Санкт-Петербургский государственный университет
Факультет прикладной математики – процессов управления
Кафедра технологии программирования

Корниенко Владислав Олегович

Выпускная квалификационная работа бакалавра

Генерация NavMesh и поиск оптимальных путей

Направление 010400
Прикладная математика и информатика

Научный руководитель,
кандидат физ.-мат. наук,
доцент
Должиков В.В.

Санкт-Петербург

2017

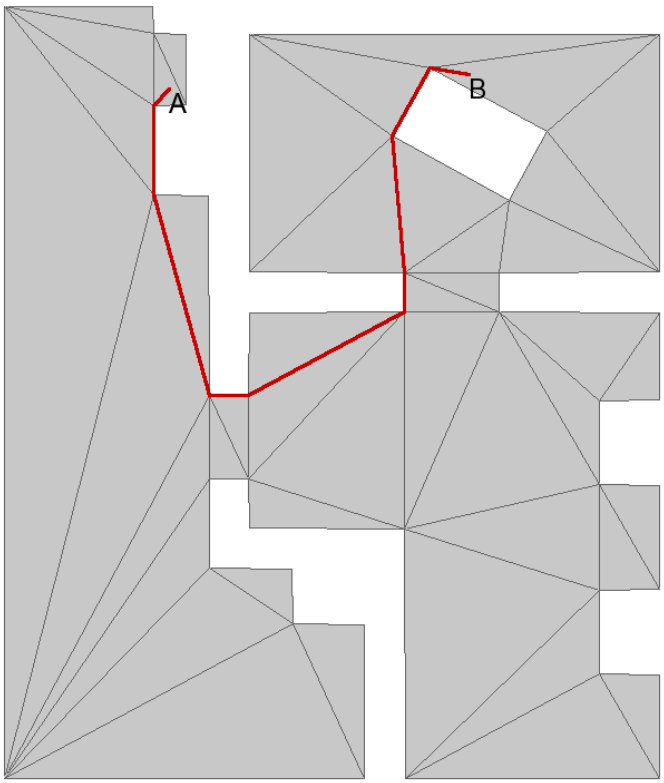
Содержание

Введение.

Для большинства современных компьютерных игр, особенно для стратегий в реальном времени, игровой опыт сильно зависит от правильной работы искусственного интеллекта. Такая часть искусственного интеллекта, как поиск пути, сильно влияет на успех игры. Поэтому игровые дизайнеры тратят огромное количество усилий, чтобы увеличить производительность алгоритмов ответственных за поиск пути. Такие алгоритмы работают с большим количеством компьютерных ресурсов, но не должны нагружать процессор.

Навигация в виртуальном мире осуществляется с помощью алгоритма A* вместе с локальным алгоритмом движения для нахождения оптимального пути. Навигационные сетки наиболее популярный подход для объединения алгоритмов поиска пути с локальными алгоритмами движения.

Рисунок 1. Пример навигационной сетки (NavMesh).



Навигационная сетка (NavMesh) - это абстрактное представление виртуального мира игры в виде ячеек, образующих доступное для перемещения компьютерных персонажей пространство (рисунок 1). Ячейки в этом пространстве являются многоугольниками. В зависимости от количества сторон у этих многоугольников, навигационные сетки делятся на триангуляционные и полигональные. Пример триангуляционной сети показан на рисунке 1 (серые многоугольники представляют пространство доступное для передвижения, белые многоугольники - препядствия). Многоугольники в навигационной сетке должны быть выпуклыми. Это необходимо для свободного перемещения персонажа игры внутри одного многоугольника. В этом случае алгоритм поиска оптимального пути может исполняться на графе, в котором ячейки являются узлами графа, а соединения между ними образуют переходы и являются ребрами графа.

Во многих случаях разработчикам требуется создавать навигационные сетки вручную, что занимает много времени и может привести к ошибкам, при которых образуются области недоступные для передвижения игровых персонажей, либо персонажи начинают застревать в углах и препятствиях. Поэтому существуют специальные алгоритмы генерации навигационных сеток для предотвращения таких ошибок.

Постановка задача

В рамках данной работы были поставлены следующие задачи:

Во-первых, сформировать общий алгоритм генерации навигационных сеток.

Во-вторых, реализовать некоторые алгоритмы поиска путей на навигационных сетках, выявить из них наиболее оптимальный.

В-третьих, необходимо создать полноценный проект, представив в нём работу оптимального алгоритма поиска путей на сгенерированной навигационной сетки в виде Unity проекта – игры в стиле TowerDefence, основной задачей которого будет преображение геймплея, с целью усложнения игрового процесса, и получения пользователем нового игрового опыта.

Обзор литературы

Для ознакомления с построением навигационных сеток и поиском путей в играх была полезна статья [1]. Генерация же навигационных сеток и алгоритм триангуляции хорошо описаны в статье [2] и книге [3]. Задача поиска пути на графах и большая часть алгоритмов рассмотрены в книгах [5] и [8]. Проблемы оптимизации алгоритма Дейкстры, эвристика и сам алгоритм A* подробно описаны в книгах [6] и [7]. Работа с игровым движком Unity, принципами его работы, написания на нём скриптов и работа с его графической составляющей в достаточной степени подробности описаны в [9] и [10].

Обзор алгоритмов генерации навигационных сеток.

Навигация может осуществляется через карты путей, диаграммы Воронова или иерархического представления виртуального окружения. Так же существуют множество приёмов локального передвижения внутри выпуклых многоугольников. Лернер (Lerner, A.) описал алгоритм, который автоматически генерирует граф ячеек и переходов, который работал как для внутреннего, так и для внешнего случая. Однако в этом алгоритме метод генерации ячеек, не гарантировал, что они будут выпуклыми. Хаумант (Haumount) создал алгоритм для создания такого графа, основанный на вокселях.

Хертел и Мехлорн (Hertel and Mehlhorn) представили неоптимальное разбиение диагоналями, которое работало только для многоугольников без дырок. Алгоритм сначала делил многоугольник на треугольники, а затем удалял несущественные диагонали. Разбиение, основанное на диагоналях, часто используется, когда необходимо сохранить общее количество вершин многоугольника. Но применяя это разбиение для навигации, нет особой необходимости сохранять количество вершин, и поэтому могут быть созданы новые вершины, если они стоят в более подходящих позициях.

Коллмен (Kallman) представил автоматический генератор триангуляционных навигационных сеток (состоящий из треугольников), основанный на триангуляции Делоне, который генерирует максимально возможное количество вырожденных треугольников. Использование триангуляционных сеток выгодно при первом подходе, так как она гарантирует, что каждая созданная ячейка будет выпуклой. Так же геометрические преобразования треугольников очень эффективные. Основным недостатком такого метода является создание множества лишних ячеек, что увеличивает время вычисления пути между двумя ячейками.

Во многих случаях навигационные сетки создаются вручную. Некоторые игровые движки и отдельные программы предлагают инструменты для автоматического создания навигационной сетки для

заданной карты. Но они либо генерируют большое количество вырожденных ячеек, либо сильно зависят от геометрии карты. Например, Velve использует генератор навигационной сетки, основанный на делении виртуального пространства на квадраты. Такой метод генерирует неоптимальное выпуклое разбиение и не очень хорошо подходит картам с произвольной геометрией.

Unreal Engine имеет свой собственный генератор навигационных сеток. Но он создаёт большое количество плохо обусловленных полигонов, может повлиять на эффективность локальных алгоритмов движения и качества созданного пути. Recast – это открытый генератор навигационных сеток, он часто используется в популярных играх, но он часто создаёт лишние ячейки, которые можно было бы легко слить вместе, что увеличивает общее количество клеток.

Глава 1. Алгоритм генерации навигационной сетки.

При создании навигационной сетки существуют два основных ограничения. Во-первых, в соответствии с алгоритмом поиска пути, требуется уменьшить количество ячеек. Это необходимо для того, чтобы алгоритм нашёл оптимальный путь настолько быстро, насколько это возможно. Во-вторых, ячейки должны быть выпуклыми, чтобы обеспечить передвижение по прямой линии внутри одной ячейки.

Ниже приводится алгоритм создания навигационной сетки в двух измерениях, при котором стартовый многоугольник может представлять поверхность, по которой будут передвигаться персонажи, а дырки в нём будут представлять неподвижные препятствия. На входе алгоритма – любой простой многоугольник (без самопересечений) с препятствиями или без. Алгоритм превращает его в оптимальную совокупность выпуклых многоугольников-ячеек, которая подходит для алгоритмов нахождения пути и избегает вырожденных многоугольников и всех случаев плохо обусловленных многоугольников.

Три основных задачи генератора навигационной сетки:

1. Создать настолько меньшее количество ячеек, насколько возможно.
2. Создать переходы как можно короче.
3. Избежать ячеек с внутренними углами близкими к нулю, так как это затрудняет движение внутри ячейки и может привести к тому, что персонаж находится в двух ячейках одновременно.

После создания совокупности ячеек, автоматически генерируется граф представляющий систему, в котором узлы это выпуклые многоугольники, получившиеся в результате разделения, а ребра графа – соединения между смежными многоугольниками.

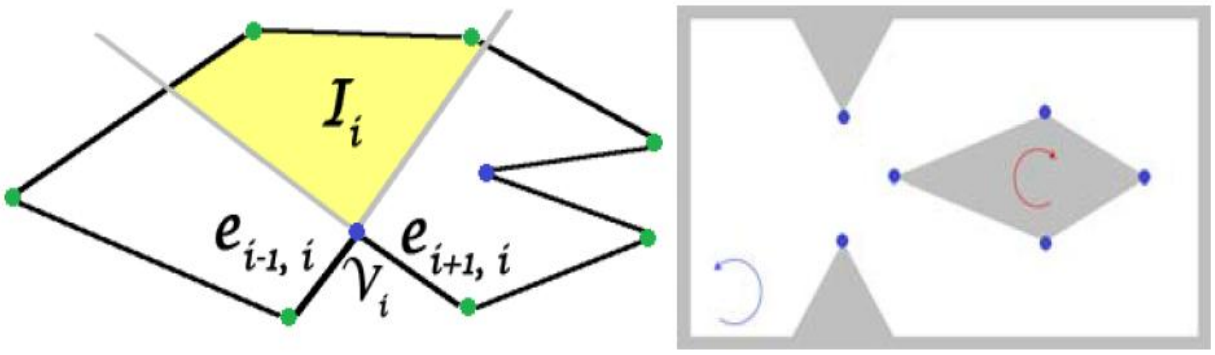
1.1. Начало алгоритма.

Существует два варианта разделения многоугольника на совокупность выпуклых многоугольников. Первый подразумевает добавление диагоналей, каждая из которых, соединяют две вершины оригинального многоугольника. Второй основан на создании отрезков между вершинами многоугольника и новыми точками, созданными на границах многоугольника или препятствий. Рассматриваемый алгоритм основан на создании новых точках, поэтому он не будет ограничен положением вершин в оригинальном прямоугольнике.

Алгоритм начинается с нахождения вершин в многоугольнике, внутренний угол которых больше π. Далее каждую такую вершину разделяют на два выпуклых угла. Это гарантирует, что в результате алгоритма начальный многоугольник будет состоять только из выпуклых ячеек.

