Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики и технологий программирования

Дисциплина: Объектно-ориентированное программирование

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

на тему

Компьютерная игра «Tanks in the Maze» в жанре Shoot ‘em up

Студент гр.853506 Головенко Д.И.

Руководитель: ассистент кафедры информатики Рогов М.Г.

Минск 2020

Оглавление

[Введение 6](#_Toc42111556)

[1. Постановка задачи 7](#_Toc42111557)

[1.1. Цель 7](#_Toc42111558)

[1.2. Основные требования к функционалу: 7](#_Toc42111559)

[2. Теоретические основы ООП 8](#_Toc42111560)

[3. Обоснование выбора языка и среды разработки 10](#_Toc42111561)

[3.1. Критерии выбора 10](#_Toc42111562)

[3.2. Преимущества Unity3D перед другими движками 10](#_Toc42111563)

[4. Разработка компьютерной игры 12](#_Toc42111564)

[4.1. Краткие сведения об игре 12](#_Toc42111565)

[4.2. Генерация лабиринта 13](#_Toc42111566)

[4.3. Разработка консоли 15](#_Toc42111567)

[4.3.1. Регистрация команды 15](#_Toc42111568)

[4.3.2. Выполнение команды 15](#_Toc42111569)

[4.4. Контроллер танка 16](#_Toc42111570)

[4.4.1. Класс Tank 16](#_Toc42111571)

[4.4.2. Класс TankController 16](#_Toc42111572)

[4.4.3. Класс AITank 18](#_Toc42111573)

[4.4.4. AITankController 21](#_Toc42111574)

[4.5. Разработка лазера 23](#_Toc42111575)

[4.6. Разработка системы сохранения и загрузки 23](#_Toc42111576)

[4.7. Перезагрузка статических элементов 23](#_Toc42111577)

[4.8. Анимация 24](#_Toc42111578)

[4.8.1. Анимация выстрела 24](#_Toc42111579)

[4.8.2. Анимация движения 24](#_Toc42111580)

[4.9. Меню 25](#_Toc42111581)

[4.10. GameOver 25](#_Toc42111582)

[Заключение 26](#_Toc42111583)

[Список использованных источников 27](#_Toc42111584)

[ПРИЛОЖЕНИЕ А. Диаграмма классов. Иерархия наследования классов 28](#_Toc42111585)

# Введение

С развитием цифровых технологий компьютеры все больше вливаются в жизнь человека. Если раньше ЭВМ использовались исключительно для сложных математических вычислений, то сегодня сфера их применения существенно расширилась. Компьютерные игры - одно из наиболее массовых применений электронных вычислительных машин.

Бесспорно видеоигры сегодня один из самых популярных видов проведения досуга. Развитие игровой индустрии шло стремительным темпом. Первые игры отличались простотой интерфейса и логики, но со временем они становились все сложнее и сложнее, над их созданием работал уже не один человек, а целая команда разработчиков. В играх подростки не прочь надеть на себя роль полководца, джедая, летчика, кого угодно. Игры позволяют им почувствовать порой то, что настоящий мир им дать не может.

Не самые маленькие поклонники видеоигр могут помнить о такой замечательной штуке как Nintendo Entertainment System(NES), легендарную игровую приставку, на которую вышла очень известная в русском пространстве игра “Battle City” или, как у нас ее называли, “Танчики”.

Игра “Tanks in the maze”, представленная в данном проекте берет свое начало именно с “Battle City”. Но несмотря на это она имеет достаточно отличительных черт, таких процедурно генерируемый лабиринт, отрикашечивание снарядов и другие вещи.

# Постановка задачи

## Цель

Применение теоретических знаний и практических навыков для создания компьютерной игры «Tanks in the maze» в жанре «shoot ‘em up» с использованием среды разработки Unity3D и языка программирования С#.

В данном проекте предполагается реализовать компьютерную игру с несколькими режимами игры:

* Друг против друга;
* Игра против бота.

Помимо общей механики особый интерес вызывает разработка искусственного интеллекта для бота.

## Основные требования к функционалу:

* Основное меню, в которых производится настройка игры, ее сохранение, загрузка и выбор режима;
* Процедурная генерация лабиринта;
* Консоль, которая запускает разрешенные команды;
* Искусственный интеллект для бота;
* Физика;
* Цветовая проработка;
* Анимация;
* Текстуры;
* Звук;

# Теоретические основы ООП

История становления программирования богата какими-либо нововведениями, которые коренным образом меняют всю предшествующую систему. Одним из первых и наиболее ярких таких открытий можно считать выход в свет книги «Программирование без GOTO». Следующим наиболее важным шагом по праву считают переход к процедурно-функциональному программированию, когда программы разбивались на ряд независимых блоков, и в итоге просто грамотно соединялись в единое целое. Ну и третьим, наверное, самым большим открытием можно считать появление объектно-ориентированного программирования. ООП объединило в себе одновременно процедурные методы, для чего создавались самостоятельные модули, структурное проектирование и стимулировало творческий подход к созданию программ.

