



ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

Тема: Оптимизация на светофари в симулация на трафик

Дипломант:

Борис Веселинов Ханджиеев

Дипломен ръководител:

Иван Кръстев

СОФИЯ

2025



ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Дата на заданието: 28.10.2024 г.

Утвърждавам:.....

Дата на предаване: 28.01.2025 г.

/проф. д-р инж. П. Яков/

ЗАДАНИЕ за дипломна работа

ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ

по професия код 481020 „Системен програмист“

специалност код 4810201 „Системно програмиране“

на ученика Борис Веселинов Ханджиев от 12 В клас

1. Тема: Оптимизация на светофари в симулация на трафик

2. Изисквания:

- Да се разработи симулация на трафик с автомобили и светофари;
- Поведението на автомобилите да имитира това в реалния свят;
- Да може да се оптимизират светофарите чрез алгоритъм за машинно обучение;
- Да има елементарен потребителски интерфейс;
- Да има алгоритъм за генериране на случайна пътна мрежа;
- Да може да се събират данни от симулацията.

3. Съдържание 3.1 Теоретична част
3.2 Практическа част
3.3 Приложение

Дипломант :.....

/ Борис Ханджиев /

Ръководител:.....

/ Иван Кръстев /

ВРИД Директор:.....

/ ст. пр. д-р Веселка Христова /



ТЕХНОЛОГИЧНО УЧИЛИЩЕ "ЕЛЕКТРОННИ СИСТЕМИ"
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

СТАНОВИЩЕ
КЪМ ДИПЛОМНА РАБОТА

Тема на дипломната работа: Оптимизация на светофари в симулация на трафик

Ученник: Борис Веселинов Ханджиев

Клас: 12 В

Професия: код 481020 „Системен програмист“

Специалност: код 4810201 „Системно програмиране“

Дипломен ръководител: Иван Кръстев Кръстев

Дипломантът успешно е реализирал целите, поставени в своята дипломна работа.

Разработена е симулация на трафик, която включва автомобили и светофари, като поведението на автомобилите реалистично отразява условията в реалния свят. В хода на работата беше и внедрен алгоритъм за машинно обучение, който оптимизира работата на светофарите с цел подобряване на трафика и намаляване на задръстванията. Също така беше и реализиран алгоритъм за генериране на случайна пътна мрежа, което допринася за приложимостта на разработената система. Системата също включва и елементарен потребителски интерфейс, който позволява лесно управление и наблюдение на симулацията.

Процесът на работа протече организирано, с регулярни срещи и с последователен подход към решаването на задачите. Дипломантът демонстрира задълбочени познания по програмиране, самостоятелност в работата си, както и способност за критично мислене и анализиране на проблеми. При възникване на затруднения той сам проявява инициативност в намирането на решения.

Предлагам за рецензент:

Марин Петров Петров

Квалификация: Senior software developer

Имейл: mpetrov@nemetschek.bg

Телефон: 0889191163

Всичко изложено дотук дава основание ученикът Борис Ханджиев да бъде допуснат до защита пред комисията за провеждане на държавен изпит за придобиване на професионална квалификация по теория и практика на специалността и комисията да оцени дипломната работа отлично.

28.01.2025 г.

Дипломен ръководител: 

Увод

Най-големият недостатък на светофарните системи е, че те често работят по статични графици, което ги прави неспособни към ефективното адаптиране към промените на трафика. Този проблем, заедно с разрастването на градовете и нарастващия обем на превозни средства по съществуващата инфраструктура, води до множество проблеми, като: повишени вредни емисии от автомобилите, по-дълго време за пътуване до работни места и домове и повишенна раздразненост у водачите, което е опасно, както и за тях, така и за останалите водачи на пътя.

Светофарните системи спадат към понятието за критични инфраструктури^[1]. Това са системи, или инфраструктура, чиито неизправности или спирания биха донесли значителни опасения за безопасността и здравето на населението и околната среда. В случая на механизмите, които управляват светлинните сигнали, тези опасения са катастрофите и евентуалната загуба на живот.

Поради тази причина, светофарните алгоритми и контролери винаги се изprobват в симулирана среда преди тяхното внедряване в истинския живот. Освен безопасност, също може да се анализират различни показатели, като цялостна ефикасност, отделени емисии, цени и т.н. Още едно предимство на експериментирането в симулационна среда, е възможността за нагласяването на отделни параметри, като гъстота на задръстването, различни ограничения на скоростта, различен брой ленти, час в деновонощието и др.

Тази дипломна работа реализира примитивна апликация, която симулира трафик и подпомага оптимизирането на светофарите на кръстовищата чрез алгоритъм за машинно обучение.

Първа глава

Проучване на технологии и съществуващи решения

1.1 Технологии за създаване на симулация на трафик

1.1.a Игрови двигатели

Гейм ендженни като Unreal Engine, Unity и Godot, които са едни от най-известните на пазара, предлагат мощни възможности за 2D/3D визуализация и физични симулации, което ги прави подходящи за създаване на сравнително точна трафик симулация. С употребата на игрови двигатели също се премахва нуждата за огромно количество „boilerplate“ код, което позволява бързото създаване на прототипи и разработване на софтуер.

1.1.b Програмни езици и библиотеки от ниско ниво

За по-леки, бързи и контролиращи, спрямо изискванията на разработчика, симулации, бързи езици като C/C++ и библиотеки като raylib или SDL (Simple DirectMedia Layer) са идеалния избор. Те предоставят опростен и сравнително лесен за използване интерфейс за програмиране на видеоигри. Минусите на този подход са плюсовете на игровите двигатели, а именно, че прототипирането е много по-бавно и „boilerplate“ кода е повече, а също и графичните и физичните възможности са много по-ограничени.

1.2 Съществуващи симулации на трафик

1.2.a SUMO

SUMO (Simulation of Urban MObility)^[2] е софтуерен пакет с отворен код, който може микроскопично да симулира големи мрежи от пътища. SUMO е разработен от Германския аерокосмически център, както и от потребители от общността и е написан на C++, Java и Python. Той е достъпен

като СОК от 2001 г., а от 2017 г. е проект на фондацията Eclipse. SUMO позволява използването на различни софтуерни инструменти за настройване на параметри и анализиране на пътния трафик и се използва за изследователски цели, като: прогнозиране на трафика, оценка за ефективността на светофарите и избор на маршрути.

SUMO може да симулира различни автомобили, обществен транспорт и пешеходци и идва с множество средства за създаването на уникални трафик сценарии. Понеже SUMO е по-лек и по-елементарен софтуер в сравнение със следващия (Vissim), той е по-подходящ и се използва по-често за мащабни, ориентирани към данни, изследователски цели. Независимо от това възможността за интегриране с външни софтуерни инструменти, факта, че SUMO е отворен код и цялостната гъвкавост на софтуера, поставят SUMO на едно от членните места, при избиране на софтуер за трафик симулация.

1.2.b PTV Vissim

PTV Vissim^[3] е многомодулен софтуерен пакет за микроскопично симулиране на трафиков поток. Той е разработен от PTV Planung Transport Verkehr AG в Карлсруе, Германия през 1992 г. Името му е абревиатура на "Verkehr In Städten - SIMulationsmodell" (На английски - "Traffic in cities - simulation model"). Приложенията на PTV Vissim варират от пътно инженерство, обществен транспорт, противопожарни защити (симулации на евакуации) до 3D визуализация за илюстративна цел.

В Vissim може да се симулират различни участници в движението, като: коли, автобуси, камиони, рейсове, велосипеди, мотоциклети, пешеходци, каруци и др., както и различна обществена инфраструктура, като: паркинги, летища, гари и големи сгради. В сравнение с предишния софтуер (SUMO), Vissim се използва повече за детайлно планиране на трафик в истинския живот, тъй като Vissim е с по-голяма прецизност и

физична акуратност от SUMO. Широкият му спектър от трафик агенти, заедно със възможността за интеграция с други PTV продукти като PTV Visum (водещ световен софтуер за планиране на трафик), превръщат Vissim в индустриален стандарт.

1.2.c Aimsun Next

Aimsun Next^[4] е мощен многомащабен софтуерен пакет за симулация на трафик, който може да работи на микроскопично, мезоскопично и макроскопично ниво. Разработен е от компанията Aimsun, част от Yunex Traffic, и е използван от инженери, градски плановици и изследователи по целия свят. Благодарение на способността си да моделира сложни транспортни мрежи, Aimsun Next е широко приложим в области като интелигентни транспортни системи (ITS), управление на трафика и анализ на въздействието на инфраструктурни проекти.

Aimsun Next може да симулира различни участници в движението, включително автомобили, обществен транспорт, пешеходци и велосипедисти, както и специфични сценарии като автономни превозни средства и споделена мобилност. В сравнение със SUMO и PTV Vissim, Aimsun Next предлага гъвкавост при избора на мащаб на симулацията, което го прави подходящ както за детайлно моделиране на отделни кръстовища, така и за мащабни транспортни стратегии на градско и регионално ниво. Възможността за интеграция с други транспортни модели и данни в реално време утвърждава Aimsun Next като водещ инструмент за анализ и оптимизация на транспортните системи.

1.3 Съществуващи методи за оптимизиране на трафик

В реалния свят ефективното оптимизиране на трафика е възможно само с устройства, които позволяват комуникацията Vehicle-to-Everything (V2X), което позволява на превозни средства, светофари и инфраструктура да обменят данни в реално време.

Методи за проследяване на трафик като сензори с индуктивна верига, радар, LiDAR, камери и проследяване, базирано на GPS, предоставят важна информация за потока на превозните средства и нивата на задръстванията. Комуникационни методи като Dedicated Short-Range Communication (DSRC), Cellular V2X (C-V2X) и 5G позволяват на свързаните превозни средства да взаимодействат със системите за трафик за по-добра координация.

В контекста на симулационна среда обаче, не е нужно да се тревожим за комуникационни предизвикателства в реалния свят – позицията, скоростта и намерението на всяко превозно средство вече са известни, което улеснява прилагането и тестването на стратегии за оптимизиране на трафика. Това, което следва са някои от методите за ефективно управление на светофари:

1.3.а Сигнали фиксиранi във времето

Контролът с фиксирано време е метод за оптимизиране на светофара, при който продължителността на зелената, жълтата¹ и червената светлина е предварително програмирана въз основа на фиксирани времетраения или исторически данни за трафика. Циклите на сигнала се изпълняват по повтарящ се график, независимо от действителните условия на трафика, което го прави ефективен в предвидими, стабилни среди на трафика, но неефективен при променливи количества автомобили.

¹ В закона в различните държави продължителността на жълтата светлина обикновено е фиксирана, като тя често е пропорционална на максималната разрешена за пътя скорост.

Разновидност и подобрение на този метод са предварително зададени пътни сигнали, които използват различни планове спрямо часа в деновонощието. Примери за такива планове са: план за час пик (17:00 – 18:00), нощен план (12:00 – 6:00) и др. Въпреки че все още не отговаря на трафика в реално време, този подход подобрява ефективността чрез адаптиране към очакваните трафикови обеми през цялото деновонощие.

Текущо това е оспоримо най-разпространеният метод по света, поради леснотата на имплементация и ниската цена. Все повече обаче се наблюдава преминаването към по-адаптивни методи за управление на светофарите, като тези изброени по долу.

1.3.b Задействане на сигнали

Метод на задействане на сигналите е подход за оптимизиране, който настройва времената на сигнала въз основа на наличието на трафик в реално време. В случая на обикновено светофарно кръстовище - системата открива присъствието на ППС-та и съответно адаптира продължителността на зеления сигнал. В реална обстановка, както бе посочено горе, това става възможно с различни сензори, но в симулация – всички тези данни вече се знаят.

Този метод позволява по-гъвкав и по-отзовчив контрол в сравнение със системите с фиксирано време, като намалява ненужното време на чакане и подобрява трафика, особено в периоди на слабо натоварване.

Разновидност на задействаните сигнали са полузадействани пътни сигнали, където се съчетават този и горния метод. Пример е, когато главен път работи с фиксиран график, а пък страничните улици, работят на принципа на задействане, тогава, когато се появи трафик по страничните улици, те се изчистват възможно най-рано, като това осигурява минимално време за чакане.