Чтобы убедиться, нужно ли создавать новый отрезок для разбиения вершины на два новых угла, рассматривается зона I_i вершины v_i. заданная продолжениями двух граней $e_{i-1,i}$и $e_{i,i+1}$. Эта зона показана на рисунке 2, где $e_{i-1,i}$ это грань, соединяющая v_{i-1} с v_i, а $e_{i,i+1}$ это грань, соединяющая v_i с v_{i+1}.

Рисунок 2. Определение области I_i.



На рисунке 2 слева показана рассматриваемая зона I_i вершины v_i. Внутренний угол зелёных вершин меньше π, их нет необходимости разделять. Синие вершины – вершины подходящие для разделения. Справа показан простой пример начального пространства с препятствиями, который

идёт на вход алгоритму. Начальный многоугольник обозначен белым цветом, препятствия - серым.

Пространство, по которому передвигается персонаж игры, задано в виде простого многоугольника, нумерация вершин которого идёт против часовой стрелки. Любое препятствие внутри него задано многоугольником, нумерация вершин которого идёт по часовой стрелке. Препятствия представлены в виде дырок в начальном многоугольнике. Вместе они представляют пространство, идущее на вход алгоритму (рисунок 1, справа).

Пространство, идущее на вход, содержит многоугольник P, включающий в себя другие многоугольники $H_1 \ldots H_2$, которые являются дырками или пустыми многоугольниками. Пусть δP – это граница многоугольника P, и δH_i – это граница препятствия H_i. Тогда выполняются следующие условия:

1. $\delta P \cap \delta H_i = \emptyset, \forall i = 1, \ldots, h$
2. $H_i \cap H_j = \emptyset, \forall i \neq j$

Первый шаг алгоритма заключается в определении, какие из вершин являются выпуклыми, а какие являются вогнутыми. Для этого вычисляется область, треугольник, определённый тремя последовательными вершинами v_i, v_{i+1}, v_{i+2}:

$$A(v_i, v_{i+1}, v_{1+2}) = \frac{1}{2}\begin{vmatrix} \overline{v_i, v_{i+1}}, x & \overline{v_i, v_{i+2}}, x \\ \overline{v_i, v_{i+1}}, y & \overline{v_i, v_{i+2}}, y \end{vmatrix}$$

Если область $A(v_i, v_{i+1}, v_{1+2})$ положительна, то вершина v_{i+2} лежит слева от грани $e_{i,i+1}$, которая задана предыдущими вершинами v_i и v_{i+1}. Если она отрицательна, то это значит, что v_{i+2} лежит справа от грани $e_{i,i+1}$. Для основного многоугольника, вершины у которого заданы против часовой стрелки , это означает, что v_{i+1} вогнутая, и её нужно разделить. Для препятствий, вершины которых заданы по часовой стрелке, так же находится вершина, область которой отрицательна. Все разделяемые вершины мы

записываем в список $\mathcal{V}$ по порядку. Трудоемкость этого шага $O(n)$, где n – общее количество вершин в начальном пространстве.

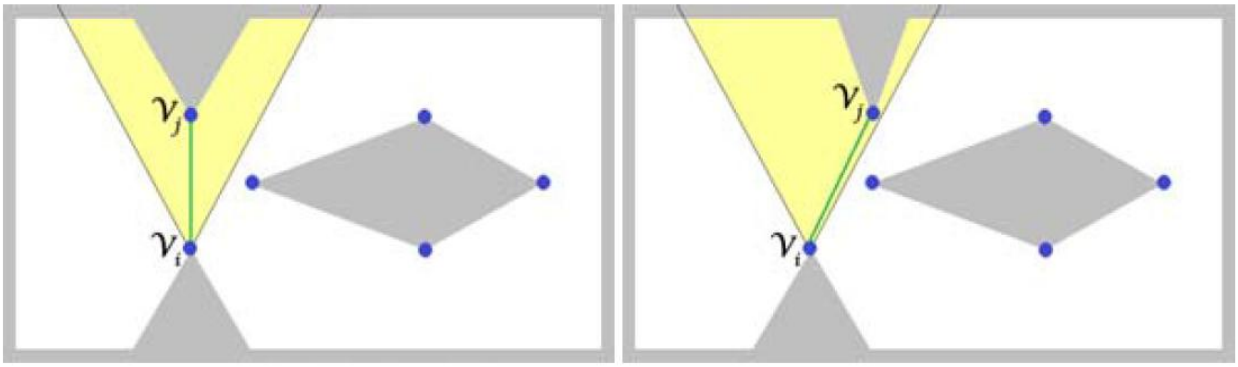
1.2. Создание переходов.

Для каждого v_i в $\mathcal{V}$ алгоритм ищет ближайший элемент, лежащий внутри области I_i, и создаёт переход до него. Трудоёмкость этого шага $O(n-r)$, где n, – количество вершин, а r – количество подходящих для разделения вершин. Таким элементом может быть другая вершина, грань начальной области или переход. В зависимости от выбора элемента можно выделить три типа переходов: вершина-вершина, вершина-грань, вершина-переход. Каждый из типов рассматривается отдельно.

1.3. Переход Вершина-Вершина.

Когда ближайший элемент к v_i это другая вершина v_j, алгоритм просто создаёт переход ρ_i между v_i и v_j вершинами. Как показано на рисунке 3, созданный переход гарантированно разделит вершину v_i на две выпуклые области. Если вторая вершина v_j так же содержится в $\mathcal{V}$ (то есть её внутренний угол является вогнутым), то алгоритм проверяет, будет ли переход ρ_i разбивать v_j на два выпуклых угла. Это произойдёт только, когда v_i находится внутри I_j, как показано на рисунке 3.

Рисунок 3. Переход вершина-вершина.



На рисунке 3 слева v_i входит внутрь I_j, поэтому её можно удалить из $\mathcal{V}$. Справа v_i не входит внутрь I_j, поэтому эту вершину нужно обработать отдельно.

1.4. Переход Вершина-Грань.

Когда ближайший элемент к v_i это грань $e_{j,j+1}$, алгоритм создаёт переход ρ_i между v_i и точкой q, лежащей на отрезке $e_{j,j+1}$. Так как отрезки, являющиеся переходами, должны быть как можно короче, находится ближайшая точка на грани, которая вычисляется как проекция v_i на $e_{j,j+1}$. В данном случае q=($proj_e\ v_i$).

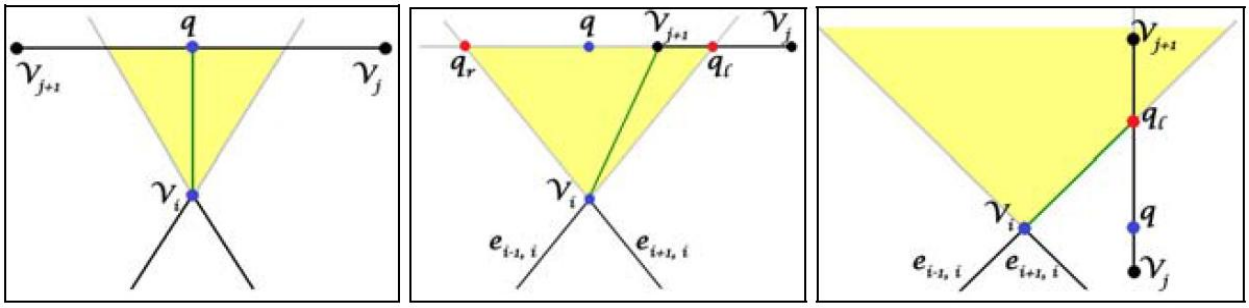
Если q находится внутри области I_i, создаётся новый переход, и алгоритм берёт следующую вершину из $\mathcal{V}$ (рисунок 4, слева). Однако, несмотря на то, что грань будет ближайшим элементом к вершине, проекция может лежать за пределами отрезка $e_{j,j+1}$ либо за пределами области I_i. И тогда перехода между вершиной и проекцией q будет недостаточно, чтобы разделить v_i на два выпуклых угла (рисунок 4, в центре и слева).

Если проекция q не подходит для создания перехода, то алгоритм выбирает точку из следующих вариантов:

- Концы отрезка $e_{j,j+1}$, v_j и v_{j+1} (рисунок 4, в центре).
- Точки пересечения q_l и q_r (если такие существуют), где q_l является точкой пересечения грани $e_{j,j+1}$ продолжением отрезка $e_{i-1,i}$ (грань слева от v_i), а q_r является точкой пресечения $e_{j,j+1}$ продолжением отрезка $e_{i,i+1}$ (грань справа от v_i) (рисунок 4, справа). Так же может быть, что точки пересечения будут являться концами отрезка $e_{j,j+1}$.

Среди четырёх возможных вершин, указанных выше, алгоритм выбирает ближайшую, лежащую внутри области I_i, и создаёт переход между v_j и выбранной вершиной.

Рисунок 4. Переход вершина-грань.



На рисунке 4 искомая точка q является проекцией (слева), искомая точка является концом отрезка $e_{j,j+1}$ (центр), искомая точка – точка пересечения (справа).

Случаи слева (проекция) и справа (точки пересечения) единственные случаи, при которых создаются новые вершины. Эти вершины всегда будут выпуклыми, поэтому алгоритму не нужно обрабатывать их далее.

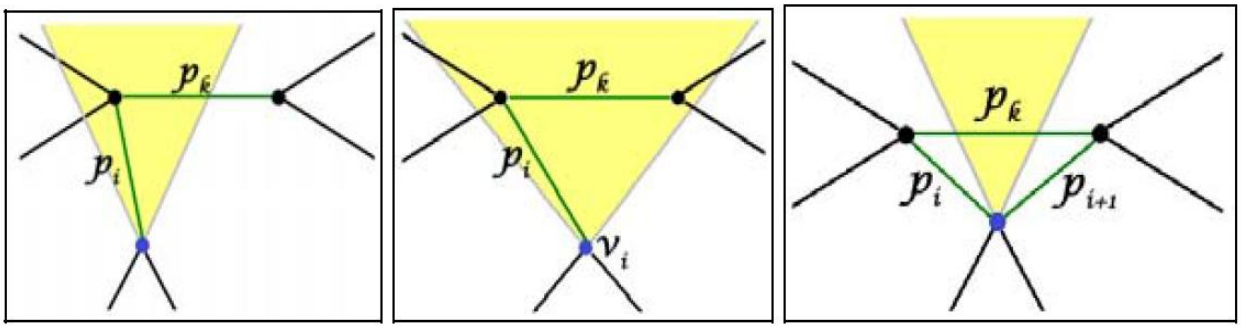
1.5. Переход Вершина-Переход.

В случае, когда ближайший элемент к v_i это уже созданный переход, создание перехода отличается от типа Вершина-Грань. На переходах не могут лежать точки, являющиеся проекциями вершин, поэтому созданный переход не должен пересекать уже существующий.

Поэтому, если ближайшим элементом является переход ρ_k, необходимо создать новый переход ρ_i с любым концом отрезка ρ_k. Алгоритм выбирает ближайшую из таких точек, которые лежать в области I_i (рисунок 5, слева и в центре). Только вершины, лежащие внутри области I_i, гарантируют, что v_i поделится на два выпуклых угла. В случае, если ни один из концов отрезка ρ_k не удовлетворяет этому требованию (рисунок 5, справа), алгоритм создаёт два перехода вместо одного. Новыми переходами будут ρ_i, который соединяет v_i с левым концом ρ_k, и ρ_{i+1}, который соединяет v_i с правым концом ρ_k. В этом случае внутренний угол между ρ_i и

ρ_{i+1} всегда будет меньше π, и это гарантирует, что при создании новых двух переходов ν_i разделится на три выпуклых области.