Сегодня ни одна программа не представляется без наличия в ней объектов. Объектно-ориентированное программирование вышло на новую ступень своего развития, когда внедрить объект в программу не составляет большого труда, а порой, это за вас сделает визуальная среда. Таким образом, такой мощный инструмент, как объект стал доступным даже для начинающего программиста. Теперь любой программист может использовать всю мощь и эффективность объектно-ориентированного программирования.

Неформально можно сказать, что объектно-ориентированное программирование стоит на трёх китах, таких как инкапсуляция, полиморфизм и наследование. Что же это такое?

Инкапсуляция - это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать обязательства абстракции от их реализации.

Наследование означает такое отношение между классами (отношение родитель - потомок), когда один класс заимствует структурную или функциональную часть одного или нескольких других классов (соответственно, одиночное и множественное наследование).

Полиморфизм: одно и то же имя может означать объекты разных типов, но, имея общего предка, все они имеют и общее подмножество операций, которые можно над ними выполнять.

Главным звеном программы в ООП является объект (например класс).Главным в понятии объекта является объединение идей абстракции данных и алгоритмов. Объекты обладают целостностью, которая не может быть нарушена. Свойства, которые характеризуют объект и его поведение, остаются неизменными.

Объектно-ориентированное проектирование - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Для ООП важную роль играет такое понятие, как абстрагирование.

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных.

Класс делится на две части: интерфейс и реализацию.

Интерфейс отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса.

Внутренняя реализация описывает представление этой абстракции и механизмы достижения желаемого поведения объекта.

Для удобного представления программы используются модульность и иерархия.

Модульность - это разделение программы на фрагменты, которые компилируются по отдельности, но могут устанавливать связи с другими модулями.

# Обоснование выбора языка и среды разработки

## Критерии выбора

Поскольку на написание собственного игрового движка требуется достаточное количество времени, выбор пал на готовые решения. Перед началом разработки был проведен детальный анализ популярных сред разработки компьютерных игр. В некоторых из них используются свои языки программирования, которые порой не являются даже объектно-ориентированными. Из ознакомленных мной раннее языков программирования выбор падал на Lua, Python, C++ и C#. Так же среди немаловажных критериев при выборе были: наличие хорошей документации, большого комьюнити, графического редактора и широкого спектра возможностей игрового движка. Для разработки 2D игры идеально под все критерии вписывается Unity3D.

Также требовалось выбрать программу для анимации. Существует огромное количество программ для создания анимации. Лично мне в данном проекте хотелось потрогать руками, что такое скелетная анимация. К сожалению, для нее большинство ПО является платным, как например Spine, который является лучшим выбором, однако на форумах была посоветована программа DragonBonesPro, в которой есть достаточно ресурсов для создания подобного рода анимации.

Написание кода происходило в Microsoft Visual Studio 2019 за отличный Debugger, совместимый с Unity3D

## Преимущества Unity3D перед другими движками

Первое преимущество – язык на котором ведется разработка, С#. Данный язык высокоуровневый и позволяет программисту легко войти в разработку игры. Это важный момент, потому что в отличие от других движков, где используется язык С++, в C# есть много элементов и приемов, которые уже реализованы, и программисту нужно только воспользоваться ими.

Второе преимущество – кроссплатформенность, т. е. один и тот же код, написанный на движке Unity, с минимальными изменениями может перенесен на различные платформы (PC, Mac, Android, iOS, Web, игровые консоли). Это огромный плюс, который сокращает время на разработку игры в несколько раз.

Третье преимущество – хорошее комьюнити. Это означает, что у различных функций движка есть четкое описание с примерами на сайте разработчика, обратиться к которому можно в любой момент. Если что-то все же осталось непонятным, служба поддержки обязательно ответит на возникший вопрос.

И четвертым преимуществом является Asset Store, где имеется огромное количество различных плагинов и ресурсов для создания игры. Разумеется, какие-то из них бесплатные, какие-то платные, но все они собраны в одном месте с удобным поиском и возможностью загрузить, интегрировать и получить сразу рабочий функционал.

Еще одним плюсом было то, что разработчики вышеупомянутого DragonBonesPro написали библиотеку для совместимости с Unity3D.

