Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _комп'ютерних наук_____
(повна назва)

Кафедра _програмної інженерії_____
(повна назва)

АТЕСТАЦІЙНА РОБОТА (ПРОЕКТ)
Пояснювальна записка

_____Бакалавр_____
(освітньо-кваліфікаційний рівень)

_____
(позначення документа)

_____Програма пошуку найкоротшого шляху для ігрових додатків_____.
_____.
_____.
(тема)

Виконав: студент _4_ курсу, групи _ПІ-12-1__
напряму підготовки (спеціальності) _____
_6.050103 «Програмна інженерія»_____
_____
(шифр і назва напряму, спеціальності)

_Піляєв Д.В._____
(прізвище, ініціали)
Керівник _Качко О.Г._____
(прізвище, ініціали)
Рецензент_____
(прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ _____
(підпис) (прізвище, ініціали)

2016 р.

Харківський національний університет радіоелектроніки

Факультет_комп'ютерних наук_____
Кафедра_програмної інженерії_____
Освітньо-кваліфікаційний рівень _Бакалавр_____
Напрям підготовки _6.050103 «Програмна інженерія»_____
(шифр і назва)
Спеціальність_6.050103 «Програмна інженерія»_____
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _ З.В.Дудар____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ

НА АТЕСТАЦІЙНУ РОБОТУ (ПРОЕКТ)

студентові_____
(прізвище, ім'я, по батькові)

1. Тема роботи (проекту)__«Програма пошуку найкоротшого шляху для ігрових додатків»_____
_____
затверджена наказом по університету від "___"_____20 __ р. № _____
2. Термін подання студентом роботи (проекту) _____
3. Вихідні дані до роботи (проекту)_розробити програму пошуку найкоротшого шляху для ігрових додатків, використовуючи мову С++_____
_____
_____
_____
_____
_____
_____
_____
_____
4. Зміст пояснювальної записки (перелік питань, що потрібно розробити) _вступ, аналіз проблемної області і постановка задачі, проєктування системи, опис розробленої програмної системи, аналіз результатів, висновки_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслеників, плакатів)_____
_____
_____
_____
_____
_____
_____
_____
_____

6. Консультанти розділів роботи (проекту)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	Качко О.Г.		

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи (проекту)	Термін виконання етапів проекту (роботи)	Примітка

Студент _____
(підпис)

Керівник роботи (проекту) _____ _____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 42 сторінок, 4 розділів, 23 рисунків, 11 джерел.

Об’єкт проектування – програма для пошуку найкоротших шляхів.

Метою роботи є створення оптимізованної програми для пошуку найкоротших шляхів в ігрових додатків.

Методи розробки базуються на язиці C++11, та бібліотеках threadpool11, SFML і SFGUI

У результаті роботи була здійснена програмна реалізація програми для пошуку шляхів в ігрових додатків, а саме алгоритмів A*, JPS та Goal Bounding. В процессі ці алгоритми були проаналізовані та оптимізовані.

БИБЛИОТЕКА, ПОИСК ПУТЕЙ, C++, A*, JPS, GOAL BOUNDING, HAA*, КВАДРАТНАЯ СЕТКА, ЭВРИСТИЧЕСКАЯ ФУНКЦИЯ.

Explanatory note: 42 pages, 4 sections, 23 figures, 11 sources.

Subject of architecturing – library for path finding.

The aim is to create optimized library for path finding in games.

The development methods are based on C++11 and libraries threadpool11, SFML and SFGUI.

The result of work is implemented library for path finding in games which includes A* and JPS algorithms. During development this algorithms were analyzed and optimized.

LIBRARY, ПРАТН FINDING, C++, A*, JPS, GOAL BOUNDING, HAA*, SQUARE GRID, HEURISTIC FUNCTION.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задачей нахождения кратчайшего пути является поиск оптимального и короткого пути между двумя точками. Проблема нахождения кратчайших путей возникает в таких случаях как: оптимизация перевозки грузов и пассажиров, оптимальная маршрутиризация пакетов в сети, навигация искусственного интеллекта и игрока в компьютерных играх, а так же навигация роботов в пространстве. На данный момент большинство компьютерных игр имеют поиск путей в том или ином виде, поэтому скорость и точность алгоритма часто влияет на качество искусственного интеллекта и восприятие игры игроком.

Разнообразие жанров компьютерных игор приводит к разнообразию представлений карт с которыми приходится работать алгоритмам поиска путей. Одним из таких представлений является квадратная сетка, которая часто используется в играх с двумерной картой, например игры жанра RTS. Хотя квадратная сетка в большинстве случаев является неоптимальным представлением области поиск, с ней очень просто работать и легко модифицировать, что значительно упрощает программную работу с игровой картой. В следствии неоптимальности представления карты появляется необходимость в оптимизации алгоритмов поиска работающих с ней посредством общей оптимизации логики работы алгоритма, введение некоторых допущений и ограничений на область поиска, а так же проведение низкоуровневых оптимизаций.

Одними из методов нахождения кратчайшего пути являются алгоритмы A* и JPS, которые широко используются в игровых приложениях. Данные алгоритмы не требуют предварительных расчётов, однако такие расчёты могут ускорить поиск пути при определённых условиях [1].

Таким образом, задача нахождения кратчайшего пути на области представленной квадратной сеткой является актуальной проблемой, которая будет рассмотрена и исследована в данной работе.

Целью выпускной работы является проведение анализа существующих алгоритмов поиска путей, разработка различных вариантов алгоритмов и их оптимизаций, создание оптимизированной универсальной библиотеки для нахождения оптимальных маршрутов и анализ полученных результатов.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Алгоритмы нахождения кратчайших путей

Задачей нахождения кратчайшего пути является поиск оптимального и короткого пути между двумя точками. Решения этой задачи в большинстве случаев основаны на алгоритме Дейкстры [2] для нахождения кратчайшего пути во взвешенных графах.

Простейшие алгоритмы для обхода графа, такие как поиск в ширину и поиск в глубину могут найти какой-то путь от начальной до конечной вершины, но не учитывают стоимость пути. Одним из первых алгоритмов поиска пути с учётом его стоимости был алгоритм Беллмана – Форда, который проходит по всем возможным маршрутам и находит наиболее оптимальный, вследствие чего имеет временную сложность $O(|V||E|)$, где V – количество вершин, а E – количество рёбер. Однако для нахождения пути близкого к оптимальному не обязательно перебирать все пути, а можно отсекать неперспективные направления на основе некой эвристики, что может дать таким алгоритмам нижнюю оценку $O(|E|\log(|V|))$. Такими алгоритмами являются алгоритм Дейкстры, A* и их модификации.

1.1.1 Алгоритм A*

Алгоритм A* (A звёздочка) - это алгоритм общего назначения, который может быть использован для решения многих задач, например для нахождения путей. A* является вариацией алгоритма Дейкстры и используя эвристическую функция для ускорения работы, при этом гарантируя наиболее эффектное использование онной [3].

Алгоритм A* поочерёдно рассматривает наиболее перспективные неисследованные точки или точки с неоптимальным маршрутом до них, выбирая пути которые минимизируют $f(n) = g(n) + h(n)$, где n последняя точка в пути, $g(n)$ – стоимость пути от начальной точки до точки n, а $h(n)$ – эвристическая оценка стоимости пути от n до конца пути. Когда точка исследована, алгоритм останавливается если это конечная точка, иначе все её соседи добавляются в список для дальнейшего исследования.

Для нахождения пути от начальной до конечной точки, кроме стоимости пути до точки, следует записывать и её предка (точку из которой мы пришли в неё).

Свойства алгоритма A*:

- Алгоритм гарантирует нахождение пути между точками, если он существует;
- Если эвристическая функция $h(n)$ не переоценивает действительную минимальную стоимость пути, то алгоритм работает наиболее оптимально;
- A* оптимально эффективен для заданной эвристики $h(n)$.

На оценку сложности A* влияет использованная эвристика, в худшем случае количество точек рассмотренных A* экспоненциально растёт по сравнению с длинной пути, однако при эвристике $h(n)$ удовлетворяющей условию $|h(x) - h^*(x)| \leq O(\log h^*(x))$, где $h^*(n)$ – оптимальная эвристика, алгоритм будет иметь полиномиальную сложность. Так же на временную сложность влияет выбранный способ хранения закрытых и открытых точек.

1.1.2 Алгоритм JPS

Jump Point Search (JPS) - эффективная техника для нахождения и отброса симметричных путей [4]. На рисунке 1.1 показаны симметричные пути из стартовой до конечной точки - все они имеют одинаковую длину и при наличии такой симметрии такие алгоритмы как A* вынуждены рассматривать почти все оптимальные пути. Так же A* вынужден рассматривать многообещающие точки, которые не лежат на каком-либо оптимальном пути. JPS представляет собой модификацию алгоритма A* с двумя правилами отброса точек, которое позволяет избежать рассмотрения большинства симметричных путей и рассматривать только те точки, которые могут находится на оптимальном пути. Применения этих правил позволяет увеличить производительность поиска путей на квадратной сетки на порядок - без препроцессинга и дополнительной памяти.

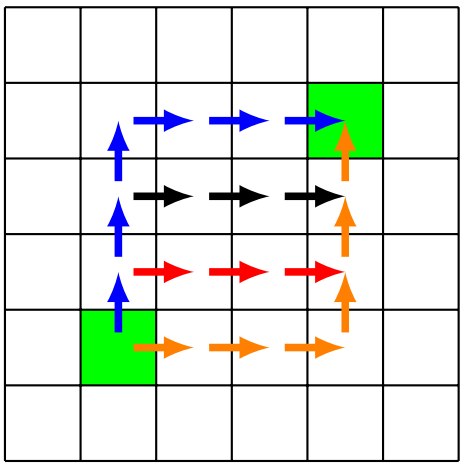

Рисунок 1.1 — Симметричные пути

JPS действует в предположении, что прохождение по клеткам карты намного быстрее добавления их в открытые и закрытые списки. При этом одна клетка может рассматриваться много раз в течении одного поиска.

В JPS существует два набора правил: правила отброса клеток и правила прыжков.

Имея клетку x, достижимую из родительской клетки p, мы отбрасываем из соседей x любую клетку n для которой исполняется хотя бы одно из условий:

- существует путь от p до n равный $\pi' = (p, x, n)$ строго короче чем путь от p до n через клетку x равный $\pi = (p, x, n)$;

- существует путь от p до n равный $\pi' = (p, n)$ такой же длины как путь от p до n через клетку x равный $\pi = (p, x, n)$, но путь π' имеет диагональное перемещение раньше чем путь π.

Для вычисления каждого из правил достаточно проверки только соседей данной точки.

Преимуществами JPS является отсутствие препроцессинга, дополнительного потребления памяти, постоянное ускорение A* на порядок. Главным недостатком является, то что он работает только на картах с одинаковой стоимостью прохода по клеткам, однако существует теоретическое решение данной проблемы, но любое её решение приводит к уменьшение преимущества JPS над простым A*.