Рисунок 5. Переход вершина-переход.



На рисунке 5 только один конец ρ_k находится в области I_i (слева), оба конца находятся в области I_i (в центре), не один из концов ρ_k не находится в области I_i (справа).

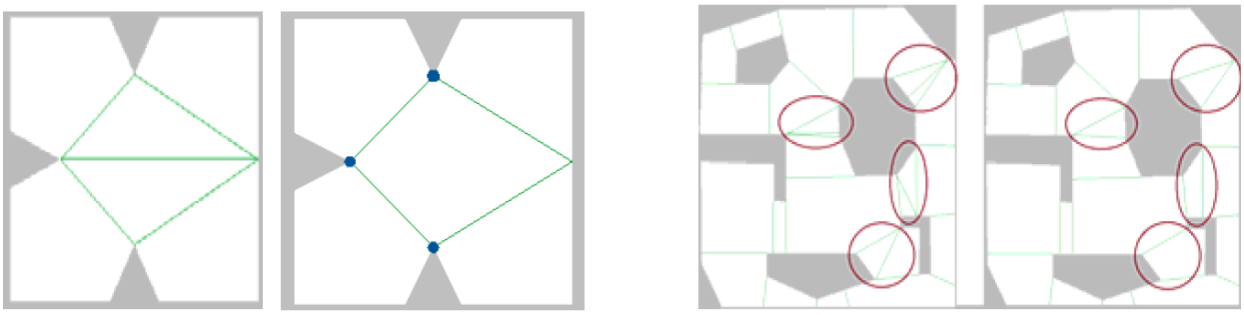
Этот случай единственный, при котором создаётся два новых перехода на одну вершину. Однако часто, создавая два перехода, есть возможность удалить оригинальный переход ρ_k, при этом на каждую вершину будет приходиться по одному переходу.

1.6. Удаление ранее созданных переходов.

Когда переход ρ_i типа вершина-переход находится между вершиной ν_j и ранее созданным переходом ρ_k, существует как минимум одна вершина ν_i, где оба перехода соединяются. Чтобы определить, необходимо ли соединить две ячейки, разделённые ρ_k, алгоритм проверяет, является ли ρ_k всё ещё необходимым переходом. При добавлении ρ_i к вершине ν_j, она уже может быть разделена на две выпуклых области, и делить её ещё раз не имеет смысла. Чтобы удалить переход ρ_k необходимо проверить нуждаются ли правая и левая вершины в ρ_k, чтобы быть выпуклыми. Для этого вычисляется внутренний угол между двумя соседними отрезками перехода ρ_k при каждой вершине на его концах (второй отрезок может быть гранью начального многоугольника или другим переходом) и проверяется на выпуклость. Если

каждый угол оказался выпуклым, то мы можем удалить ρ_k, тем самым объединив две выпуклые ячейки в одну большую (рисунок 6).

Рисунок 6. Удаление перехода.



На рисунке 6 изображено удаление ранее созданного перехода ρ_k, при создании новых переходов ρ_i и ρ_{i+1} (слева). Этот же случай изображён с несколькими переходами (справа).

1.7. Слабое определение выпуклости.

Вершина является выпуклой, если её внутренний угол меньше или равен π, в противном случае, она вогнутая. Таково математическое определение выпуклости, однако, в некоторых приложениях, таких как создание навигационных сеток, иметь такое строгое определение не обязательно. Как было отмечено ранее, навигационные сетки состоят из набора выпуклых многоугольников-ячеек с переходами, которыми являются пересекающие их отрезки. Движение внутри выпуклой клетки и между клетками описано в соответствии с локальным алгоритмом, который легко справляется с вогнутыми вершинами и препятствиями, которые встречаются в пространстве. В зависимости от реализованного алгоритма, можно ослабить определение выпуклости, введя постоянный предел τ. Это приводит к меньшему количеству переходов, так как больше многоугольников-ячеек можно соединить вместе в τ -выпуклые ячейки, где τ – предел, который задаёт локальный алгоритм движения. Получаются следующие определения:

Определение: Вершина v_i обладает τ -выпуклостью, если её внутренний угол меньше, чем $\pi + \tau$.

Слабое определение влияет не только на классификацию вершин, но и на определение рассматриваемой области I_i у вогнутой вершины.

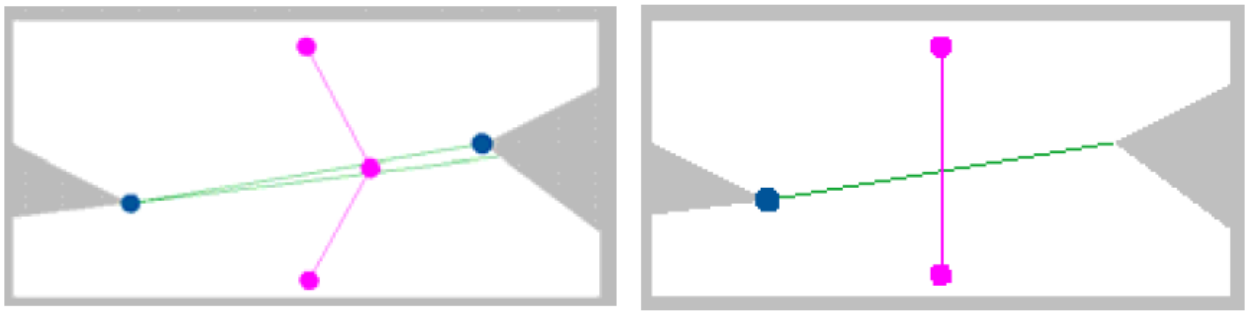
Определение: Рассматриваемая область I_i обладает τ-выпуклостью, если её внутренний угол равен $a_i + \tau$, где a_i угол I_i до введения ослабления определения выпуклости.

Для того чтобы избежать вырождений или непростых многоугольников, необходимо улучшить определение предела для вершины. Таким образом, мы гарантируем, что подходящий элемент для любой вершины v_i никогда не будет лежать за гранью, на которой находится v_i или пересекать границу начального многоугольника. Достигается это, ограничением $a_i + \tau < \pi$.

Определение: Многоугольник можно разделить на τ -выпуклые многоугольники-ячейки, только если все вершины обладают τ-выпуклостью.

Благодаря этому увеличивается внутренний угол области I_i, что позволяет отбирать большее количество элементов, входящее в эту область. Это не только уменьшит количество ячеек, но и так же означает уменьшение количества вырожденных многоугольников (рисунок 7).

Рисунок 7. Отличие между слабым определением выпуклости и обычным.



На рисунке 7 слева ячейка создана в соответствии со строгим определением выпуклости, справа – в соответствии с ослабленным определением выпуклости.

1.8. Результат алгоритма.

Данный алгоритм генерирует навигационную сетку, которая может эффективно использоваться для поиска пути для перемещения между ячейками сетки. В полученной сетке во всех случаях количество ячеек меньше, чем количество вогнутых вершин в начальном многоугольнике. Трудоёмкость этого алгоритма $O(r-n)$, где r – это количество вогнутых вершин, а n – количество выпуклых вершин.

Глава 2. Поиск пути в навигационных сетках.

Поиск пути это процесс определения набора перемещений объекта из одной точки в другую, без столкновения с любыми препятствиями. Выбор оптимального пути для каждого объекта является одной из самой важных задач искусственного интеллекта в коммерческой игре.

Оптимальный путь должен обладать двумя свойствами. Первое свойство называется периодом действия и является наиболее общей мерой, показывающей есть ли на пути препятствия. Второе свойство называется оптимальностью и определяет показатель расстояния или времени, требуемое для прохождения пути. При использовании показателя расстояния, оптимальный путь является кратчайшем путём. Это означает, что дистанция от начала движения до финиша не длиннее любого другого маршрута.

Время – это другая часто используемая мера. Она определяет оптимальный путь, как быстрейший путь. Это означает, что время, затраченное на прохождение оптимального пути, всегда меньше, чем при прохождении любого другого маршрута. Во многих случаях, кратчайший путь является самым быстрым, но бывают и особые случаи.

Нахождение оптимального пути состоит как минимум из трёх стадий. На перовой стадии пространство игрового мира трансформируется в геометрическое представление, которое является навигационной сеткой. На второй стадии в сгенерированной сетке выполняется поиск пути по заданному алгоритму. Однако, найденный путь может оказаться недостаточном оптимальным. Для решений этой проблемы на третьей стадии использует локальный алгоритм оптимизации.

2.1 Алгоритм поиска в ширину(*breadth-first search*, BFS).

Одним из самых первых, и, пожалуй, простейших алгоритмом поиска кратчайшего пути на графах является алгоритм поиска в ширину, разработанный независимо Муром и Ли в 1959 и 1961 годах. Очень часто

данный алгоритм сравнивают с неконтролируемым пожаром, когда при поджоге одного объекта, загораются все соседние, которые ещё не зажжены. То же самое происходит и со всеми соседними вершинами, т.е. получается, что огонь распространяется «в ширину».

Перед началом выполнения алгоритма необходимо произвести инициализацию некоторых объектов, таких как: структуру для хранения посещённых вершин и очередь, которую будем заполнять вершинами, необходимыми для посещения. Также каждой вершине будет присваиваться уровень глубины(аналог расстояния).

Алгоритм:

1. Выбираем начальную вершину, задаём ей уровень нулевой уровень глубины.
2. Всех её соседей добавляем в очередь для посещения, а саму помечаем как посещённую и больше к ней не обращаемся
3. Берём элементы из очереди для посещения, считая их уровень глубины, как: уровень глубины родителя + 1. Добавляем эти соседние вершины в очередь для посещения
4. А текущую вершину, помечаем как посещённую и удаляем из очереди для посещения
5. Повторяем шаги 3 и 4, пока не обойдём весь граф

Пример:

Рассмотри граф следующего вида (Рис 1)

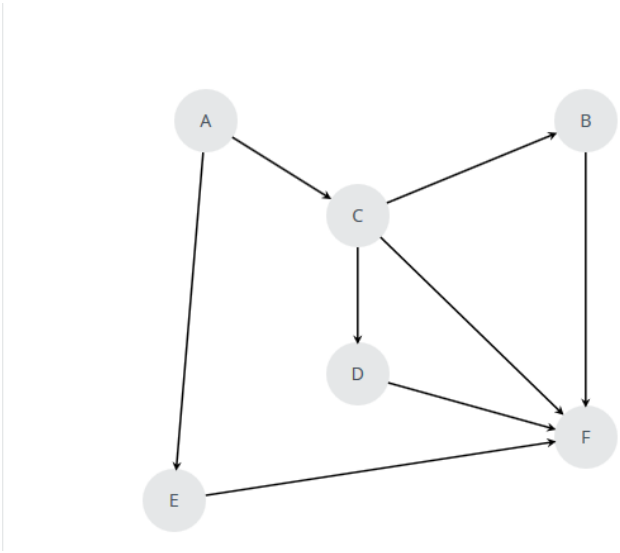


Рис 1

На первом шаге выберем начальную вершину A и зададим ей начальный уровень глубины 0 (Рис 2). А всех соседей: B, C, E добавляем в очередь для посещения, а саму вершину A помечаем, как посещённую