# Разработка компьютерной игры

## Краткие сведения об игре

Механика

* **2** режима игры:
* 2 игрока:

*в этом режиме каждому игрокам даются два одинаковых танка, различающихся только цветом*

* игрок против бота
* уникальность каждого уровня при помощи процедурной генерации
* выпущенные снаряды сразу не поглощаются стенами, а отрикашечивают от стен какое-то n раз
* ограничение по кол-ву снарядов, существующих одновременно на карте ( вы выпустили k снарядов, пока хотя бы один не исчезнет, выпустить еще один не выйдет)
* броня, поглощающая одно попадание
* управление осуществляется только клавишами, без мыши

Бот

* танк поддерживает один из сбалансированных режимов. В чем-то он лучше человека, в чем-то хуже.
* Для нахождения кратчайшего пути используется матрица достижений по алгоритму Форда-Уоршелла

Принцип механики бота

* танк генерирует waypoint в рандомном месте в пределе следующей ячейки лабиринта, куда ему надо добраться и едет туда
* когда танк "нащупывает" лазером цель, он также сразу выстреливает и переходит в режим прицеливания на время
* таймер обновляется, когда танк находит цель
* при истечении таймера, танк снова начинает движение

Консоль

* Имеется консоль для навигации по игре, в ней можно перейти в меню, включить или выключить графическое зрение бота, то есть игрок не будет видеть, на какие элементы бот опирается. Так же можно найти пасхалку, найти звук и вызвать меню help и т.д.

Пасхалка

* если ввести в консоль /battlecity, то игрок найдет пасхалку, которая отсылается к той самой игре battlecity на денди

## Генерация лабиринта

Образованием лабиринта занимаются 2 класса: MazeGenerator и Maze-Spawner. Первый занимается программной генерацией, второй – отображением его в самой игре.

Существует достаточное количество различных алгоритмов создания лабиринтов. Из основных – Recursive Backtracking, Hunt-and-Hint, Binary Tree, Wilson's algorithm.

MazeGenerator использует алгоритм Recursive Backtracking, который, с моей личной точки зрения, является наилучшим выбором для построения лабиринта в большинстве случаев.

Преимущества:

1. Быстрота
2. Простота в реализации
3. Создает лабиринты, которые очень эстетичны
4. Из всех алгоритмов он создает наименьшее количество тупиков
5. В результате имеет особенно длинные и извилистые пути.

Недостаток только один : Для генерации потребуется достаточный объем памяти для хранения всего лабиринта, и для этого потребуется пространство стека, пропорциональное размеру лабиринта, поэтому для исключительно больших лабиринтов это может быть довольно неэффективно. Однако, для избежания данной проблемы, алгоритм можно реализовать итерационно.

Перед началом описания алгоритма стоит уточнить, что построение начинается с построения сетки с последующим «ломанием» стен между клетками, образуя проходы.

Алгоритм построения:

1. Выбираем отправную точку.
2. В этой точку случайным образом выбираем стену и делаем проход в соседнюю ячейку, но только если она еще не посещена. Данная клетка становится текущей.
3. Если все соседние ячейки были посещены, вернитесь к последней ячейке, которая имеет «непосещенных соседей», и повторите шаг 2.
4. Алгоритм заканчивается, когда на третьем шаге возврат происходит к начальной ячейке и оказывается, что у нее так же все соседние ячейки были посещены.

Лабиринт готов, однако он не подходит для игры, т. к. на данном этапе у нас отсутствуют циклы, из-за чего из каждой ячейки в другую ведет только один путь.

Для создания циклов дополнительно написана функция LoopAdding. Ее суть в том, чтобы сломать случайные стены. В этом случае и будут создаваться циклы.