1.3.c Адаптиране на сигнали

Адаптивният контрол е последният метод, който ще бъде разгледан и той представлява динамично настройване на времената на сигналите в реално време въз основа на данни за непрекъснат трафик.

Използвайки усъвършенствани алгоритми, системата непрекъснато следи условията на трафика и коригира продължителността на различните светлини, за да оптимизира трафика. Този метод е много ефективен при справяне с непредсказуеми вълни от трафик, внезапни скокове или часове извън пиковите натоварвания.

Разновидност на адаптивния контрол е централизиран адаптивен контрол, където данните от множество кръстовища в мрежа се събират и обработват централно, което позволява координирани корекции в поширока област за допълнително оптимизиране на потока на трафика и за получаването на явления като зелени вълни¹.

1.4 Техники за създаване на адаптивни трафик системи

1.4.a Прости алгоритми базирани на правила

Простите алгоритми, базирани на правила, представляват детерминирани методи за управление на светофарите, които разчитат на предварително зададени логически правила и прагови стойности. Те не използват машинно обучение, а директно боравят с предварително дефинирани критерии, като: дължината на опашката от автомобили, времето на чакане, потоци от автомобили или исторически данни за трафика.

¹ Явление, при което няколко светофари координирано пропускат една непрекъсната вълна от трафик през няколко кръстовища в една главна посока

Примери за такива алгоритми са Webster's method^[5], SCOOT (Split Cycle Offset Optimization Technique)^[6] или SCATS^[7], които анализират и правят корекции в реално време.

Предимствата им са, че те са прости, детерминистични и предсказуеми, с по-малка изчислителна сложност и евтини за интегриране. Недостатъците са като цяло плюсовете на метода с машинно обучение – скалируемост и адаптация.

1.4.b Машинно обучение

Различните категории на машинно обучение включват: Supervised Learning, Deep Learning, Deep Reinforcement Learning (deep RL) и др. Едни от най-популярните библиотеки за обобщено машинно обучение са Tensorflow и PyTorch, а пък за RL са OpenAI Gym^[8], Learning Agents в контекста на Unreal Engine и ML-Agents в контекста на Unity.

Идеята е, че чрез трафик данни, използвайки една от горепосочените библиотеки, може да се тренира модел, който да се научи да се адаптира към състоянието на пътната обстановка и да превключва светофарните сигнали по ефикасен начин.

Предимството на ИИ-базиран подход за адаптивните трафикови системи спрямо стандартни алгоритми е, че алгоритъмът с машинно обучение е значително мащабирам – възможност е приложението му за цели градове, а не само за единични кръстовища. Също, има потенциал за по-добра крайна оптимизация на светофарите – особено измежду няколко кръстовища и пътни коридори.

Втора глава

Изисквания, избор на средства, преглед на проекта

2.1 Изисквания към софтуерния продукт

2.1.a Изисквания от заданието на дипломната работа

От заданието на дипломната работа (2 стр.), това са някои от задължителните изисквания:

- Да се разработи симулация на трафик с автомобили и светофари;
- Поведението на автомобилите да имитира това в реалния свят;
- Да може да се оптимизират светофарите чрез алгоритъм за машинно обучение;
- Да има елементарен потребителски интерфейс;
- Да има алгоритъм за генериране на случайна пътна мрежа;
- Да може да се събират данни от симулацията.

2.1.b Допълнителни изисквания

Освен гореизброените изисквания, тези долу са изисквания, които би било хубаво да има в този проект, понеже те обогатяват възможните действия, преживяването на потребителя или улесняват някои задачи:

- Да има система за запазване/зареждане на симулации чрез файлове;
- Да има настройки и параметри на симулацията, които да открият по-голям брой възможности на симулацията;
- Да има прости настойки за контролиране на времето: паузиране и забързване;
- Да има възможност за селектиране на автомобили или кръстовища, където да се визуализират пътя на автомобила и допълнителни данни.

2.2 Технологии

2.1.a Игрови двигател – Unity

За дипломната работа и разработката на трафик симулация е избран гейм ендженът Unity. Той е мощен и широко използван гейм енджен предимно за разработката на игри, но напълно е приложим за филми, анимации, симулации, научни цели и т.н. Той предлага гъвкавост и богата екосистема от инструменти за работа с 3D и 2D среди. Някои от чертите на Unity, правещи го добър избор, са следните: кросплатформена поддръжка, C# като скриптов език, който е лесен за използване и има гигантска колекция от функционалности в .NET фреймуърка, Universal RP, Unity Asset Store, вградена физика, надграждаща PhysX^[9] на NVIDIA и много други.

Причините за избиране на Unity пред Unreal Engine са следните:

- Unreal Engine има по-малък брой asset-и от Unity, заема по-малък процент от пазара и използва C++, който е по-сложен и по-времеемък за програмиране.
- Unreal Engine всъщност набляга повече на графичните си способности – това го прави по-употребяван специфично за CGI цели и за AAA игри за разлика от Unity. Това не е проблем обаче, тъй като една симулация за трафик не се нуждае от тежки графики.

Също причината, поради която не е избран Godot, е че на него му липсват повече asset-и, ресурси онлайн и интеграция с различни софтуери.

Последно, избора на Unity пред език от ниско ниво, като C++ и библиотеки като raylib и SDL, може да се аргументира лесно с факта, че много аспекти би трябвало да се имплементира почти от нулата и работата за физика, рендъринг и потребителски интерфейс би изисквала много повече време.

2.2.b Файлова структура на обикновен Unity проект

Всеки Unity проект използва специфична структура, която се състои от няколко основни директории и файлове. Тази структура е важна за правилната организация на ресурси, скриптове и настройки. Някои от директориите са:

- **Assets директория:**

Това е най-важната папка, където се съхраняват всички ресурси (asset-и) на проекта. Всичко, което се намира тук, се вижда в Unity Editor. Честа конвенция е Assets директорията да се раздели на главни поддиректории, както следва:

- Scenes – съдържа сцените на проекта (файлове с разширение .unity).
- Scripts – за всякакви C# скриптове.
- Materials – съдържа материалите за обекти и текстури.
- Prefabs – готови за използване игрови обекти.
- Animations – анимационни клипове и контролери.
- Audio – звукови файлове и ефекти.
- Shaders – графични шейдъри за визуални ефекти.
- Resources – съдържа файлове, които могат да бъдат манипулирани по време на изпълнение чрез Unity класа Resources.

- **Library директория:**

Тази папка се използва от Unity за кеширане на данни и компилирани файлове. Тя не трябва да се променя ръчно, защото Unity автоматично я обновява. Това означава, че в системи за контрол на версии (VCS - например Git, SVN) тя обикновено се пропуска, подобно на node_modules при използване на npm или venv при виртуални среди с Python.

- ProjectSettings директория:

Тази директория е предназначена за всички настройки на проекта и плъгини, които се отнасят за рендъринг, аудио, входяща информация от външни устройства, като мишки и клавиатури, тагове, слоеве и др. Съответно често срещани файлове тук са:

- AudioManager.asset
- EditorSettings.asset
- ProjectSettings.asset
- QualitySettings.asset
- TagManager.asset
- InputManager.asset

- Packages директория:

Съдържа списък с инсталираните Unity пакети в файлове като manifest.json и packages-lock.json, а примери за често употребявани пакети са Physics, ML-Agents, Terrain, MemoryProfiler и други.



Фиг 2.1. Примерна файлова структура на един прост Unity проект:

Отляво – основната директория на проекта, както е през File Explorer
Отдясно – Изгледът на Assets директория през Unity Editor-a.

2.2.c Въведение към Unity – понятия и концепции

Unity е мощен и широко използван игрови енджен, но за да може да се използва в най-пълните му възможности, трябва първо да се разбират някои основни термини и концепции за това как работи.

В Unity всеки обект в сцената е GameObject, който може да съдържа различни компоненти, като класът Component представлява базов клас за всичко, което може да се закачи за даден обект. Примерни компоненти са:

- Transform – запазва позицията, ротацията и машаб на обекта. Той винаги присъства в GameObject-ите. Не е възможно да се премахне от обект или пък да се създаде обект без него.
- MeshRenderer, който рендерира модел на обект – всички обекти, които могат да се видят в сцената вероятно имат такъв компонент.
- Collider, което е базов клас за всички обекти, които трябва да имат възможността да засичат колизии с други колайдери.
- Scripts – Това е основното средство, с което разработчик работи. Скриптовете също могат да се закачат за обекти, понеже наследяват MonoBehaviour класа, който наследява Behaviour, а той от своя страна наследява Component.

Всеки скрипт изглежда по подобен на следния начин:

```
public class Script : MonoBehaviour
{
    void Awake()
    {
        Debug.Log("Script loaded");
    }

    void Start()
    {
        Debug.Log("First frame");
    }

    void Update()
    {
        Debug.Log($"Time since last frame: {Time.deltaTime}");
    }
}
```

Фиг 2.2.

Най-важните методи са Awake, който се вика при зареждането на скрипт – независимо дали е активиран или не, Start, който се вика преди първото извикване на Update само ако скриптът е активиран и Update, който се вика всеки кадър. Като цяло редът на изпълнение на функциите на един MonoBehaviour е:

- Awake()
 - OnEnable()
 - Start()
 - FixedUpdate()
 - OnTrigger...
 - OnCollision...
 - Update()
 - LateUpdate()
 - OnApplicationQuit()
 - OnDisable()
 - OnDestroy()
-
- Изпълняват се един път в началото на жизнения цикъл на скрипта.*
- Може да се изпълнят по няколко пъти.*
- Изпълняват се един път в края живота на скрипта.*

Някои често използвани класове са: Vector2 и Vector3 – за обикновена математика с позиции в пространството и вектори, Quaternion – за работа с ротации, Time – за движение, което не зависи от броят кадри в секунда (FPS) и ускоряване на скоростта на времето, Mathf – за всякакви математически функции, Random, Debug и т.н.

За манипулиране на обектите в една сцена най-много се използват Instantiate() – за създаване на GameObject от prefab, Destroy() – за изтриване на обект, GetComponent<>() – за намиране на компонент в даден обект, FindObjectsOfType() и FindGameObjectsWithTag() – за намиране на обекти и т.н.

2.2.d Алгоритъм за оптимизиране на трафика - самообучение с утвърждение

За оптимизацията на светофари, дипломната работа използва алгоритъм с самообучение с утвърждение. Самообучение с утвърждение (на английски: reinforcement learning) е подходящ за оптимизация на трафика, защото позволява на алгоритмите да се учат чрез проби и грешки, като постепенно подобряват управлението на светофарите. RL може да вземе предвид множество фактори като дължина на опашките, средно време на чакане и натовареност на различни пътни участъци.

Основните предимства на RL спрямо традиционните алгоритми включват:

- Този метод на трениране може да се адаптира към динамични и непредсказуеми, както и различни от обичайното, трафикови условия, особено чрез експериментация и достатъчно време за трениране;
- Възможно е автоматичното откриване на най-оптималните за ситуацията стратегии, това е голямо предимство пред адаптивни трафик механизми с конкретни зададени правила и прагови стойности;
- Подходите базирани на машинно обучение често са скалируеми, в конкретния случай на трафик симулация, този алгоритъм може да се транслира към съседни кръстовища, региони и градове.

2.2.e Въведение към самообучение с утвърждение

RL позволява на агенти, като това са единиците, които се тренират (трафик контролери), да научат стратегии и действия чрез взаимодействие с дадена среда (трафик симулацията) и да получават награда или наказание в резултат на своите действия. Вместо да се предоставят директни етикети или примери за правилните действия (dataset), както при традиционното

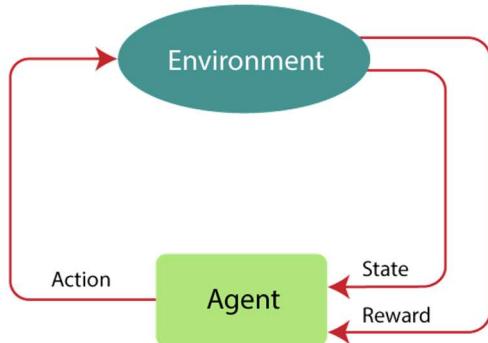
машинно обучение, агентът в RL сам открива какви действия водят до максимални награди, като се адаптира към променящите се условия в средата. В процеса на самообучение агентът използва серия от взаимодействия със средата (environment) – често наричани епизоди (episodes), за да взима решения и да подобрява своето поведение с течение на времето.