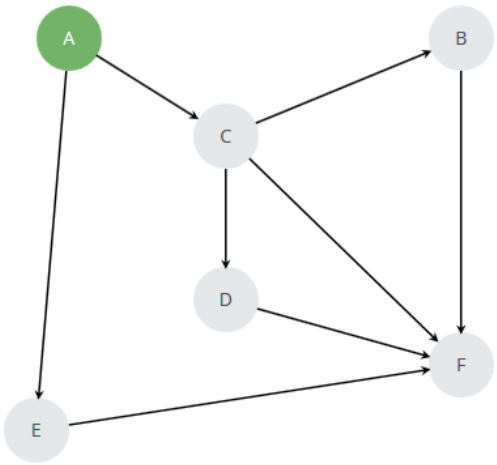


Рис 2

На следующих шагах идём по очереди обходим вершины : B, C, E увеличивая их уровень глубины на 1 (уровень глубины родителя A +1)

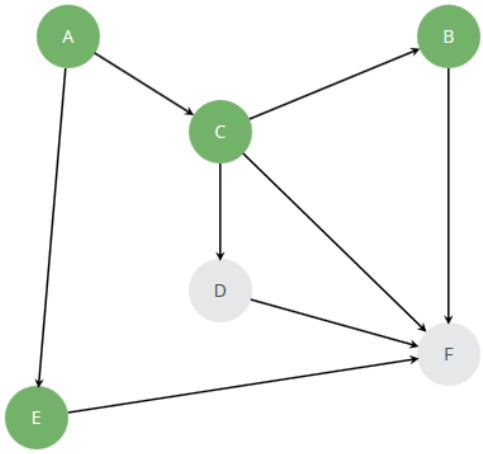


Рис 3

Затем, следуя алгоритму, посещаем все вершины графа (Рис 4)

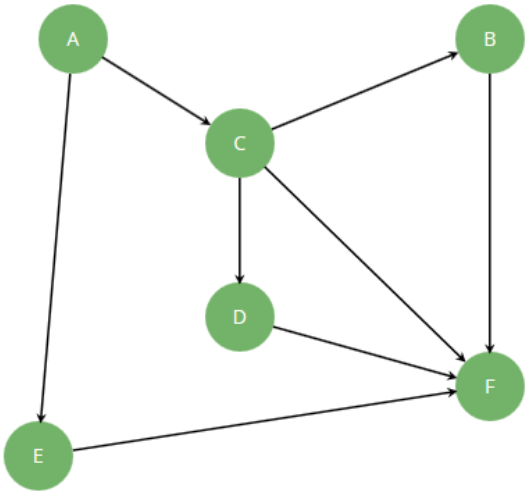


Рис 4

Ниже приведена реализация алгоритма на Python:

```
G = [     # матрица смежности для графа
    [1,3], # 0
    [0,3,4,5], # 1
    [4,5], # 2
    [0,1,5], # 3
    [1,2], # 4
    [1,2,3] # 5
]

level = [-1] * len(adj) # список уровней вершин
```

```
def bfs(s):
   level[s] = 0 # уровеньглубины начальной вершины
   stack = [s] # добавляем начальную вершину в очередь
   while stack: # пока очередь не пуста
      v = stack.pop(0) # извлекаем начальную вершину
      for w in adj[v]: # запускаем обход из вершины v
         if level[w] is -1: # проверка, посещена ли вершина
            stack.append(w) # добавление соседей в очередь
            level[w] = level[v] + 1 # подсчитываем уровень глубины вершины w

bfs(0) # поиск из вершины 0
print(level[2]) # уровень глубины до  вершины 2
```

Сложность алгоритма считается равной O(N*M), где N- количество вершин, а M – количество ребер. Исходя из данной оценки, можно легко заметить, что время выполнения поиска во многом зависит от количества ребёр в графе, т.е. чаще всего от того, как глубоко проводится поиск по графу.

2.2. Алгоритм Дейкстры.

Создан нидерландским учёным Э. Дейкстрой в 1959 году. Данный алгоритм работает на графах, находя кратчайшие пути от одной из вершин графа до всех остальных.

На начальной стадии всем вершинам, кроме начальной, присваивается вес равный бесконечности, а начальной вершине присваивается нулевой вес. Обходя граф, посещаются ближайшие к начальной вершины и им присваивается вес, состоящий из суммы веса предыдущей вершины плюс вес до этой вершины. После посещения всех ближайших вершин, текущую помечают как посещённую и больше к ней не обращаются, а текущей становится вершина с минимальным весом. Алгоритм останавливается,

дойдя до конечной вершины, а весом кратчайшего пути становится её вес. Сам алгоритм можно пошагово представить следующим образом:

Алгоритм

1. Задаём набор расстояний, в котором будут храниться кратчайшие расстояния от исходной вершины до всех остальных вершин графа, а также набор посещённых вершин.
2. Начальной вершине задаём значение равное нулю, а всем остальным бесконечности.
3. Пока в набор посещенных вершин не добавлены все вершины делаем следующее:
   a) Выберем вершину, не включённую в наш набор посещённых вершин, значение которой минимально
   b) Добавим её в наш набор посещённых вершин
   c) Обновим значения всех смежных вершин, как: минимальное значение суммы
   d)
   e) текущей вершины и веса ребра

Пример:

Пусть задан следующий граф (Рис 1)

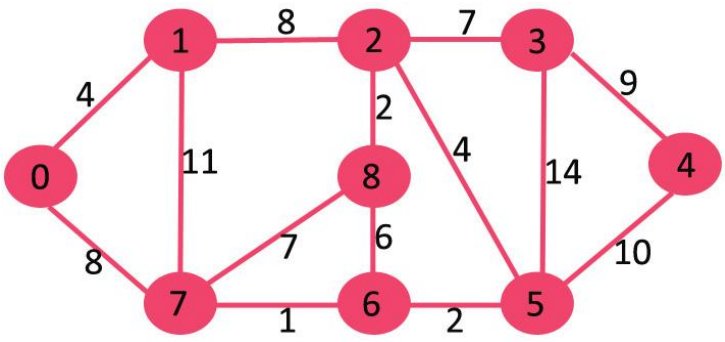


Рис 1

На первом и втором шаге проводим инициализацию, в результате которой будем иметь набор расстояний, вида: {0, INF, INF, INF, INF, INF, INF, INF, INF, INF}, набор посещённых вершин: {}.

Далее выполняем итерацию

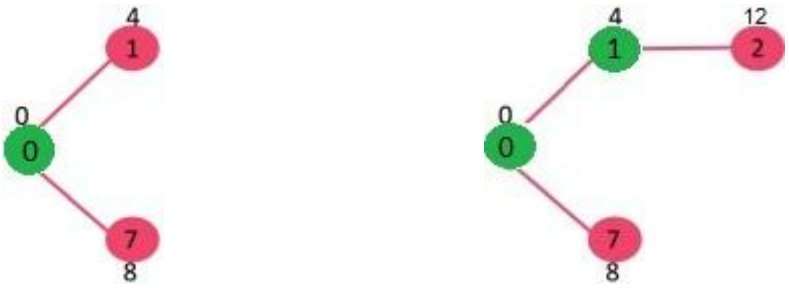


Рис 2

Посчитали значения смежных с начальной (0) вершин (1,7) и выбрали для следующей итерации вершину с минимальным значением (1), добавив её в наш набор расстояний, который теперь выглядит следующим образом: {0,4, INF, INF, INF, INF, INF, INF, INF, INF}, а набор посещённых вершин(отмечены зелёным цветом): {0,1}. Для этой вершины также посчитали минимальные значения для смежных вершин.

Продолжая выполнять итерации получим граф следующего вида (Рис 3)

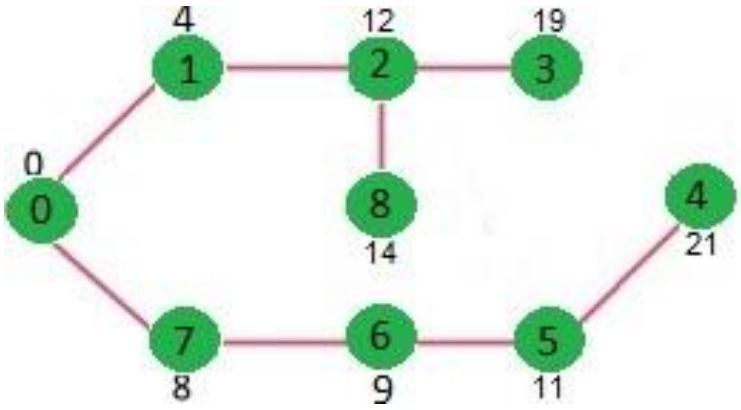


Рис 3

Для которого набор расстояний выглядит следующим образом:{0,4,8,9,11,12,14,19,21}

Набор посещённых вершин: {0,1,7,6,5,2,8,3,4}

Ниже представлен пример реализации на C++:

```
// C++ Алгоритм поиска кратчайшего пути
// Программа представлена для работы с графом, представленном в матричном виде

#include <stdio.h>
#include <limits.h>

// Кол-во узлов в графе
#define V 9

// Функция для поиска узлов с минимальным значением расстояния из множества узлов, ещё не включённых в набор посещённых узлов
int minDistance(int dist[], bool sptSet[])
{
  // Initialize min value
  int min = INT_MAX, min_index;

  for (int v = 0; v < V; v++)
    if (sptSet[v] == false && dist[v] <= min)
        min = dist[v], min_index = v;

  return min_index;
}

// Функция дял печати построенного массива расстояний
int printSolution(int dist[], int n)
{
  printf("Vertex   Distance from Source\n");
  for (int i = 0; i < V; i++)
     printf("%d \t\t %d\n", i, dist[i]);
}

// Функция для поиска кратчайшего расстояний алгоритмом Дийкстры от узла src до i для графа, представленного в виде матрицы смежности
void dijkstra(int graph[V][V], int src)
{
```

```
    int dist[V];     // Итоговый массив, в котором будут кратчайшие расстояние от узла src до i


    bool sptSet[V]; // sptSet[i] true, если вершина включена в набор посещённых или он уже сформирован окончательно
    // Инициализируем все расстояния узлов как бесконечность and stpSet[] как false
    for (int i = 0; i < V; i++)
       dist[i] = INT_MAX, sptSet[i] = false;

    // Расстояние исходной вершины задаем равное 0
    dist[src] = 0;

    // Находим кратчайшие расстояние для всех узлов
    for (int count = 0; count < V-1; count++)
    {
      // Выбираем узел с минимальным расстоянием, который ещё не был добавлен в набор посещённых
      int u = minDistance(dist, sptSet);

      // Помечаем как посещённый
      sptSet[u] = true;

      // Обновляем расстояние дял смежных узлов
      for (int v = 0; v < V; v++)

        //Обновляем только если смежный ещё не посещён и полученное расстояние меньше текущего
        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                                    && dist[u]+graph[u][v] < dist[v])
          dist[v] = dist[u] + graph[u][v];
    }

    // Печатаем массив расстояний
    printSolution(dist, V);
}

// Тест
int main()
{
   /* Граф, в виде матрциы смежности из рассмотренного примера */
   int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                      {4, 0, 8, 0, 0, 0, 0, 11, 0},
                      {0, 8, 0, 7, 0, 4, 0, 0, 2},
```

```
                {0, 0, 7, 0, 9, 14, 0, 0, 0},
                {0, 0, 0, 9, 0, 10, 0, 0, 0},
                {0, 0, 4, 14, 10, 0, 2, 0, 0},
                {0, 0, 0, 0, 0, 2, 0, 1, 6},
                {8, 11, 0, 0, 0, 0, 1, 0, 7},
                {0, 0, 2, 0, 0, 0, 6, 7, 0}
               };

   dijkstra(graph, 0);

   return 0;
}
```

Сложность данного алгоритма во многом зависит от способа хранения графа и его обработки. В простейшем же случае, когда граф хранится в виде матрицы смежности, то время выполнения имеет порядок O(n^2), где n – количество узлов графа.

