Пловдивски Университет

Факултет
"Математика и Информатика"

Дипломна работа

Оптимизиране на транспортната мрежа в град Пловдив

Дипломант:

Венелин Вълков

фак. № 0901261093

Научен ръководител:

гл. ас. д-р Ангел Голев

кат. „Компютърни технологии“

03. 07. 2012 г.

Съдържание

Въведение **3**

1 Създаване на системата **6**

1.1 Проучване 6

1.2 Моделиране 16

1.3 Използвани технологии 24

1.4 Реализация 30

2 Ръководство за потребителя **38**

Заключение **41**

Списък на фигурите **45**

А Използвани съкращения **47**

Б Библиография **48**

Резюме

Настоящата дипломна работа се състои от въведение, създаване на симулационна система, ръководство за потребителя и заключение.

Във въведението се описват проблемът и целта на дипломната работа. За постигане на целта са посочени задачи, които трябва да бъдат изпълнени.

Основната задача е създаване на симулационна система. Тя съдържа четири основни части:

- *Проучване* - кратко разглеждане на съществуващи решения и симулационни системи
- *Моделиране* - запознаване със същността на моделирането
- *Използвани технологии* - преглед на използваните технологии за изграждане на решението
- *Реализация* - описва процеса на създаване на системата

Ръководство за потребителя съдържа описание на основните функционалности на системата, как могат да се ползват и какво може да се променя. Това е допълнено с екрани от програмата.

В заключението се предоставят резултати от наблюдаваните експерименти и система, решаваща поставените задачи. Описват се приносите на автора и проблемите, срещнати по време на разработката на програмата. Накрая се дават препоръки за възможни подобрения.

В края се предоставят списъци от библиография, използвани алгоритми, фигури и използвани съкращения.

Въведение

Искали ли сте да прекарате повече време с детето си на закуска, да се приберете по-рано от работа и да прекарате време с любимия човек или да излезнете с приятели? Може и да поспите повече, разбира се. Колко хубаво би било това да е правило, вместо изключение.

Всеки пловдивчанин, използващ пътната мрежа на града, губи средно около 60 мин. на ден в трафика. Времето, необходимо за достигане на квартал Тракия от Нова сграда на ПУ е 15 мин, при липса на задръствания. Това време се увеличава до 40 мин в периодите около 18:00-19:00 ч. Времето, прекарано в условия на трафик се отразява сериозно на здравето ни. [HEI] Би било добре, това време да бъде прекарвано по друг, по-приятен начин.

Към проблема е подхождано по множество различни начини, много от които предоставят реални и действащи решения. Генерално решение би подхождало при условие, че то може да бъде модифицирано в много голяма степен, така че да пасва на локалния проблем. Нуждата от локално и много специализирано решение, обаче, подсказва липса на генерално, добре пасващо такова.

Цел

Целта на дипломната работа е да предложи среда, в която да се изследват стратегии за намаляне на времето, прекарано в пътната мрежа на град Пловдив.

Изисквания

- Цената за използването на градската мрежа трябва да остане същата или да е по-ниска.
- Повишаване на комфорта и спокойствието при използването на градската мрежа.
- Намаляне на вредните емисии във въздуха.
- Намаляне на броя катастрофи.

Постигане на целта

За постигане на целите на дипломната работа, спрямо поставените изисквания, се разглеждат:

- Модерни технологични решения в подобни ситуации.
- Проучване на съществуващи симулационни системи.
- Създаване на симулация за намиране на критичните точки от транспортната мрежа и тяхното оптимизиране.
- Получените резултати се анализират, за да се получи частично или пълно решение на проблема.

Целеви групи

Хора, които използват транспортната мрежа, през най-натоварените часове на денонощието. Важно за всеки от участниците е бързото достигане на съответна точка от града, без това да пречи на личното му здраве и комфорт.

Желан резултат

Дипломната работа може да се сметне за успешна, когато бъде създаден прототип на система, който да позволява изследването и намалянето на трафика на града. Системата трябва да бъде лесна за промяна и разширяване, така че да предоставя реална подкрепа при разработване на начини за оптимизация на трафика.

Задачи

За постигане на желания резултат е необходимо следните задачи да бъдат изпълнени:

- Проучване върху методите за изграждане на симулации
- Избор между съществуваща и специализирана за целта система
- Избор на програмен език
- Избор на технологии
- Програмиране на самата система
- Провеждане на експерименти
- Отчитане на резултатите от експериментите
- Препоръки за усъвършенстване на системата

Глава 1

Създаване на системата

1.1 Проучване

За решаване на голяма част от проблемите, описани по-горе, се разработват и използват Интелигентни транспортни системи (Intelligent Transportation Systems) (ИТС) в държави като Япония, Сингапур, Южна Корея, САЩ и други. Употребата им води до значително нарастване на производителността, включително намаляване на задръстванията, подобряване на сигурността и удобството по време на пътуването [Ezell]

1.1.1 ИТС

ИТС черпят своя "интелект" от ситуацията на пътя, който наблюдават. Участници в пътната обстановка са превозни средства, пешеходци, пътни платна, кръстовища, светофари и други. Основната цел на интелигентните системи е да позволи на всеки наблюдаван участник да достигне желаната дестинация - бързо, сигурно и удобно.

Приложения на ИТС

Системи за уведомяване при аварирали превозни средства (Emergency vehicle notification systems), позволяват автоматично или ръчно свързване с

оператор, когато се случи инцидент. По време на обаждането, освен гласовите данни, се предава информация за времето на инцидента, вида му, местоположението, идентификацията на превозното средство и др.

Автоматично засичане на нарушения на пътищата (Automatic road enforcement) е система състояща се от камера и устройство за наблюдаване на превозни средства. Използва се за засичане на нарушители на ограничението на скоростта, преминаване на червен светофар, неправомерно използване на пътни платна, предназначени за автобуси, неправилно изпреварване и др.

Вариращи ограничения на скоростта (Variable speed limits) се използват, когато има налично задръстване. В такива ситуации се променят минималната и максималната допустима скорост. Най-често се използват при наличие на отсечка от пътя, която се употребява от множество превозни средства.

Основополагащи технологии за ИТС

Глобална позиционна система (GPS) устройства се поставят в превозните средства, за да изчисляват текущото местоположение. Точността е до 10 м. Държави като Холандия и Германия, я използват за изчисляване на изминатите разстояния от отделните превозни средства. Това спомага по-точното таксуване при използване на пътищата.

Специални съобщения в малък обсег (DSRC) е канал за комуникация, оперираш на 5.8 или 5.9 гигахерцова честота, специално разработена за превозни средства. Най-важната особеност на системата е позволяването на двупосочна комуникация между превозните средства и апаратурата до пътя. Каналът позволява и директна комуникация между участниците в пътното движение.

Безжични мрежи (Wireless Networks) позволяват комуникация на устроиства до няколко стотин метра. Този проблем се решава чрез добавяне на възможност за всеки участник в пътното движение да бъде приемник и изпращач на данни. Използва се същата технология, която използват безжичните домашни рутери.

Мобилна телефония (Mobile Telephony) се използва от ИТС за трансфер на информация по мобилни мрежи от трета и четвърта генерация. Предимство на тези мрежи е наличието им в градовете. При използване на мобилна мрежа се наблюдават увеличение на изпратените и получените данни, които трябва да

бъдат покрити от мобилните оператори или потребителите. При спешна нужда за бързо прехвърляне на голямо количество информация, мобилните мрежи може да не покрият изискванията.

Преглед на ИТС

Интелигентна пътно-информационна система (IRIS) е проект с отворен код, разработен от отдела за транспорт на Минесота, САЩ. Използва се за наблюдение и обработване на междущатски и магистрален трафик.

Нуждата за създаване на IRIS идва от факта, че предишни системи са разработени с модули, използващи затворен код и висока цена за използването им. [Darter]

Някои от възможностите на системата са:

- Наблюдение и контрол на вариращи скоростни ограничения
- Контрол на пътните платна
- Засичане на задръствания
- Автоматизирана система за предупреждения
- Камери за наблюдение и контрол на трафика

Системата е успешно интегрирана в Минеаполис, САЩ.

MITSIMLab е проект с отворен код, разработен от Масачузетския технологичен институт. Той представлява симулационно-базирана лаборатория, която е разработена за разглеждане на ефектите от различни начини за управление на трафика.

Софтуерът се състои от три основни модула - микроскопичен симулатор на трафик, мениджър на трафика и Графичен потребителски интерфейс (Graphical user interface) (ГПИ).

Системата е успешно интегрирана в Стокхолм, Швеция. Тя се грижи за интелигентната промяна на сигналите, подавани на светофарите и даване на приоритети при наближаване на автобуси.

MITSIMLab е избрана, защото предоставя много възможности, които я правят реалистичен симулатор на обстановката на пътя. Дизайнът на системата е високо модулизиран. Това предоставя възможност за лесна променя и добавяне на допълнителни елементи при създаване на различни сценарии. [Rehunathan]

1.1.2 Нужна ли е ИТС?

Има няколко модела за създаване на транспортни симулации. За постигане на целите, изложени по-горе, ще се съсредоточим върху микроскопични модели, които показват поведението на различните превозни средства и симулация на групи от такива. Проучвания показват, че такива модели показват реалистични резултати [Nagel].