1.1.3 Алгоритмы HPA* и HAA*

HPA* (Hierarchical Path-Finding A*) - добавляет алгоритму A* иерархическую абстракцию, разбивая карту на прилегающие друг к другу кластеры, которые соединены входами [5]. Одной из основных идей алгоритма является то, что расчёт пути в A* каждый раз происходит с нуля, что можно исправить добавив сохранение кратчайших путей между определёнными точками.

Первым этапом алгоритма является препроцессинг карты для построения кластеров и их входов. При этом возможно построение нескольких уровней графа кластеров используя один и тот же алгоритм рекурсивно на созданных во время предыдущего прохода кластерах.

Во время исполнения программы запрос нахождения пути выполняется рекурсивно находя и уточняя путь на графе начиная с самых крупного уровня кластеров. После нахождения пути может применятся его сглаживание.

Алгоритм HPA* работает с такими допущениями:

- Все актёры имеют одинаковый размер, при этом все части навигационной сетки проходимы ими;

- На всех участках карты агенты имеют одинаковую проходимость.

Иерархическая структура карты сильно ускоряет поиск пути, однако алгоритм HPA* работает не учитывая такие важные параметры как размер агентов и проходимость местности.

В итоге размер агентов и проходимость карты должны учитываться при нахождении пути, что в случае с алгоритмом HPA* приводит к тому, что эти параметры должны учитываться при оценки путей между входами.

Алгоритм иерархического поиска является достаточно абстрактным для учёта указанных проблем. Для этого на основе алгоритма HPA* был создан алгоритм HAA* (Hierarchical Annotated A*), который при создании путей между входами кластера учитывает размеры актёров и проходимость местности.

Основная разница с алгоритмом HPA* у алгоритма HAA* состоит в шаге формирования пути между транзитными точками в рамках кластера и дальнейшем шаге их оптимизации. При нахождении пути между транзитными точками в кластере следует найти пути для всех агентов разных размеров и проходимости

В итоге алгоритм HAA* имеет такие же преимущества как HPA*, а так же возможность учёта размера агента и проходимости карты. В зависимости от выбранной тактики устранения похожих путей между транзитными точками кластеров конечный путь будет хуже оптимального до 4-8%.

1.2 Различные представления области поиска

Для поиска пути требуется некоторое представление области поиска, от его выбора зависит скорость работы алгоритма, точность его работы и качество пути. Так же выбор способа представления влияет на занимаемое областью место в оперативной памяти.

Глобально пространства поиска можно разделить на дискретные и непрерывные, далее будут рассматриваться только дискретные пространства (для непрерывных пространств существуют алгоритмы сведения их к дискретным).

Можно выделить такие способы представления: квадратная сетка, quadtree (дерево квадрантов), navmesh (навигационная сетка), waypoints (путевые точки).

1.2.1 Квадратная сетка

Квадратная сетка является самым простым и очевидным представлением области поиска. Подходит для игр жанра TD (tower defence) и некоторых RTS игр.

Обладает такими преимуществами: проста в использовании и представлении в памяти, можно быстро и легко добавлять и удалять препятствия, легко поддерживать различную проходимость карты и различные размеры агентов.

К недостаткам можно отнести большое занимаемое в памяти место, сложность поддержки многоуровневых карт а так же наименьшую скорость работы алгоритма A* с данным представлением.

Некоторые алгоритмы поиска путей не работают на других представлениях кроме квадратной сетки, например JPS.

1.2.2 Quadtree (дерево квадрантов)

Дерево квадрантов является усовершенствованной версией квадратной сетки. Позволяет сильно уменьшить занимаемое место за счёт объединения одинаковых участков, так же позволяет ускорить алгоритм поиска пути. Минусом можно считать усложнение добавления и удаления элементов из карты.

1.2.3 Navmesh (навигационная сетка)

Навигационная сетка - набор выпуклых многоугольников, которые описывают проходимую часть трёхмерного мира.

Является популярной техникой для описания трёхмерных карт. Навигационная сетка часто автоматически генерируется из трёхмерного мира, однако при этом возникает проблема оптимизации числа полигонов для ускорения работы поиска путей и уменьшении веса сетки.

Преимуществами навигационной сетки являются: возможность представить многоуровневый мир, низкое потребление памяти, скорость поиска пути, точность. Недостатком является сложность динамического изменения мира.

1.2.4 Waypoints (путевые точки)

Путевые точки – набор точек и рёбер по которым можно ходить. Обычно создаются вручную.

Преимущества: просты в использовании, возможно описать пути любой сложности (в воздухе, в воде и т.д.), простота изменения, гибкость работы.

Недостатки: медленней чем навигационная сетка, пути выходят хуже (более длинные и неестественные).

1.3 Эвристические функции

Эвристическая функция $h(n)$ вычисляет для алгоритмов поиска путей основанных на A* приблизительную стоимость маршрута от точки до цели. Эвристика может использоваться для контролирования поведения алгоритмов [6]:

- если $h(n)$ равно нулю, то A* вырождается алгоритм Дейкстры;

- если $h(n)$ всегда меньше или равен стоимости пути от точки до цели, то A* гарантирует нахождение кратчайшего пути, однако чем больше разница между $h(n)$ и реальной стоимостью, тем больше точек раскрывает алгоритм и тем медленней работает;

- если $h(n)$ равен стоимости пути от точки до цели, то A* следует только по оптимальному пути, что делает его очень быстрым, однако такая эвристика невозможна в общих случаях;

- если $h(n)$ переоценивает стоимость пути, то A* не гарантирует нахождение кратчайшего пути, однако это может ускорить его;

- если $h(n)$ очень сильно переоценивает стоимость пути, то алгоритм вырождается в жадный.

Исходя из этого можно сделать вывод, что при выборе эвристики осуществляется выбор между скоростью и точностью.

Что бы получить эвристику, которая всегда равна стоимости от точки до цели, нужно предрасчитать пути между каждыми парами точек, что не осуществимо на реальных картах. Однако существует некоторые упрощения и модификации этой эвристики:

- покрыть карту более крупной сеткой и уже для неё рассчитывать данную эвристику;

- выделить некоторые путевые точки и рассчитывать эвристику между ними;

- сохранять не все расстояния от точки до остальных, а только по одной области для каждого направления, содержащие все точки до которых оптимальный путь идёт в выбранном направлении (Goal Bounding).

Для различных представлений области поиска существуют разные эвристики. Для области представленной квадратной сеткой можно выделить такие эвристики:

- расстояние городских кварталов (Manhattan distance) - расстояние равное сумме модулей разностей координат точек $|x_1 - x_2| + |y_1 - y_2|$;

- евклидова метрика (Euclidean distance) - равна расстоянию между точками вычисленному по теореме пифагора $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$;

- эвристики препятствующие прохождению по путям с одинаковой стоимостью (Tie breaaking).

1.4 Препроцессинг RSR

Алгоритм JPS позволяет избавляться от симметричных путей во время нахождения пути, в свою очередь от симметричных путей можно избавится так же с помощью препроцессинга карты. Алгоритм RSR (Rectangular Symmetry Reduction) позволяет найти симметрии разбивая карту на пустотелые прямоугольные регионы. При этом избежать

симметрии можно рассматривая только периметр таких прямоугольников во время поиска пути. Для оптимального перемещения по прямоугольника вводится возможность прыжка с одной стороны на противоположную.

При этом препроцессинг RSR выполняется очень быстро и имеет низкое потребление памяти, является оптимальным и позволяет ускорить поиск на порядок, так же как и JPS. RSR можно использовать на изменяющихся картах, так как изменение участка карты влечёт изменение только части прямоугольников и происходит очень быстро.

Недостатком RSR является невозможность корректно работать на картах с неоднородными весами клеток.

Алгоритм RSR сочетается с A* и может работать с JPS, так как они отталкиваются от разных предположений.

1.5 Препроцессинг Goal Bounding

Goal Bounding - техника отброса заранее неподходящих точек, которая позволяет значительно ускорить поиск пути [7]. Goal Bounding можно разделить на два этапа:

- оффлайн этап – представляет собой препроцессинг карты с целью найти и сохранить прямоугольники для каждого из направлений, ограничивающие минимальную область на карте, которая включает в себя все точки карты до которых путь через выбранное направление является самым оптимальным.

- онлайн этап – прекращение итераций в выбранном направлении если целевая точка не входит в его ограничивающий прямоугольник.

Данная техника применима для всех областей поиска и для всех алгоритмов – не имеет значение одинакова ли стоимость прохода по всем клеткам или нет. Её недостатком является время препроцессинга и занимаемое место результирующими данными. Препроцессинг работает за $O(n^2)$, где n – количество узлов на карте, что делает невозможным пересчёт во время исполнения программы. Количество занимаемой памяти – линейно, и для квадратной сетки с 8 направлениями равно $4 * 32 * n$ байт.

Этап препроцессинга легко поддаётся распараллеливанию, что позволяет работать даже с большими картами, имея достаточное количество вычислительной мощности.

1.6 Выбор алгоритма и представления области поиска

Выбор алгоритма сильно зависит от выбранного типа области поиска, в данной работе была выбрана область поиска представленная квадратной сеткой. Нахождения пути

на ней является самым затратным по производительности, поэтому оптимизация алгоритмов работающих с этим представлением является актуальной задачей.

Как базовый алгоритм был взят A*. При выборе между JPS, HPA* и HAA* был выбран JPS, так как по сравнению с JPS алгоритмы HPA* и HAA*, работают дольше и являются намного сложнее в реализации, которая может быть несоразмерна с полученной от них выгодой, однако они могут выдавать начальные участки пути намного быстрее чем JPS и A*, что в некоторых случаях оправдывает их написание.

Для препроцессинга карты был выбран алгоритм Goal Bounding.

1.7 Анализ аналогичных библиотек

Такие игровые движки как Unreal Engine и Unity имеют свои встроенные модули для нахождения пути, которые в основном основаны на навигационных сетках и нацелены на игры от первого и третьего лица, и в меньшей степени на RTS игры.

Как популярные отдельные библиотеки можно выделить “Recast & Detour” и “A* Pathfinding Project”. “Recast & Detour” бесплатная библиотека с открытым исходным кодом, написана на C++ и работает с навигационными сетками, содержит инструменты для создания сетки по имеющемуся уровню и её редактирование. “A* Pathfinding Project” библиотека имеющая платный и бесплатный вариант, имеет открытый исходный код, написана на C# для Unity и умеет работать с различными областями представления и использует как A*, так и JPS алгоритмы. “A* Pathfinding Project” так же содержит возможности сглаживания пути и его локальной коррекции. Данная библиотека имеет самые широкие возможности из доступных в интернете библиотек.

Так же существует множество минималистичных библиотек реализующих алгоритм A*. Библиотек использующих алгоритм JPS и другие модификации A* намного меньше и в основной массе они являются частью научных публикаций и не поддерживаются авторами после их создания.

Разрабатываемая в данной работе библиотека нацелена на предоставление простого интерфейса для поиска путей на квадратной сетке с

1.8 Постановка задачи

Целью работы является проведение анализа существующих алгоритмов поиска путей, разработка различных вариантов алгоритмов и их оптимизаций, создание оптимизированной

универсальной библиотеки для нахождения оптимальных маршрутов в контексте игровых приложений и анализ полученных результатов.