2.3 Алгоритм A*.

Следующим шагом в развитии алгоритмов поиска кратчайшего пути на графах, является алгоритм A*. Его создателями являются Питером Хартом, Нильсом Нильсоном и Бертрамом Рафаэлем в 1968 году, впервые описавшие его. Данный алгоритм является усовершенствованием алгоритм Дейкстры за счёт введения эвристики. А именно: переход осуществляется в ту вершину, предположительный путь от которой до конечной является кратчайшим. Чаще всего пользуются эвристикой Манхеттена, но помимо неё возможно использование эвристик Чебышева и Евклида.

A* на сегодня является самым распространённым алгоритмом для поиска пути в игровой индустрии, ввиду своей универсальности, относительно

низким потреблением памяти и весьма быстро справляющимся со своей задачей, в большинстве случаев. Чащей всего, быстрая скорость выполнения достигается как раз за счёт введения эвристики.

Алгоритм во многом похож на Алгоритм Дейкстры, у нас также имеется 2 списка: открытый список для хранения узлов, которые нужно обойти и проверить и закрытый список, в котором храним уже проверенные узлы, к которым больше не будем обращаться. Для того, чтобы восстановить кратчайший путь необходимо хранить родителя для каждого узла, а также необходимо для каждого узла хранить значение целевой функции F, равной сумме функции G и H, где G – стоимость пути до данного узла, а H – значение функции эвристики для данного узла.

Функция эвристики – это примерная стоимость передвижения от данного узла до конечного. Её так называют, потому что это предположение Мы действительно не узнаем длину пути, пока не найдём его, так как в процессе поиска мы можем столкнуться препятствиями, которые могут существенно повлиять на длину пути. Существуют 3 популярные эвристики: Манхеттена, Чебышева и Евклида, их расчет происходит по следующим формулам:

- Если мы можем перемещаться в четырех направлениях, то в качестве эвристики стоит выбрать манхэттенское расстояние $h(v) = |v.x - goal.x| + |v.y - goal.y|$.
- Расстояние Чебышева применяется, когда к четырем направлениям добавляются диагонали: $h(v) = \max(|v.x - goal.x|, |v.y - goal.y|)$.
- Если передвижение не ограничено сеткой, то можно использовать евклидово расстояние по прямой: $h(v) = \sqrt{(v.x - goal.x)^2 + (v.y - goal.y)^2}$.

В случае навигационных сеток, обычно пользуются манхэттенское расстояние, так как чаще всего оно позволяет не только недооценить расстояние до конечного узла, но и не переоценить его.

Алгоритм работы A* можно представить следующим образом:

1) Добавляем начальный узел в открытый список для хранения узлов

2) Повторяем следующее:

a) Останавливаемся, если:

- Добавили конечный узел в открытый список, в этом случае путь найден.
- Или открытый список пуст и мы не дошли до конечного узла. Здесь получаем, что не существует пути до конечного узла

b) Ищем в открытом списке узел с наименьшим значением целевой функции F. Делаем его текущим узлом

c) Помещаем этот узел в закрытый список. (И удаляем с открытого)

d) Для каждого из соседних узлов делаем следующее

- Если узел находится в закрытом списке, игнорируем ее. В противном случае делаем следующее.
- Если узел еще не в открытом списке, то добавляем его туда. Делаем текущий узел родительским для это клетки. Рассчитываем стоимости F, G и H узла.
- Если узел уже в открытом списке, то проверяем, не дешевле ли будет путь через этот узел. Для сравнения используем стоимость G. Более низкая стоимость G указывает на то, что путь будет дешевле. Если это так, то меняем родителя узла на текущий узел и пересчитываем для него стоимости G и F

3) Сохраняем путь. Двигаясь назад от целевого узла, проходя от каждого узла к его родителю до тех пор, пока не дойдем до начального узла. Это и будет наш путь.

Пример, рассмотри, навигационную сетку следующего вида(Рис. 1). Где, зелёная клетка – стартовая, синие клетки - преграды, а красная клетка – конечная. Необходимо найти кратчайший путь из зелёной в красную. Синие клетки сразу добавляются в закрытый список, ввиду того, что они являются препятствиями и к ним нельзя обращаться.

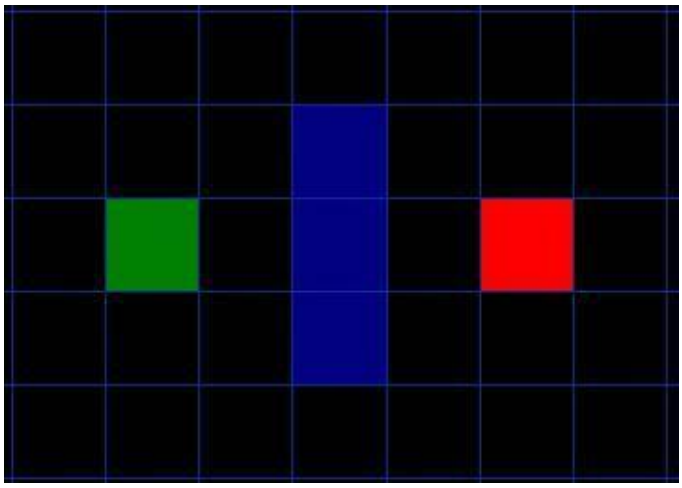


Рис. 1

Далее добавляем начальную клетку в открытый список, проходим всех её соседей (8 клеток выделенных зелёным контуром), добавляем их в открытый список, рассчитываем для них значения целевой функции, а также функций G и H, и для всех них указываем начальную клетку, как родительскую. Добавляем начальную клетку в закрытый список и удаляем из открытого (выделен голубым контуром). (Рис. 2)

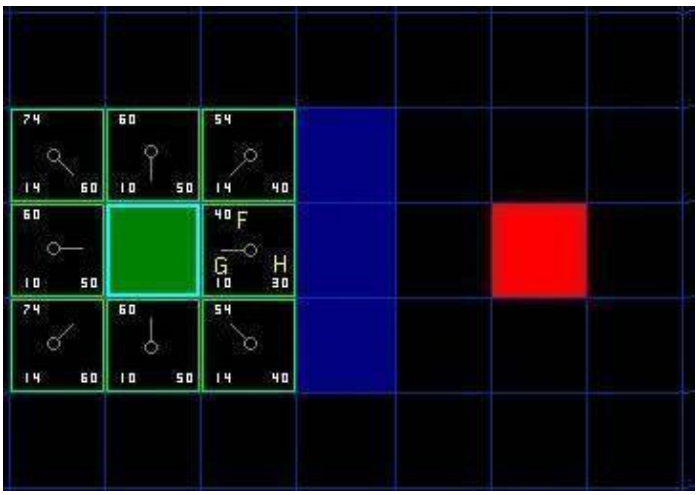


Рис. 2

Затем из открытого списка, выбирается клетка с минимальным значением целевой функции (справа от зелёной). Родительская для неё задаётся зелёная, её соседи уже добавлены в открытый список и мы, разве что, можем рассчитать новые значения целевой функции для соседей, но они получаются больше, чем имеющиеся, поэтому мы их не обновляем, клетку добавляем в закрытый список и перемещаемся по списку к клетке, расположенной по диагонали, значение целевой функции которой на данный момент минимально.(Рис 3)

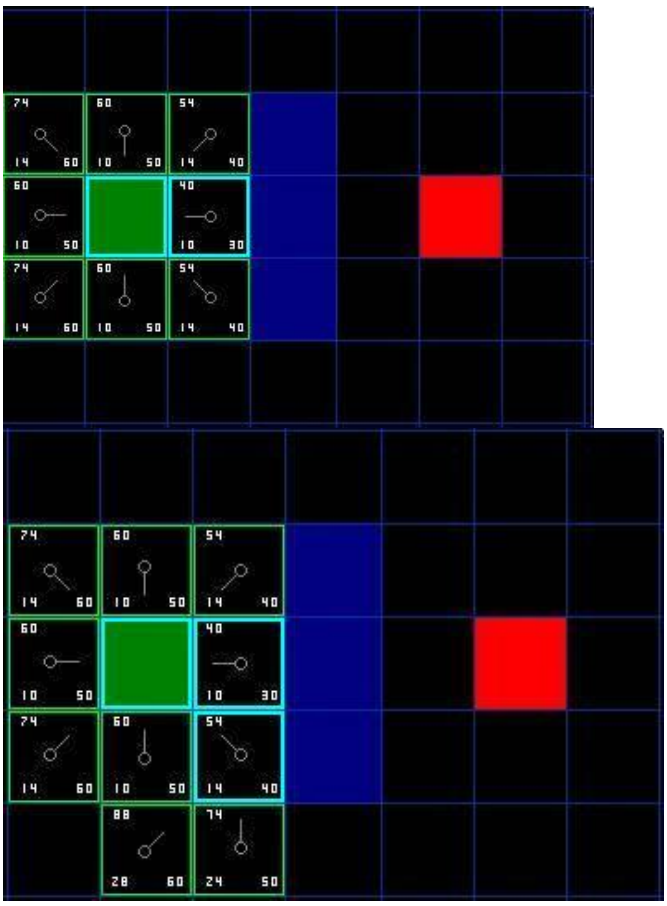


Рис 3

Продолжая выполнять работу по заданному алгоритмы, мы в итоге добавляем конечную клетку в открытый список, что является признаком остановки, и по родителям имеющихся клеток, восстанавливаем полученный, кратчайший маршрут(отмечен красными точками Рис 4).