В дипломния проект съответно:

- Средата (на английски Environment) е самата симулация - пътната мрежа и автомобилите.
- Състоянието (на английски State) е информация за трафика в даден момент, която се подава на агента, като: опашки на кръстовище, конфигурацията на светофарите и т.н.
- Агент (Agent) е всеки трафик контролер, като крайната цел е той да оптимизира светофарните сигнали и съответно трафика.
- Действието (Action) е решението, което агентът предприема в даден момент, като това решение променя средата.
- Наградата (на английски Reward) е количествена стойност, която агентът получава след изпълнението на действие. Наградата може да бъде положителна, когато агентът предприеме успешно действие, или отрицателна, ако резултатът е нежелан.

Принципът на самообучението с утвърждение се състои в това агентът да учи какви действия водят до най-добрая дългосрочен резултат, като оптимизира стойността на получаваните награди през времето.

Диаграма, показваща цикъла на алгоритъма за самообучение с утвърждение може да се види долу:



Фиг 2.3. Диаграма описваща взаимоотношението между агента и средата в самообучение с утвърждение.

2.2.f Библиотека за самообучение с утвърждение - ML-Agents

За целите на самообучение с утвърждение е избран ML-Agents. ML-Agents е Unity проект^[10], който улеснява трениране чрез Deep Reinforcement Learning (deep RL) на модели в симулации вървящи в гейм енджина. Сам по себе си ML-Agents не извършва тренирането, а питонски бакенд, който използва TensorFlow или PyTorch, като за deep RL има различни методи на трениране, като PPO^[11], DQN^[11] и други.

ML-Agents е първият кандидат, тъй като е най-популярен в Unity екосистемата и също има много документация и ресурси онлайн. Пакетът, който може да се свали в игровия двигател, предоставя кодов интерфейс за лесно взаимодействие с трениращия процес.

2.2.g Въведение към ML-Agents

Процесът на създаване на агент с ML-Agents е възможно с наследяване на Agent класа, който е предоставен от ML-Agents пакета. Всеки агентов скрипт има структура подобна на следния пример:

Agent класът открива няколко методи, които разработчикът да имплементира, а именно OnEpisodeBegin(), CollectObservations() и OnActionReceived(). С ML-Agents процесът на създаване на агенти и алгоритъм за самообучение чрез утвърждение е улеснен многократно, обаче двете най-критични и сложни части от всеки RL алгоритъм е интегрирането на агента със средата и дефинирането на системата за награди.

```
public class MyAgent : Agent
{
    public override void OnEpisodeBegin()
    {
        // Episode start
    }

    public override void CollectObservations(VectorSensor sensor)
    {
        // Collect environment state
    }

    public override void OnActionReceived(ActionBuffers actions)
    {
        // Perform an action
    }

    public override void Heuristic(float[] actionsOut)
    {
        // Default actions
    }
}
```

Фиг 2.4.

Някои други аспекти, свързани с пакета ML-Agents са:

- **Конфигурация** на обучителния процес – това са настройки във формата на YAML файл^[12], които се подават на ML-Agents процеса при началото на трениране. Тези настройки определят: специфичен алгоритъм на трениране (PPO, SAC и др.), хиперпараметри (например learning rate),

параметри на невронната мрежа (например брой слоеве), максимален брой стъпки на трениране (max_steps) и др.

- **Обучението**, което започва като се изпълни mlagents-learn, като някои от неговите аргументи са:

```
mlagents-learn [config.yaml] [--run-id RUN_ID] [--base-port BASE_PORT]  
[--torch-device DEVICE] ...
```

Като:

- config.yaml е гореспоменатия конфигурационен файл.
- --run-id е идентификатор на това трениране;
- --base-port е порт за комуникация с Unity средата, тъй като двата процеса си комуникират през сокет;
- --torch-device указва устройството, на което да се извършва тренирането (“сру” или „cuda“);
- **Прилагането** на вече обучен модел, което е възможно като се зареди asset, представляващ модела, през Unity и се закачи за BehaviourParameters (скрипт, който описва различни параметри на агента) на агента.

2.2.h Интегрирана среда за разработка – Visual Studio

За разработката на проекта е избран Visual Studio като основна интегрирана среда за разработка (IDE). Това е мощен и широко използвани инструмент за програмиране, който предлага пълна поддръжка за C# - Unity.

Причините за избора са: възможност за директно дебъгване на Unity програмата чрез точки на прекъсване, богат IntelliSense и широкия набор от инструменти за анализиране на апликацията.

Други IDE-та, които може да се използват за разработка на приложение са JetBrains Rider и MonoDevelop^[13], който идва с Unity, но изборът на средата за разработка е най-вече избор на разработчика, понеже, с достатъчно опит, с всичките IDE-та, може да се напишат едни и същи скриптове.

Трета глава

Реализация на дипломаната работа

3.1 Реализация на симулацията

Преди самата симулация, тряба да дефинираме няколко ключови типове, без които симулацията няма да работи:

- Клас Clock

Класът Clock е singleton клас, целта на който е да проследява времето на симулацията и да абстрактира класа Time. Той съдържа текущото време (DateTime datetime), дали времето е паузирано (bool isPaused), съотношението между симулационното и реалното време (float clockRatio),

```
// Singleton class representing time in the simulation
public class Clock : SingletonMonobehaviour<Clock>
{
    public DateTime datetime { get; set; }
    public bool isPaused = false;
    private static readonly float clockRatio = 1f;

    private float effectiveTimeScale;
    public float timeScale {
        get => effectiveTimeScale;
        set
        {
            if (!isPaused)
            {
                Time.timeScale = value;
                effectiveTimeScale = value;
            }
            else effectiveTimeScale = value;
        }
    }
}
```

Фиг 3.1.1.

като стойността по подразбиране е 1, времева скала (float timeScale) и ефектива времева скала (float effectiveTimeScale). Нуждата от последните две, е поради факта, че при паузиране timeScale обикновено се задава на 0, но тогава при отмяна на паузирането, timeScale няма как да знае към коя стойност да се върне.

- Именно пространство Utils

Тук се съдържат статични класове и методи, които симулацията извиква постоянно. Класовете в namespace Utils са:

- клас Arrays
- клас Random
- клас Math
- клас Pathfinding
- клас SingletonMonobehaviour<T>
- клас Modelingю

- Клас GameManager

Този singleton клас е отговорен за първоначалното зареждане на сцената в Unity, както и за различни настройки свързани с моделите на елементите на пътя, генерирането на пътна мрежа, автомобили и др.

```
public class GameManager : SingletonMonobehaviour<GameManager>
{
    [System.NonSerialized] public Simulation simulation;

    [Header("Prefabs")]
    public GameObject simulationPrefab;
    public GameObject roadStraightPrefab;
    public GameObject roadTurnPrefab;
    public GameObject roadJoinPrefab;
    public GameObject roadCrossPrefab;
    public GameObject roadEndPrefab;
    public GameObject buildingPrefab;

    [Header("Generation Settings")]
    [SerializeField] public float tileSize = 15f;
    [SerializeField] public int gridSize = 50;
    [SerializeField] public int junctionGap = 5;

    public override void Awake()
    {
        base.Awake();
        string loadMethod = PlayerPrefs.GetString("Load method");
        if (string.IsNullOrEmpty(loadMethod))
            loadMethod = "generate";
        switch (loadMethod)
        {

```

Фиг 3.1.2. Класът GameManager и неговия Awake метод.

При първоначалното зареждане на сцената за симулацията и съответно този скрипт се създава симулация или чрез алгоритъма за генериране, или от запазен файл с разширение „.tsf“ (Traffic simulation file).

- Клас Simulation

Това е MonoBehaviour класът, който върви симулацията. Той е закачен за обект, и при зареждането му се инициализират класове, които менажират автомобилите (VehicleManager vehicleManager) и сградите (BuildingManager buildingManager) съответно. Освен тях в Start метода се зареждат списъците с кръстовища (List<Junction> junctions) и пътища (List<Road> roads).

```
public class Simulation : MonoBehaviour
{
    public VehicleManager vehicleManager;
    public BuildingManager buildingManager;

    public List<Junction> junctions;
    public List<Road> roads;

    public void Awake()
    {
        vehicleManager = GetComponent<VehicleManager>();
        buildingManager = GetComponent<BuildingManager>();
    }
}
```

Фиг 3.1.3.

3.1.a Имплементация на кръстовища и пътища

Една пътна мрежа се състои от множество кръстовища и пътища между тях. С други думи, една пътна мрежа е граф, където върховете са кръстовищата, а ребрата са пътищата. Затова се имплементират два основни класа – Junction и Road.

За да се знае едно кръстовище кои пътища свързва и един път, към кои кръстовища води, всяка инстанция на тези класове трябва да съдържа референции към обекти от другия клас. Ако зареждането на симулацията става чрез алгоритъма за генериране, тогава референциите се откриват динамично с алгоритми като DFS, а ако се зарежда симулация от файл, то референциите се съдържат в файла.

- Клас Junction

Това е класът представляващ кръстовище. Едно кръстовище има тип (Type type), списък с пътища, които свързва (List<Road> roads), кеширани изходни точки на всеки Road обект за бързи изчисления (List<Vector3> exitPoints), симулация към, която принадлежи (Simulation simulation) и светофарен контролер (TrafficController trafficController), който отговаря за светофарните сигнали.

Единствената особеност, е че задънените улици също се запазват като Junction обект, чийто тип е None. Това е, за да се запази графообразната структура на пътната мрежа и за да може да се приложи алгоритъм за намиране на път в граф до тези участъци.

```
// A junction is either a crossroad, joinroad (t-junction) or an end road (for pathfinding)
public class Junction : MonoBehaviour
{
    public enum Type
    {
        None,
        Stops,
        Lights,
    }

    public Type type;
    public List<Road> roads;
    public List<Vector3> exitPoints;

    public Simulation simulation;
    public TrafficController trafficController;

    public BoxCollider boxCollider;
```

Фиг 3.1.4.

- Клас Road

Този клас представлява път. Един път има начално кръстовище (Junction junctionStart), крайно кръстовище (Junction junctionEnd), списък от точки, описващи формата на пътя (List<Vector3> path), кеширана стойност,

представляваща дължината на пътя (float length) и симулация към, която принадлежи (Simulation simulation).

```
// Straight roads and turns are considered a part of the road path
public class Road
{
    public Simulation simulation;

    // Both of the junctions at the road's ends
    public Junction junctionStart { get; set; }
    public Junction junctionEnd { get; set; }

    // A list of points that describe the path of the road
    public List<Vector3> path;
    public float length;
```

Фиг 3.1.5.

За разлика от Junction класа, Road не наследява MonoBehaviour. Това е защото още от алгоритъма за генериране на пътна мрежа остава факта, че цялата симулация е изградена от prefab-и, които представляват квадратни клетки. За абстрактиране от това, Road не е прикачен към Unity обект, а има само списък от точки, които проследяват обектите в сцената. Така надеждата, е че този клас би могъл да се употреби с пътни мрежи различни от квадратни.

3.1.b Имплементация на сгради

- Клас BuildingManager

Класът BuildingManager е закачен за същия обект като Simulation скрипта и той е предназначен за съхранението и за извършването на прости операции с сградите в симулацията. Той съдържа симулацията към която принадлежи (Simulation simulation), списък със сгради (List<Building>

buildings), както и кеширана хеш таблица между тип сграда и самата сграда за забързано търсене (Dictionary<Type, List<Building>> buildingsByType).

```
public class BuildingManager : MonoBehaviour
{
    private Simulation simulation;

    public List<Building> buildings;
    public Dictionary<Building.Type, List<Building>> buildingsByType;
```

Фиг 3.1.6.

