scope of this project, however.

We developed this series of interactive experiences and games in order to test the performance of the reinforce learning in free-range, autonomous driving agents for video-games in real-time. For the most part, early prototypes cast positive results: as agents interact with the environment, the rate of learning improves, as does the stability of the process of learning. The design helps agents take better decisions over time and, thus, improve their performance. The games engendered with it seem to be fun and playable and, in the words of our players, "no two deathmatches are alike." It seems we've come full circle to the original problem of weaving a more intelligent problem space for the player, and the testing done for the game supports our conclusion that we have, in fact, created more interesting AI for players to compete against and absorb.

References

[1] Ian Millington. *AI for Games*. CRC Press, 3rd Edition; March 2019.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Bradford Books, 2nd Edition; November 2018.

[3] Jesse Schell. *The Art of Game Design*. AK Pters/CRC Press, 3rd Edition; August 2019.

[4] Raph Koster. *A Theory of Fun*. O'Reilly Media, 2nd Edition; December 2013.

[5] Jeannie Novak. *Game Development Essentials: An Introduction*. Cengage Learning, 3rd Edition; August 2017.

[6] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial Modern Classics, 1st Edition; July 2008.

[7] Katie Salen Tekinbas and Eric Zimmerman. *Rules of Play: Game Design Fundamentals*. The MIT Press; September 2003.

[8] Edward O. Wilson. *On Human Nature*. Hardvard University Press, 2nd Edition; October 2004.

[9] Yuval Hoah Harari. *Sapiens: A Brief History of Mankind*. Harper; February 2015.

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition; November 1994.

[11] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 1st Edition; November 2014.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Daan Wierstra, Martin Riedmiller and Ioannis Antonoglou. *Playing Atari with Deep Reinforcement Learning*. DeepMind Tehcnilogies, `https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf`.

[13] Harshit Sethy, Amit Patel and Vineet Padmanabhan. *Real Time Strategy Games: A Reinforcement Learning Approach*. Eleventh International Multi-Conference on Information Processing-2015 (IMCIP-2015).

[14] M. Bowling, J. Fürnkranz, T. Graepel, and R. Musick, *"Machine learning and games"*. vol. 63, no. 3, pp. 211–215, May 2006, doi: 10.1007/s10994-006-8919-x.

[15] C. Jennett, A. L. Cox, P. Cairns, S. Dhoparee, A. Epps, T. Tijs, and A. Walton. *Measuring and defining the experience of immersion in games, International Journal of Human-Computer Studies 66 (9) (2008) 641 – 661*. DOI: `http://dx.doi.org/10.1016/j.ijhcs.2008.04.004.1010`. URL: `http://www.sciencedirect.com/science/article/pii/S1071581908000499`.