На базата на тези познания се отхвърля нуждата от създаване на сложна ИТС. Това, което ни трябва, е начин за достигане на някаква дестинация по бърз, лесен, удобен и безопасен начин. Използване на симулация е основен инструмент при анализ на пътен трафик. Още повече, наличието на градски транспорт води до значително намаляне на разходите, риска и вредните газове за отделните участници.

1.1.3 Какво е моделиране?

Моделирането е процесът по създаване на модел. Моделът е представяне на съществуващ обект от анализираната система. Той е близък, но по-прост от обекта в реалната система. Когато създава модел, анализаторът използва модела, за да предскаже промените в системата. От една страна, моделът трябва да е близък до реалния обект и притежава неговите свойства. От друга страна, трябва да е лесен за разбиране и променяне. Добрият модел е добър компромис между простота и реализъм [Anu]

Препоръчва се, усложняването на модела да става итеративно (iteratively). Важно е, моделът да продължава да бъде верен през този процес. Някои техники за това включват:

- Симулиране, включваики модела, с познати входни и изходни данни

- Пресечено валидиране (cross-validation) [Mahoney]

В зависимост от средствата, използвани за построяването им, моделите биват: физически (обекти, процеси и явления, евентуално различни по физическата си природа от оригинала, но с аналогични свойства), математически (теории, методи и обекти – функции, уравнения, редове и други), информационни (информационни методи, обекти и процеси), компютърни (програми, данни и други) и така нататък [Totkov]

Компютърното моделиране на информацията и автоматизирането на информационните дейности предполагат въвеждането, изучаването и използването на различни модели на информацията за обектите и за дейностите, в които те участват. Информационните обекти и информационните процеси са абстрактни модели на информацията и информационните дейности, наречени абстрактни (концептуални) информационни модели. В компютърната информатика абстрактните информационни модели се проектират и създават под формата на алгоритми и структури от данни. [Totkov]

Симулационните компютърни модели представляват интерес за научните и стопанските дейности. Компютърният модел дава възможност за разкриване на структурата, механизма и закономерностите, на които се подчиняват различни природни феномени или проблеми от реалния свят. По този начин е възможно прогнозиране, управление или изкуствено възпроизвеждане на тези модели, което намалява средствата, влагани в експериментални изследвания. [Iliev]

1.1.4 Какво е симулация?

Симулация е процес, при който се извършват различни действия/експерименти върху построения модел, вместо върху реална система. В най-общото си значение, симулация е инструмент за наблюдаване на това как една система работи. Симулацията може да се тества под различни условия и за различни времеви диапазони.

Симулация може да се използва, когато искаме да:

- Намалим риска за недостигане на поставен срок
- Намираме непредвидени пречки и проблеми

- Оптимизираме поведението на системата

Видове:

- Непрекъсната симулация (Continuous simulation) (НС) променя състоянието си в неопределени моменти във времето, използва диференциални уравнения
- Дискретно-събитийна симулация (Discrete-event simulation) (ДСС) променя състоянието си в определени моменти във времето, използва събития

1.1.5 Визуална симулация

Интересно за поставената цел е, че е необходима направата на визуален компонент за системата. По този начин, лесно ще може да бъдат наблюдавани различни движения на обектите. Това може да доведе до по-успешна и лесна оптимизация на цялостната пътна мрежа в града.

Такъв тип симулация налага използването на знания от компютърната графика. Още повече, доближава ни (и дори довежда) до използването на системи за съставяне на компютърни игри. Изненадващо, игрите не са разглеждани задълбочено, или поне тяхната структура и начин на работа, в академичните среди [Holzkorn].

Визуалната част на системата е критична за това дали системата ще бъде използвана. Лесното и удобно използване са важни за всеки софтуерен продукт. Потребителят е този, за който системата ни трябва да се грижи добре. [Microsoft]

Следва разглеждане на някои от основните и най-добри симулационни системи[1]. Целта е да се направи най-добър избор на такава, която е в съответствие с поставените ни задачи и изисквания.

1.1.6 Преглед на системи за създаване на симулация

Simplex3 е система за симулации, която може да работи под Windows и Unix операционни системи. Тя е универсално приложима, когато имаме диск-

[1] `http://en.wikipedia.org/wiki/List_of_computer_simulation_software`

ретни модели, процеси лесно моделируеми с помощта на опашки или транспортни модели.

Позволява лесно и бързо научаване на системата, като това не изисква научаването на нов програмен език от високо ниво.

Притежава собствен език за създаване на модел, наречен Simplex-MDL. Той позволява описанието на почти всякакъв вид модели. Въпреки това, модели, използващи частни диференциални уравнения не могат да се представят лесно.

Благодарение на универсалността си, Simplex3 може да се използва за академични цели. Лесно може да се приспособи за случаи, в които няма разработен специализиран симулационен софтуер.

От сайта на системата, не става ясно до колко тя е поддържана. Липсва добра документация. [Simplex3]

AnyLogic е платена система за общ вид моделиране и симулационен инструмент за дискретни, непрекъснати и хибридни системи.

Системата предоставя ГПИ за създаване на моделите, които допълнително могат да бъдат разширявани с помощта на програмният език Java. AnyLogic предоставя моделиране чрез UML-базирано обектно-ориентирано моделиране, блок-схеми, диаграми на автомати, диференциални и алгебрични уравнения и други.

Предоставя възможност за създаване на Java аплети, които позволяват лесно споделяне на симулациите, както и поставянето им в интернет.

Моделиращият език на системата е разширение на UML-RT - голяма колекция от най-добри практики, доказали се при моделирането на големи и сложни симулации.

Тя е полезна, когато искаме да разглеждаме симулации относно:

- Контролни системи
- Производство
- Телекомуникации
- Обучение

- Логистика(Logistics)
- Компютърни системи

AnyLogic е доста обширна и понякога тежка система. Предоставя голям набор от полезни инструменти. Тя е *много скъпа*[2], което е в разрез с изискванията ни. [AnyLogic]

Tortuga е симулационна система с отворен код, разработена от MITRE Corporation [3] в периода от 2004 до 2006 година. Тя е ориентирана изцяло към дискретно-събитийни симулации.

Симулация в системата може да се представи като взаимодействия между процеси или планирани (scheduled) събития. Системата изисква множество от написани (от потребителя) класове на Java, инсталиран Java средства за разработка (Java development kit) (JDK), Ant - инструмент за автоматично билдване (build) и самата симулационна система. Предоставя се и ГПИ за управление и наблюдение на самата симулация.

Tortuga е изцяло написана на Java. Това прави възможно използването ѝ под Windows и Unix базирани системи. Интересното тук е, че за постигане на своите цели Tortuga използва AOP (Aspect Oriented Programming) [AOP], като не се изисква от потребителя да има познания в тази насока, стига да компилира програмата си по специфичен начин.

Системата използва програмна парадигма, която многократно намаля сложността при създаване на симулация. Тя третира всяка единица от симулацията като отделна нишка. Java виртуалната машина ограничава броя на нишките [4], което води до ограничение броя на симулационните единици.

Tortuga може да бъде използвана самостоятелно или като част от по-голям проект. Тя предоставя споделяне на симулации в Java аплети, също както и AnyLogic.

Лицензът който използва е Lesser General Public License (LGPL)[5], което прави системата добър кандидат за по-нататъшно разглеждане. Възможността

[2]Цената на най-евтиното издание на продукта е 4800€

[3]`http://www.mitre.org/`

[4]Броят варира спрямо имплементацията, настройки при пускане и хардуера

[5]`http://www.gnu.org/licenses/lgpl.html`

й за използване под много платформи (благодарение на Java виртуалната машина) е още един голям плюс. Не трябва да забравяме използването и на *AOP*, което я изкарва от "стандартна"Java програма. Според сайта на системата, от 2008 година, поддръжката за нея е спряна. [Tortuga]

SimPy е система с отворен код, създадена през 2002 г. от Klaus G. Muller и Tony Vignaux. След това към тях се присъединяват много други разработчици. Тя предоставя възможност за създаване на дискретно-събитийни симулации.

Симулациите в системата могат да се представят като взаимодействия между процеси. Те се изграждат чрез програмен код. За създаването на проста симулация се изискват около 10 реда код, които включват познания по основни класове и функции от системата. Има GUI(използващо библиотеката Tk[6]) за предоставяне на входни данни и наблюдаване на симулациите.

SimPy е написана на Python[7] и поддържа версии от 2.3 до 3.2 включително. Благодарение на Python виртуалната машина (при използване на стандартната CPython имплементация[8], SimPy може да се изпълнява под Windows и Unix операционни системи.

Системата използва обектно-ориентиран подход към създаване на симулации. Благодарение на езика, на който е написана, с нея бързо може да се направи дори и по-сложен модел.

SimPy може да бъде използвана самостоятелно, както и навякъде, където има Python интерпретатор. Предоставя се набор от пакети за изобразяване на графики и управление на структури от данни, които могат да бъдат използвани извън рамките на библиотеката. Готови симулации лесно могат да бъдат пакетирани като изпълними файлове и предоставени на други потребители.

Лицензирана е под LGPL лиценз. Системата изглежда добре поддържана. Последната версия е 2.3, излезнала през Декември 2011 г. Използва лесен и достъпен език, но без много допълнителни инструменти за по-бързо и лесно създаване на симулация. [SimPy]

[6] `http://www.tcl.tk/`
[7] `http://www.python.org/`
[8] `http://wiki.python.org/moin/CPython`

GarlicSim е безплатен продукт с отворен код[9], който се опитва да предостави нова концепция за това как всъщност може да се използват симулациите. Автор на системата е Ram Rachum[10].