Создание библиотеки состоит из следующих частей:

- анализ алгоритмов нахождения путей;
- анализ алгоритмов препроцессинга карт;
- разработка алгоритма A*;
- разработка алгоритма JPS;
- разработка препроцессинга Goal Bounding;
- интеграция Goal Bounding с алгоритмом A*;
- интеграция Goal Bounding с алгоритмом JPS;
- разработка удобного визуализатора для наглядной оценки и проверки алгоритмов;
- создание модуля для сравнения и валидации алгоритмов;
- анализ полученных результатов.

Для написания библиотеки был выбран язык C++ стандарта C++11. Для визуализации результатов была выбрана библиотека SFML и SFGUI. Для написания параллельных алгоритмов – библиотека threadpool11, которая реализует пул потоков. Компилятором был выбран g++ версии 5.3.

2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Программное обеспечение

Для написание библиотеки нахождения кратчайших путей был выбран язык программирования C++ стандарта C++11. C++ является современным языком высокого уровня, который предоставляет широкие возможности по оптимизации кода, в отличии от интерпретируемых языков и языков с JIT оптимизациями. Так же C++ широко используется в игровых приложениях. Для создания библиотеки язык C++ был выбран по следующим причинам: кроссплатформенность, быстродействие, возможность низкоуровневых оптимизаций, простота внедрения библиотеки в существующие программные продукты.

Стандарт C++11 привнёс в язык многие функции, которые позволяют писать более понятный и современный код, уменьшить количество случайных ошибок и повысить читаемость. В следствии чего были устранены многие недостатки в сравнении с другими языками программирования.

В C++11 для измерения времени используется стандартная библиотека chrono. В chrono существует несколько реализаций измерения времени:

- system_clock – общесистемное время;
- steady_clock – монотонное время, которое никогда не подстраивается;
- high_resolution_clock - наиболее точное время с наименьшим доступным периодом.

Для измерения времени была выбрана реализация high_resolution_clock.

Библиотека SFML (Simple and Fast Multimedia Library) – кроссплатформенная библиотека для создания мультимедийных приложений. Имеет простой платформонезависимый интерфейс для рисования графики.

SFGUI - библиотека работающая совместно с библиотекой SFML и предоставляющая возможности для отрисовывания интерфейсов.

Для использования многопоточности была подключена библиотека threadpool11, которая реализует пул потоков [8].

Для написания кода библиотеки была выбрана среда разработки CLion, которая имеет редактор с поддержкой синтаксиса C++11 и его подсветкой, имеет средства рефакторинга и поддерживает различные CVS, например Git. Так же имеется поддержка CMake – системы кроссплатформенной сборки проектов, управления зависимостями и тестами.

В качестве системы контроля версий был выбран Git. Git – распределённая система контроля версий, которая направлена на скорость работа и целостность данных, имеет гибкую и простую систему создания и объединения веток. Репозиторий проекта был размещён на удалённом сервисе BitBucket.

2.2 UML-моделирование ПО

Унифицированный язык моделирования (UML) – язык общего назначения для визуализации, спецификации, конструирования и документации программных систем [9]. Язык UML объединяет в себе семейство разных графических нотаций с общей метамоделью.

Преимуществами UML являются:

- объектно-ориентированность, что делает его близким к современным объектного-ориентированным языкам;
- расширяем, что позволяет вводить собственные текстовые и графические стереотипы;
- прост для чтения;
- позволяет описать системы со всех точек зрения.

В UML используется три вида диаграмм:

- структурные диаграммы – отражают статическую структуру системы;
- диаграммы поведения – отражают поведение системы в динамике, показывают, что должно происходить в системе;
- диаграммы взаимодействия – подвид диаграмм поведения, которые выражают передачу контроля и данных внутри системы.

Для моделирования системы проектируемой в данной аттестационной работе будут использованы структурная и поведенческая диаграммы, а именно диаграмма классов (Class diagram) и диаграмма вариантов использования (Use case diagram).

Диаграмма классов является статическим отображением системы, которая демонстрирует её классы, их аттрибуты, методы и взаимосвязи. Диаграмма классов является прямым отображением классов присутствующих в системе.

Диаграмма состоит из классов и связей между ними. В свою очередь классы состоят из имени класса, его полей и методов. Поля и методы могут иметь модификаторы доступа к ним: публичный – знак плюс, приватный – знак минус, защищённый – решётка и некоторые другие. Методы могут иметь аргументы и возвращаемое значение и их типы.

Связи между классами делятся на связи между созданными объектами класса и на связи на уровне самих классов. Связи между объектами включают в себя:

- зависимость – односторонняя зависимость между двумя классами, когда изменения в одном влияют на второй;
- ассоциация – набор схожих связей, которые обозначают, что один объект выполняет некоторые действия над другим, такие как: вызов метода или посылка сообщения;
- агрегация – включение одним классом другого, однако при этом их жизненные циклы не зависимы;
- композиция – включение одним классом другого, при этом существует зависимость между жизненным циклом контейнера и включаемого класса.

Связи между классами делятся на:

- обобщение – обозначает, что один из двух классов является подклассом второго;
- реализация – обозначает, что класс реализует поведение определённое во втором.

В разрабатываемой библиотеки можно выделить центральный интерфейс, который является основным интерфейсом для взаимодействия с ней, это PathFinder см. рис. 2.1. Он является шаблонным классом и параметризуется типом координат (целочисленный тип), он имеет один метод – Find, который принимает начальную позицию, конечную позицию и тип актёра (проходимость актёра), а возвращает вектор клеток являющихся найденным маршрутом. Его реализуют классы SimpleAStar и JPSAstar, которые в свою очередь имеют следующие шаблонные параметры: тип карты, эвристика, включение Goal Bounding и включение режима отладки.

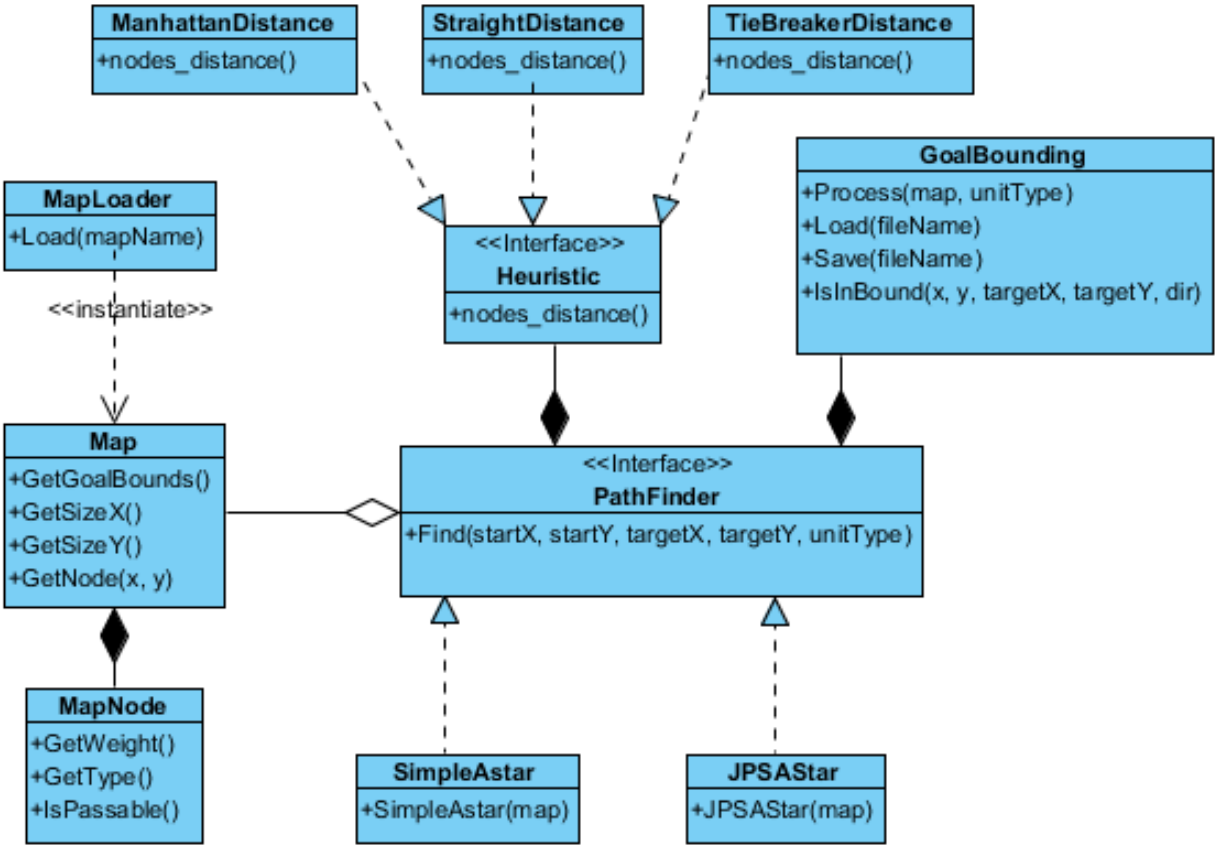

Рисунок 2.1 — Диаграмма классов

Интерфейс для реализации эвристика, Heuristic, содержит метод nodes_distance, который принимает текущую точку, конечную точку и начальную точку, а возвращает приближённо оценённую стоимость маршрута от текущей до конечной точки. Эвристика включается в PathFinder как шаблонный класс.

Класс GoalBounding реализует одноимённый алгоритм и содержит методы для препроцессинга карты с заданным типом актёра, для загрузки и сохранения информации полученный при препроцессинге, а так же для отброса направлений при поиске пути. Данный класс включается в класс карты и используется в классах реализующих PathFinder.

Класс карты является базовой реализацией карты и содержит методы для получение данных о препроцессинге, размеров картыб проверки вхождения точки в область карты и метод получения клетки карты. Класс карты передаётся в PathFinder через конструктор и хранится в нём. Карта включает в отдельные клетки MapNode, которые имеют вес, проходимость и метод для проверки на проходимость.

Карта загружается по средством класса MapLoader, который имеет единственный метод для загрузки карты по её имени.

Диаграмма вариантов использования позволяет отразить представленную систему в динамике, а именно собрать требования к системе отражающие внешние и внутренние воздействия. Она позволяет показать взаимодействие различных пользователей и частей системы. Целями диаграммы вариантов являются:

- сбор требований к системе;
- получение вида системы со стороны;
- идентификация внутренних и внешних воздействий на систему;
- показ взаимодействия требований и актёров.

Диаграмма вариантов использования состоит из вариантов использования, которые представляют собой некоторую высокоуровневую функцию системы определённую через анализ требований, актёров, которые являются чем-то, что взаимодействует с системой и связей между вариантами использования и актёрами [10].

Актёром может быть как человек или организация, которая использует систему, так и внешний сервис взаимодействующий с ней. Графически актёр обычно представляется в виде человечка.

Взаимодействие актёра и варианта использования отражается посредством стрелки между ними.

При создании варианта использования стоит уделять внимание выбору его имени. Оно должно чётко определять выполняемую функцию и начинаться с глагола.

