

РЕФЕРАТ

Пояснительная записка содержит 18 страниц, 2 части, 3 рисунков и 5 источников.

Целью работы является анализ существующих алгоритмов поиска путей для игровых приложений, их разработка и оптимизация, а так же анализ результатов. Поставленная задача решалась с помощью языка C++11, библиотеки threadpool11, SFML и SFGUI. В результате были проанализированы, выбраны для реализации и реализованы алгоритмы нахождения путей A* и JPS с Goal Bounding модификацией.

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

СОДЕРЖАНИЕ

Введение	3
1 Анализ предметной области	4
1.1 Алгоритмы нахождения кратчайших путей	4
1.1.1 A*	4
1.1.2 JPS	5
1.1.3 HPA* и HAA*	6
1.2 Различные представления области поиска	7
1.2.1 Квадратная сетка	7
1.2.2 Quadtree (дерево квадрантов)	7
1.2.3 Navmesh (навигационная сетка)	8
1.2.4 Waypoints (путевые точки)	8
1.3 Эвристические функции	8
1.4 Goal Bounding	9
1.5 Выбор алгоритма и представления области поиска	10
1.6 Постановка задачи	10
2 Проектирование программного обеспечения	12
2.1 Программное обеспечение	12
2.2 Архитектура ПО	13
2.3 UML-моделирование ПО	13
2.4 Требования к ПО	15
2.5 Описание интерфейса взаимодействия с библиотекой	16
Выводы	17
Список литературы	18

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 маршрутизация пакетов в сети, навигация искусственного интеллекта и игрока в компьютерных играх, а так же навигация роботов в пространстве. Все компьютерные игры имеют поиск путей в том или ином виде, поэтому скорость и точность алгоритма часто влияет на качество искусственного интеллекта и восприятия игрока.

Существует несколько представлений пространства для проведения поиска путей, одним из них является квадратная сетка, которая проста для создания и используется в стратегиях реального времени и других играх с двумерной картой. Хотя квадратная сетка в большинстве случаев является неоптимальным представлением области поиска, 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.

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 для нахождения кратчайшего пути в 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 отсекал перспективные 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 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 функцию для ускорения работы, при этом гарантируя наиболее эффективное использование онной.

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 длиной 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 JPS

Jump Point Search (JPS) - эффективная техника для нахождения и отброса симметричных путей [1]. JPS представляет собой модификацию алгоритма A^* с двумя правилами отброса точек. Применение этих правил позволяет увеличить производительность поиска путей на квадратной сетки на порядок - без препроцессинга и дополнительной памяти.

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 выполняется хотя бы одно из условий [1]:

- 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, отсутствие 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 уменьшению преимущества JPS над простым A^* .

1.1.3 НРА* и НАА*

НРА* (Hierarchical Path-Finding A*) - добавляет алгоритму A* иерархическую абстракцию, разбивая карту на прилегающие друг к другу кластеры, которые соединены входами [2]. 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 применяться его сглаживание.

Алгоритм НРА* 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 НРА* 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 НРА* приводит к тому, что эти параметры должны учитываться при оценке путей между входами.

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

Основная разница с алгоритмом НРА* у алгоритма НАА* 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 НАА* имеет такие же преимущества как НРА*, 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).

Сожно 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 [3]:

- 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 существуют 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 breaking).

1.4 Goal Bounding

Goal Bounding - техника отброса заранее неподходящих точек, которая позволяет значительно ускорить поиск пути [4]. 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 поиска. 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 $4 \cdot 32 \cdot n$ байт.

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

1.5 Выбор алгоритма и представления области поиска

Выбор алгоритма сильно зависит от выбранного типа области поиска, в данной выпускной 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^* . По сравнению с JPS алгоритмы НРА* и НАА*, 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. В данной работе был выбран алгоритм JPS.

1.6 Постановка задачи

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. Создание библиотеки состоит из заданий:

- анализ алгоритмов нахождения путей;
- разработка алгоритма A*;
- разработка алгоритма JPS;
- разработка препроцессинга Goal Bounding;
- интеграция Goal Bounding в алгоритм A*;
- интеграция Goal Bounding в алгоритм JPS;
- разработка визуализатора для наглядной оценки и проверки алгоритмов;
- создание модуля для сравнения и валидации алгоритмов;

Для написания библиотеки был выбран язык C++. Для визуализации результатов была выбрана библиотека SFML и SFGUI. Для использования параллельности - библиотека threadpool11.

2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Программное обеспечение

Для написания 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 отличие 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, быстроедействие, низкоуровневые оптимизаций, возможность внедрения библиотеки в существующие программные продукты.

Стандарт C++11 привнес 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 подстроено;
- `high_resolution_clock` - наиболее точное время с наименьшим доступным периодом.

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.

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 Архитектура ПО

2.3 UML-моделирование ПО

Унифицированный язык моделирования (UML) - язык общего назначения для визуализации, спецификации, конструирования и документации программных систем [5]. Язык UML объединяет в себе семейство разных графических нотаций с общей метамоделью.

Преимуществами UML являются:

- объектно-ориентированность, что делает его близким к современным объектно-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 выпускной работе будут использованы структурная и поведенческая диаграммы, а именно диаграмма классов (Class diagram) и диаграмма вариантов использования (Use case diagram).