Проектът обещава да напише повтарящата се част от кода, необходим за една симулация, вместо нас и ни оставя да работим върху по-важната част от проекта - самата симулация. Самата тя се създава чрез писане на програмен код. Може да постигнем направата на проста симулация с 5 реда код!

GarlicSim е изцяло написана на Python и официално все още не поддържа Python 3.x сериите. Системата предлага изчистен и лесен GUI за управление и наблюдение на симулациите.

За създаване на системата е използван изцяло обектно-ориентиран подход, който допринася за лесното научаване на основните стъпки при правене на симулация. Основна дейност при извършване на симулация е т.нар. симулационен пакет (simulation package), който съдържа функция, която определя стъпката за дадената симулация.

Според автора, системата е достатъчно обща, за да позволи симуларинето на каквато и да е симулация. Дадени възможности са:

- Физични
- Теория на игрите
- Разпространение на епидемии
- Квантова механика
- Електрически

Лицензирана е под LGPL лиценз. GarlicSim е един чудесен продукт за начални опити за създаване на симулации. Основен проблем е, че е в алфа версия. Няма сведения за успешни употреби в някаква насока. Има добра, но не напълно достатъчна документация. За създаване на собствена симулация е необходимо написването на симулационен пакет, което до някъде обезсмисля обещанието за това да пишем по-малко код. [GarlicSim]

[9] `https://github.com/cool-RR/GarlicSim`
[10] `http://ram.rachum.com/`

1.2 Моделиране

След анализа върху съществуващите симулационни системи изглежда, че система, която да реши поставените проблеми, не съществува. Повечето от тях имат примитивни ГПИ, които не предоставят необходимата гъвкавост на точно определена създадена за целта такава.

Под голям въпрос е и качеството на документацията, както и състоянието на наличните примери и качество на кода. Това оставя възможността да бъде разгледан един нетрадиционен вариант за решаване на такъв проблем.

За създаването на нашия модел и цялостната програма ще се използва итеративен подход [ArtOfAgile]. Това означава, че ще изградим основните части от системата - първо[11]. Допълнителна функционалност ще добавяме само когато сме доволни от качеството на текущата програма.

Основната част от програмата ще контролира заложените закони и взаимодеиствия между обектите. При всяка нова стъпка от симулацията, обектите участващи в нея ще предоставят информация за състоянието, в което се намират. То може да се променя, като това е изразено в промяна на техните свойства.

Необходимостта за изобразяване на самата симулация, по време на изпълнение и когато тя вече е приключила, води до някои допълнителни ограничения. Системата трябва да работи достатъчно бързо[12], за да предоставя гладка и лека работа за потребителя.

С така поставените ограничения и подходи, използването на ДСС предоставя:

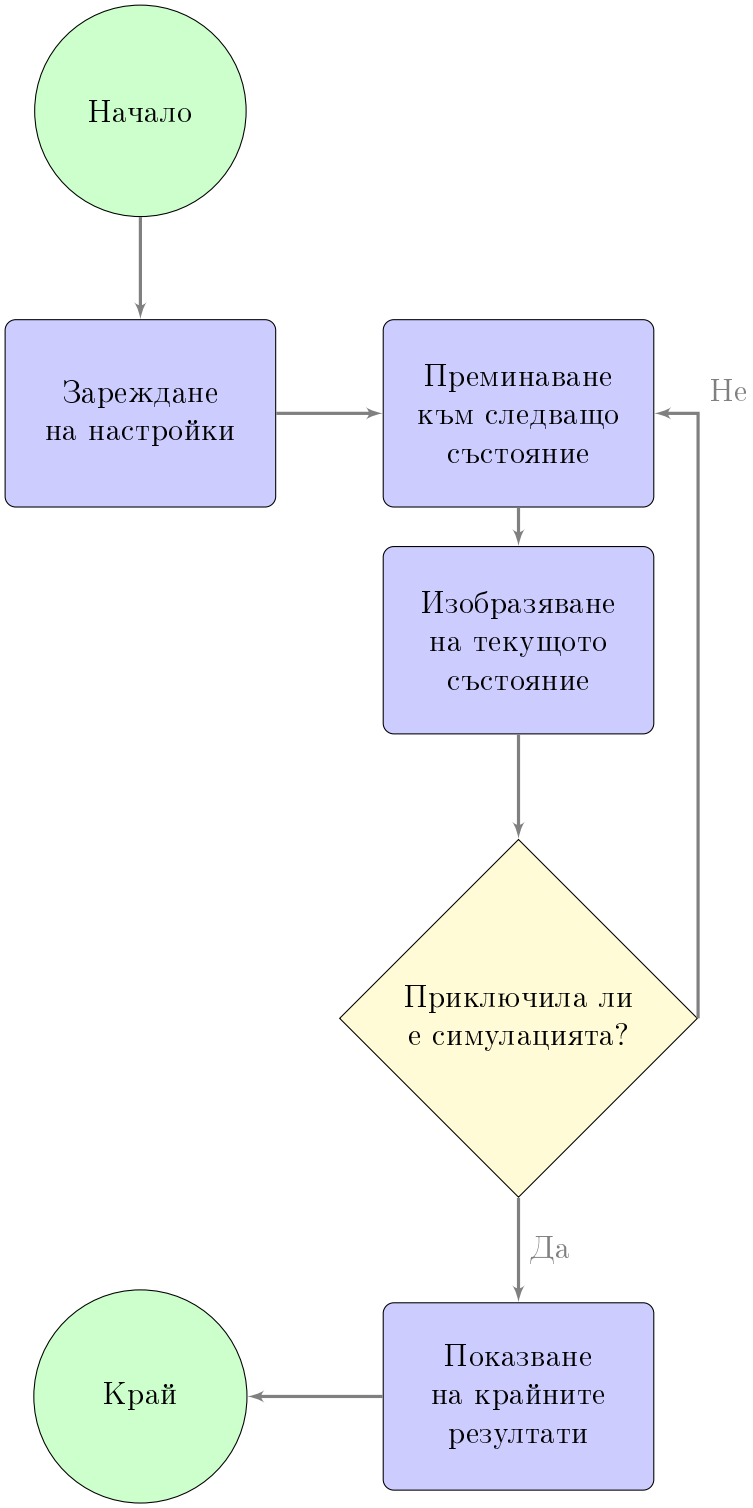
- Гъвкавкост
- Лесна имплементация
- Лесна промяна
- Лесно скриване на детайлите от самия потребител

[11] Посоченият подход е лична интерпретация
[12] Между 30 и 60 кадъра в секунда

Още повече, за симулация на трафик е необходима симулация от тип опашка, която в повече от случаите се имплементира с помощта на ДСС [Barlas].

За настройка на симулацията, потребителят ще трябва да отвори и промени конфигурационен файл. Когато симулацията започне, на потребителя се дава възможност визуално да наблюдава симулацията на всяка своя стъпка. След приключването ѝ се предоставят данни за това как е протекла симулацията. На фиг. 1.1 е изобразена диаграма, която показва работата на така зададения модел.

Фигура 1.1: Модел на симулационната система



1.2.1 Компоненти на симулацията

Една ДСС се представя като поредица от случващи се събития. Всяко събитие в определен момент и породената промяна на състоянието на системата.

Основни изграждащи единици

Всяка ДСС се нуждае от логика, която да отговаря на това какво се прави, когато дадено събитие се случи. Трябва и да се определят точните времена, в които те ще се случат. Някои други компоненти обаче, също са задължителни:

- *Часовник* грижи се за запазването и променянето на текущото време за симулацията. Единиците, в които времето се измерва, се избират спрямо самата симулация. Благодарение на незабавното случвания на събитията, времето подскача (hops).

- *Участници в симулацията* са конкретни обекти, които взаимодействат със други такива и влияят пряко на резултатите на изпълнение. Определянето на това, какви точно трябва да бъдат те, зависи изцяло от модела. Тяхното количество в точно определен момент на симулацията, зависи от входните данни, както и текущата стъпка.

- *Списък на събитията* представлява подредено множество от събития, които още не били изпълнени. Много от тях не са предварително дефинирани, а са създадени в следствие на изпълнение на предшестващи ги събития. Събитие е описано чрез тип и време на случване.

  Понякога е необходимо да изпълняваме различни задачи, които се простират извън границите на един момент от времето. Обикновено това се моделира като поредица от събития. Те биват създвани като се използва диапазон от време.

  Системата може да поддържа еднонишково(singnle-threaded) и многонишково(multi-threaded) изпълнение на събития в даден момент. При втория вид може да се породят проблеми със синхронизацията между

различните участници в симулацията. От друга страна, той допринася за по-бързото й представяне на потребителя.

За имплементация на списъка със събития се използва приоритетна опашка. Приоритетът е времето за случване на събитието. Няколко алгоритъма за създаване на приоритетни опашки са се доказали като подходящи при създаването на ДСС [Jones], най-вече "разширено дърво"(splay tree).

- *Случайни числа* се използват за различни нужди по време на изпълнение на симулацията. Тя се нуждае от случайно генерирани променливи, в зависимост от модела. Използват се псевдо-случайни числа, вместо случайни числа, за да се използват еднакви данни, когато искаме да проведем симулация с еднакво поведение.

- *Статистики* се събират по време на изпълнение на симулацията. Тяхното естество отново е зависимо от модела на симулацията. Те представляват данните, които потребителят вижда след края на всяка симулация.