Варианты использования могут включать другие и расширять их. Варианты использования создаются и рисуются от самых крупных уровней к самым мелким.

В разрабатываемой библиотеки можно выделить программиста, который использует её, как актёра. В свою очередь как варианты использования можно выделить части упрощённого интерфейса библиотеки см. рис. 2.2, что даёт нам три варианта использования:

- “Выполнить препроцессинг” карты;
- “Найти путь”, который реализован двумя разными алгоритмами и может быть расширен с помощью Goal Bounding алгоритма;
- “Загрузить карту”, который реализован в виде загрузки простой текстовой карты.

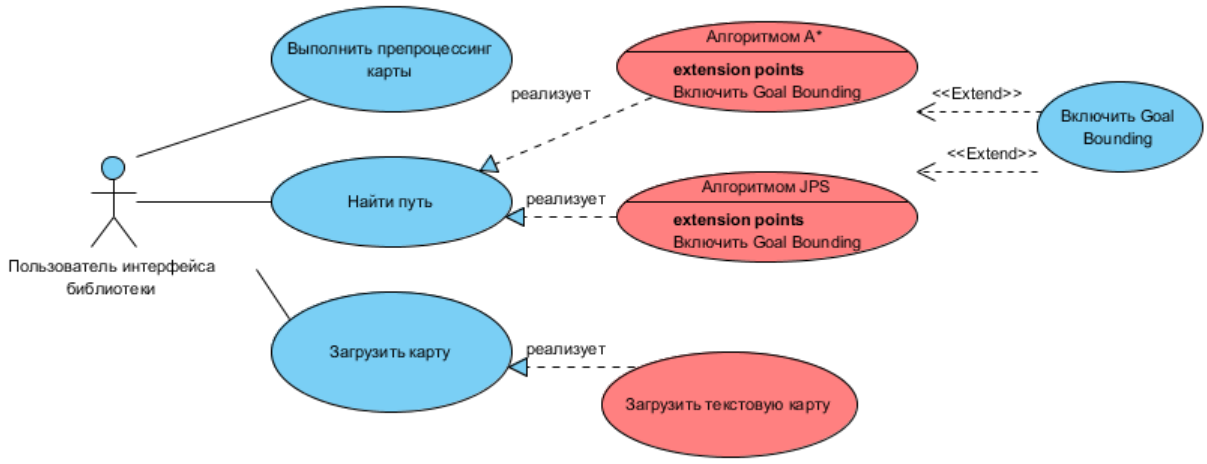

Рисунок 2.2 — Диаграмма Use case

2.3 Требования к ПО

Библиотека должна предоставлять интерфейс для нахождения путей различными алгоритмами с поддержкой:

- различных реализаций карт;
- различных эвристик;
- включения и выключения Goal Bounding алгоритма;
- возможностью опционально получить информацию для отладки алгоритмов.

Должен иметься интерфейс для работы с Goal Bounding алгоритмом включающий в себя: препроцессинг карты, сохранение и загрузка результатов.

Так же библиотека должна предоставлять базовую реализацию карты и её загрузчика.

Библиотека должна зависить только от стандартной библиотеки и библиотеки threadpool11.

Библиотека не должна включать платформозависимый код и в следствии чего должна компилироваться на различных платформах, таких как: x32, x86-64, arm.

Библиотека должна иметь простой способ включения в другие проекты.

2.4 Описание интерфейса взаимодействия с библиотекой

Для работы с библиотекой требуется её включение в проект по средством CMake или её подключение в бинарном виде.

Основным интерфейсом взаимодействия с библиотекой является интерфейс нахождения путей PathFinder<CoordsType> см. рис. 2.3, который имеет метод find с аргументами начальной,

конечной точки и типа актёра (его проходимость), а возвращать массив точек представляющих собой найденный маршрут.

```
template<class CoordsType>
class PathFinder
{
public:
virtual std::vector<Point<CoordsType>> find( CoordsType startX,
          CoordsType startY,
          CoordsType targetX,
          CoordsType targetY,
          uint32_t unitType ) = 0;
};
```

Рисунок 2.3 — Интерфейс нахождения путей

Так же предоставляется интерфейс для модификации работы алгоритмов (типа карты, эвристики, Goal Bounding) посредством шаблонов. Из-за того, что количество параметров шаблонов большое, библиотекой предоставляется несколько предзаданных вариантов.

Библиотекой предоставляется интерфейс и реализация базовой версии карты, который включает в себя загрузку карты, получение её параметров и отдельных клеток, а так же выполнение препроцессинга Goal Bounding.

3 ОПИСАНИЕ ПРОГРАММНОЙ РЕАЛИЗАЦИИ

При реализации библиотеки и визуализатора возникли задачи не связанные с реализацией самих алгоритмов, такие как: корректное измерение времени работы алгоритмов и включение отдельных методов в зависимости от типа с которым работает алгоритм.

Для корректного измерение времени был создан класс MeasureUtils, который включает в себя методы для измерения скорости работы обычных функций и методов классов. Данные методы являются шаблонными, что позволяет добиться их универсальности. Они принимают настройки тестирования, которые содержат количество вызовов функции для прогрева и количество вызовов для измерения скорости, так же передаётся сама функция и её аргументы. Прогрев нужен что бы данные и инструкции гарантировано оказались в кэше процессора, чем уменьшили влияние времени на пересылку данных из оперативной памяти и кэшей нижнего уровня. Время выполнения функции равно общему времени выполнения её в цикле поделённому на количество итераций в цикле.

Включение и выключение методов в данной работе требуется для работы со знаковы и беззнаковыми целыми типами - для беззнаковых типов не требуется проверка на то, является ли переменная отрицательной, что нужно для проверки находится ли точка в границах карты. Данная проблемы решена двумя макросами “PF_FUN_ENABLE_IF_SIGNED” и “PF_FUN_ENABLE_IF_UNSIGNED” см. рис. 3.1.

```
#define PF_FUN_ENABLE_IF_SIGNED( ValueType, ReturnType ) \
template<class Q = ValueType> \
enable_if_t<std::is_integral<Q>::value &&  std::is_signed<Q>::value, ReturnType>

#define PF_FUN_ENABLE_IF_UNSIGNED( ValueType, ReturnType ) \
template<class Q = ValueType> \
enable_if_t<std::is_integral<Q>::value &&  std::is_unsigned<Q>::value, ReturnType>
```

Рисунок 3.1 — Макросы для включения методов

3.1 Реализации алгоритма A*

Алгоритм A* является базовым и наиболее часто используемым алгоритмом для поиска путей. Для работы алгоритм использует открытый и закрытый список. В открытый список добавляется начальная точка, открытый список содержит точки, которые мы уже нашли, но ещё не рассмотрели. После чего пока в открытом списке существуют точки – выбирается точка с наименьшей стоимостью и рассматриваются её соседи. Если стоимость пути через данную точку до соседа меньше записанной в него стоимости, то стоимость пути до него изменяется

и его родитель меняется на текущую клетку см. рис. 3.2. После рассмотрения точки - она добавляется в закрытый список. Если открытый список оказался пуст – значит пути до конечной точки не существует. Если из открытого списка была взята конечная точка - это означает, что путь существует и можно реконструировать его. Для этого мы берём родителя конечной точки и рекурсивно рассматриваем её родителей в массив, пока не дойдём до начальной точки см. рис. 3.3.

```
NodeInfo& nextNodeInfo = m_nodesInfo[nextNodeIdx];
if( !nextNodeInfo.IsVisited() || newCost < nextNodeInfo.GetNodeCost())
{
  nextNodeInfo.SetVisited( true );
  nextNodeInfo.SetNodeCost( newCost );
  nextNodeInfo.SetCameFrom( best.x, best.y );
  float priority =
    newCost +
    Heuritic::nodes_distance( nextX, nextY, targetX, targetY, startX, startY );
  openList.push( PriorityNode<CoordsType>( nextX, nextY, priority ));
}
```

Рисунок 3.2 — Добавление в открытый список в A*

```
std::vector<Point<CoordsType>> path;
auto current = Point<CoordsType>( targetX, targetY );
path.push_back( current );
while( !current.Equals( startPoint ))
{
  auto nodeIdx = coords_to_contiguous_idx( current.x, current.y, m_map.GetSizeX());
  const auto& info = m_nodesInfo[nodeIdx];
  if( current == info.GetCameFrom())
  {
    break;
  }

  current = info.GetCameFrom();
  path.push_back( current );
}
```

Рисунок 3.3 — Реконструкция пути

При написании A* для открытого списка была использована очередь с приоритетом std::priority_queue, а для закрытого списка - вектор имеющий размер карты. Такая реализация закрытого списка дала возможность делать проверку на вхождение в него за константное время. Память для закрытого списка выделяется один раз при создании экземпляра алгоритма поиска и при каждом поиске у всех клеток сбрасывается флаг посещённости.

3.2 Реализация алгоритма JPS

Алгоритм JPS представляет собой усовершенствование алгоритма A*. Недостатком A* является то, что он добавляет в открытый список все клетки, которые являются прямыми соседями рассматриваемой и имеют больший вес чем вес текущая клетка плюс стоимость пути между ними, JPS в свою очередь предоставляет возможность пропуска добавления многих клеток на основе возможной симметрии путей. Множество симметричных путей возникает на отрытых пространствах. Пути называются симметричными, потому что они практически идентичны. Алгоритм A* не учитывает возможность симметричности путей, тогда как JPS использует её вводя некоторые допущения о карте для увеличения производительности поиска.

Для пропуска добавления лишних клеток вводятся функции прыжков, которые делятся на горизонтальные, вертикальные и диагональные.

Логика работы горизонтального и вертикального прыжка одинакова. Рассмотрим горизонтальный прыжок право - остальные варианты происходят по аналогии с ним. При этом мы можем сделать следующие допущения [11]:

- клетка из который мы пришли может быть проигнорирована;
- клетки по диагонали позади рассматриваемой, мы тоже можем игнорировать, так как они были достигнуты из родительской клетки см. рис. 3.4;
- клетки выше и ниже рассматриваемой могут быть достигнуты оптимальнее из её родителя см. рис. 3.5;
- клетки правее и выше/ниже рассматриваемой могут быть достигнуты оптимальнее из клеток на одну левее;

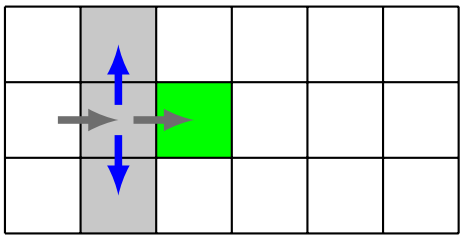

Рисунок 3.4 — Отброшенные клетки сзади

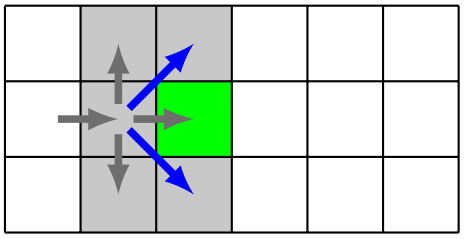

Рисунок 3.5 — Отброшенные клетки сверху и снизу

Эти допущения приводят к тому что алгоритм должен рассматривать только клетки правее от текущий пока путь не содержит препятствий см. рис. 3.6. Однако путь не всегда свободен от препятствий, что ломает приведённой допущение. Это происходит в том случае, когда клетка сверху или снизу рассматриваемой является препятствием, что делает утверждение о том, что оптимальный путь до диагональной клетки не проходит через рассматриваемую см. рис. 3.7. В такой ситуации прыжок останавливается и клетка по диагонали (такая клетка называется вынужденным соседом) и текущая клетка добавляются в открытый список для дальнейшего рассмотрения.