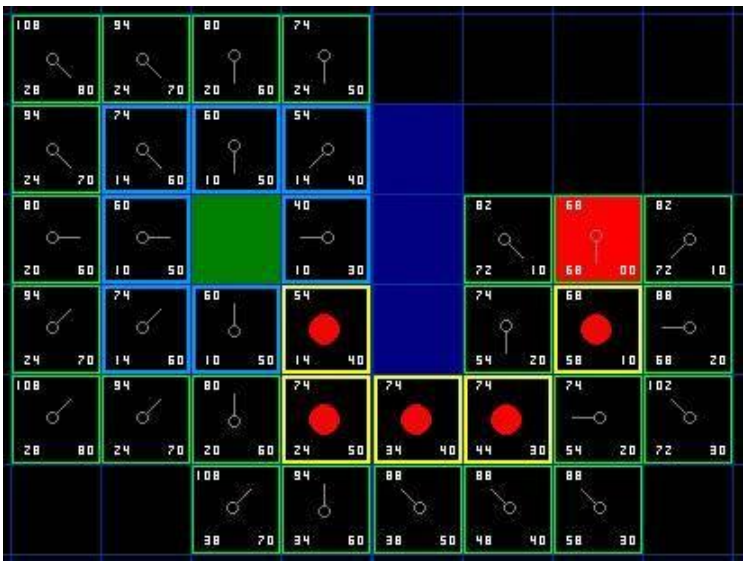
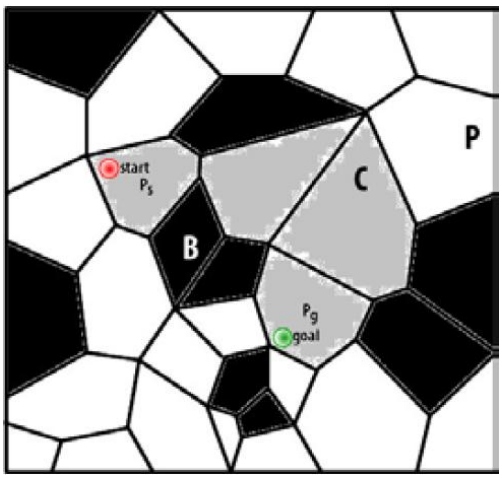


Рис 4

2.4 A* в навигационных сетках.

Ниже показан пример работы алгоритма A* в навигационных сетках. Пусть G – это граф, с P многоугольниками, доступными для передвижения, и B многоугольниками, которые представляют собой препятствия (рисунок 8).

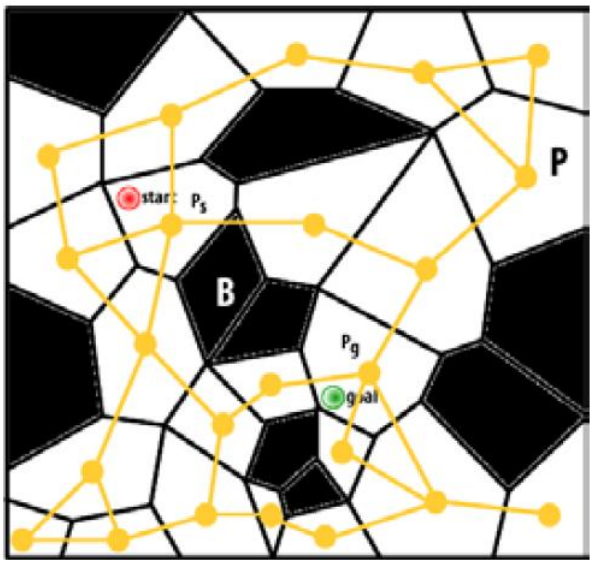
Рисунок 8. Набор многоугольников C.



Сначала находится набор многоугольников $C \subseteq P$, через который проходит оптимальный путь. Если P_s - многоугольник, в котором находится начало пути, то P_s должен быть первым многоугольником в C. Если P_g – многоугольник, в котором находится конец пути, тогда P_g должен быть последним многоугольником в C. Чтобы найти недостающую часть C, необходимо внести некоторые изменения в карту. Каждый многоугольник в

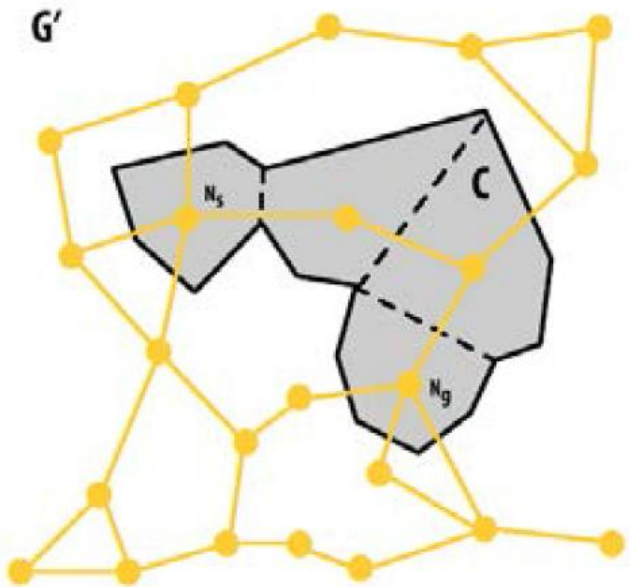
G сопоставляется с узлом G’. Например, P_s сопоставляется с N_s, а P_g сопоставляется с N_g. Каждая граница между двумя многоугольниками в G сопоставляется грани, соединяющей два узла в G’ (рисунок 9).

Рисунок 9. Соответствие G с G’.



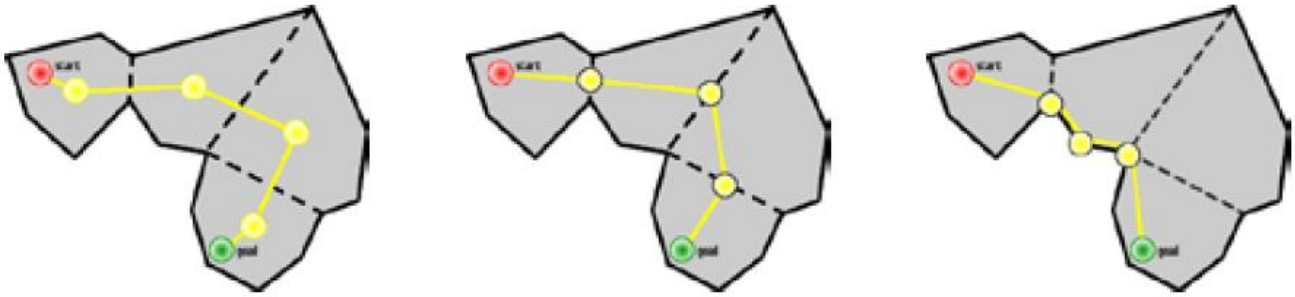
Тогда оптимальный путь p из N_s в N_g из G’ может быть найден с помощью A*. Каждый узел в G’ соответствует многоугольнику в G, и каждый узел в p соответствует многоугольнику в C (рисунок 10).

Рисунок 10. Граф навигации G’.



C не является настоящим оптимальным путём, это всего лишь набор многоугольников. Рисунок 11 показывает три разных способа поиска настоящего пути в C. Не важно, какой из этих путей будет использоваться, ни один не сможет гарантировать оптимальный путь.

Рисунок 21. Варианты поиска оптимального пути.



2.5 Сравнение и выявление лучшего из рассмотренных алгоритмов поиска.

Для проведения тестов и выявления лучшего из 3-х представленных алгоритмов была проведена работа с сервисом https://qiao.github.io/PathFinding.js/visual/, в котором реализованы в единой оболочке большинство алгоритмов поиска путей. В качестве критериев оценивания были выбраны следующие критерии: скорость нахождения кратчайшего пути, количество итераций, выполненных в процессе поиска и длина найденного пути. Т.к. алгоритмы реализованы на одной и той же платформе (java script), то в расчёт не берётся скорость компилятора. Нам дана сетка из квадратов, на которой мы выбираем начальную точку (зелёный квадрат), конечную току(красный квадрат) и задаём препятствия (серый

квадрат). На выходе же мы получаем наш путь(жёлтая линия). Во всех алгоритмах дополнительным параметром указывает отсутствие срезов углов, т.к. в условиях реального проекта может случится выход за сетку или провал в текстуры или невозможность прохождения.

Результаты сравнения работы рассмотренных алгоритмов

Тест	Алгоритм	Длина пути	Время(ms)	Кол-во операций
1	A*	12.49	1.52	119
	Дейкстра	12.49	5.8	946
	BFS	12.49	3.16	847
2	A*	20.83	1.64	281
	Дейкстра	20.83	8.7	2111
	BFS	20.83	5.6	2445
3	A*	37.66	1.23	488
	Дейкстра	37.66	5.6	2594
	BFS	37.66	6.7	2701

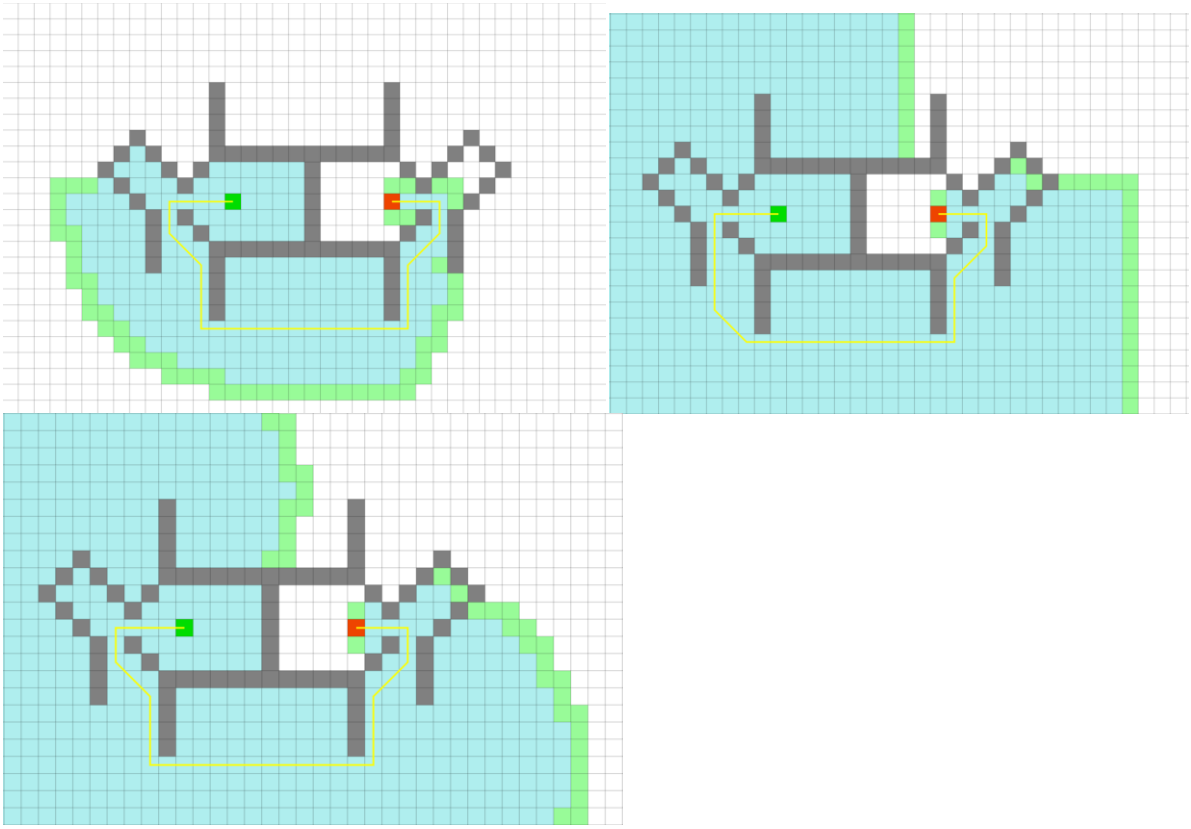


Рис 1. (Тест номер 3)

Из таблицы проведённых тестов видно, что алгоритм A* находит кратчайший путь как за меньше время, так и совершает наименьшее количество итераций по сравнению с другими рассмотренным алгоритмами поиска кратчайших путей. Также в виду специфики поставленной задачи (реализация игры в стиле TowerDefence) необходим алгоритм, который будет обходить препятствия и быстро искать путь из одной точки в другу, с чем A* справляется в большинстве случаев лучше других алгоритмов. Таким образом лучшим алгоритмом поиска кратчайшего пути в поставленной задачи был выбран A*.