- Клас Building

Всяка сграда има закачен Building скрипт, който не се изпълнява всеки кадър, а съдържа данни за симулацията. Всяка сграда съдържа тип на сградата (Type type), списък с улици, на които се намира сградата (List<Road> roads), хеш таблица за най-близката точна на улицата, за всяка улица (Dictionary<Road, Vector3> spawnPoints), най-близкото кръстовище (Junction closestJunction) и последно кой менажира сградата (BuildingManager buildingManager).

```
public class Building : MonoBehaviour
{
    public enum Type
    {
        None,
        Home,
        Work
    }
    public Type type;

    public List<Road> roads = new List<Road>();
    public Dictionary<Road, Vector3> spawnPoints = new Dictionary<Road, Vector3>();

    public Junction closestJunction;

    private Simulation simulation;
    private BuildingManager buildingManager;
```

Фиг 3.1.7.

3.1.c Имплементация на светофари

- Клас TrafficController

Класът TrafficController е закачен за детето на Junction обекта и той контролира светофарите в кръстовището, като предоставя интерфейс за сменянето на fazите на светофарите. Класът задържа информация за кръстовището, на което принадлежи (Junction junction), списъка от светофари, които управлява (List<TrafficLight> lights), режим на управление (Mode mode) и агентът, който участва в процеса на трениране (TrafficAgent agent).

```
public class TrafficController : MonoBehaviour
{
    public enum Mode
    {
        Single,
        Double,
    }

    public Mode mode;

    private bool switchMode = false;
    private Mode newMode;

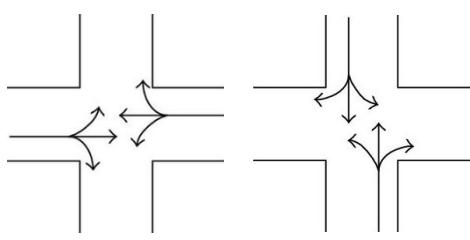
    public Junction junction;
    public TrafficLightAgent agent;
    public List<TrafficLight> lights;

    public float elapsedTime = 0.0f;
    private int activeLight = 0;

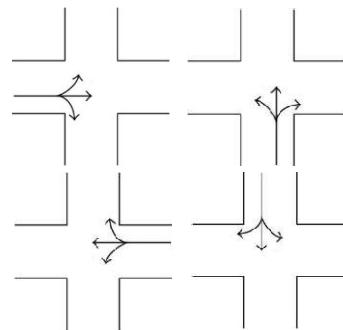
    public static readonly float defaultGreenInterval = 20f;
```

Фиг 3.1.8.

Всеки контролер на трафик работи в един от два режима – Single и Double. Съответно в Single режим се пуска зелен сигнал на всяка улица една по една в 3/4 фази (в зависимост от кръстовището), а в Double – зеленият сигнал се пуска на противоположни улици в 2 фази. На обикновено кръстовище това би изглеждало по следния начин:

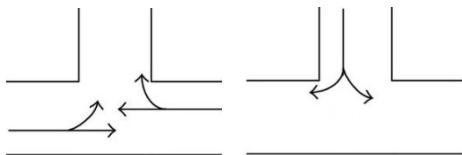


Фиг 3.1.9. 2-те фази в режим
Double при обикновено
кръстовище.

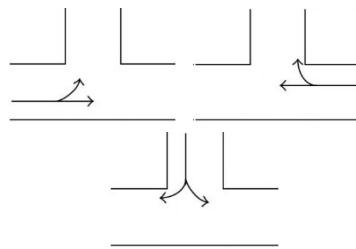


Фиг 3.1.10. 4-те фази в режим
Single при обикновено
кръстовище.

На Т-образно кръстовище светофарните режими изглеждат така:



Фиг 3.1.11. 2-те фази в режим
Double при Т-образно
кръстовище.



Фиг 3.1.12. 3-те фази в
режим *Single* при Т-образно
кръстовище.

- Клас TrafficLight

Всеки светофар е отделен обект дете на трафик контролера и си има собствен скрипт. Всеки светофар има статус, който определя моментния сигнал (Status status), път, на който се намира светофара (Road road), списък с автомобили, които сами се записват в него, като се наредят на червена светлина (List<Vehicle> vehicleQueue), конфигурираната в момента продължителност на зеления интервал (float greenInterval) и трафик контролера, който съдържа този светофар (TrafficController trafficController).

```

public class TrafficLight : MonoBehaviour
{
    public static readonly float minGreenInterval = 5.0f,
        maxGreenInterval = 40.0f,
        yellowInterval = 4.0f,
        redIntervalBuffer = 4.0f;

    public enum Status
    {
        Red,
        Yellow,
        Green
    };

    public Road road;
    public Vector3 roadDirection; // The direction to the road

    public List<Vehicle> vehicleQueue;
    public TrafficController trafficController;

    private Status _status;
    public Status status
    {
        get => _status;
        set
        {
            _status = value;

            meshRenderer.material.color = TrafficLight.StatusToColor(_status);

            if (value == Status.Green)
                vehicleQueue.Clear();
        }
    }

    public float greenInterval = 10.0f;
    private MeshRenderer meshRenderer;
}

```

Фиг 3.1.13.

3.1.d Имплементация на автомобил

За симулацията се симулират отделни автомобили. Това определя симулацията като микроскопична, тъй като се изчисляват отделните единици от пътното движение. Максималният брой автомобили в града е пропорционален на броя сгради. Това ограничение на броя автомобили е абсолютно и никога не се превишава. Изчислява се по следния начин:

$$n = kb$$

Фиг !. Формула за определяне на максималния брой автомобили в града, където:

n – максималният брой автомобили

b – броят сгради

k – скалиращ фактор

Освен тази величина има и ефективен брой автомобили, което е броят, който в момента трябва да има в града. Съществува също и настройка за зависим от времето количество трафик.

Посоката на движение на автомобилите бива два типа:

- Насочено – от дом към работно място или обратното.
- Произволно – от една сграда до друга, независимо каква е тя.

Вероятностите за вида движение се изчисляват от функцията `Utils.Modeling.CalculateTrafficFlowFromTime()`.

Жизненият цикъл на автомобил е следния:

- Определя се какъв тип движение ще има – насочено или произволно, спрямо променливи, които диктуват съответните вероятности. Реализира се в `CreatePath()`;
- Определя се начална сграда и краяна сграда. Реализира се в `CreateDirectedVehiclePath()` или `CreateRandomVehiclePath()`;
- Изчислява се най-краткия път от най-близките сгради на тези сгради чрез A^* ^[14], което е алгоритъм за намиране на път в графи и се създава списък от точки, описващи пътя;
- Автомобилът се добавя в опашка от коли, за да може да бъде активиран;
- През определен период се проверява пътя, на който се намира началната сграда и ако той е с автомобили, възпрепятстващи задействането на

текущия, тогава той остава в опашката за по-късен опит на активиране. В случая, че пътя е чист, тогава автомобилът се задейства и се премахва от опашката;

- При задействане, автомобилът почва да проследява списъка от точки от алгоритъма за намиране на път, като се движи според другите автомобили на пътя и светофарните сигнали;
- При пристигане до крайната сграда, автомобилът се унищожава и това поставя край на жизнения цикъл на един автомобил.

Важни класове, свързани с автомобилите са:

- **Клас VehicleManager**

VehicleManager е скрипт на същия обект като Simulation и е отговорен за всичко свързано с автомобили. Той съдържа абсолютния максимален брой автомобили (int maxVehicleCount), ефективния максимален брой автомобили (int simulatedMaxVehicleCount), опашка за задействане на автомобили (List<Vehicle> spawnQueue), типове автомобили, с които VehicleManager може да работи (List<VehiclePreset> types) и параметри описващи вероятностите за насочен трафик (float homeToWorkTrafficChance, workToHomeTrafficChance).

```
public class VehicleManager : MonoBehaviour
{
    private Simulation simulation;

    public int maxVehicleCount;
    [System.NonSerialized] public int currentVehicleCount;
    [System.NonSerialized] public int simulatedMaxVehicleCount;

    private float minVehicleTravelDistance = 75.0f;
    private float minVehicleSpawnDistance = 30.0f;
    private float turnRadius=7.5f;
    private int turnResolution=5;

    private List<VehiclePreset> types = new List<VehiclePreset>();

    public List<Vehicle> spawnQueue = new List<Vehicle>();

    public float homeToWorkTrafficChance = 1.0f;
    public float workToHomeTrafficChance = 1.0f;
```

Фиг 3.1.14.

Всеки кадър се проверява дали може да се задейства автомобил и ако може се активира като обект.

- Клас VehiclePreset

VehiclePreset е ScriptableObject, който представлява нещо като шаблон за автомобил. Надеждата, е че така би могло лесно да се добавят повече видове автомобили.

```
public class VehiclePreset : ScriptableObject
{
    /*
     * Vehicle settings
     */

    public GameObject prefab;

    public float maxVelocity = 25.0f;

    public float accelerationRate = 5.0f;
    public float decelerationRate = 10.0f;

    public float lookAheadDistance = 20.0f;
    public float stoppedGapDistance = 3.0f;
```

Фиг 3.1.15.

- Клас VehiclePath

VehiclePath е клас съдържащ данни за пътя на един автомобил – запазва списък от точки, по които да се движи автомобилът (List<Vector3> points), началната и крайна сграда (Building from, to) и списък с елементи от пътя, по които минава пътя (List<Road> roads, List<Junction> junctions). Пътят на автомобил се изчислява от Utils.Pathfinding.FindCarPath(), където се изпълнява A* между най-близките кръстовища на началната и крайна сграда.

```
public class VehiclePath
{
    public Building from, to;

    public List<Road> roads;
    public List<Junction> junctions;

    public List<Vector3> points;
```

Фиг 3.16.

- Клас Vehicle

Vehicle е абстрактен клас, който имплементира базовата логика за движението на автомобилите. Всеки автомобил има шаблон (VehiclePreset preset), път (VehiclePath path), мениджър (VehicleManager), статус (Status status) и др.

```
public abstract class Vehicle : MonoBehaviour
{
    public enum Status
    {
        DRIVING,
        WAITING_RED,
        WAITING_CAR,
    };
    public Status status { get; private set; }

    private VehicleManager vehicleManager;

    public VehiclePreset preset;

    public VehiclePath path;

    protected Vector3 bumperOffset;
    protected Vector3 bumperPosition;

    protected float velocity;
    protected float acceleration;
    protected float distanceThisFrame;
```

Фиг 3.1.17.

- Клас Car

Car наследява Vehicle и имплементира функция, която връща Collider, който Vehicle може да използва за собствените си логика.

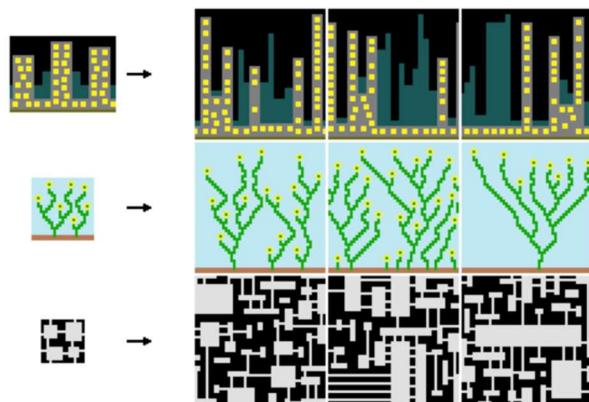
```
public class Car : Vehicle
{
    // Returns the center of the front of the box collider
    public override Vector3 GetBumperOffset()
    {
        BoxCollider boxCollider = GetComponent<BoxCollider>();
        return new Vector3(
            boxCollider.center.x,
            boxCollider.center.y,
            boxCollider.center.z + boxCollider.size.z / 2
        ) + 0.01f * transform.forward;
    }
}
```

Фиг 3.1.18.