Последним допущением является то, что если при прыжке препятствие блокирует продвижение в заданном направлении - весь прыжок может быть отброшен см. рис. 3.8.

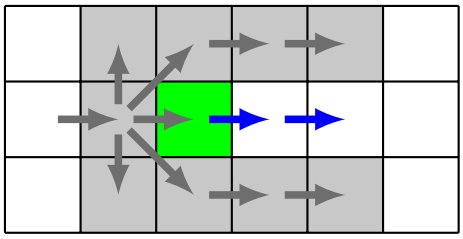

Рисунок 3.6 — Отброшенные клетки сверху и снизу

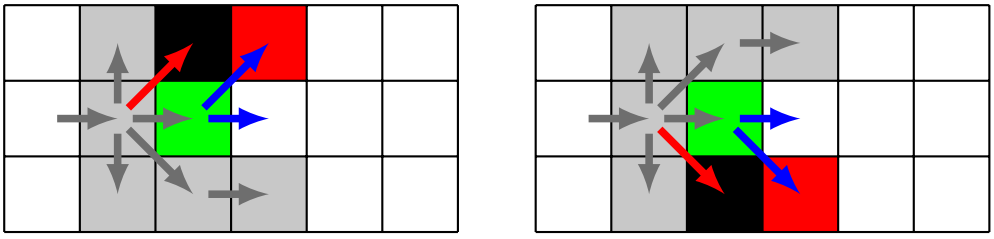

Рисунок 3.7 — Вынужденный сосед

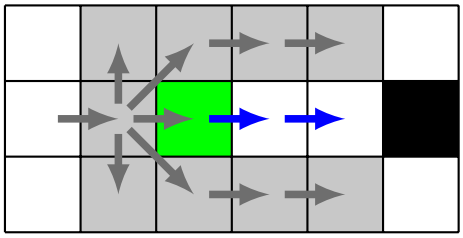

Рисунок 3.8 — Заблокированный путь

Такие же правила и допущения верны для диагональных прыжков. Рассмотрим прыжок вправо вверх по диагонали, можно предположить, что клетки снизу, снизу справа, слева и слева сверху можно оптимально достичь через родителя рассматриваемой клетки. В следствии чего остаётся рассмотреть три клетки: сверху, справа и по диагонали см. рис. 3.9. В отличии от горизонтального и вертикального прыжка в данном случае осталось три клетки для рассмотрения, однако для двух из них требуется вертикальный и горизонтальный прыжок. Так

как требуемые прыжки уже определены - вначале происходят они, затем если в результате прыжков не было найдено клеток для дальнейшего рассмотрения - происходит прыжок по диагонали на одну клетку и процесс повторяется.

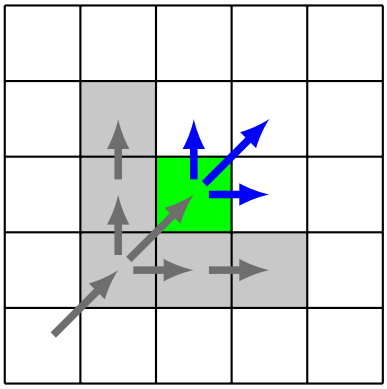

Рисунок 3.9 — Прыжок по диагонали

По аналогии с горизонтальным и вертикальным прыжком при диагональном прыжке так же встречаются вынужденные соседи, когда слева или под текущей клеткой находится препятствие, клетка слева вверху или справа внизу соответственно являются вынужденными соседями см. рис. 3.10.

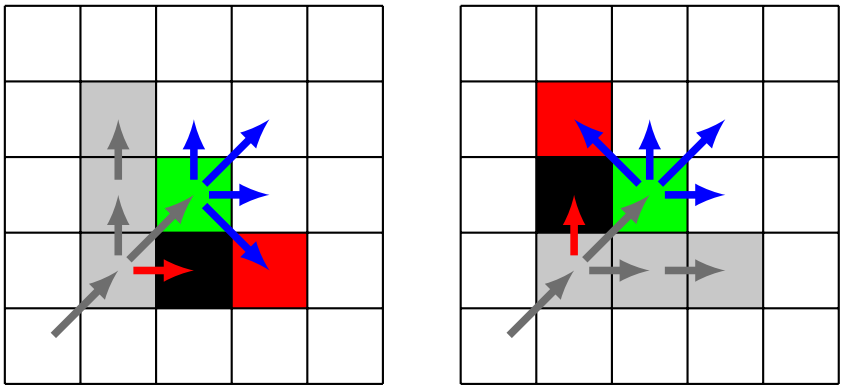

Рисунок 3.10 — Вынужденный сосед при диагональном прыжке

Когда прыжок закончен берётся клетка с наименьшим весом из открытого списка и из неё происходит прыжок в направлении в котором алгоритм пришёл в неё.

JPS начинается с того, что из начальной точки происходят прыжки во все восемь сторон. Затем выполняется цикл по открытому списку и если в нём существует клетка, то из неё происходит вертикальный и горизонтальный прыжок, затем выполняется проверка не допускающая прохождение между двумя непроходимыми клетками. После чего запускается диагональный прыжок.

Диагональный прыжок происходит пока следующая клетка проходима, не является конечной клеткой и находится на карте. Пусть текущая клетка имеет координаты (x_c, y_c), а направлением прыжка является пара дельт (d_x, d_y) принимающих значения

$d_x, d_y \in \{-1, 0, 1\}$. Сперва рассматривается соседняя клетка $(x_c, y_c + d_y)$ и если она проходима и соседняя $(x_c - d_x, y_c)$ не проходима, а клетка с координатами $(x_c - d_x, y_c + d_y)$ проходима (путь не заблокирован см. рис. 3.11), то текущая клетка проверяется на возможность добавления в открытый список. Такая же проверка происходит с тройкой точек $\{(x_c + d_x), (x_c, y_c - d_y), (x_c + d_x, y_c - d_y)\}$ соответственно. Если прыжок вперёд заблокирован двумя клетками по горизонтали и вертикали $\{(x_c, y_c + d_y), (x_c + d_x, y_c)\}$ см. рис. 3.12, то прыжок прекращается, иначе проводится горизонтальный и вертикальный прыжок и если хотя бы один из них нашёл клетку для дальнейшего рассмотрения, то текущая клетка добавляется в открытый список, при этом такие прыжки в стороны не добавляют клетки в открытый список.

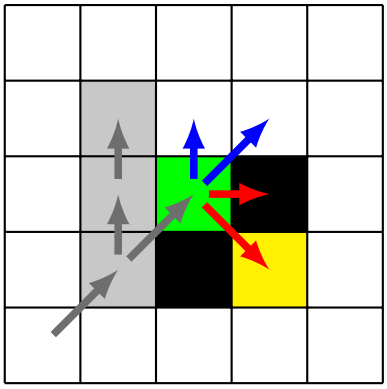

Рисунок 3.11 — Заблокированный вынужденный сосед

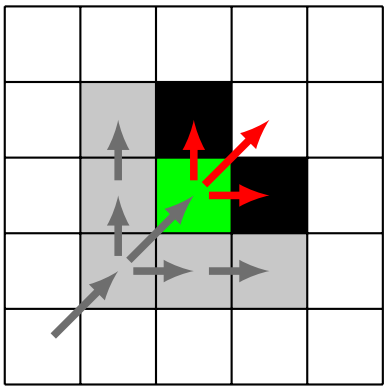

Рисунок 3.12 — Заблокированный прыжок по диагонали

Горизонтальный и вертикальный прыжки происходят пока следующая клетка проходима, не является конечной клеткой и находится на карте. На каждом шаге происходит проверка на наличие вынужденных соседей и если такой сосед найден, то он добавляется в открытый список и метод прерывается.

Когда открытый список исчерпан - происходит восстановление пути так же как и в A*.

3.3 Реализация и интеграция GoalBounds

Алгоритм Goal Bounding можно разделить на два этапа: этап препроцессинга и проверка направления во время выполнения поиска пути. Во время препроцессинга происходит прохождение по всем клеткам карты, для каждой клетки выполняет волновой алгоритм, который заполняет все клетки стоимостью путей до них от начальной клетки и изначальным направлением по которому алгоритм пришёл в данную клетку. После чего для данной клетки определяется 8 ограничивающих прямоугольников см. рис. 3.14, по одному на каждое направление см. рис. 3.13. Определяются они по такому алгоритму:

- делаем минимальную точку прямоугольника равную координатам правой верхней точки карты, а максимальную - равной левой нижней;

- идём по всем точкам и расширяем прямоугольник соответствующий направлению в котором точка была достигнута из начальной, что бы прямоугольник включал её.

```
auto& bbNode = m_bbMap.GetNodeBB( startX, startY );
for( uint16_t i = 0; i < map.GetSizeX(); i++ )
{
  for( uint16_t j = 0; j < map.GetSizeY(); j++ )
  {
     auto& ffNode = ffMap->GetNode( i, j );
     if( map.IsPassable( i, j, type ))
     {
       auto& dirBB = bbNode.boundingBoxes[static_cast<int>(ffNode.originalDir)];
       dirBB.minX = std::min( dirBB.minX, i );
       dirBB.maxX = std::max( dirBB.maxX, i );
       dirBB.minY = std::min( dirBB.minY, j );
       dirBB.maxY = std::max( dirBB.maxY, j );
     }
   }
 }
```

Рисунок 3.13 — Определение ограничивающих прямоугольников

Так как алгоритм вычисляет ограничивающие прямоугольники для каждой клетки независимо, то его можно распараллелить. Для этого был использован пул поток, реализованный библиотекой threadpool11. Для каждой клетки создаётся своя задача и добавляется в очередь, после добавления всех задач происходит ожидание их завершения. Каждый поток имеет свою карту для волнового алгоритма которая создаётся один раз для одного потока. После исполнения алгоритма результат записывается в файл рядом с файлом карты, в файле поочерёдно, для каждой клетки, записано 32 целочисленных значения - четыре значения для каждого из восьми направлений из клетки, которые являются верхней левой и правой нижней точкой прямоугольника. Если файл с результатами вычислений уже существует на диске, то загружается он, что бы не проводить трудоёмкие вычисления заново.