- *Условие за край* е това, което трябва да е изпълнено, за да приключи симулацията. При отсъствие на такова, теоретично, една симулация може да продължи безкрайно. Създателят на симулацията трябва да прецени какво точно да бъде то, за да може тя да бъде прекратена.

1.2.2 Архитектура

За да предостави възможност за лесно развитие и промяна на стратегиите за оптимизация, системата трябва да бъде лесно разширима и да базира работата си на интерфейси, а не конкретни имплементации. Това води до нуждата от създаването на интерфейс. Един задължителен такъв е модел.

Моделите се създават като преизползват вече съществуващи части и поведения от системата. Големите модели се създават като множество от други модели, някои от които са атомарни. Атомарните модели не могат бъдат композирани от по-малки такива. Те съдържат поведението на отделните елементи в симулацията.

Симулационният двигател работи като използва интерфейсите на двата вида модели. Това му позволява да се фокусира върху създаването на стратегии

за симулация, обявяване на събития и други задачи.

Визуализацията, взаимодействието и други функционалности са отговорности на софтуера, който използва симулационният двигател. Този софтуер се нарича *симулационнен клиент*. Той контролира как времето в симулацията напредва, като това може да е подобно на нормално течащото време или по някакъв друг зададен подход. Клиентът бива оведомяван, когато някой модел промени състоянието си и има нужда да бъде обновен на потребителския екран, събрана статистика, отстранена грешка и др. Последната основна роля на клиента е възможност за събиране на данни от потребителя и предоставянето им като вход на симулационния двигател.

Симулационният двигател за създаване на ДСС е най-лесният за имплементиране. Разширяването му до по-сложна система за симулации е лесно, щом основните компоненти на двигателя са създадени. Структурата на работа остава постоянна, създават се смесени и атомарни компоненти, които се използват при самото осъществяване на самата симулация. Симулациите основно се различават според алгоритъма, по който времето напредва по време на симулацията.

Атомарни модели

Атомичният модел е динамична система, която се променя спрямо обграждащата я среда. Промените в самия модел се отразяват и на обграждащата го среда. Съществува множество от входни променливи, които имат ефект върху модела - X. Множество от изходни променливи, които влияят на обграждащата го среда - Y. Състонието на системата се съдържа в множество от променливи - S.

Променливото поведение на системата е описано като функции от време до входни, изходни и множества, описващи състоянието. Множеството от времена, използвани от системата, е наречено база за времето (time base) и се бележи с N.

Траекториите биват три вида - входни, изходни и такива, обвързани със промяна на състоянието. Всеки вид съответства на X, Y и S. Стойността на траекторията z в специфично време t се записва като $z(t)$.

Траекториите се дефинират в интервал от време и се отбелязват като

$z < t_0, t_n >$, в общия случай. За прецизиране на интервалите се използват стандарните математически интервали т.е.

$$\begin{aligned} t \in [t_0, t_n] &\Longleftrightarrow t_0 \leq t \leq t_n \\ t \in (t_0, t_n] &\Longleftrightarrow t_0 < t \leq t_n \\ t \in [t_0, t_n) &\Longleftrightarrow t_0 \leq t < t_n \\ t \in (t_0, t_n) &\Longleftrightarrow t_0 < t < t_n \end{aligned} \tag{1.1}$$

Промяна на състоянието се осъществява чрез цялостната функция Δ, която премества системата от s до $s\prime$ като отговор на входната траектория $x[t_0, t_n)$

$$s\prime = \Delta(s, x[t_0, t_n)) \tag{1.2}$$

Не всички такива функции са цялостни, Δ трябва да удовлетворява

$$\Delta(s, x_1 \bullet x_2) = \Delta(\Delta(s, x_1), x_2) \tag{1.3}$$

$$\Delta(s, x[t_0, t_n)) = s \tag{1.4}$$

Уравнение (1.4) изисква от системата да не променя състоянието си в празен интервал от време. Практически е много полезно да се изисква

$$\Delta(s, t(h, x)) = \Delta(s, x) \tag{1.5}$$

Уравнение (1.5) показва, че реакцията на системата, при зададен вход, не зависи от времето, в който е бил даден той.

Algorithm 1.1 Итеративна процедура за изчисление на функцията за промяна на състоянието

```
s ← s_0
for t_k = t_0 → t_n do
    print(t_k, s)          ▷ Изписва на екрана времето и текущото състояние
    s ← F(s, x(t_k))
end for
```

Смесени модели

Този вид модели се състоят от две части: множеството от компоненти, от които е изграден, и описание на връзките между модела и съдържащите се в него модели. Изходът от смесените модели може да бъде свързан като вход за други компоненти. Входът за самия модел може да бъде вход за неговите вложени компоненти.

Смесените модели, също както и атомарните, имат множество от входни данни X и множество от изходни данни Y. Мрежата от модели има ролята на входна траектория, която създава изходна такава. Промяната от входни до изходни данни се извършва от вложените компоненти. Тяхното общо състояние определят и цялостното състояние на мрежата. Функцията за промяна на състоянието е дефинирана като функция от функциите за промяна на състоянието на отделните компоненти. Изходната функция има същата дефиниция.

Смесеният модел е запознат само с входа и изхода на вложените в него компоненти. Вътрешната имплементация на отделните модели е скрита. Това води до възможността, всеки смесен модел да бъде представен като атомарен такъв, който точно препокрива начина му на действие.

За да се даде ясна дефиниция на резултата, получен от смесен модел, е необходимо точно описание на неговите части. Самият модел се записва с N. Има множество от входни данни - X и изходни данни - Y. Множеството D съдържа всички компоненти на модела. За всеки модел $d \in D \bigcup U$ има множество от компоненти, които оказват влияние върху d.

Различните части на смесените модели са:

- X_n = входно множество
- Y_n = изходно множество
- D_n = множество от компоненти

Резултатът на смесен модел е атомарен модел, който се представя като мрежа от модели. Той има множество S_r от състояния, X_R множество от входни данни, Y_r множество от изходни данни, δ_r функция за промяна на състоянието и λ_r изходна функция.

Множеството от състояния на резултантната се създава рекурсивно с функцията *State*. Тази функция приема като аргумент модел и връща множество от състоянията му. В зависимост от типа на подадения модел, тя може да върне множество на атомарен модел или продукт от множествата на смесен модел. Функцията може да бъде изразена по следния начин

$$STATE(d) = \left\{ \begin{array}{ll} S_d & \text{ако } d \text{ е атомарен модел} \\ X_{d\prime} \in D_d & \text{ако } d \text{ е смесен модел} \end{array} \right\}$$

Множеството от състояния на резултата на смесения модел N е

$$S_r = STATE(N) \tag{1.6}$$

и резултатът и смесеният модел има еднакви входно-изходни множества

$$X_r = X_n \tag{1.7}$$

$$Y_r = Y_n \tag{1.8}$$

Това множество от дефиниции за създаване на атомарен модел е достатъчно за създаване на двигател за симулации. Въпреки това, софтуерът ще бъде организиран и реализиран по различен начин. Сравнително простите дефиниции ни показват какво трябва да върши един двигател за симулации, не точно как да го прави. От части, всяка ДСС е идентична, два симулатора с еднакви подадени модели като входни данни, трябва да предоставят идентични изходни данни.

1.3 Използвани технологии

1.3.1 Lua

Lua е малък скриптов език, който кара да се усмихнеш, когато го използваш. Според авторите му той е: [13]

- Бърз

[13] `http://www.lua.org/about.html`

- Лек
- Лесен за вграждане (embeddable)
- Доказан

Той е напълно безплатен за употреба. Разпространява се под *MIT*[14] лиценз. Доказал се е като добър избор за скриптов език за множество комерсиални игри[15].

Автори на езика са Roberto Ierusalimschy, Waldemar Celes и Luiz Henrique de Figueiredo. Те разработват езика в университета PUC-Rio[16], като част от нуждите за тяхната група от технологии за компютърна графика.

Името на езика идва от португалската дума Lua, която означава "Луна".

Езикът е разработен като инструмент за вътрешна разработка на софтуер, но по-късно е бил използван в няколко комерсиални проекти по света. В момента е най-широко разпространен в индустрията, отговорна за създаване на игри. [Ierusalimschy]

Lua е смесица между обектно-ориентиран, функционален и програмиране свързано с данни (data-driven development) подход към създаването на програми. Има лесен за научаване синтаксис и сравнително малко на брой концепции.

Имплементиран е като библиотека за програмния език *C*[17]. Поради тази причина, той няма концепция за"основна програма" (main program). Нуждае се от приемник (host), който извиква различни части от код, написан на *Lua*.

С помощта на *C* функции, *Lua* може да бъде пригоден за работа в различни, непредвидени от авторите му, области и да споделя вече съществуващия синтаксис.

Lua има дългогодишна история като вграден скриптов език използван в игрите. Производителността му е далеч по-добра от други скриптови езици, дори без имплементация на Точно на време компилатор (Just in time compiler) (ТВК). Написан е на *ANSI C* и е изключително лесно да се създаде връзка със съществуващи библиотеки на *C* и *C++*. Това става чрез указатели към

[14] `http://www.opensource.org/licenses/mit-license.php`
[15] `http://en.wikipedia.org/wiki/Category:Lua-scripted_video_games`
[16] `http://www.puc-rio.br/index.html`
[17] `http://en.wikipedia.org/wiki/C_(programming_language)`

функции, които нямат нужда от скъпа сериализация на параметрите, между *Lua* и *C*.