3.2 Реализация на генериране на произволен път

3.2.a Теория на алгоритъма

Ключовата част от генерирането се изпълнява от алгоритъма WFC^[15]. Това е алгоритъм за рестриктивно процедурно генериране, който се основава на локални шаблони и правила за съседство между елементи. Той най-често се използва в контекста на решетка от елементи, като съседи са докосващи се клетки. Долу са посочени примерни резултати от алгоритъма:



Фиг 3.2.1. Примери на WFC.

Отляво – тайлсет, който е вход на алгоритъма.

Отдясно – генерирана решетка от клетки.

WFC е подходящ избор за генериране на пътища, поради неговия мрежов характер. Конкретно всеки WFC се състои от следните няколко стъпки:

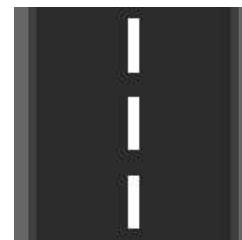
- **Дефиниране на тайлсет:**

WFC работи над даден тайлсет – това е множеството от възможните елементи и техните правила за съседство в една клетка от решетката. Може да дефинираме често срещани елементи от пътната инфраструктура (и всичките им ориентации) като:

- Прав път:

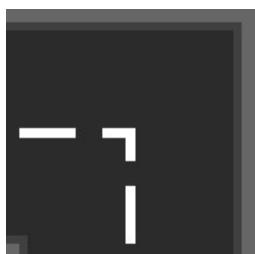


Фиг 3.2.2. Прав път с
хоризонтална ориентация и
съседство на изток и запад.



Фиг 3.2.3. Прав път с
вертикална ориентация и
съседство на север и юг.

- Завой:



Фиг 3.2.4. Завой
със съседство на
запад и юг.



Фиг 3.2.5. Завой
със съседство на
запад и север.



Фиг 3.2.6. Завой
със съседство на
север и изток.



Фиг 3.2.7. Завой
със съседство на
изток и юг.

- Т-образно кръстовище:



Фиг 3.2.8. Т-
образно
кръстовище със
съседство на север,



Фиг 3.2.9. Т-
образно
кръстовище със
съседство на запад.

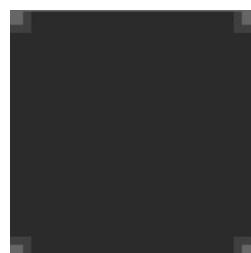


Фиг 3.2.10. Т-образно
кръстовище със
съседство на север,
юг и изток.



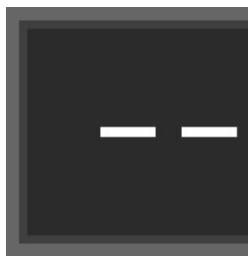
Фиг 3.2.11. Т-образно
кръстовище със
съседство на север,
изток и запад.

- Обикновено кръстовище:



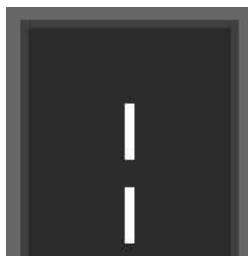
Фиг 3.2.12.
Кръстовище със
съседство във всички
посоки.

- Задънена улица:



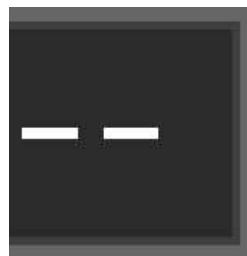
Фиг 3.2.13.

Задънена улица със съседство на изток.



Фиг 3.2.14.

Задънена улица със съседство на юг.



Фиг 3.2.15.

Задънена улица със съседство на запад.



Фиг 3.2.16.

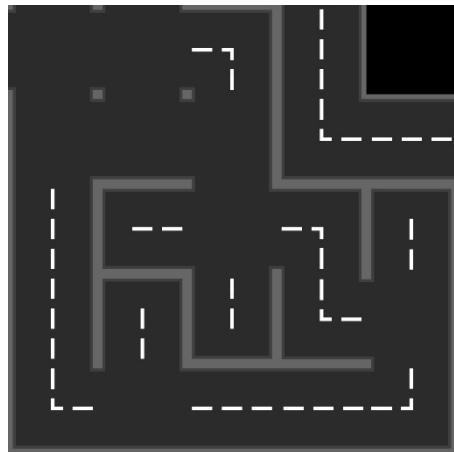
Задънена улица със съседство на север.

- „Колапс на вълнова функция“:

Това е същинската част от алгоритъма. Избира се начална клетка и понеже се генерира град, това може да е обикновено кръстовище в центъра на града. Разглеждат се всички околни клетки и се прилагат ограниченията, която тя поставя върху съседите ѝ, според дефинираните ѝ съседства. След това се избира съседа с най-малко възможности и се избира произволен елемент от пътя, като след поставянето целият процес се повтаря до изпълване на решетката. В случая, че няма възможност за клетка, тя може да се остави празна, както в примера долу.

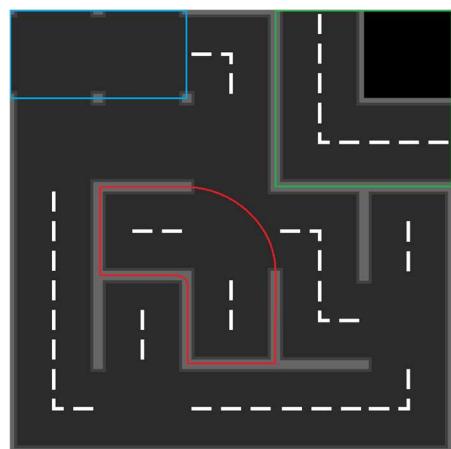
Понеже клетките, които представляват елементи от пътя, са със съседства, където води пътя, в края на този алгоритъм решетката би трябвало да се изпълни с пътни клетки, които са свързани логично – или път за път, или тротоар за тротоар.

В края на тази стъпка би трявало да има решетка, която изглежда по подобен начин на долупосочения пример:



Фиг 3.2.17. Примерен генериран град с размер на решетката 5.

Алгоритъмът до този момент има 3 основни проблема:



Фиг 3.2.18. Три проблема в генериран град по-първия метод. Съответно:
В синьо – пътищата излизат от ограничението на решетката.
В зелено – изолирани части от пътната мрежа.
В червено – задънени улици близо до центъра на града (подобно многото кръстовища близо до ръба на града – горе ляво).

Тези проблеми може да се отстраният по следните начини:

- За клетки на ръба на решетката, се ограничават възможностите до само задълни улици. Ако няма останали възможности пък, както клетката най-горе вдясно в примера, тогава клетката се оставя празна.
- За изолираните части от пътя може да се използва DFS алгоритъм, който да открие всички свързани сегменти от пътища, после да се избере централния и да се съкратят останалите, като се заместят с празни клетки .
- За последния проблем, може да се дефинират вероятностни функции за всички участъци от пътя, които дават шанса даден участък да се появи спрямо разстоянието му от центъра на града. Кръстовища ще имат по-голяма вероятност да се появят близо до центъра, а задълни улици не толкова. Далече от центъра, близо до ръба на решетката, обратното е вярно.

- Раздалечаване на клетките

Последната стъпка на алгоритъма за генериране е раздалечаване на клетките, чрез разширяване на решетката с някакъв константен брой клетки, за да се придобие вид на град, понеже до момента съседни кръстовища и завои са с нулево разстояние между тях. Това може да се постигне като текущите клетки се преместят в увеличена решетка, която е с размер:

$$n_2 = n_1 + k(n_1 - 1) + 2$$

Фиг !. Формула за размера на разширена решетка, където:

n_2 – размер на разширена решетка

n_1 – размер на оригиналната решетка

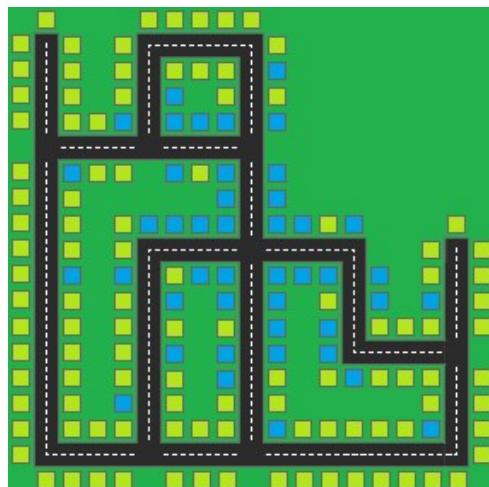
k – брой клетки на разширение

Причината за добавяне на 2, е за да може да се осигури място за клетки сгради, които са по ръбовете на решетката - по най-далечните улици.

След разширяването на решетката, може да удължим тайлсете на алгоритъма, като добавим клетка представляща сграда. Итерирайки през решетката, постигаме две цели:

- Запълването на разстоянията между кръстовищата с прости улици;
- Добавяне на сгради в клетките, които се допират до улица.

Последно се избира тип на сграда – дом или работно място. Отново, този избор е базиран на вероятностна функция спрямо разстоянието от центъра на града. Накрая пътната мрежа изглежда, подобно, на тази долу и алгоритъмът е готов, а симулацията може да започне.



Фиг 3.2.19. Примерен резултат на алгоритъма за генериране на произволна пътна мрежа:
Синя сграда – работно място
Зелена сграда – дом

3.2.b Имплементация

Преди да пристъпим към имплементация на алгоритъма, който беше описан стъпка по стъпка горе, първо трябва да дефиниране няколко класове:

- **Клас Grid**

Представлява решетката от алгоритъма горе. Всеки обект от този тип има размер от клетки (int size), самите клетки (GridTile[,] tiles) и изместване от центъра на света (Vector2 centerOffset), като по подразбиране центърът на генериране е в (0, 0, 0).

```
namespace Generation
{
    public class Grid
    {
        public int size;
        public GridTile[,] tiles;
        public Vector2 centerOffset;

        public Grid(int size)
        {
            this.size = size;

            tiles = new GridTile[size, size];
            for (int i = 0; i < size; ++i)
            {
                for (int j = 0; j < size; ++j)
                {
                    tiles[i, j] = GridTile.empty;
                    tiles[i, j].SetGrid(this, new Vector2Int(i, j));
                }
            }

            this.centerOffset = - new Vector2(
                (size / 2.0f) * GameManager.Instance.tileSize,
                (size / 2.0f) * GameManager.Instance.tileSize
            );
        }
    }
}
```

Фиг 3.2.20.

- Клас GridTile

Представлява клетка от алгоритъма горе. Всеки обект от този тип има референция към решетката към която принадлежи (Grid grid), тип на това, което представлява – път, кръстовище или сграда, като None е стойност по подразбиране (Type type), списък на съседство (List<char> validConnections), както и списък с грешни връзки (List<char> invalidConnections), които се изчисляват въз основа на списъка на съседство, координати къде се намира в решетката (Vector2Int coords), prefab-ът, който представлява (GameObject prefab) и ъгъл спрямо Y оста (int rotY).

Последните две – prefab и rotY, могат да закодират всички възможни ориентации на дадена клетка само с един модел.

```
namespace Generation
{
    public class GridTile
    {
        public enum Type
        {
            None,
            Road,
            Junction,
            Building,
        }

        // instead of 0f, it has to be an offset for the center of the city
        public Vector3 physicalPos => new Vector3(
            (coords.x + 0.5f) * GameManager.Instance.tileSize + grid.centerOffset.x,
            0,
            (coords.y + 0.5f) * GameManager.Instance.tileSize + grid.centerOffset.y
        );

        public static GridTile empty => new GridTile(Type.None, null, 0, null, null, -Vector2Int.one);

        public Type type;
        public GameObject prefab;
        public int rotY;
        public List<char> validConnections;

        private List<char> invalidConnections;
        public Grid grid;
        public Vector2Int coords;
```

Фиг 3.2.21.