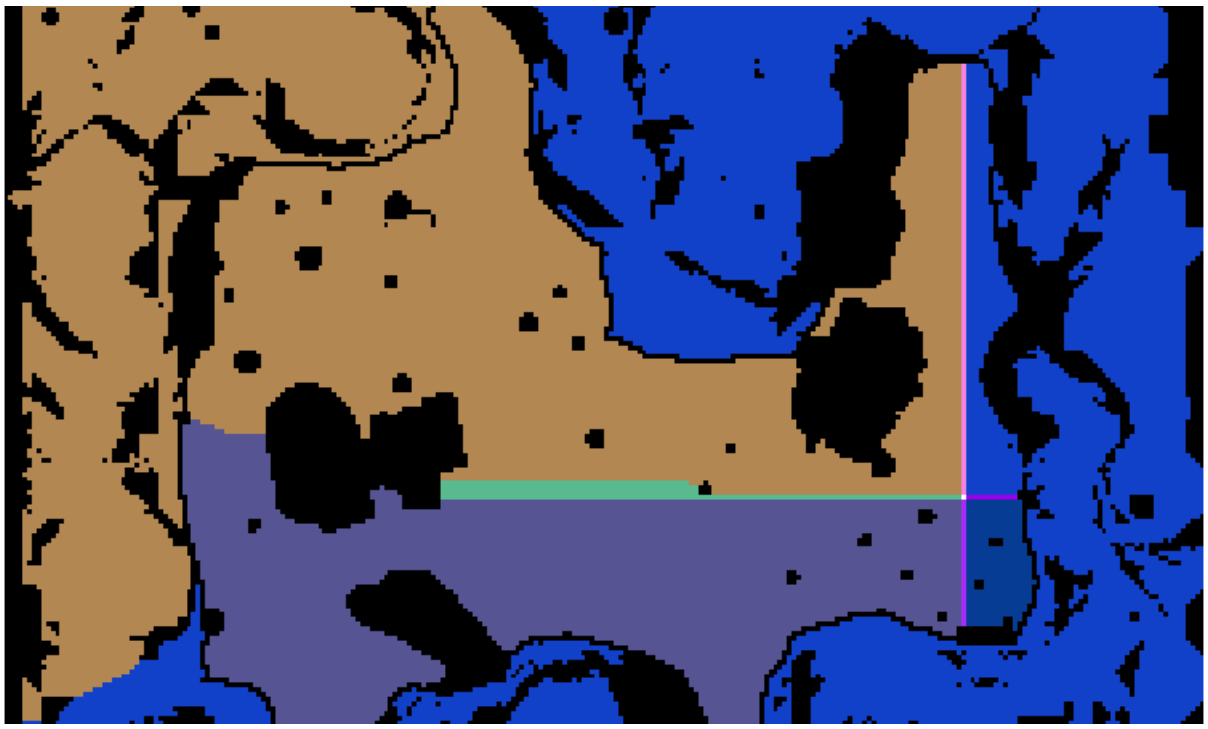

Рисунок 3.14 — Часть карты с раскрашенными в разные цвета ограничивающими прямоугольниками

Для включения Goal Bounding в A* требуется добавить проверку на правильность направления сразу после проверки на вхождение точки в границу карты при циклическом обходе соседей текущей точки см. рис. 3.15.

```
for( int i = 0; i < 8; i++ )
{
  const auto& delta = coordDeltas[i];
  const CoordsType nextX = best.x + delta.first;
  const CoordsType nextY = best.y + delta.second;
  if( m_map.IsNodeOnMap( nextX, nextY ))
  {
    if( TestGoalBounding( best.x, best.y, targetX, targetY, dirs[i] ))
    {
      const MapNode& node = m_map.GetNode( nextX, nextY );
      if( node.IsPassable( unitType ))
      {
```

Рисунок 3.15 — Проверка Goal Bounds

Для добавления Goal Bouding в JPS требуется провести аналогичное действие, но для всех функций прыжков.

3.4 Реализация визуализатора

Для реализации визуализатора были использованы кроссплатформенные библиотеки SFML и SFGUI. Карта рисуется несколькими слоями, которые для удобства и скорости отрисовываются в отдельные текстуры. Визуализатор рисует такие слои: слой карты, последний найденный путь, открытые клетки алгоритмом A*, направления прыжков алгоритма JPS, ограничивающие прямоугольники алгоритма Goal Bounding для выбранной клетки.

Нахождение пути происходит после выбора начальной и конечной точки, после чего он отображается на карте и выводится приблизительное время выполнения алгоритма.

Визуализатор, кроме отображения путей с отладочной информацией имеет возможность сравнивать скорость выполнения алгоритмов для выбранного пути.

3.5 Реализация сравнения алгоритмов

Для сравнения алгоритмов был создан отдельный программный модуль, который позволяет

4 АНАЛИЗ РЕЗУЛЬТАТОВ

4.1 Сравнительный анализ алгоритмов

Сравнительный анализ алгоритмов проводился на картах из таких игр: “Dragon Age: Origins”, “Warcraft III”, “Baldurs Gate II ”. Для выполнения анализа были взяты карты и пути с сайта “movingai.com/benchmarks/”, где собраны подготовленные наборы данных состоящие из карт и сценариев к ним. В ходе сравнительного анализа было использовано 267 различных карт: 36 из “Warcraft III”, 156 из “Dragon Age: Origins” и 75 из “Baldurs Gate II ”. Количество протестированных путей в общей сложности составило 1070000. Для Astar и JPS без модификации Goal Bounding было протестировано больше путей чем с данной модификацией, так как карты с размерностью более 512 на 512 были исключены из препроцессинга. На карты с большим размером время препроцессинга оказывалось очень большим.

Тестирование алгоритмов проводилось на компьютере со следующими характеристиками: процессором AMD FX-8320 с частотой 3.5 ghz и оперативной памятью 16gb DDR3 1333 mhz. Программа выполнялась под операционной системой Windows 10 без каких-либо запущенных сторонних процессов. Программа для тестирования была скомпилирована с ключом O3 компилятором g++ версии 5.3.

В ходе анализа алгоритмов было получено более миллиона уникальных векторов данных, которые первоначально экспортились в формат csv, однако для последующего анализа они были импортированы в реляционную базу данных PostgreSQL, что сильно упростило манипуляцию с данными.

Рассмотрим результаты выполнения алгоритмов на карте AR0011SR размером 512 на 512 см. рис. 4.1.

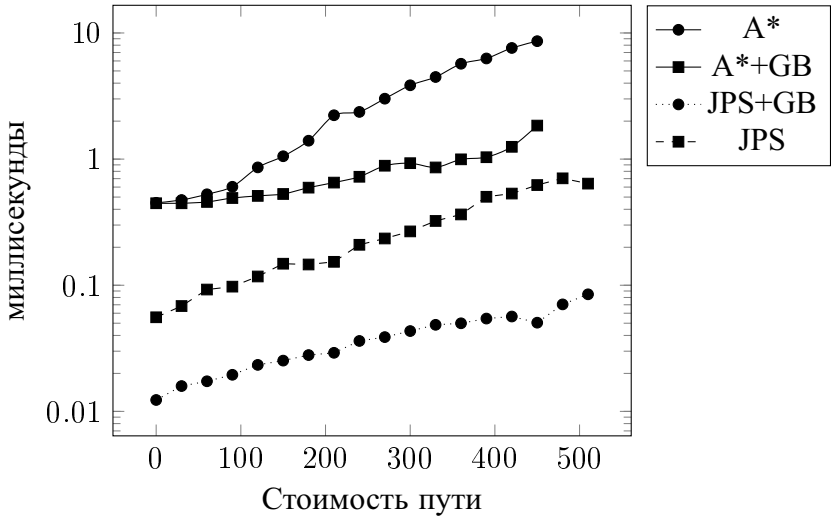

Рисунок 4.1 — Сравнение алгоритмов на карте AR0011SR

Самой быстрой является комбинация алгоритма JPS и Goal Bounding, при этом время выполнения одного вызова варьировалось от 12 до 80 микросекунд. Время выполнения JPS варьировалось от 55 до 700 микросекунд, что в 3-10 раз хуже чем с Goal Bounding см. рис. 4.2. A* с и без Goal Bounding на коротких путях показывают примерно одинаковое время. На длинных путях модификация Goal Bounding даёт преимущество в несколько раз (в 3-5 раз). Время выполнения A* доходит до 9 миллисекунд, что для игровых приложений очень долго, так как это время близко ко времени полной отрисовки кадра, а найти может потребоваться больше одного пути.

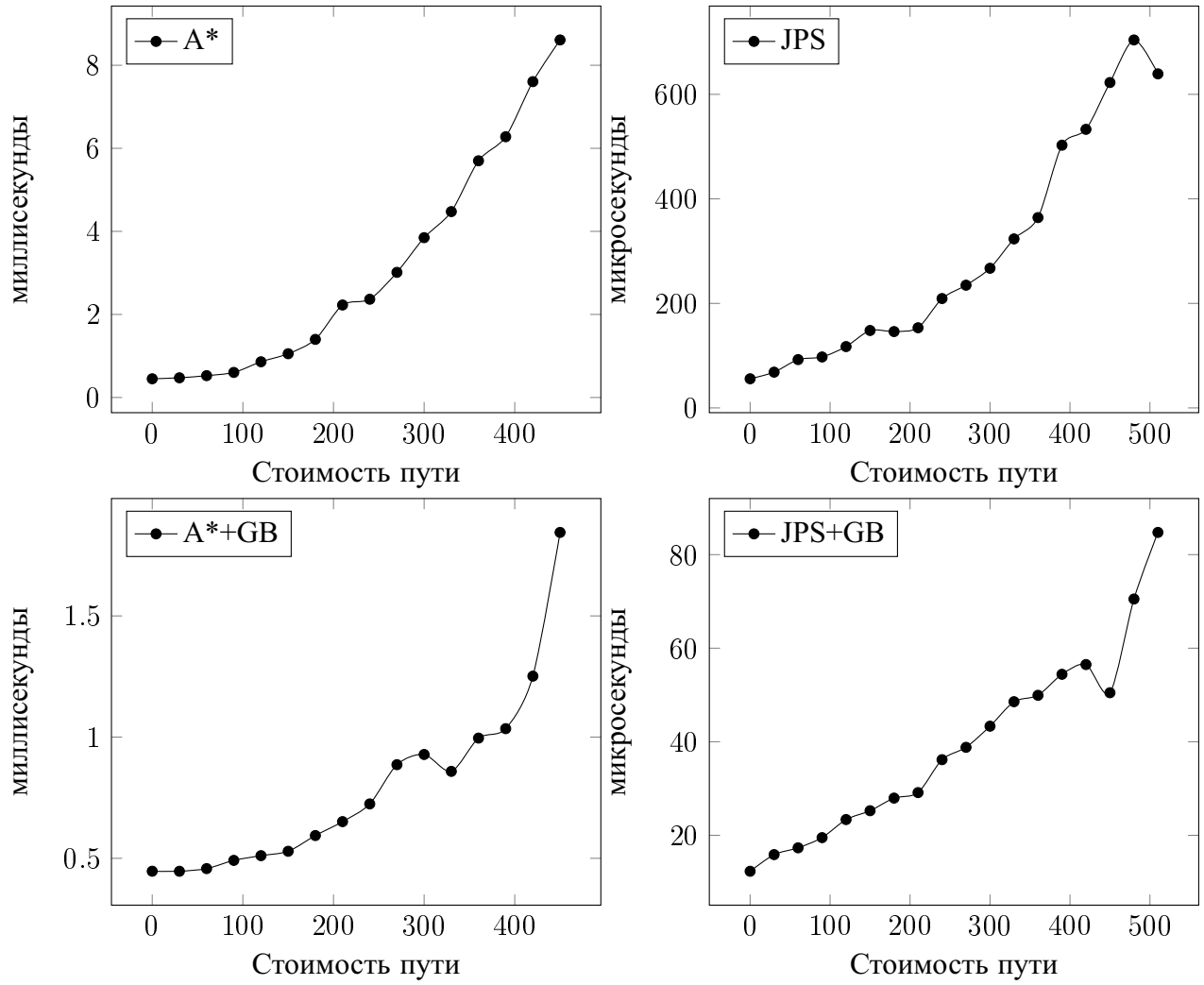

Рисунок 4.2 — Зависимость длительности одного вызова поиска пути от его стоимости для карты AR0011SR (512x512)