Lua предоставя високо детайлен контрол върху управлението на оперативната памет. Включва се постепенен Събирач на боклука (Garbage collector) (СБ), който може да бъде настройван или напълно изключван във важни за игрите моменти. Всичко споменато по-горе, заедно с възможността за създаване на богата логика, необходима за модерните игри, прави *Lua* добър избор дори и за най-тежките екшън игри.

Концепции

Lua е динамично типизиран език (dynamically typed), което означава, че само стойностите имат тип.

Има концепция за прихващане на грешките, която много се доближава до тази на изключенията от езици като C++ и Java.

Езикът предоставя и съвместни програми (coroutines). Съвместна програма в Lua представлява изпълнение на независима нишка.

Използването на класове в *Lua* е малко по-интересно в сравнение с езици като *C* и *Java*. То по-скоро напомня на прототипното наследяване в *JavaScript*[18]. Не се предоставят класове, всичко, което разработчика вижда, е по-лесен синтаксис за разработката на подобни на обектно-ориентирани системи. *Lua* не предоставя възможност за скриване на информация, т.е. всичко е в един "клас" и е достъпно от всеки.

Възможно е създаването на наследяване, нещо, което отново не се поддържа директно от езика. Проста имплементация би търсела несъществуващите полета на текущия обект в друг, предварително зададен от програмиста. [Figueiredo] Това позволява на всеки разработчик да създаде свой, собствен стил на писане на класове. Това може да е добра и лоша страна.

Имплементация

Lua използва една от първите регистрови виртуални машини в своята имплементация. Така се избягва необходимостта от неколкократното вмъкване

[18] `http://www.crockford.com/javascript/inheritance.html`

и премахване на стойности, което е необходимо при стек-базираните виртуални машини. Средният брой изпълнени инструкции, при използване на регистрови виртуални машини, е с 47% по-малко от този при стековите такива. [Khan]

За разлика от много други езици, *Lua* не предоставя масиви. Предоставени са "таблици", които са асоциативни и могат да бъдат използвани подобно на масиви. Езикът използва методи за засичане на това дали една таблица е използвана като масив. Ако случаят е такъв, то имплементацията на езика създава истински масив, за да оптимизира изпълнението на програмата.

Предоставят се анонимни функции. Те са първокласни конструкции в езика. Използва се новаторски подход, които запазва локалните променливи в стека и ги премества само когато те излезнат от обсега (scope) и се използват от вложени функции.

1.3.2 Moai

Lua е прекрасен малък език, но сам по себе си, той не доставя необходимите инструменти за лесно създаване на симулационна система. Намиране на библиотека предоставяща по-висока абстракция е задължително.

Една такава библиотека е *Moai*[19] (произнася се Moe-Eye). Тя е по-скоро двигател за игри (game engine). Разработката ѝ е започнала през 2010г. и е активно разработвана и до днес. Тя е с *отворен код*[20] и използва лицензът *CPAL*[21], който позволява да използваме софтуера за нашите цели. Разработена е от *Zipline Games*[22].

Moai е библиотека за разработване на игри върху множество от платформи, вариращи от десктоп до мобилни операционни системи. За основна графична библиотека се използва стандартът *OpenGL*[23], който позволява тази разнообразност. Поради добрата си свързаност със *C*, Moai може да постигне производителност, по-висока от тази на приложения написани на *Objective C* [24] за *iPhone*.

[19] http://getmoai.com/
[20] http://github.com/moai/moai-dev
[21] http://www.opensource.org/licenses/cpal_1.0
[22] http://www.ziplinegames.com/
[23] http://www.opengl.org/
[24] http://en.wikipedia.org/wiki/Objective-C

Moai разрешава много от проблемите, с които се сблъскват разработчиците на мултиплатформени игри във всекидневната си работа. Състои се от клиентска рамка (framework), която ускорява разработката на игри, и облачно-базирана (cloud-based) сървърна част, която спомага за създаване на игри, играни от 2-ма или повече играчи.

Обектният модел на *Moai* е изцяло написан на *C++* и използва префикс "MOAI". Тези обекти се изчистват от СБ, също както и тези в *Lua*. За това спомагат и т.нар. контейнери в *Moai*, които се грижат да не останат неизчистени обекти от паметта.

По време на създаването на *Moai*, авторите са искали да направят библиотека, която да помага много за лесното създаване на симулации[25]. Например, по подразбиране началото на координатната система е в центъра на екрана. Това позволява по-лесното позициониране на обекти като цяло, тъй като обектите които са в центъра на екрана са по-важни за наблюдателя.

Основната концепции на *Moai* са [Zipline]:

- *Мащабируемост. (Scalable)* Позволява игрите да бъдат играни от милиони играчи, без това да пречи на изживяването им. Работата на програмиста е да направи добра игра, за останалото се грижи *Moai*

- *Мултиплатформеност. (Cross-platform)* Позволява използването на един програмен език за създаване на игри на множество платформи. Предоставя единна обвивка, която намалява риска от нуждата за писане на платформено-зависим код.

- *Отвореност. (Open)* Осигурява лесна промяна на платформата, ако нуждите за текущия проект не са добре подсигурени от *Moai*. Кодът е отворен и всеки може да се възползва от това.

- *Бърза разработка. (Rapid development)* Позволява на екипите от разработчици, бързо и лесно, да променят и създават нови части, концепции и графики за своите игри. Предоставя се висока абстракция, за да не е необходимо повторно създаване на вече съществуващи концепции.

[25] `http://getmoai.com/moai-basics-series/moai-basics-part-1.html`

Облакът на *Moai*

Облакът от услуги на *Moai* е платформа за създаване, разполагане (deployment), управление и хоризонтално скалиране на онлайн компоненти за мултиплатформени игри. Голямата стойност, която облакът на Moai предоставя, е премахването нуждата за създаване и настройването на частна инфраструктура, която да има възможност за обработка на милиони потребители.

Логиката в облакът може да бъде написана на *Lua*, *C Sharp* или *JavaScript*. Това позволява на разработчиците да създават иновации в техните игри, вместо да са заключени в стандартизирано Интерфейс за приложно програмиране (Application programming interface) (ИПП).

Архитектурата на облака е 3-слойна. Тя предоставя оптимизации за хоризонтална скалируемост във всеки слой. Предоставя се възможност за разширяване на системата за добавяне на нови възможности и езици за програмиране.

Първият слой е входна точка, която приема множество уеб протоколи (HTTP, WebSockets, Flash Sockets и др) и ги трансформира до единен формат, който е познат на платформата. Това предоставя балансиране на натоварването (load balancing), проверка на съобщенията (message validation), събиране на статистики (metrics gathering) и др.

Вторият слой се грижи за събирането на съобщенията и насочването им към необходимите среди на програмните езици. Те представляват сигурни кутии (sandboxes), които не пазят състояния и изпълняват кода на разработчиците.

Третият слой представлява механизми за запазване на информация. Облакът на *Moai* използва споделени бази от данни.

Производителност

Основни проблеми при създаването на игри са ниската производителност, високата консумация на оперативна памет и размери на файловете. Производителността в игрите се измерва с разглеждане на кадрите за секунда и други метрики.

Игрите създадени с помощта на *Moai* могт да бъдат по-бързи от такива, написани на *Objective-C* или *Java*. Това се дължи на високо оптимизирания

C код и употребата на *OpenGL*. Двигателят на *Moai* автоматично запазва и минимализира заявките до *OpenGL*. Има вградена система за геометрично буфериране, която гарантира оптималната употреба на графичния хардуер, дори и за много сложни сцени. Графът на сцени, който *Moai* използва, е високо технологична имплементация, която намалява обработката във всеки кадър, само до променените обекти в сцената.

Много разработчици споделят, че *Moai* е перфектен избор за бързо разработване на производителни игри. Дори такива със сложна логика, показващи стотици хиляди ефекти са лимитирани само от графичния хардуер. Двигателят и игровата логика са толкова бързи, че единственото ограничение е самият хардуер. [Zipline]

1.4 Реализация

За да се спази итеративния подход към създаването на системата, по време на първата итерация се имплементират само някои от изискванията към системата. Също така, тези изисквания може допълнително да бъдат разбити на по-малки такива.

Първата версия има малък набор от функционалности, които, според автора, имат най-висок приоритет за изграждането на работеща програма, която може да бъде разширена лесно и бързо чрез минимални познания по системата и редове код. Версията има следната функционалност:

- Симулацията представлява пътно кръстовище с 4 посоки на движение.
- Всяка посока има само 1 пътно платно.
- Превозните средства могат да се движат само направо.
- Скоростта за движение е постоянна.

Посочените опростявания ще ни позволят да намерим грешките, които сме допуснали при създаването на предварителния модел на системата. Още повече, те ще ускорят създаването на първоначална версия на системата.