- Клас WFC

Това е класът, който е отговорен за вървенето на WFC алгоритъма. Съдържа тайлсет (`List<GridTile> tiles`), решетка (`Grid grid`), възможности от клетки за всяка клетка от решетката (`List<GridTile>[,] possibilities`) и тежести, които определят вероятността на различните клетки спрямо разстоянието им от центъра на града.

```
namespace Generation
{
    public class WFC
    {
        public List<GridTile> tiles;

        public Grid grid { get; private set; }
        public List<GridTile>[,] possibilities;

        private float straightChance = 0,
                    turnChance = 0,
                    joinChance = 0,
                    crossChance = 0,
                    endChance = 0;
    }
}
```

Фиг 3.2.22.

Самият алгоритъм е имплементиран по следния начин:

Цялостното генериране е имплементирано по следния начин, като самите клетки се инстанциират накрая като обекти в сцената.

```
namespace Generation
{
    public static class Generation
    {
        public static readonly List<GridTile> tiles = new List<GridTile>()
        {
            new GridTile(GridTile.Type.Junction, GameManager.Instance.roadCrossPrefab, 0, "NESW", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Junction, GameManager.Instance.roadJoinPrefab, 0, "WNE", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Junction, GameManager.Instance.roadJoinPrefab, 90, "NES", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Junction, GameManager.Instance.roadJoinPrefab, 180, "ESW", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Junction, GameManager.Instance.roadJoinPrefab, 270, "SWN", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Junction, GameManager.Instance.roadEndPrefab, 0, "N", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Junction, GameManager.Instance.roadEndPrefab, 90, "E", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Junction, GameManager.Instance.roadEndPrefab, 180, "S", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Junction, GameManager.Instance.roadEndPrefab, 270, "W", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Road, GameManager.Instance.roadStraightPrefab, 0, "NS", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Road, GameManager.Instance.roadStraightPrefab, 90, "WE", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Road, GameManager.Instance.roadTurnPrefab, 0, "NE", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Road, GameManager.Instance.roadTurnPrefab, 90, "ES", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Road, GameManager.Instance.roadTurnPrefab, 180, "SW", null, -Vector2Int.one),
            new GridTile(GridTile.Type.Road, GameManager.Instance.roadTurnPrefab, 270, "WN", null, -Vector2Int.one),
        };

        public static void Generate(Transform parent)
        {
            GameManager gm = GameManager.Instance;

            Grid grid = new Grid(gm.gridSize);
            WFC wfc = new WFC(tiles, grid);
            wfc.Run();

            // Remove disconnected grid tiles
            Optimization.KeepLargestRoadComponent(grid);

            // Space out grid by junctionGap
            var expandedGrid = ExpandGrid(grid, gm.junctionGap);
            var buildingTile = new GridTile(GridTile.Type.Building, gm.buildingPrefab, 0, "NESW", null, -Vector2Int.one);

            // Fill empty spaces with straight roads && buildings
            FillStraightRoads(expandedGrid, gm.junctionGap);
            FillBuildings(expandedGrid, buildingTile);

            // Get ground object rescale and position it
            Transform ground = parent.transform.GetChild(0);
            var scale = ground.localScale;
            scale.Scale(new Vector3(expandedGrid.size, 1f, expandedGrid.size));
            ground.localScale = scale;
            ground.localPosition += 0.025f * Vector3.down;

            Instantiation.InstantiateGrid(expandedGrid, new List<GridTile>(tiles) { buildingTile }, parent);
        }
    }
}
```

Фиг 3.2.23.

3.3 Реализация на алгоритъм за обучение

3.3.a Теория на алгоритъма

Преди да преминаването към конкретната имплементация на алгоритъма, е важно да се разберат основните теоретични принципи, които стоят зад него. В тази част се описва ключовата част от алгоритъма – системата за награди.

Както бе споменато във втора глава, системата за награди (reward system) е сърцевината на всеки алгоритъм за обучение с утвърждение. Тя определя как агентът ще бъде награждаван за действията, които предприема в средата. В контекста на симулацията на трафика, наградата е свързана с постигането на ефективно управление на трафика и минимизиране на задръстванията. Награждават се и се наказват следните действия и явления:

- Преминаване на автомобил – допринася 1 точка към общата награда.
- Дълго общо чакане на автомобилите на кръстовище – отнема експоненциално повече точки, спрямо общото време за чакане на всички автомобили. С други думи, колкото повече време чакат автомобили, толкова по-голямо е наказанието. Това насищава агентът да пропуска автомобили по-отрано.
- Задръстване в текущото кръстовище и околните кръстовища – отнема точки, пропорционално равни на измереното количество „задръстване“, което се смята с дълчините на опашките и времената за чакане на автомобилите.
- Кооперация със съседни агенти – допринася точки за положителна промяна в задръстването в съседните кръстовища.

„Задръстването“ е величина, която се оценява от друг скрипт (CongestionTracker) за всяко кръстовище и тя се изчислява като претеглена сума:

$$n = k_1 Q + k_2 T$$

Фиг !. Формула за приблизителна оценка на задръстването

Q – сума от дължините на опашките от автомобили

T – сума от времената за чакане на червено

k_1, k_2 - тежести

Поради спорадичността на симулацията, особено при забързано време, измерването на задръстването става чрез усредняване на списък от оценки на задръствания, всички взети в определен интервал от време.

3.3.b Имплементация

- Клас CongestionTracker

Този клас е предназначен за оценяването на задръстването на едно кръстовище. Той е MonoBehaviour, закачен за трафик контролер обекта, заедно с TrafficController и TrafficAgent. Всеки CongestionTracker измерва стойност на задръстване във времеви прозорец (float timeWindow), като добавя нова стойност всеки няколко секунди (float updatePeriod).

```
public class CongestionTracker : MonoBehaviour
{
    public float timeWindow = 90f; // Total congestion is an average over 5 minutes
    public float updatePeriod = 1.5f; // Add values every 5 seconds

    private VehicleManager vehicleManager;
    private TrafficController trafficController;

    private float maxHistoryLength;
    private Queue<float> congestionHistory;
    private float cumulativeCongestion = 0f;
    private float timeElapsed;
```

Фиг 3.3.1.

Усредненото задръстване, което се връща от GetAverageCongestion(), се изчислява като сумата на историята от задръствания (float cumulativeCongestion) делено на броя измервания (congestionHistory.Count).

- Клас TrainingManager

TrainingManager е singleton MonoBehaviour, който следи всичките агенти (List<TrafficAgent> agents) и техните им настройки (List<BehaviourParameters> behaviours), опции като: трафик, зависещ от часа в денонощието (bool timeDependentTraffic), дали кръстовищата могат да работят в двата режима (bool twoModeJunctions), като по-подразбиране са в режима Double, и дължина на един епизод на трениране (float episodeLength).

```
public class TrainingManager : SingletonMonoBehaviour<TrainingManager>
{
    [SerializeField] public bool timeDependentTraffic = false;
    [SerializeField] public bool twoModeJunctions = false;
    [SerializeField] public float episodeLength = 300.0f; // in seconds simulated time

    private List<TrafficAgent> agents;
    private List<BehaviorParameters> behaviours;
```

Фиг 3.3.2.

- Клас TrafficAgent

Класът TrafficAgent е агент, който наследява базовия Agent, който е предоставен от ML-Agents пакета. Той е закачен за същия обект като TrafficController (детето на Junction обекта). Всеки агент има конфигурация (BehaviourParameters parameters), референция към мениджъра на трениране (TrainingManager trainingManager) и референции към различни части от симулацията, като: мениджъра на автомобили (VehicleManager vehicleManager), собствения трафик контролер (TrafficController trafficController), оценител на задръстването (CongestionTracker tracker) и съедни агенти (List<TrafficAgent> neighbours).

```

public class TrafficAgent : Agent
{
    private BehaviorParameters parameters;
    private TrainingManager trainingManager;

    private VehicleManager vehicleManager;
    private TrafficController trafficController;
    public CongestionTracker tracker;
    private List<TrafficAgent> neighbours;

    private float previousCongestion;
    private float previousReward;
    private float timeElapsed;
}

```

Фиг 3.3.3.

Размера на вектора с данни в CollectObservations() се изчислява в Start()
метода, въз основата на броя съседни кръстовища и броя пътища.

```

parameters.BrainParameters.VectorObservationSize = 2 * thisJunction.roads.Count;
foreach (var junction in neighbouringJunctions)
{
    parameters.BrainParameters.VectorObservationSize += 2;
}

```

Фиг 3.3.4.

По време на тренирането се добавят наблюдения към този вектор:

```

public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(tracker.GetQueueLengths().Select(v => (float) v).ToArray());
    sensor.AddObservation(tracker.GetTotalWaitingTimes().Select(v => (float) v).ToArray());

    foreach (var agent in neighbours)
    {
        sensor.AddObservation(agent.tracker.GetQueueLengths().Sum());
        sensor.AddObservation(agent.tracker.GetTotalWaitingTimes().Sum());
    }
}

```

Фиг 3.3.5.

След като питонския бакенд, вземе решение, което агентът да изпълни, то се отразява в симулацията и се изчислява наградата, по демонстрирания горе начин.

```
public override void OnActionReceived(ActionBuffers actionBuffers)
{
    ReflectActions(actionBuffers);
    EvaluateRewards();
}
```

Фиг 3.3.6.

3.3.с Изпробване на имплементацията

След като алгоритъмът за обучение е имплементиран, следващата стъпка е изпробването му чрез стартиране на процеса на обучение с ML-Agents. За целта се използва командата `mlagents-learn`, която стартира тренирането въз основа на конфигурационен файл, който изглежда така:

```
behaviors:
  TrafficLightBrain:
    trainer_type: ppo
    hyperparameters:
      batch_size: 64
      buffer_size: 2048
      learning_rate: 0.0003
      beta: 0.001
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
    network_settings:
      normalize: true
      hidden_units: 128
      num_layers: 2
    reward_signals:
      extrinsic:
        strength: 1.0
        gamma: 0.99
      max_steps: 10000000
      keep_checkpoints: 10
```

Фиг 3.3.7.

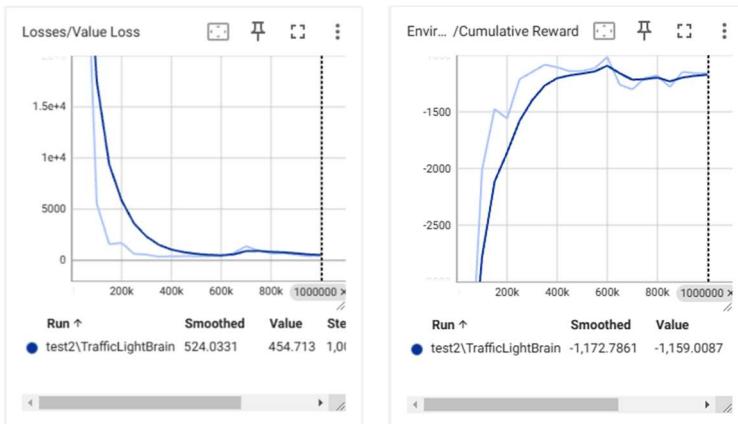
Изпълняваме mlagents-learn с config.yaml файла и с run-id и стартираме Unity апликацията, и натискаме бутона за трениране и така обучението започва. Резултатите от тренирането, като checkpoint-и и финалния модел се записват в директория зададена от --results-dir, като папката по-подразбиране е „results“. Анализирането на обучението е възможно с външни средства като TensorBoard^[16]. Примерните резултати след 90 минути трениране с горната система за награди и горния конфигурационен файл са:

Горното изображение, показва две графики:

- Първата показва Value Loss спрямо времето на обучението на модела TrafficLightBrain. Вижда се как Loss стойността намалява и се стабилизира все повече и повече с течение на времето. Това е добър знак, тъй като показва, че моделът успешно се обучава без сериозни колебания или дивергенция.

```
TrafficLightBrain. Step: 50000. Time Elapsed: 278.858 s. Mean Reward: -4067.156. Std of Reward: 7451.496. T
TrafficLightBrain. Step: 100000. Time Elapsed: 532.836 s. Mean Reward: -2015.350. Std of Reward: 2207.686.
TrafficLightBrain. Step: 150000. Time Elapsed: 756.507 s. Mean Reward: -1475.175. Std of Reward: 1130.105.
TrafficLightBrain. Step: 200000. Time Elapsed: 974.137 s. Mean Reward: -1559.084. Std of Reward: 1374.047.
TrafficLightBrain. Step: 250000. Time Elapsed: 1192.442 s. Mean Reward: -1211.615. Std of Reward: 811.624.
TrafficLightBrain. Step: 300000. Time Elapsed: 1412.548 s. Mean Reward: -1148.475. Std of Reward: 820.780.
TrafficLightBrain. Step: 350000. Time Elapsed: 1633.505 s. Mean Reward: -1084.066. Std of Reward: 654.551.
TrafficLightBrain. Step: 400000. Time Elapsed: 1854.790 s. Mean Reward: -1106.780. Std of Reward: 689.055.
TrafficLightBrain. Step: 450000. Time Elapsed: 2080.390 s. Mean Reward: -1141.660. Std of Reward: 749.833.
TrafficLightBrain. Step: 500000. Time Elapsed: 2311.188 s. Mean Reward: -1139.352. Std of Reward: 691.731.
[...]
[161] Converting to results\test2\TrafficLightBrain\TrafficLightBrain-499973.onnx
[173] Exported results\test2\TrafficLightBrain\TrafficLightBrain-499973.onnx
TrafficLightBrain. Step: 550000. Time Elapsed: 2535.545 s. Mean Reward: -1114.811. Std of Reward: 720.879.
TrafficLightBrain. Step: 600000. Time Elapsed: 2748.467 s. Mean Reward: -1016.850. Std of Reward: 707.555.
TrafficLightBrain. Step: 650000. Time Elapsed: 2962.850 s. Mean Reward: -1260.688. Std of Reward: 1053.447.
TrafficLightBrain. Step: 700000. Time Elapsed: 3176.309 s. Mean Reward: -1301.966. Std of Reward: 1216.357.
TrafficLightBrain. Step: 750000. Time Elapsed: 3386.567 s. Mean Reward: -1203.037. Std of Reward: 1101.621.
TrafficLightBrain. Step: 800000. Time Elapsed: 3599.039 s. Mean Reward: -1178.324. Std of Reward: 978.071.
TrafficLightBrain. Step: 850000. Time Elapsed: 3820.025 s. Mean Reward: -1280.680. Std of Reward: 1086.789.
TrafficLightBrain. Step: 900000. Time Elapsed: 4041.486 s. Mean Reward: -1147.824. Std of Reward: 785.065.
TrafficLightBrain. Step: 950000. Time Elapsed: 4256.251 s. Mean Reward: -1158.216. Std of Reward: 718.708.
TrafficLightBrain. Step: 1000000. Time Elapsed: 4471.978 s. Mean Reward: -1159.009. Std of Reward: 824.803.
[161] Converting to results\test2\TrafficLightBrain\TrafficLightBrain-999965.onnx
[173] Exported results\test2\TrafficLightBrain\TrafficLightBrain-999965.onnx
[187] Learning was interrupted. Please wait while the graph is generated.
[223] UnityEnvironment worker 0: environment stopping.
[237] UnityEnvironment worker 0 closing.
[240] UnityEnvironment worker 0 done.
[161] Converting to results\test2\TrafficLightBrain\TrafficLightBrain-1002497.onnx
[173] Exported results\test2\TrafficLightBrain\TrafficLightBrain-1002497.onnx
[151] Copied results\test2\TrafficLightBrain\TrafficLightBrain-1002497.onnx to results\test2\TrafficLightBrain-1002497.onnx
[81] Saved Model
[482] SubprocessEnvManager closing.
[110] UnityEnvWorker 0 got exception trying to close.
[st>]
```

Фиг 3.3.8. Изходът на mlagents-learn от изпробването на описания алгоритъм за обучение.



Фиг 3.3.9. *TensorBoard* графики от изпробването на описания алгоритъм за обучение.

- Втората показва кумулативната награда на агентите спрямо времето на обучение. Първоначално тя се увеличава бързо, след което започва да се забавя и приближава към конкретна стойност, което е знак че обучението е стабилно и моделът конвергира.

Допълнително, втората графика показва, че кумулативната награда стабилизира около много ниска отрицателна стойност, което означава, че агентът диспропорционално получава повече наказания от награди по време на обучението. Макар и, моделът да е открил стабилна стратегия, това може да се оправи с по- внимателна селекция на наградите. За да се подобри е възможно да се добавят повече начини агентът да получава положителни награди, като например поощрения за плавен трафик и за зелени вълни, както и намаляване на тежестта на наказанията.

Намирането на абсолютно оптималната система за награждаване, и съответно оптимален трафик, изисква много повече тестове и итерации, тъй като малки промени могат значително да повлият на обучението на агентите. Текущата система е далеч от перфектна – въпреки че води до стабилизация.

3.4 Имплементация на потребителски интерфейс

3.4.a Имплементация на главно меню

Главното меню се намира в собствена Unity сцена. То се състои от бутони, които като се натиснат, изпълняват предназначението им и сменят на сцената със симулацията.



Фиг 3.4.1. Главно меню с опции за зареждане на симулация от файл или генериране на нова симулация с настройки.

В сцената с главното меню има скрипт на име UIManager, който изпълнява съответните функции при натискането на бутон и зарежда следващата сцена:

```
public void Generate()
{
    PlayerPrefs.SetString("Load method", "generate");

    // pass generation settings using PlayerPrefs API
    foreach (var (name, val) in sliderValues)
        PlayerPrefs.SetInt(name, val);

    SceneManager.LoadSceneAsync(1);
}

public void LoadSimulation()
{
    PlayerPrefs.SetString("Load method", "file");

    SceneManager.LoadSceneAsync(1);
}
```

Фиг 3.4.2.

Бутона за зареждане на симулация на файл има допълнителен скрипт, който също слуша за натискане и при такова, изпълнява функцията LoadSimulation.LoadSimulationFromFileSystem(). За реализацията на това се използва пакетът StandaloneFileBrowser, който позволява използването на Windows-кия файл експлорър.

3.4.b Имплементация на heads-up дисплей



Фиг 3.4.3. Различни елементи от heads-up дисплеят в симулационната сцена:

В зелено – сегашното симулационно време;

В жълто – бутони за манипулиране на скоростта на времето;

В червено – бутоон за паузиране на времето;

В розово – графика за средното задръстване;

В синьо – резюмирани данни и статистики за симулацията

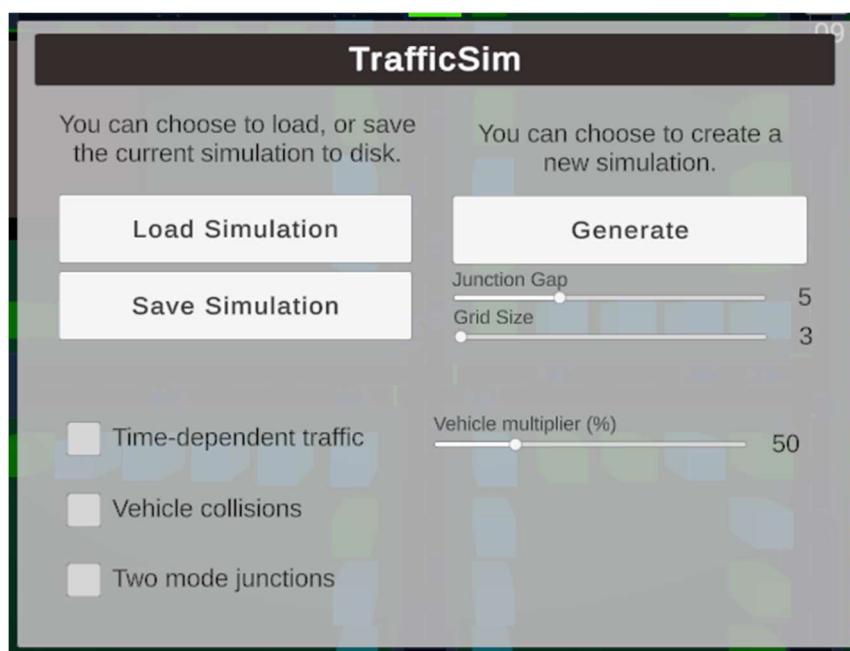
Всичките елементи от heads-up дисплеят са имплементирани със скрипт, който е закачен за тях. Един по един:

- Текстът на сегашното симулационно време е имплементиран в DatetimeText, като логиката е: четене от singleton-а Clock и записване на форматирана дата в текст полето в TextMeshPro текстов компонент.
- Бутоните за манипулиране на скоростта на времето са имплементирани в TimeScaleButtons. Всеки бутоон слуша за натискане и използва Clock класа да настрои timeScale. Бутоните могат да се навигират и с „<“ и „>“.
- Бутона за паузиране е реализиран с PlayPauseButton, като при паузиране се вика Clock.SetPaused() и се сменя Sprite-ът. Може да се паузира с „P“.

- Графиката е имплементирана в Graph, като открива един публичен метод за графиране на списък от стойности – Graph.ShowGraph().
- Последно, резюмираният данни се реализират в SimulationInfo, където се форматира стринг с всякакви данни и се записва в TextMeshPro компонент.

3.4.c Имплементация на меню за опции

Скриването и отварянето на менюто за опции става чрез деактивиране и активиране на съответния GameObject в сцената, като това отново се управлява от UIManager-а. То използва подобни елементи като главното меню – LoadSimulation, SaveSimulation и CustomSlider-и, както и CustomCheckbox-и, които взаимодействват с UIManager инстанцията. При натискане на бутона за генериране или зареждане на симулация от файл, за разлика от главното меню, сцената остава същата – просто се презарежда, а при променяне на настойките на симулацията, съответните променливи се актуализират. Менюто за опции може да се отвори и с „ESC“.



Фиг 3.4.4. Меню за опции в симулационната сцена.

Дава възможност за генериране на нов град, за запазване/зареждане на симулация като файл и за настройване на текущата симулация.

3.5 Допълнителни функционалности

3.5.a Имплементация на камера

Имплементирано е просто движение на камера, а по-конкретно – панорамиране и увеличение. Тези функционалности са имплементирани в View, който се състои от скорост на панорамиране (float panSpeed) и скорост на увеличение (float zoomSpeed):

```
public class View : MonoBehaviour
{
    private Camera cam;

    public float panSpeed;
    private Vector3 panOrigin;
    private Vector3 posOrigin;

    public float zoomSpeed;
    public float minCameraY = 100f;
    public float maxCameraY = 1000f;
```

Фиг 3.4.5.

Увеличението става, чрез колелото на мишката, а панорамирането – щракване и плъзгане.

3.5.b Имплементация на селектиране

Реализирано е селектиране на обекти от симулацията, а именно на автомобили и кръстовища. Скриптът имплементиращ това е Selection и той е закачен за главната камера, заедно с View. При селектиране на автомобил се визуализират: начална и крайна сграда, пътя на автомобила (чрез LineRenderer компонент) и данни свързани с автомобила, като време на чакане на червен светофар, а при селектиране на кръстовище се изписва: ниво на задръстване, изминалото време на TrafficController-а и данни свързани с различните светофари. Изписването на данните на автомобила или на кръстовищета става възможно през Tooltip класа и prefab-а.

3.5.c Имплементация на запазване и зареждане

- Клас PersistenceManager

Това е singleton MonoBehaviour, който е отговорен за зареждането и запазването на симулации. Симулациите се сериализират в SimulationData обект, който се записва като JSON файл на диска. Съответно сградите (Building) се сериализират с BuildingData, пътищата (Road) – в RoadData, кръстовищата (Junction) – в JunctionData и др.

Референциите между пътищата и кръстовищата се запазват като матрица на съседство с индекси, а свръзките сграда-път – със списък от индекси.

3.5.d Имплементация на осветление

Има и проста система за осветление, като тя се реализира от LightManager, който манипулира позицията на слънцето спрямо симулационното време. Цветовете на осветлението се определят от LightPreset, който е ScriptableObject.