На разных картах алгоритмы могут вести себя по разному см. рис. 4.3. В случае карты brc501d алгоритм JPS в большинстве случаев медленней чем комбинация A* и Goal Bounding, хотя всё равно остаётся намного быстрей обычного A*. В данном случае карта имеет такое строение, что многие пути имеют вид “крюков” огибающих очень длинное препятствие, в таком случае алгоритм JPS сканирует очень много клеток и хотя в открытый список добавляются немногие из них, большое количество клеток проходится безрезультатно. На рисунке 4.4

показан такой случай - непроходимые клетки обозначены чёрным цветом, красная ломаная показывает найденный путь, а сине-зелёные полоски показывают направления в которых проходил алгоритм JPS. Как можно заметить для нахождения пути алгоритму пришлось пройти всю карту больше одного раза.

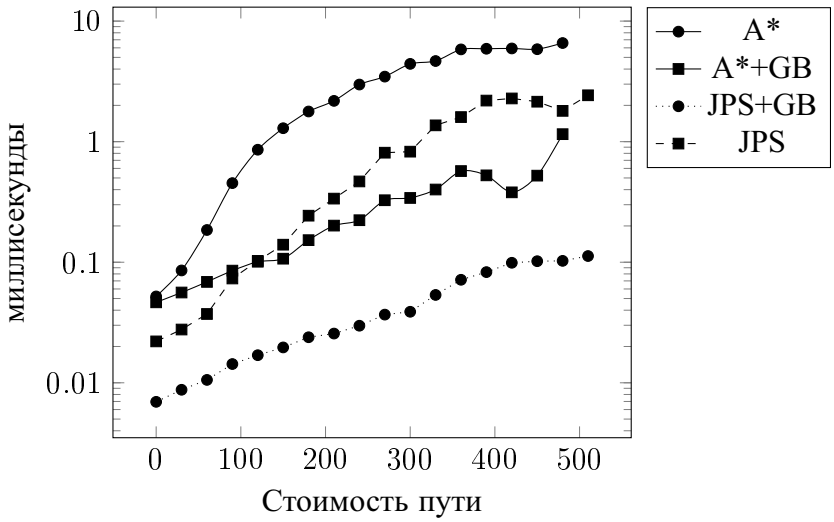

Рисунок 4.3 — Сравнение алгоритмов на карте brc501d

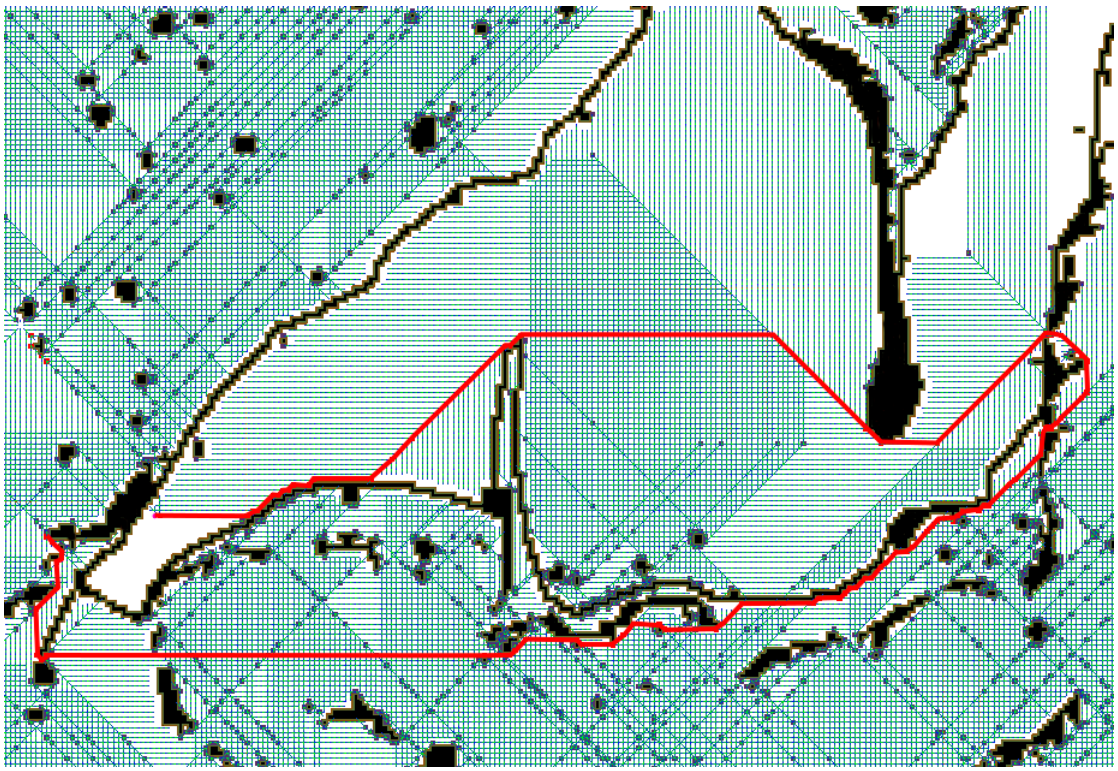

Рисунок 4.4 — Неоптимальность обхода карты brc501d алгоритмом JPS

Таблица 4.1 показывает абсолютную и относительную скорость алгоритмов в зависимости от длины пути. В таблице приведены значения продолжительности вызова одного запроса на нахождение пути в миллисекундах, так же продолжительность в процентах от алгоритма A*.

Таблица 4.1 — Сравнение скорости выполнения алгоритмов на всех картах

Стоимость	A* мс	A* + GB мс	% от A*	JPS мс	% от A*	JPS+GB мс	% от A*
000	0,365	0,160	43,80%	0,024	06,50%	0,005	01,28%
030	0,494	0,252	51,09%	0,050	10,19%	0,010	01,97%
060	0,488	0,284	58,10%	0,065	13,33%	0,012	02,45%
090	0,569	0,312	54,87%	0,084	14,81%	0,014	02,49%
120	0,722	0,337	46,63%	0,103	14,27%	0,016	02,27%
150	0,912	0,366	40,13%	0,122	13,43%	0,018	02,01%
180	1,166	0,395	33,87%	0,150	12,84%	0,021	01,76%
210	1,466	0,430	29,29%	0,179	12,23%	0,023	01,54%
240	1,782	0,462	25,93%	0,205	11,48%	0,025	01,38%
270	2,177	0,494	22,67%	0,240	11,03%	0,027	01,25%
300	2,635	0,534	20,28%	0,275	10,43%	0,029	01,12%
330	3,148	0,559	17,77%	0,320	10,15%	0,032	01,01%
360	3,745	0,581	15,53%	0,358	09,57%	0,033	00,89%
390	4,361	0,578	13,24%	0,418	09,58%	0,035	00,81%
420	4,855	0,549	11,31%	0,481	09,90%	0,039	00,80%
450	4,935	0,470	09,51%	0,502	10,18%	0,039	00,79%
480	4,971	0,417	08,38%	0,569	11,44%	0,040	00,81%
510	5,093	0,466	09,16%	0,775	15,23%	0,044	00,87%
540	5,320	0,453	08,52%	0,828	15,56%	0,050	00,93%

Исходя из приведённой таблицы можно сделать такие выводы:

- A* всегда медленней других алгоритмов;
- Goal Bounding больше ускоряет JPS чем A*;
- A* с Goal Bounding на длинных путях сравним и выигрывает по скорости JPS;
- JPS с Goal Bounding является самый быстрым алгоритмом и выполняется в от 50 до 100 раз быстрее чем A*;

В итоге можно сделать вывод, что для не изменяющихся карт наилучшим выбором является комбинация JPS и Goal Bounding, однако для больших карт Goal Bounding может не подходить из-за дополнительного места которое он требует как в оперативной памяти, так и на диске, это особо актуально для устройств с ограниченной памятью таких как смартфоны. Если карта изменяется или на целевой платформе ограничена память, то следует выбрать обычный JPS, однако и в этом случае выбор не однозначен – если карта имеет не однородную стоимость прохождения по клеткам, то JPS может не подойти, так как он при отбросе симметричных путей не учитывает возможность разной стоимости клеток. В таком случае стоит рассматривать алгоритм A* с и без Goal Bounding, так же возможно применение алгоритма HAA*, который был рассмотрен в данной работе в ходе анализа.

4.2 Возможные дальнейшие улучшения

В ходе выполнения работы были рассмотрены, реализованы и проанализированы несколько алгоритмов: A*, JPS, A* + Goal Bounding, JPS + Goal Bounding. В процессе чего были так же найдены и рассмотрены модификации и улучшения представленных алгоритмов, которые не вошли в данную работу. Ниже рассмотрены некоторые из них.

Одним из недостатков алгоритма JPS является то, что он работает только на картах с одинаковой стоимостью прохождения по клеткам. Однако в играх часто имеется потребность в неоднородных картах, что исключает использование данного алгоритма без дополнительных модификаций. В качестве модификации для поддержки неоднородных карт может быть рассмотрено следующие предположение: если на однородной карте вынужденные соседи возникали когда клетку преграждало препятствие, то на неоднородной карте как препятствие можно так же рассматривать смену стоимости прохождения через клетку. При этом если стоимости прохождения по клеткам для соседних клеток в большинстве случаев различны, то JPS теряет своё преимущество перед A*. Для исправления такой ситуации можно уменьшить разрешение стоимости пути, что бы у соседних клеток чаще была одинаковая стоимость. Или ввести некоторую адаптивную дельту если разность стоимости пути между клетками меньше её, то они считаются одинаковыми. Данная модификация является чисто теоретической, так как никем не была реализована на практике.

Алгоритм Goal Bounding позволяет существенно ускорить A* и JPS, однако при этом часть препроцессинга выполняется очень долго, результат занимает много места и не поддерживает изменение карты во время выполнения. Эти недостатки сильно сужают применимость данного алгоритма. Одним из решений может являться расчёт границ не для всех клеток на карте, а только для некоторых линий, которые могут быть определены как ограничивающие или преграждающие некоторую область. В результате скорость расчёта сильно увеличится, однако при этом увеличится и время поиска пути. Так же открытым является вопрос – существует ли алгоритм расчёта с более оптимальной асимптотикой.