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;

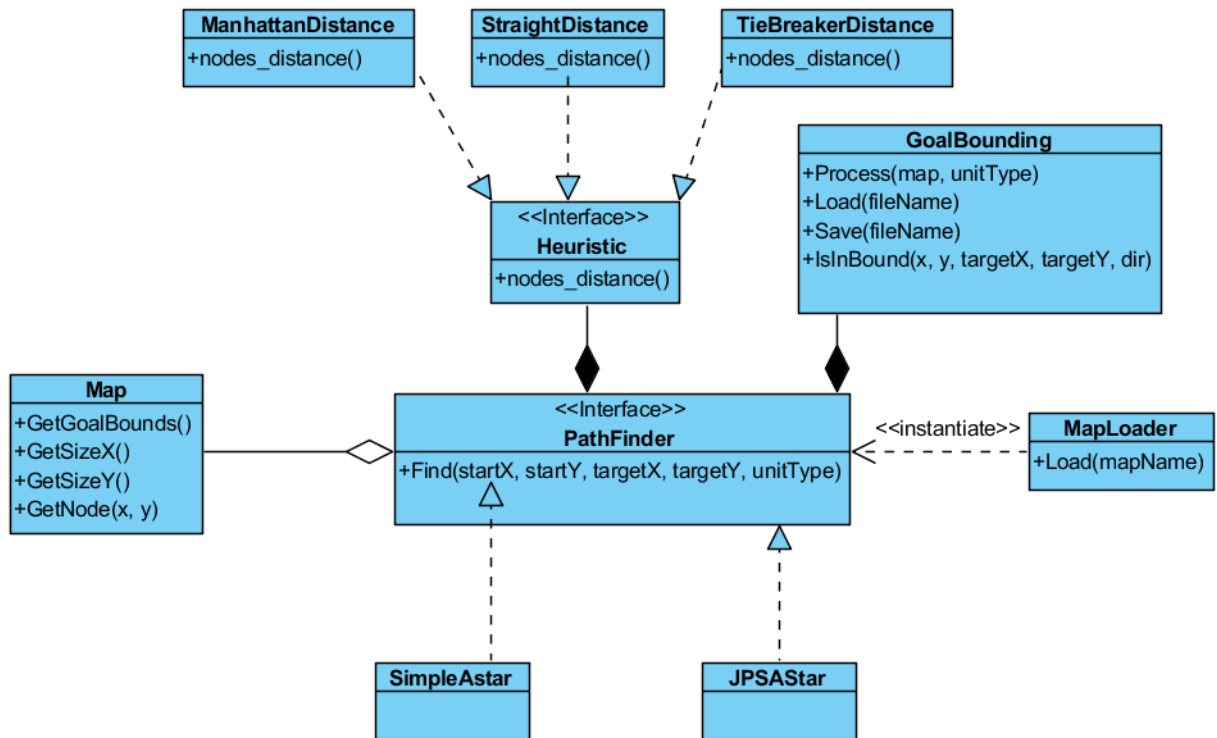


Рисунок 2.1 — Диаграмма классов

- 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.

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 библиотеке 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:

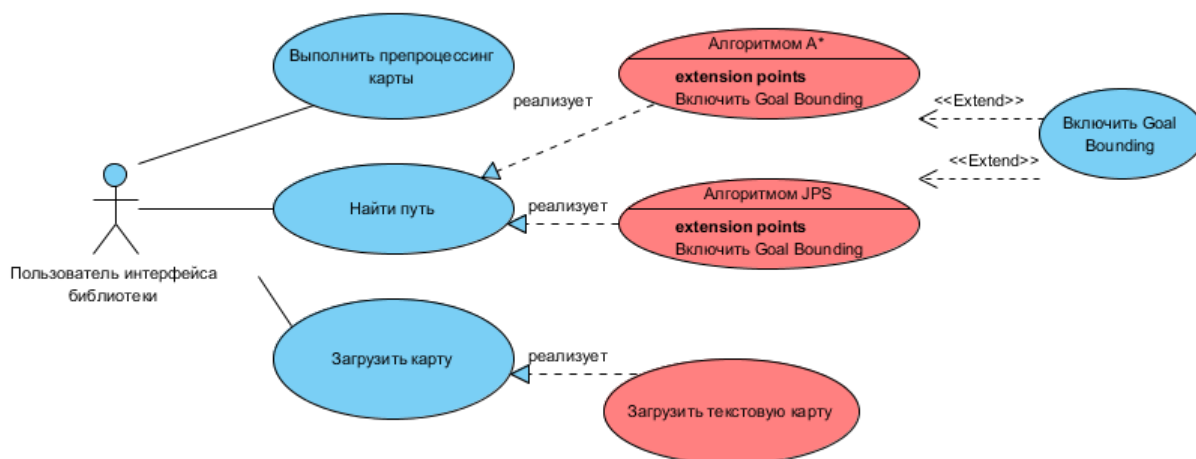


Рисунок 2.2 — Диаграмма Use case

- “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.

2.4 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 зависеть 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.

2.5 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.

ВЫВОДЫ

В ходе выпускной 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 ходе 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 библиотеки: C++11, threadpool11, SFML, SFGUI.

СПИСОК ЛИТЕРАТУРЫ

1. Harabor D. Improving Jump Point Search: статья/ Harabor D., Grastien A. - The Australian National University - 8с.
2. Botea A. Near Optimal Hierarchical Path-Finding: статья/ Botea A., Muller M., Schaeffer J. - University of Alberta - 30с.
3. Информационный портал stanford.edu - Heuristics From Amit's Thoughts on Pathfinding [Электронный ресурс]: – Режим доступа: <https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html> - Загл. с экрана.
4. Репозиторий кода GitHub - Steve Rabin JPS+ and Goal Bounding [Электронный ресурс]: – Режим доступа: <https://www.github.com/SteveRabin/JPSPlusWithGoalBounding/> - Загл. с экрана.
5. Unified Modeling Language User Guide, The, 2nd Edition / G. Booch, J. Rumbaugh, I. Jacobson - Addison-Wesley Professional, 2005. - 496 с. - ISBN 0-321-26797-4.