Ниже приведена реализация алгоритма в созданном проекте:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System;
using System.Linq;

/// <summary>
```

```
/// This class contains all Astar functionality
/// The class can return a walkable path
/// </summary>
public static class AStar
{
    /// <summary>
    /// This dictionary contains all the nodes in our grid
    /// We are using a dictionary because it is faster than a list
    /// </summary>
    private static Dictionary<Point, Node> nodes;

    /// <summary>
    /// This function creates all the nodes in our grid
    /// </summary>
    private static void CreateNodes()
    {
        //Instantiates our dictionary
        nodes = new Dictionary<Point, Node>();

        //Runs through all the tiles in our game, if we have a tile, then we need to crea
te a node
        //Tiles are the visual squares you see in the game. A node is something invisib
le we use for pathfinding
        foreach (TileScript tile in LevelManager.Instance.Tiles.Values)
        {
            //Creates a node based on the tile we just found
            nodes.Add(tile.GridPosition, new Node(tile));
        }
    }

    /// <summary>
    /// Returns a walkable path
    /// </summary>
    /// <param name="start">The start position of the path</param>
    /// <param name="goal">The end position of the path</param>
    /// <returns>A stack of nodes with the path, returns null if no path available</re
turns>
    public static Stack<Node> GetPath(Point start, Point goal)
    {
        if (nodes == null) //If we don't have any nodes, then we need to create them
        {
            CreateNodes();
        }

        //We need to reset our nodes, so that we can find a new path
```

```
        //If we don't reset our nodes, old values might be reused and we won't get the shortest path
        foreach (Node node in nodes.Values)
        {
            node.Reset();
        }

        //Sets the current node as the start node, this is part of the Astar algorithm
        Node currentNode = nodes[start];

        //Creates an open list for nodes that we might want to look at later
        HashSet<Node> openList = new HashSet<Node>();

        //Creates a closed list for nodes that we have examined
        HashSet<Node> closedList = new HashSet<Node>();

        //Adds the current node to the open list (we have examined it)
        openList.Add(currentNode);

        while (openList.Count > 0) //As long as the openlist has nodes in it then we need to keep searching for a path
        {
            for (int x = -1; x <= 1; x++) //These two forloops makes sure that we all nodes around our current node
            {
                for (int y = -1; y <= 1; y++)
                {
                    //Stores the position of the current neighbour we are looging at
                    Point neighbourPos = new Point(currentNode.GridPosition.X - x, currentNode.GridPosition.Y - y);

                    //If there is a neighbout at the position, and it isn't start or the current node, then we need to examine it
                    if (LevelManager.Instance.InBounds(neighbourPos) && neighbourPos != start && LevelManager.Instance.Tiles[neighbourPos].IsEmpty && neighbourPos != currentNode.GridPosition)
                    {
                        //Stores a reference to the node
                        Node neighbour = nodes[new Point(neighbourPos.X, neighbourPos.Y)];

                        //Sets the gCost to 0 to makes sure that we get a value later
                        int gCost = 0;
```

```
                    //If the node is horizontal or vertical positioned
                    if (Math.Abs(x - y) % 2 == 1)
                    {
                        gCost = 10; //The gscore for a vertical or horizontal node is 10
                    }
                    else //If the node is diagonally positioned
                    {
                        if (!ConnectedDiagonally(currentNode, neighbour))
                        {
                            continue;
                        }

                        gCost = 14; //The gscore for a diagonally node is 14
                    }

                    if (openList.Contains(neighbour))//If the open list contains the neig
hbour
                    {
                        if (currentNode.G + gCost < neighbour.G) //Then we need to chec
k if this node is a better parent
                        {
                            neighbour.CalcValues(currentNode, nodes[goal], gCost);
                        }
                    }
                    else if (!closedList.Contains(neighbour)) //If the openlist doesn't con
tain the neighbout and the close list doesn't contain the neighbour.
                    {
                        neighbour.CalcValues(currentNode, nodes[goal], gCost);//Then w
e need to calc the nodes values

                        if (!openList.Contains(neighbour)) //An extra check for openlist c
ontaining the neighbout
                        {
                            openList.Add(neighbour); //Then we need to add the node to th
e openlist
                        }
                    }
                }
            }
        }

        //The current node is removed fromt he open list
        openList.Remove(currentNode);

        //The current node is added to the closed list
```

```
            closedList.Add(currentNode);

            if (openList.Count > 0) //If the openlist has nodes on it, then we need to sor
t them by it's F value
            {
                currentNode = openList.OrderBy(x => x.F).First();//Orders the list by th
e f value, to make it easier to pick the node with the lowest F val
            }

            if (currentNode == nodes[goal]) //If our current node is the goal, then we fo
und a path
            {
                //Creates a stack to contain the final path
                Stack<Node> finalPath = new Stack<Node>();

                //Adds the nodes to the final path
                while (currentNode.GridPosition != start)
                {
                    //Adds the current node to the final path
                    finalPath.Push(currentNode);
                    //Find the parent of the node, this is actually retracing the whole path
back to start
                    //By doing so, we will end up with a complete path.
                    currentNode = currentNode.Parent;
                }

                //Returns the complete path
                return finalPath;
            }
        }

        //If we didn't manage to find a path, then we return null
        return null;

    }

    /// <summary>
    /// A helper method, that determines if two nodes are connected diagonally witho
ut anything blocking the way
    /// </summary>
    /// <param name="currentNode">The first node</param>
    /// <param name="neighbour">The second node</param>
    /// <returns>True if the nodes are in bounds</returns>
    private static bool ConnectedDiagonally(Node currentNode, Node neighbour)
    {
```

```
        //Get's the direction
        Point direction = currentNode.GridPosition - neighbour.GridPosition;

        //Gets the positions of the nodes
        Point first = new Point(currentNode.GridPosition.X + (direction.X * -
1), currentNode.GridPosition.Y);
        Point second = new Point(currentNode.GridPosition.X, currentNode.GridPosi
tion.Y + (direction.Y * -1));

        //Checks if both nodes are empty
        if (LevelManager.Instance.InBounds(first) && !LevelManager.Instance.Tiles[
first].IsEmpty)
        {
            return false;
        }
        if (LevelManager.Instance.InBounds(second) && !LevelManager.Instance.Til
es[second].IsEmpty)
        {
            return false;
        }