В алгоритме JPS ключевой частью является скорость сканирования карты для нахождения препятствий. Эту операцию можно ускорить посредством анализа не одной клетки а линии из нескольких клеток за раз – используя то, что карту можно представить как битовую матрицу, где 0 – это отсутствие препятствия, а 1 – его наличие. Используя такую организацию карты можно значительно уменьшить количество чтений из памяти и количество выполнений условных операторов. Для нахождения тупика можно использовать битовую операцию ffs (find first set), которая существует на большинстве платформах и возвращает позицию первого бита равного 1. Для нахождения вынужденных соседей требуется считать по одной линии выше и ниже текущей, вынужденный сосед появляется когда за препятствием идёт свободная клетка, что для линии можно проверить простой операцией $f(L) = (L << 1)\ \&\ !L$. При этом при помощи команды ffs можно получить место потенциального вынужденного

соседа и сравнить с имеющимся тупиком на текущей линии. Так же отдельная проверка для того, что бы не пропустить цель.

Для JPS кроме Goal Bounding имеет смысл другой алгоритм препроцессинга карты - нахождение прыжковых точек для горизонтальных, вертикальных и диагональных прыжков. Такой препроцессинг очень быстр, так как в отличии от Goal Bounding ему не надо для каждой клетки исследовать всю карту и в следствии чего может выполнятся сразу при изменении карты, при чём возможно изменение информации лишь для части клеток карты.

Для изменяемых во время выполнения карт может подойти алгоритм RSR, который даёт примерно такое же ускорение как и Goal Bounding, однако не может работать на картах с разной стоимостью прохождения по клеткам. Данный алгоритм может работать как с A*, так и теоретически с алгоритмом JPS.

ВЫВОДЫ

В результате работы был проведён анализ существующих алгоритмов нахождения пути, способов представления области поиска и эвристик. Областью представления была выбрана квадратная сетка, выбранными алгоритмами поиска стали A* и JPS с модификацией Goal Bounding.

В результате работы была разработана архитектура библиотеки и интерфейса взаимодействия с ней. Была создана диаграмма вариантов использования и классов, а так же были определены программные требования. Были выбраны следующие технологии и библиотеки для создания программного продукта: C++11, threadpool11, SFML, SFGUI.

Разработана библиотека, которая реализует выбранные алгоритмы, и визуализатор для анализа эффективности функций библиотеки. Библиотека может быть легко интегрирована с проектами, в которых необходимо находить кратчайшие пути. Выполнен анализ функций библиотеки. Определена зависимость времени поиска от стоимости пути. Анализ показал, что алгоритм A* без модификаций не подходит для нахождения пути на больших картах, модификация Gaol Bounding ускоряет A* на длинных путях в несколько раз и делает его пригодным в некоторых случаях, в свою очередь алгоритм JPS постоянно быстрее A* в 5 – 10 раз и в несколько раз быстрее комбинации A* и Goal Bounding. Комбинация алгоритмов JPS и Goal Bounding дают ещё большее ускорение – до одного порядка. В результате если карта является неизменной, то JPS с Goal Bounding предоставляет самую большую скорость выполнения, иначе стоит выбрать просто JPS.

Разработанная библиотека для нахождения путей является актуальной в связи с потребностью оптимизации рассмотренных алгоритмов. Библиотека для нахождения путей в игровых приложениях решает проблему скорости поиска путей на квадратной сетке.

ПЕРЕЧЕНЬ ССЫЛОК

1. Algorithmics of Large and Complex Networks [Текст] / D. Delling, P. Sanders , D. Schultes, D. Wagner - Springer-Verlag Berlin Heidelberg, 2009. - 401 с. - ISBN 978-3-642-02093-3.

2. Алгоритмы: построение и анализ. Второе издание [Текст] / Х. Т. Кормен, Ч. И. Лейзерсон , Р. Л. Ривест, К. Штайн - М.: “Вильямс”, 2006. - 1296 с. - ISBN 0-07-013151-1.

3. Russell S. Artificial Intelligence: A Modern Approach [Текст] / S. Russell, Norvig P. - Prentice Hall, 2009. - 1152 с. - ISBN 978-0-13-604259-4.

4. Harabor D. D. Improving Jump Point Search. Pathfinding Algorithm [Текст] / D. D. Harabor, A. Grastien // тез.докл.науч.-практ. конф. ICAPS. – 2015.

5. Botea A. Near Optimal Hierarchical Path-Finding [Текст] / A. Botea, M. Muller, J. Schaeffer // Журн. Journal of Game Development. - 2011. №1.

6. Информационный портал stanford.edu - Heuristics From Amit’s Thoughts on Pathfinding [Электронный ресурс]: – Режим доступа: https://theory.stanford.edu/ amitp/GameProgramming/Heuristics.html - 12.05.2016г. - Загл. с экрана.

7. Репозиторий кода GitHub - Steve Rabin JPS+ and Goal Bounding [Электронный ресурс]: – Режим доступа: https://www.github.com/SteveRabin/JPSPlusWithGoalBounding/ - 12.05.2016г. - Загл. с экрана.

8. Mattson T. Patterns for Parallel Programming [Текст] / T. Mattson, S. Beverly, B. Massingill - Addison-Wesley, 2005. - 355 с. - ISBN 0-321-22811-1.

9. Booch G. Unified Modeling Language User Guide, The, 2nd Edition [Текст] / G. Booch, J. Rumbaugh, I. Jacobson - Addison-Wesley, 2005. - 496 с. - ISBN 0-321-26797-4.

10. Диаграмма вариантов использования (use case diagram) [Электронный ресурс] / UML Теория - Режим доступа: http://www.info-system.ru/designing/methodology/uml/theory/use_case_diagram_theory.html - 02.06.2015 р. - Загл. с экрана.

11. Информационный портал zerowidth - Jump Point Search Explained [Электронный ресурс]: – Режим доступа: http://zerowidth.com/2013/05/05/jump-point-search-explained.html - 12.05.2016г. - Загл. с экрана.

Приложение А

Слайды презентации

Приложение Б

Примеры программного кода

```
for( int i = 0; i < 8; i++ )
{
 const auto& delta = coordDeltas[i];
 const CoordsType nextX = best.x + delta.first;
 const CoordsType nextY = best.y + delta.second;
 if( m_map.IsNodeOnMap( nextX, nextY ))
 {
  if( TestGoalBounding( best.x, best.y, targetX, targetY, dirs[i] ))
  {
   const MapNode& node = m_map.GetNode( nextX, nextY );
   if( node.IsPassable( unitType ))
   {
    float newCost = 0;
    if( i >= 4 )
    {
     if( !m_map.IsPassable( nextX - delta.first , nextY, unitType )
        &&
     !m_map.IsPassable( nextX, nextY - delta.second, unitType ))
     {
      continue;
     }

     newCost = currentCost + node.GetWeight() * std::sqrt( 2 );
    }
    else
    {
     newCost = currentCost + node.GetWeight();
    }
    const auto nextNodeIdx = coords_to_contiguous_idx( nextX,
       nextY, m_map.GetSizeX());
    NodeInfo& nextNodeInfo = m_nodesInfo[nextNodeIdx];

    if( !nextNodeInfo.IsVisited() || newCost <
       nextNodeInfo.GetNodeCost())
    {
     nextNodeInfo.SetVisited( true );
     nextNodeInfo.SetNodeCost( newCost );
     nextNodeInfo.SetCameFrom( best.x, best.y );
     float priority =
     newCost +
     Heuritic::nodes_distance( nextX, nextY, targetX, targetY,
        startX, startY );

     openList.push( PriorityNode<CoordsType>( nextX, nextY,
        priority ));
    }
   }
  }
 }
}
```

```
void checkAndAddJumpPoint( CoordsType currentX, CoordsType
   currentY, CoordsType nextX, CoordsType nextY,
 short dx, short dy )
{
 const auto jumpNodeIdx = coords_to_contiguous_idx( nextX, nextY,
    m_map.GetSizeX());
 NodeInfo& jumpNodeInfo = m_closedNodes[jumpNodeIdx];

 static const auto MIN_COST_DELTA = 0.01f;

 const auto prevNodeIdx = coords_to_contiguous_idx( currentX,
    currentY, m_map.GetSizeX());
 NodeInfo& prevNodeInfo = m_closedNodes[prevNodeIdx];
 auto cost = prevNodeInfo.GetNodeCost() +
 StraightDistance::nodes_distance<CoordsType>( currentX, currentY,
    nextX, nextY, 0, 0 );

 if( jumpNodeInfo.GetNodeCost() < 0 || ( cost + MIN_COST_DELTA ) <
    jumpNodeInfo.GetNodeCost())
 {
  jumpNodeInfo.SetNodeCost( cost );
  jumpNodeInfo.SetCameFrom( currentX, currentY );

  uint32_t priority =
  cost +
  Heuritic::nodes_distance( nextX, nextY, m_targetX, m_targetY,
  m_startX, m_startY );

  m_openList.push( JPSPriorityNode<CoordsType>( nextX, nextY,
     priority, dx, dy ));
 }
}



bool horizontalJump( CoordsType currentX, CoordsType currentY,
     short dx, bool addJumpPoint )
{
 CoordsType nextX = currentX + dx;
 bool result = false;

 if( !m_map.IsPassableXSafe( nextX, currentY, m_unitType ))
 {
  return false;
 }
 if( nextX == m_targetX && currentY == m_targetY )
 {
  if( addJumpPoint )
  {
   checkAndAddJumpPoint( currentX, currentY, nextX, currentY, 0, 0
      );
  }
  return true;
 }
 if( !TestGoalBounding( currentX, currentY, m_targetX, m_targetY,
```

```
 dx > 0 ? GoalBounding::Direction::RIGHT :
    GoalBounding::Direction::LEFT ))
 {
  return false;
 }

 uint32_t testForcedPrev = ~( m_map.IsPassableYSafe( nextX,
    currentY + 1, m_unitType ) |
 (( m_map.IsPassableYSafe( nextX, currentY - 1, m_unitType )) << 1
    ));

 nextX += dx;
 while( nextX < m_map.GetSizeX())
 {
  if( !m_map.IsPassableXSafe( nextX, currentY, m_unitType ))
  {
   return false;
  }

  AddDbgLine( nextX - 2 * dx, currentY, nextX - dx, currentY );

  uint32_t testForcedCurrent = ( m_map.IsPassableYSafe( nextX,
     currentY + 1, m_unitType ) |
  (( m_map.IsPassableYSafe( nextX, currentY - 1, m_unitType )) << 1
     ));

  auto forcedPos = testForcedPrev & testForcedCurrent;     // 1 or
     2, where 1 - down and 2 up
  if( forcedPos || ( nextX == m_targetX && currentY == m_targetY ))
  {
   if( addJumpPoint )
   {
    short dy = forcedPos == 1 ? 1 : -1;
    checkAndAddJumpPoint( currentX, currentY, nextX - dx, currentY,
       dx, dy );
   }
   return true;
  }
  testForcedPrev = ~testForcedCurrent;
  nextX += dx;
 }
 return false;
}
```