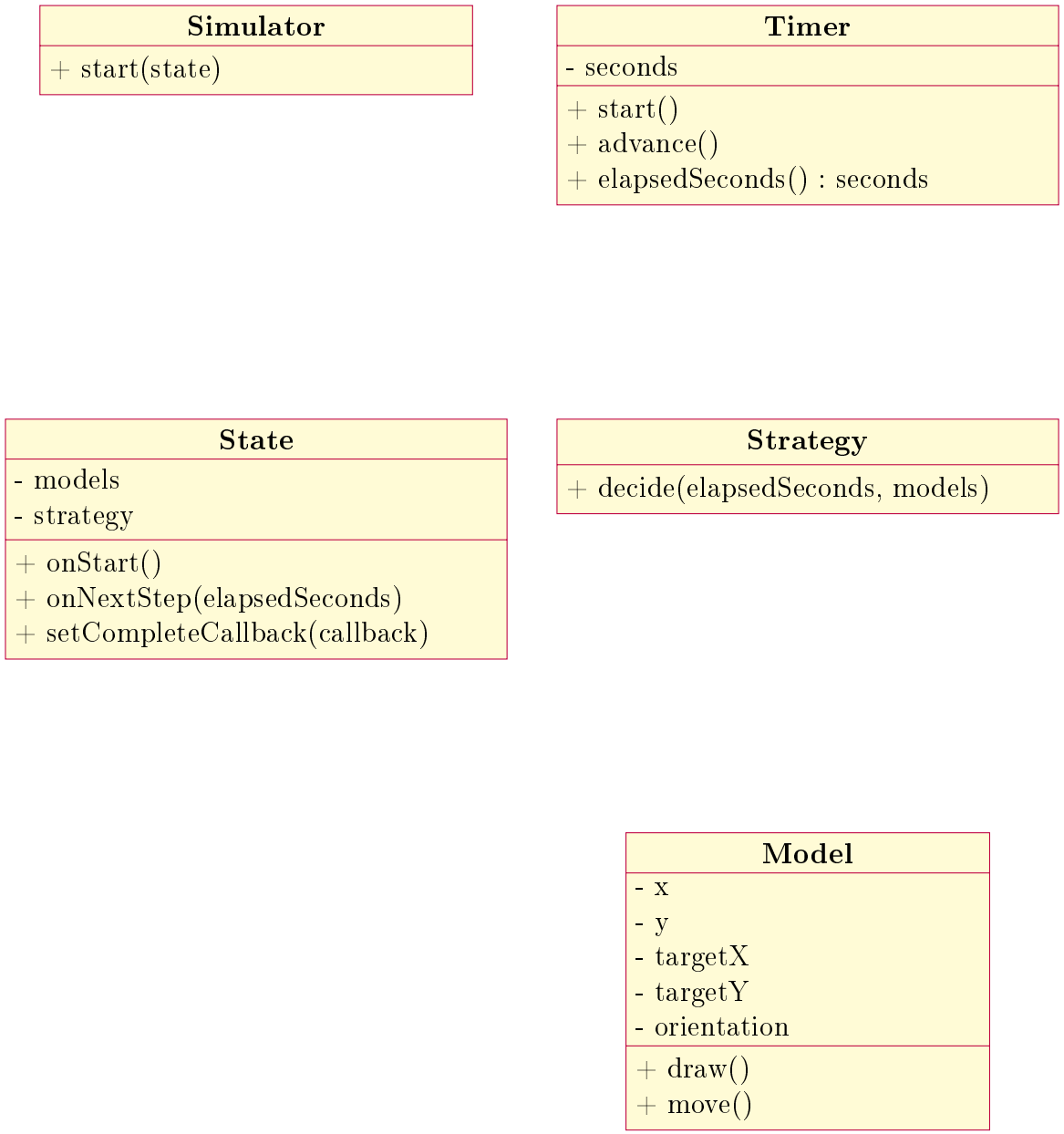
1.4.1 Итерация I

След оглед на по-горе посочените необходими функционалности се забелязва необходимостта от изграждане на проста и гъвката архитектура, която да ни позволи по-лесно добавяне на нови възможности. Добавянето на нови такива е задължително, поради избора на начина на разработка на симулационния софутер.

Опростенията, които бяха избрани, допълнително правят задачата по-бърза за имплементиране, но не трябва да се забравя, че те са само временни и задължително трябва да бъдат премахати в следващи фази от разработката. Така начинът, по който създаваме архитектурата на системата, не трябва да зависи от детайлите на специфичните за първата итерация възможности, а да разчита на по-абстрактната архитектура, описана в 1.2.2.

За целите на първата итерация, имплементираната архитектура е зададена като клас диаграма на фиг. 1.2. Представени са само основните класове, за да се представи основната идея на имплементацията, без да се разглеждат детайли, обвързани с изобразяването на отделните обекти и конкретни имплементации.

Фигура 1.2: Клас диаграма на симулационната система



Представената архитектура използва краен автомат за промяна на текущото състояние. Това много напомня на начина, по който се създава дизайн и имплементация за мобилни, настолни и др. видове игри. Концепция, доказала се при създаването на множество успешни видео игри. Тук, разбира се, тя е по-проста и олекотена, но запазва добрата основа, върху която може да бъде изграден по-сложен софтуер. По-обстойно, подобна архитектура е описана в [Rollings].

Симулатор

В основата на симулацията е класът *Simulator*. Неговата задача е да стартира симулацията и инициализира правилно началното състояние. За да направи това, е нужен и часовник, който се грижи за поддържане на изминалото време от началото на симулацията.

Когато текущото състояние приключи своята работа, симулацията трябва да предаде контрола върху следващото или да прекрати работата на програмата. Комуникацията се извършва чрез съобщение (и по-точно указател към функция), което се изпраща на симулацията, когато текущото състояние приключи работата си. Така, симулаторът, се грижи за стартиране на текущото състояние и стартиране на следващо, когато това се наложи.

Описаният по-горе часовник, като компонент на симулацията, базира времето си на изминалите кадри. Това е така, защото не е ясно дали компютърната система, на която тече симулацията може да поддържа оптимилания брой кадри за секунда. Алгоритъмът е показан на алг. 1.2.

Algorithm 1.2 Засичане на изминало време в симулацията

if $frames \bmod 60 = 0$ **then**
  $seconds \leftarrow seconds + 1$
  $notifySystemTimeChange(seconds)$
end if

Състояние

State представя едно текущо състояние от симулация. То има няколко основни момента от своето съществуване.

Когато бъде стартирано, се извиква методът *onStart()*. Той има за цел да инициализира текущото състояние и да го приготви за работа. Извиква се само толкова пъти, колкото състоянието се стартира.

Методът *onNextStep()* се извиква, когато симулацията е напреднала с една стъпка. Като аргумент се предава времето, което е изминало от стартирането на симулацията. Методът се извиква n пъти, където n е броят стъпки на симулацията.

onFinish() се извиква, когато състоянието се прекратява. Възможно е да се премахне този метод и всяко състояние да отговаря за приключването си, само. Това би направило невъзможно прекратяване на текущо състояние от симулацията и добавяне на допълнителни условия за приключване. Ефекти, които е добре да се избегнат. Въпреки това, има необходимост от начин за прекратяване на по-нататъшно обработване на текущото състояние. Така всяко състояние ще може да капсулира логиката за прекратяването си. Механизмът за това е извикване на метода *finish()*.

Стратегия

Този интерфейс е имплементация на шаблона за дизайн *Strategy* от книгата [Gamma], пригоден за нуждите на системата. Той предоставя основна точка за разширение на начина, по който симулацията може да работи.

Единственият важен метод е *decide()*. Той приема, като аргументи, текущото време и участващите модели. Негова задача е да определи кога и кои модели, участващи в симулацията, ще извършат определени действия.

Имплементацията на този интерфейс може да се променя лесно. Това става чрез създаването на нова, отговаряща на нуждите на новите изисквания и промяна на конфигурационна настройка.

Модел

Model има за цел да предостави интерфейс на атомарен или сложен модел. Основно звено в симулационната система.

Методът *draw()* се използва за изчертаване на модела върху екрана. Той използва предварително зададените атрибути - начална позиция, крайна по-

зиция и ориентация за да се изобрази правилно. Знанието на всеки модел за начина на собственото си изчертаване е опростяване на имплементацията, което в тази фаза от разработката не противоричи на избраната абстракция.

Методът *move()* се извиква, когато текущата стратегия реши, че точно този модел трябва да изпълни някакво действие. За нуждите на текущата реализация, той кара различните обекти да се придвижват, но това лесно може да бъде надградено, когато изискванията станат по-сложни.

1.4.2 Итерация II

След имплементиране на всички функционалности, изисквани от първата итерация, е ред да се изберат следващите по-важни такива.