Четвърта глава

Ръководство за потребителя

4.1 Локално инсталиране на проектът

Проектът е съхранен в GitHub хранилище, което се намира на <https://github.com/BvHand487/TrafficSimTest>, като то може да се свали чрез:

- git clone https://github.com/BvHand487/TrafficSimTest.git

или с просто сваляне на ZIP файла.

Създава се питонска виртуална среда с Conda:

- conda create -n ENV_NAME python=3.10.12

Активира се виртуалната среда с:

- conda activate ENV_NAME

Възможно е без Conda и виртуална среда, но инсталацията е възможно да стане по-сложна и непредвидим

Свалят се необходимите питонски пакети:

- pip install torch~=2.2.1 --index-url https://download.pytorch.org/whl/cu121
- pip install mlagents==1.1.0

Директорията ./TrafficSimTest се отваря като Unity 6 проект през Unity Hub.

4.2 Симулацията на трафик

След като проектът е успешно стартиран в Unity, потребителят може да използва симулацията на трафика.

- Стартриране на симулация

За да се стартира симулацията, е необходимо да се:

- Зареди сцената с главното меню.
- Натисне бутона Play
- Да се натисне или Generate бутона, който създава произволен град с параметрите в слайдерите или Load Simulation, при който се отваря файл експлорър и се избира файл завършващ на tsf.

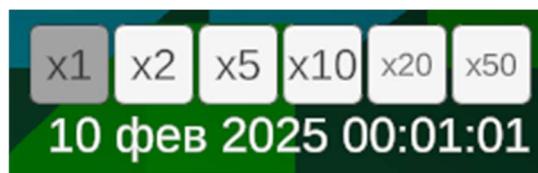


Фиг 4.1.

- Контрол на симулацията

Бутоните горе дясно позволяват контролиране на скоростта на времето:

- Бутоните може да се натискат или да се използва клавиатурата, като с „<“ времето се забавя, ако може и с „>“ времето се забързва, ако може.
- Текстът под бутоните показва симулационното време.



Фиг 4.2.

Зеленият бутон най-горе в центъра позволява паузирането на времето.

- Бутона за паузиране може да се кликне или задейства с „P“ на клавиатурата.



Фиг 4.3.

Меню с настройките е скрито меню, което може да се появи и скрие с „ESC“.

- Generate и Load Simulation бутоните изпълняват същата функционалност, като Generate и Load Simulation бутоните в главното меню. Важното е, че текущата симулация не се запазва автоматично, затова потребителят трябва да натисне Save Simulation, ако желае да запази тази симулация.
- При натискането на Save Simulation бутона се отваря файл експлорър и се избира име и мястото, на което да се запише файлът.
- Има и настройки за самата симулация, като: количество трафик зависещ от часа в денонощието, дали кръстовищата да работят в двата режима (Single и Double), мащабираща стойност за максималния брой автомобили и автомобилни колизии.



Фиг 4.4.

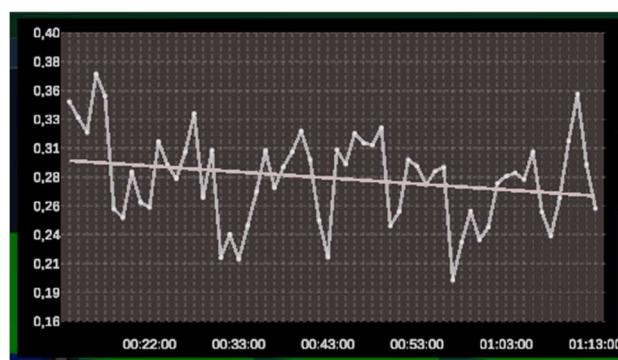
- Данни за симулацията

Текстът в лявата част на екрана показва обобщена информация за симулацията, автомобилите, сградите и т.н.

```
Vehicle Info:  
- Time-dependent vehicle amount: False  
- Vehicle collision: False  
- Vehicle multiplier: 50%  
- Max vehicles: 48  
- Simulated max vehicles: 48  
- Current vehicles: 21  
- Vehicles spawn queue: 27  
- HTW / WTH / RND: 40% / 40% / 20%  
  
Building Info:  
- Building count: 97  
- H / W: 67 / 30  
  
Road Info:  
- Two mode junction: False  
- Junction count: 5  
- Road count: 6
```

Фиг 4.5.

Графиката горе дясно показва средното задръстване измежду кръстовищата. За да почне да се чертае, трябва да се изчака малко време от началото на симулацията, за да може да се съберат данни, които да се покажат.



Фиг 4.6.

4.3 Обучение

Unity апликацията трява да е спряна в началото на тази стъпка. В основната директория на проекта има config.yaml, което е конфигурационния файл за тренирането. За да се стартира обучението, се отива в терминал, активира се виртуалната среда на conda и се изпълнява mlagents-learn:

- mlagents-learn config.yaml --results-dir=results --run-id=ID (--debug)

Резултатите от изпълняването на mlagents-learn би трявало да изглеждат така:



Фиг 4.7.

Когато програмата го изисква, се натиска Play бутона в Unity, зарежда се симулацията през файл (или с генериране), натиска се Train бутоњът горе дясно и тренирането започва, като е важно да се на променят настойките на симулацията докато се обучава модел.

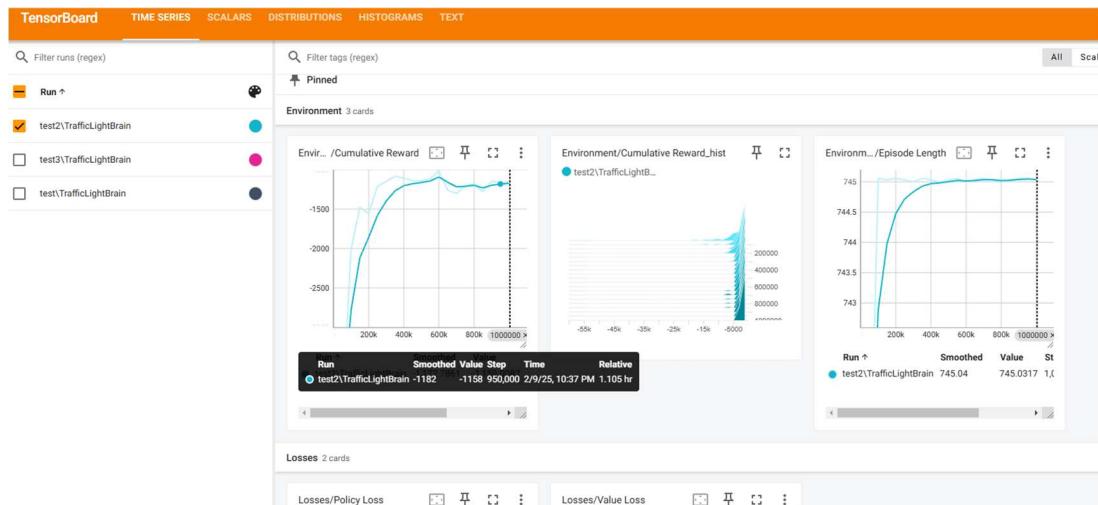
```
JG [learn.py:265] run_seed set to 5106  
JG [torch.py:59] default Torch device: cuda  
JG [stats_writer.py:51] Initializing StatsWriter plugins: default  
JG [stats_writer.py:54] Found 3 StatsWriters for plugin default  
D [environment.py:222] Listening on port 5004. Start training by pressing the Play button in the
```

Фиг 4.8.

По времето на трениране, терминалът вървящ процеса mlagents-learn и Unity апликацията трябва да останат отворени. За да се наблюдава тренирането може да се използва TensorBoard:

```
(TrafficSim) D:\Projects\TrafficSimTest>tensorboard --logdir=results
TensorFlow installation not found - running with reduced feature set.
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all
TensorBoard 2.18.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

Фиг 4.9.



Фиг 4.10.

За да се спре обучението, може да се прекъсне mlagents-learn процесът с CTRL + C. Моделът се записва в results директорията като файл с .onnx разширение.

За да се използва моделът в апликацията, може да се изпълнят тези стъпки:

- Спиране на апликацията;
- Импортиране на onnx модела като Unity asset;
- Закачване на модела за BehaviourParameters на децата на prefab-ите Junction3 и Junction4 в ./TrafficSimTest/Assets/Resources/Prefabs;
- Стартиране на апликацията и зареждане на симулация.

Заключение

Разработена е дипломна работа, която удовлетворява изискванията. Разработена е симулация на трафик, създадена на игровия двигател Unity, както и алгоритъм за самообучение с утвърждение, чрез който може да се тренират светофарите.

Бъдещото развитие на проекта е много широко, като някои от посоките, в които може да се развие са следните:

- Добавяне на повече елементи – кръгови, пътни ленти и повече видове сгради;
- По-голям спектър от участници на пътя – пешеходци и пешеходни пътеки, мотоциклети, автобуси, релсови транспортни средства и др.;
- Добавяне на аварийни ситуации – симулиране на инциденти и МПС-та със специален режим на движение;
- Моделиране на екологичен ефект и цени – възможност за измерване на CO₂ и анализ на цени за гориво.
- Добавяне на функционалността за импортиране на пътни мрежи от популярни формати като: OpenStreetMap (.osm, .pbf), OpenDRIVE (.xodr), CityGML (.gml) и др.;
- Метеорологични условия – влияние от дъжд, сняг, мъгла и различни сезони;

Литература

Съдържание

Увод	4
Първа глава	5
1.1 Технологии за създаване на симулация на трафик	5
1.1.a Игрови двигатели	5
1.1.b Програмни езици и библиотеки от ниско ниво	5
1.2 Съществуващи симулации на трафик	5
1.2.a SUMO	5
1.2.b PTV Vissim	6
1.2.c Aimsun Next	7
1.3 Съществуващи методи за оптимизиране на трафик	8
1.3.a Сигнали фиксирали във времето	8
1.3.b Задействане на сигнали	9
1.3.c Адаптиране на сигнали	10
1.4 Техники за създаване на адаптивни трафик системи	10
1.4.a Прости алгоритми базирани на правила	10
1.4.b Машинно обучение	11
Втора глава	12
2.1 Изисквания към софтуерния продукт	12
2.1.a Изисквания от заданието на дипломната работа	12
2.1.b Допълнителни изисквания	12
2.2 Технологии	13
2.2.a Игрови двигател – Unity	13
2.2.b Файлова структура на обикновен Unity проект	14
2.2.c Въведение към Unity – понятия и концепции	16
2.2.d Алгоритъм за оптимизиране на трафика - самообучение с утвърждение	18
2.2.e Въведение към самообучение с утвърждение	18
2.2.f Библиотека за самообучение с утвърждение - ML-Agents	20
2.2.g Въведение към ML-Agents	21
2.2.h Интегрирана среда за разработка – Visual Studio	22
Трета глава	24
3.1 Реализация на симулацията	24
3.1.a Имплементация на кръстовища и пътища	26
3.1.b Имплементация на сгради	28

3.1.c	Имплементация на светофари	30
3.1.d	Имплементация на автомобил	32
3.2	Реализация на генериране на произволен път.....	37
3.2.a	Теория на алгоритъма.....	37
3.2.b	Имплементация	43
3.3	Реализация на алгоритъм за обучение	47
3.3.a	Теория на алгоритъма.....	47
3.3.b	Имплементация	48
3.3.c	Изprobване на имплементацията	51
3.4	Имплементация на потребителски интерфейс.....	54
3.4.a	Имплементация на главно меню	54
3.4.b	Имплементация на heads-up дисплей.....	55
3.4.c	Имплементация на меню за опции.....	56
3.5	Допълнителни функционалности.....	57
3.5.a	Имплементация на камера	57
3.5.b	Имплементация на селектиране	57
3.5.c	Имплементация на запазване и зареждане	58
3.5.d	Имплементация на осветление	58
	Четвърта глава	59
4.1	Локално инсталиране на проектът	59
4.2	Симулацията на трафик.....	60
4.3	Обучение	63
	Литература	66
	Съдържание	67