        //The ndoes are empty
        return true;
    }
}
```

Глава 3. Реализация Unity проекта.

В современной игровой индустрии большое внимание уделяется получению качественного игрового опыта, большая часть которого сильно зависит от работы AI, одной из главной часть которого является построение путей, по которым перемещаются внутриигровые объекты. Поэтому в рамках данной ВКР было решено создать игру на игровом движке Untiy по типу TowerDefence, но преобразив сам геймплей, сделав не статический, путь, как в классической игре, когда монстрам задаётся изначально траектория их движения, а игрок, по пути следования монстров уже сам расставляет защитные сооружения, а динамический, когда противники сами ищут кратчайший путь из стартовой точки в конечную, обходя башни, расставленные игроком. Таким образом возможности геймплея достаточно широко расширятся в виду приспособления противников к стилю игры игрока.

Сначала была созданная следующая сцена (Рис 1).

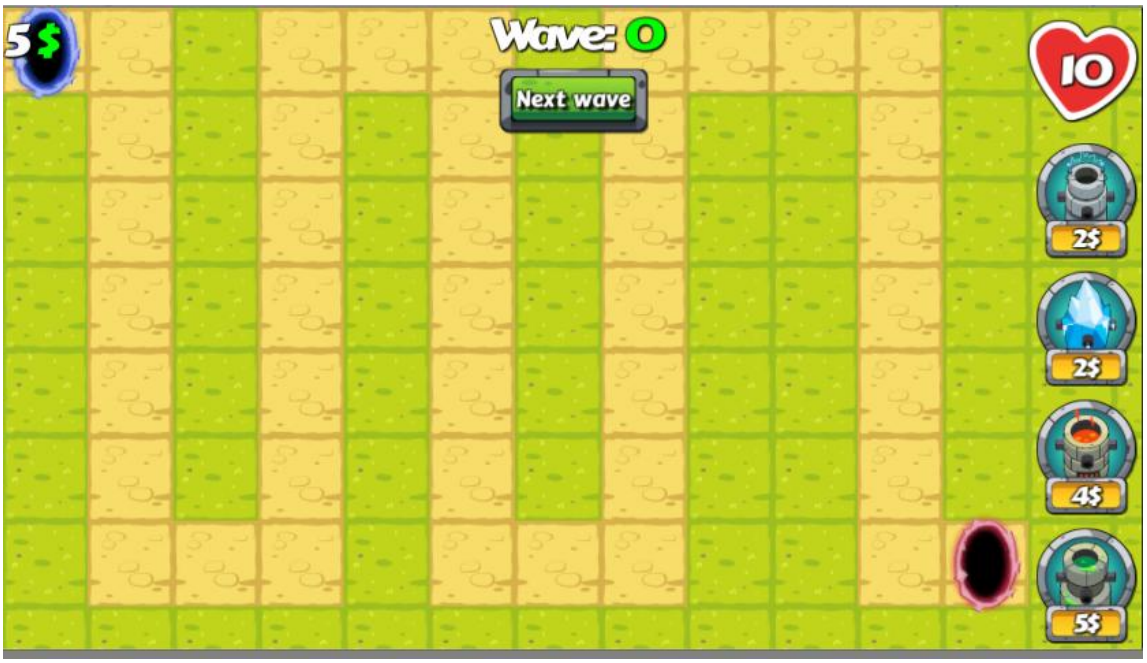


Рис 1

Необходимо реализовать следующее: монстры выходят из синего портала и идут в красный, генерирую себе путь алгоритм A* и обходя все защитные башни, расставленные игроком, причём каждый раз, как игрок расставляет свои башни, монстры ищут новый кратчайший путь, подстраиваясь, под действия игрока, усложняя геймплей.

Задачу поиска кратчайшего пути в данном проекте можно разделить на 2 этапа: генерация навигационной сетки, по которой монстры смогут передвигаться и непосредственно сам поиск кратчайшего пути на этой сетке.

Так как сцена составлена из квадратных клеток земли и каждая клетка имеет свой узел, то для удобства был создан класс Node, в котором были определены поля для хранения положения узла клетки на сцене, координаты узла на сцене, родителя узла и целевой функции, вместе с её составляющими.

Затем были создан словарь для инициализации узлов, в который соответственно были занесены все клетки нашей сцены.

Таким образом мы получили готовую навигационную сеть, по которой можем в дальнейшем передвигаться.

Далее необходимо реализовать непосредственно сам поиск по построенной навигационной сетке алгоритмом поиска кратчайшего пути A*.

На каждой волне, после расстановки игроком защитных башен выполняется следующее:

Создаются открытый список, содержащий в себе узлы изначально начальную точку и узлы, которые необходимо посетить. И закрытый список, в котором хранятся узлы, на которых расположены препятствия и уже посещённые узлы, которые игнорируют.

Затем в открытый список добавляются все соседние. Соседними узлами считаются те, расстояние между которыми равно 10 или 14, где 10 узлы, расположенные горизонтально или вертикально по отношению к текущему, а 14 – диагонально(т.к. по теореме Пифагора диагональ квадрата в нашем случае равна корню квадратному из 2). Мы используем целочисленные значения для расчета расстояний между узлами, т.к. с ними компьютер работает намного быстрее, нежели с вещественными.И для всех соседей рассчитываются значений целевых функций.

Далее выбирается узел с минимальным значением целевой функции, родителем которого обозначается текущий, а текущий добавляется в закрытый список. Этот узел становится текущим.

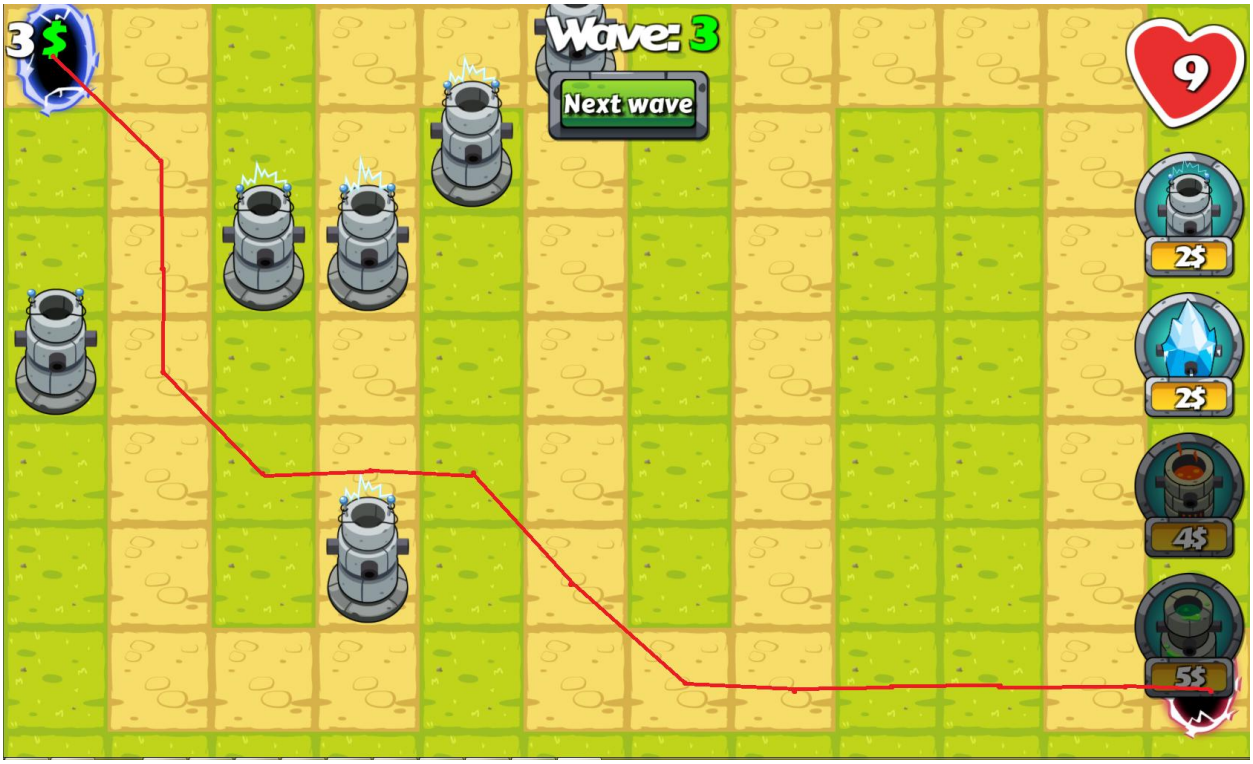
Затем для всех соседей из открытого списка обновляются значения целевых функций, если текущие меньше получившихся, а недобавленные в открытый список в него добавляются.

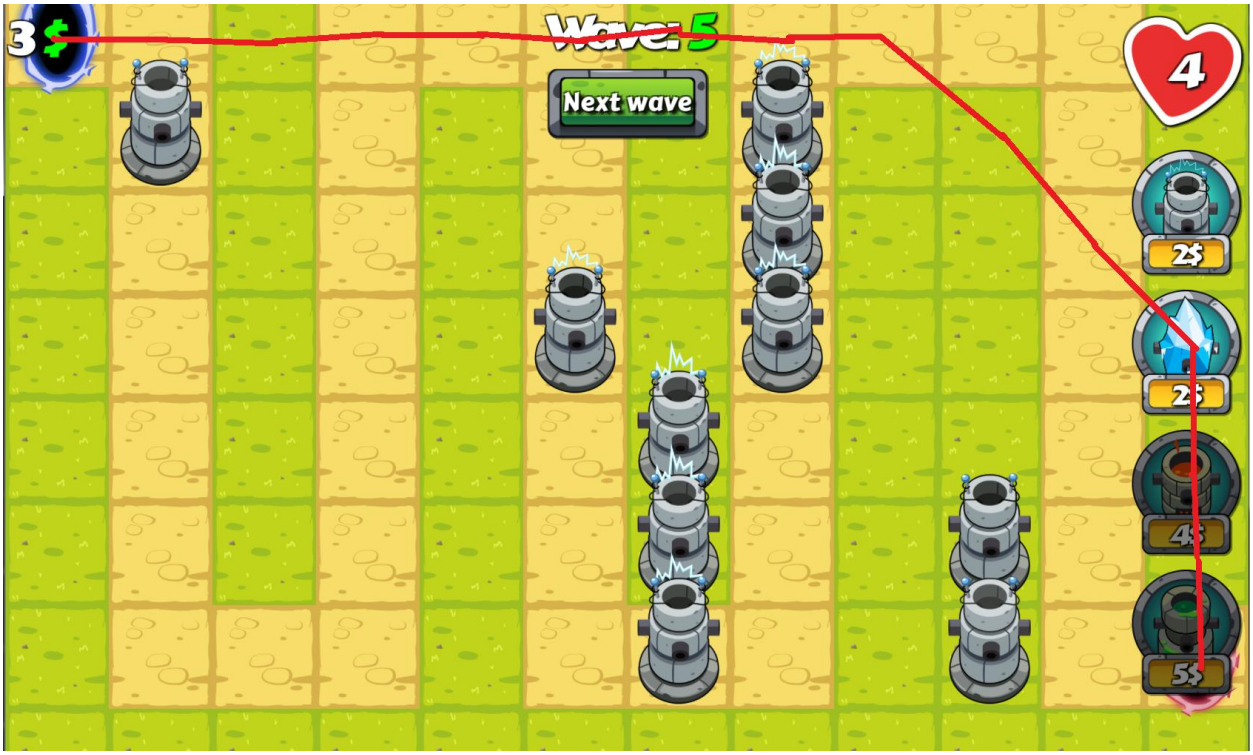
Повторяем эти действия, пока не дойдём до конечного узла с красным порталом.

Таким образом на выходе мы будем иметь готовый кратчайший путь для наших монстров в виде последовательности центров квадратов(узлов спрайтов), которые будут обходить башни, расставленные игроком и каждый раз, когда игрок заново расставит башни, подстраиваться под его действия, ища новый путь, усложняя геймплей, значительно его преобразив.

3.1. Примеры работы.

Ниже приведены примеры игрового процесса, в виде отображения траекторий, по которым следуют монстры:





(Рис 1,2,3) – Примеры работы A*

Выводы

Подводя итог проделанной работе можно сказать, что все поставленные задачи выполнены. Был показан общий алгоритм генерации NavMesh, как проводить триангуляцию и решение проблемы построения переходов при генерации NavMesh. Также рассмотрен способ оптимизации этого процесса, путём слабого определения выпуклости и удаления лишних, ранее созданных переходов. Кроме этого рассмотрены и реализованы алгоритмы поиска кратчайших путей на навигационных сетках, такие как: алгоритм Дейкстры, алгоритм A* и алгоритм поиска в ширину BFS. Так как они используются для поиска пути в различных ситуациях, была проведена серия тестов на конкретной задачи и среди них выявлен наиболее подходящий: A*, показавший лучшее время и меньшие затраты памяти. Как итоговый проект создана 2D игра в стиле TowerDefence поиск путей в которой реализовывался алгоритмом A* с целью существенного усложнения игрового процесса. Для этого был изучен игровой движок Unity, получены навыки работы с ним.

Заключение

Проделанная работа показала, что даже такое с первого взгляда простое изменение искусственного интеллекта в игре, как изменение способа построения путей для внутриигровых персонажей, может существенно изменить геймплей, преобразив получаемый игровой опыт. Когда компьютер может подстраиваться под игровые действия, игроку приходится придумывать каждый раз новую стратегию или изменять имеющуюся, что несомненно делает игровой процесс более интересным.

Список литературы

1. Xiao Cui, Hao Shi. An Overview of Pathfinding in Navigation Mesh// IJCSNS International Journal of Computer Science and Network Security, 48 VOL.12 No.12 - December 2012

2. Ramon Oliva, Nuria Pelechano. Automatic Generation of Suboptimal NavMeshes//Motion in Games, 4th International Conference, MIG 2011, Edinburgh, UK, November 13-15, 2011.

3. *Delaunay B.* Sur la sphère vide. A la mémoire de Georges Voronoï // Изв. АН СССР. VII серия. Отделение матем. и естеств. наук. — 1934. — № 6. — С. 793—800

4. *Dijkstra E. W.* A note on two problems in connexion with graphs// *Numer. Math* — Springer Science+Business Media, 1959. — Vol. 1, Iss. 1. — P. 269–271. — ISSN 0029-599X; 0945-3245— doi:10.1007/BF01386390

5. *Евстигнеев В. А.* Глава 3. Итеративные алгоритмы глобального анализа графов. Пути и покрытия // Применение теории графов в программировании/ Под ред. А. П. Ершова. — Москва: Наука. Главная редакция физико-математической литературы, 1985. — С. 138-150. — 352 с.

6. *Hart P. E., Nilsson, N. J., Raphael, B.* A Formal Basis for the Heuristic Determination of Minimum Cost Paths // IEEE Transactions on Systems Science and Cybernetics SSC4. — 1968. — № 2. — С. 100 — 107.

7. *Hart P. E., Nilsson, N. J., Raphael, B.* Correction to «A Formal Basis for the Heuristic Determination of Minimum Cost Paths» // SIGART Newsletter. — 1972. — Т. 37. — С. 28 — 29.

8. *T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein.* Introduction to Algorithms. — 3rd edition. — The MIT Press, 2009. — ISBN 978-0-262-03384-8.. Перевод 2-го издания: *Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л.*

9. Christoph Romstöck Generating 2D Navmeshe//GDOL (Gamedev.net Open License)

10. Unity Documentation // https://docs.unity3d.com/ru/current/Manual/Overview2D.html

Приложение

С полным проектом можно ознакомиться по ссылке https://www.dropbox.com/s/qnwustrcy87a6vb/TowerDefenseComplete.rar?dl=0