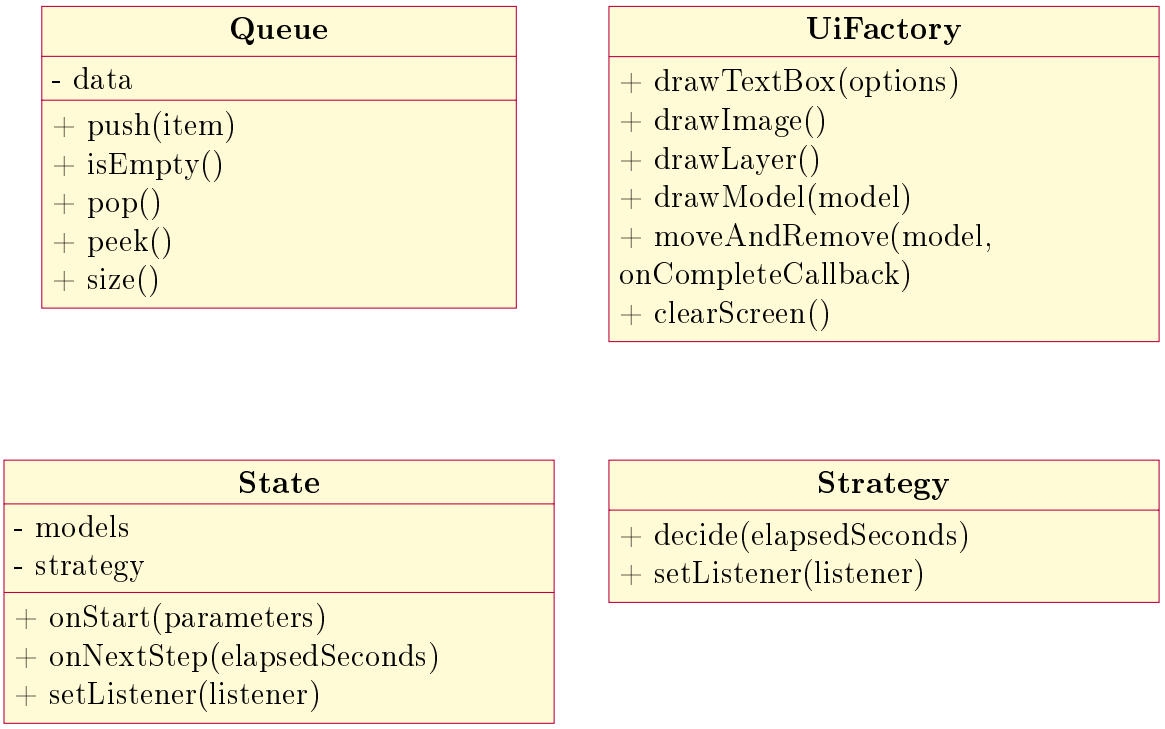
Нова функционалност:

- Представяне на статистика и различни метрики от проведената симулация.
- Моделите се подреждат на случаен принцип в опашки.
- Лесна конфигурация на системата.
- Приоритизиране на отделни модели

Последното от изискванията може да бъде лесно удовлетворено, чрез създаване на специфична стратегия, която да следи за текущите модели, които са в различните опашки. Примерна стратегия, даваща висок приоритет на автобуси от градския транспорт, е изградена във файла *bus-priority.lua*.

За целите на втората итерация, имплементираната архитектура е зададена като клас диаграма на фиг. 1.3. На диаграмата са представени само нови/променени класове. Особено впечатление прави запазването на основните класове и методи. Добавят се нови методи и параметри за предишни такива. Добавени са два нови, основни, класа. *Queue* - поддържа моделите в опашка и *UiFactory* - отговаря за изчертаването на различните елементи върху потребителския екран.

Фигура 1.3: Клас диаграма на новите елементи след 2-рата итерация



Състояние

В класът *State* са добавени механизми за предаване на параметри между различните състояния и лесен начин за изпращане на съобщения, за всички, които се интересуват от тях. И двата проблема са разрешени с помощта на указатели към методи/функции. За избиране на следващото активно състояние се грижи текущото такова.

Подреждането на моделите и тяхното създаване е грижа на състоянието, което работи с тях. Статистиката от проведената симулация се събира и предава на следващото състояние. Показването на статистиките става в ново състояние, което получава като параметри всички метрики, чрез механизма описан по-горе.

Опашка

В езика *Lua*, имплементация на *FIFO* структура от данни не съществува. Направената имплементация е стандартна с малко изключение. Вместо *nil*, когато в опашката няма следващи елементи, тя връща, като резултат, обект от тип *Optional*. Този обект може да има стойности по подразбиране и да бъде проверяван за това дали съдържа стойност или не. Това има много предимства, които са довели специалисти от *Google* да създадат подобен клас за своята библиотека *Guava*[26]. От там идва и идеята за имплементацията в системата.

Фабрика за Графични Елементи

За изграждане на ГПИ е необходимо писането на специфичен код, който се грижи за изобразяване на различни елементи върху екрана. Класът *UiFactory* предоставя абстракция за състоянията. Те може да използват предоставените статични методи за изграждане на отделни елементи върху екрана. Моделите, също използват методи на *UiFactory*, за да се изобразят и бъдат преместени. По този начин лесно може да сменим графичната библиотека, без това да налага промени по класове, отговорни за логиката на симулацията.

[26] `http://code.google.com/p/guava-libraries/`

Глава 2

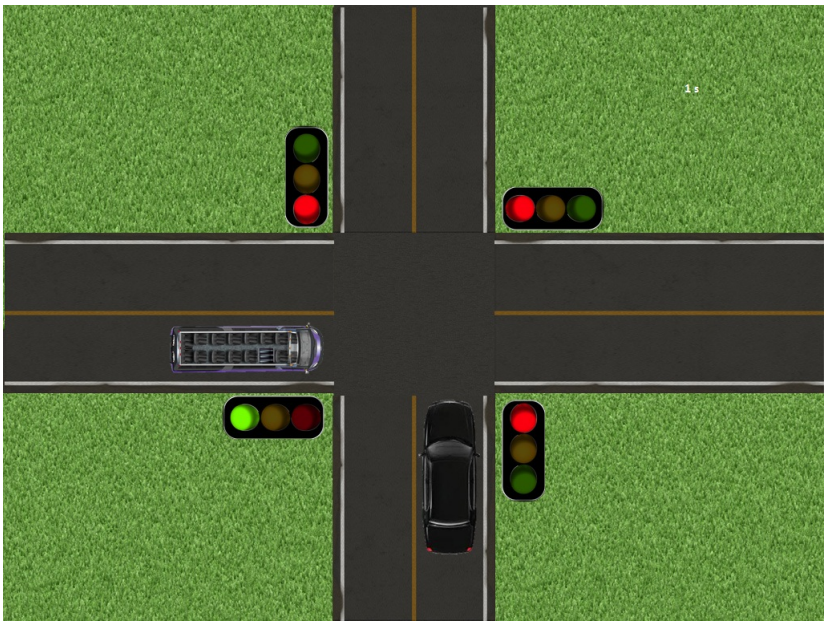
Ръководство за потребителя

За използване на системата е необходима библиотеката *Moai SDK*, версия в диапазона 1.0-1.2 и *Windows* базирана система. Стартирането става чрез файла "run.bat". При изпълнението на файла на екрана на потребителя се изобразява нещо подобно на 2.1. Изобразява се екран с модели, готови за работа и работеща симулация. Горе в дясно, потребителят, може да наблюдава изминалото време от симулацията.

След удовлетворяване на условието за край на симулацията, на екранът се изобразява статистика, представляваща различни метрики от симулацията. Подобна такава е представена на 2.2. Представената информация е:

- *Running Time* време, за което симулацията се е изпълнявала
- *Vehicles Passed* броят модели, които са успели да преминат кръстовището
- *Congestion Factor* метрика, показваща ефективността на използваната стратегия. По-голяма стойност - по-добър резултат.

Използваната стратегия за симулация по подразбиране е *simple*, която дава време от 3 секунди зелена светлина на всяка страна от кръстовището. Друг предоставен алгоритъм е *bus-priority*. Тази стратегия дава зелена светлина на първия срещнат автобус, независимо от текущото състояние на кръстовището.



Фигура 2.1: Екран при стартиране

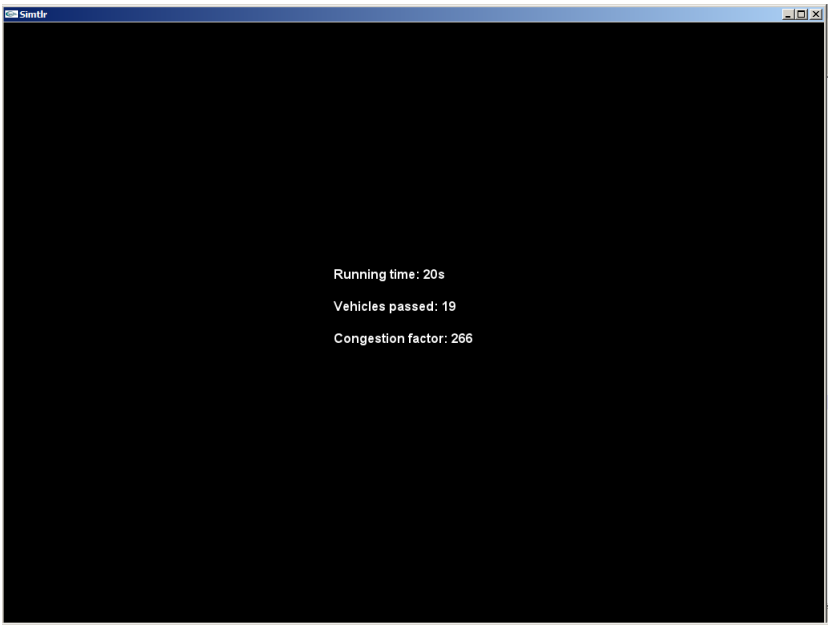
Конфигурация

За конфигуриране на симулационната система се използва файла *config.lua*, намиращ се в директорията *src*. В нея може да се променят следните симулационни настройки:

- *STRATEGY* - избор на използваната стратегия за симулацията
- *TOTAL RUNNING TIME* - времето за което симулацията е активна
- *MAXIMUM MODELS PER SIDE* - максималния брой модели, които могат да бъдат поставени в опашката на някоя от страните на кръстовището

За конфигурарине на коефициентите на ефективност в кръстовището се използват:

- *BUS EFFECTIVENESS MULTIPLIER* - определя коефициент на ефективност за автобуси
- *CAR EFFECTIVENESS MULTIPLIER* - определя коефициент на ефективност за коли



Фигура 2.2: Екран след приключване на симулация

Заключение

Резултати

За постигане на целите на дипломната работа е създадена примерна симулационна система. Тя е модуларна и разширяема чрез механизма за добавяне на нови стратегии с различна приоритизация на моделите. За създаването ѝ е използван итеративен подход, като са добавени само най-необходимите функционалности. Това намалява сложността за доразвитие и улеснява разработката на нови възможности.

Употребата на Обектно-ориентирано програмиране (Object-oriented programming) (ООП) прави лесно допълването на нови модели, начини за изобразяване на различните елементи и нови състояния. Разширяването се състои в изграждане на нови подкласове, отговарящи на определен интерфейс, които следват Лисков принцип на заместването (liskov substitution principle) (ЛПЗ).

Липсата на типове в *Lua*, прави възможна динамичната замяна на конкретни имплементации за отделните стратегии, без необходимост от явни интерфейси.

Създадената примерна имплементация е прототип на система, която може да бъде вградена в съвременната система за управление на транспорта в град Пловдив. Проведени са експерименти със стратегии с ясно изразен приоритет на специфични модели. Те показват по-голям коефициент на ефективност и превозени пътници за единица време. За да бъдат приети или отхвърлени, получените резултати трябва да се сравнят с данни от реални експерименти.

Приноси

Като приноси на автора могат да се опишат:

- Проучване на различните ИТС и употребата им в страни, различни от България
- Проучване на безплатни и платени системи за провеждане на симулаци
- Създаване на прототип на ДСС, чрез употребата на двигател за игри
- Провеждане на експерименти с различно приоритизирани модели

Проблеми по време на разработка

Изказаните по-долу мнения са изцяло авторски и могат да бъдат разгледани като истина или некомпетентност от страна на автора.

За разработката на системата беше използвана *Windows 7*[1] базирана система. Липсата на мощни интрументи (по подразбиране) в командния ред, които да спомогнат бързата разработката, беше осезаема. Проблеми бяха срещнати и при настройка на кодирането(encoding).

От друга страна, инсталациите на *Lua* и *Moai* бяха бързи и удобни. Редакторът, *Notepad++ (v6.1.2)*[2], също предложи приятен и удобен интерфейс за писане на *Lua*.

Lua Макар и разработван от дълго време, езикът има доста празнини, в сравнение с езици като *C++* или *Java*. Употребата му в големи компютърни игри не е довела до значително нарастване на помощните функции в стартовата библиотека. Напротив, повечето потребители на езика са запазили написаното като затворен код или просто не са отделили време да го споделят с останалите.

За целите на симулацията, бяха необходими създаването на структури от данни, съществуващи в стандартните библиотеки на почти всеки друг програмен език от високо ниво. Макар и съществуващите *таблици* да предлагат почти цялата функционалност, необходима за създаване на каквито и да е видове

[1] `http://windows.microsoft.com/en-US/windows/home`
[2] `http://notepad-plus-plus.org/`

структури от данни, тя не е лесна за използване и преизползване, ако се ползва в "суров" вид. Това налага нуждата от създаването на обвивки(wrappers).

Липсата на стандартна библиотека за тестване, под какъвто и да е вид, направи използване на добър итеративен подход невъзможно. Още повече, това доведе до цялостно забавяне на процеса на разботка на системата.

LuaRocks *LuaRocks*[3] е добър опит да се направи лесно и бързо споделянето на *Lua* код под формата на модули, наречени "скали"(rocks). Те съдържат информация за зависимостите(dependencies) които имат, чрез която гарантират наличие на всички необходими компоненти на системата.

Инсталацията на *LuaRocks* е бърза и лесна. Текущата версия *(v2.0.8)*, обаче, просто не тръгва под *Windows*. Проблемът се дължи в неправилно конфигуриране на пътищата из цялата програма.

Липсата на *LuaRocks* означава и липса на *инсталирана* библиотека за тестване. Много обещаваща такава изглежда библиотеката за поведенческо тестване(behaviour driven testing) *luaspec*[4].

Броят на пакетите в системата също не е впечатляващ, около 200 бр. За сравнение, разработчиците на *Ruby* разполагат с над 2530 библиотеки[5]. Те варират от малки с десетки редове код до огромни със стотици хиляди редове[6].

Moai След срещането на горепосочените проблеми, логичен избор би бил преминаването към *Linux* базирана система (напр. Ubuntu[7]). Това би решило, практически, всички тях.

За съжаление, за момента *Moai* не поддържа лесна инсталация и разработка под Linux платформа. Тя работи прекрасно под *Mac*, но такава система би излезнала "малко" по-скъпо[8].

По време на разработката на симулационната система беше използвано *Moai SDK v1.1*. То идва с редица полезни примери, повечето от които имат

[3] `http://luarocks.org`
[4] `https://github.com/mirven/luaspec`
[5] `https://rubygems.org/gems`
[6] `https://www.ohloh.net/p/rails/analyses/latest`
[7] `http://www.ubuntu.com/`
[8] Към момента на писането, най-евтиния вариант е 2395 лв. `http://bit.ly/JIawqv`

неправилно зададени пътища, а други използват вече несъществуващи методи от по-стари версии.

Документацията е полезна, но непълна. Целта на библиотеката е създаване на средно големи игри под всякакви платформи, но примерен код за такива няма. За момента няма и издадени книги, които да предлагат подобни примери.

Бъдещо развитие

Пред системата има много възможности за развитие. Тя може да бъде изградена напълно наново, взимайки отделни идеи или код. Възможно е пълното запазване на кода, но това най-вероятно няма да доведе до добре изградена система.

Възможни допълнения включват:

- Премахване нуждата за промяна на конфигурационния файл, това да става чрез ГПИ
- Подобряване на потребителския интерфейс, с цел по-лесно използване на системата
- Добавяне на нови стратегии
- Добавяне на нови модели
- Добавяне на логика за сблъсъци
- Добавяне на нови статистики
- Паралелна обработка на изчисленията на стратегиите

Списък на фигурите

1.1 Модел на симулационната система 18
1.2 Клас диаграма на симулационната система 32
1.3 Клас диаграма на новите елементи след 2-рата итерация 36

2.1 Екран при стартиране 39
2.2 Екран след приключване на симулация 40

Списък на алгоритмите

1.1 Итеративна процедура за изчисление на функцията за промяна на състоянието 22
1.2 Засичане на изминало време в симулацията 33

Приложение А

Използвани съкращения

ИПП Интерфейс за приложно програмиране (Application programming interface)

JDK Java средства за разработка (Java development kit)

НС Непрекъсната симулация (Continuous simulation)

ДСС Дискретно-събитийна симулация (Discrete-event simulation)

ГПИ Графичен потребителски интерфейс (Graphical user interface)

LGPL Lesser General Public License

СБ Събирач на боклука (Garbage collector)

ТВК Точно на време компилатор (Just in time compiler)

ИТС Интелигентни транспортни системи (Intelligent Transportation Systems)

ООП Обектно-ориентирано програмиране (Object-oriented programming)

ЛПЗ Лисков принцип на заместването (liskov substitution principle)

Приложение Б

Библиография

[AOP] Yuliyan Kiryakov and John Galletly. Aspect-oriented programming – case study experiences. 2003.

[Anu] Maria Anu. Introduction to modeling and simulation.

[AnyLogic] `http://www.xjtek.com/anylogic/why_anylogic/`.

[ArtOfAgile] James Shore. *The Art of Agile Development*. 2007.

[Barlas] Onur Ozgun and Yaman Barlas. Discrete vs. continuous simulation: When does it matter? 2009.

[Darter] Michael Darter, Kin Yen, Travis Swanston, Bahram Ravani, and Ty Lasky. Research and development of open-source advanced transportation management system hardware and software components. 2009.

[Ezell] Stephen Ezell. Explaining international it application leadership: Intelligent transportation systems. 2010.

[Figueiredo] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua - an extensible extension language. 1996.

[Gamma] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.

[GarlicSim] `http://garlicsim.org/`.

[HEI] HEI. Traffic-related air pollution: A critical review of the literature on emissions, exposure, and health effects. 2010.

[Holzkorn] Peter Holzkorn. Physics simulation in games. 2008.

[Ierusalimschy] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0. 2003.

[Iliev] Антон Илиев и Георги Христозов и Тодорка Терзиева. Софтуерна среда за представяне на детерминирани динамични модели с възможност за статистика. 2006.

[Jones] Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. 1986.

[Khan] Sohail Khan, Shahryar Khan, and Syed Hammad Khalid Banuri. Analysis of dalvik virtual machine and class path library. 2009.

[Mahoney] Kevin Mahoney. Model validation techniques. 2010.

[Microsoft] Microsoft Corporation. Usability in software design. 2000.

[Nagel] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. 1992.

[Rehunathan] Devan Bing Rehunathan, Boon-Chong Seet, and Trung-Tuan Luong. Federating of mitsimlab and ns-2 for realistic vehicular network simulation. 2007.

[Rollings] Andrew Rollings and Dave Morris. *Game Architecture and Design: A New Edition*. 2004.

[SimPy] `http://simpy.sourceforge.net/`.

[Simplex3] `http://www.simplex3.net/Body/Introduction/English/indexAbstract.html`.

[Tortuga] `http://code.google.com/p/tortugades/`, 2004.

[Totkov] Георги Тотков. Концептуално и компютърно моделиране на езикови структури и процеси (с приложения за българския език). 2004.

[Zipline] Zipline Games. Moai technical white paper. 2012.