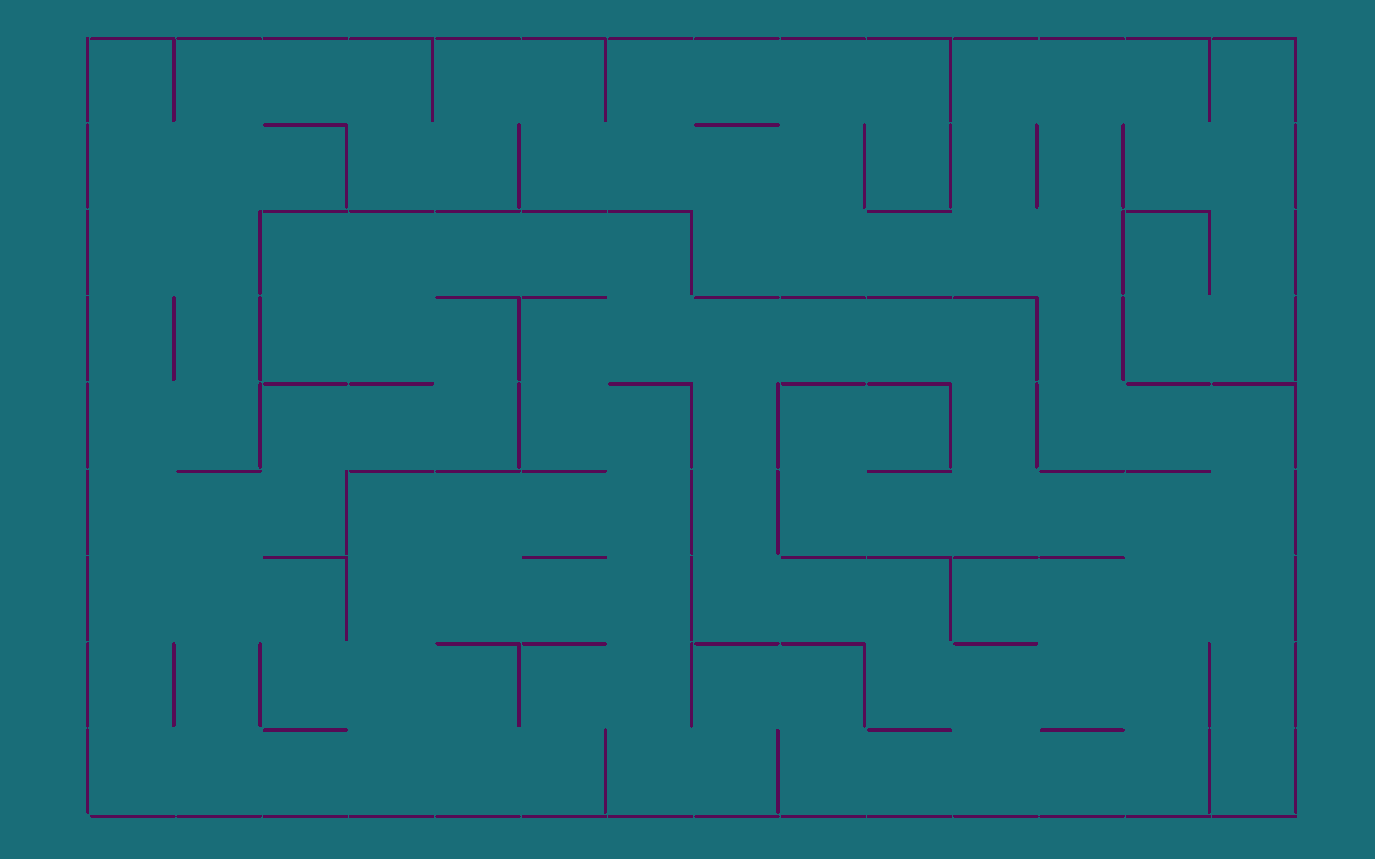
Пример сгенерированного лабиринта можно увидеть на рисунке №1

Рис. 1

## Разработка консоли

Консоль – полезная утилита, дающая возможность запускать функции прямо во время игры. Проектирование консоли разрабатывалась таким образом, чтобы облегчить внедрение новых консольных команд независимо от других функций.

### Регистрация команды

В классе ConsoleController реализована функция registerCommand, которая и отвечает за регистрацию новых команд, она принимает название комманды, функцию, выполняющую действие и help-строку, которая объясняет, что она делает данная команда.

### Выполнение команды

При вводе команды в консоль, она запускает runCommandString, которая разделяет команду на имя\_команды и ее аргументы и передает их функции runCommand, которая уже проверяет, есть ли такая команда и запускает функцию команды в случае положительного результата проверки.

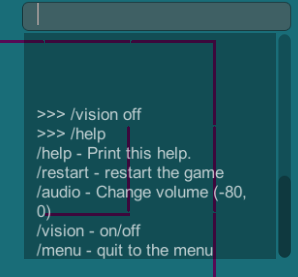
То, как в итоге выглядит консоль видно на скриншоте ниже:

Рис. 2

## Контроллер танка

### Класс Tank

В данном классе содержатся все характеристики танка (количество

жизней, брони и т.д.) и функцию, отвечающую за выстрел, которая может быть запущена только по истечению таймера, который запускается во время выстрела. Данный таймер добавлен для того, чтобы танк не мог выстрелить допустим 10 выстрелов одновременно, а чтобы была эмуляция перезарядки.

### Класс TankController

Данный класс отвечает за управление танком и запуск анимации.

Функция Update, которая в Unity запускается каждый кадр, проверяет, была ли нажата одна из указанных клавиш, если да, то танк начинает двигаться в соответствии с нажатой клавишей.

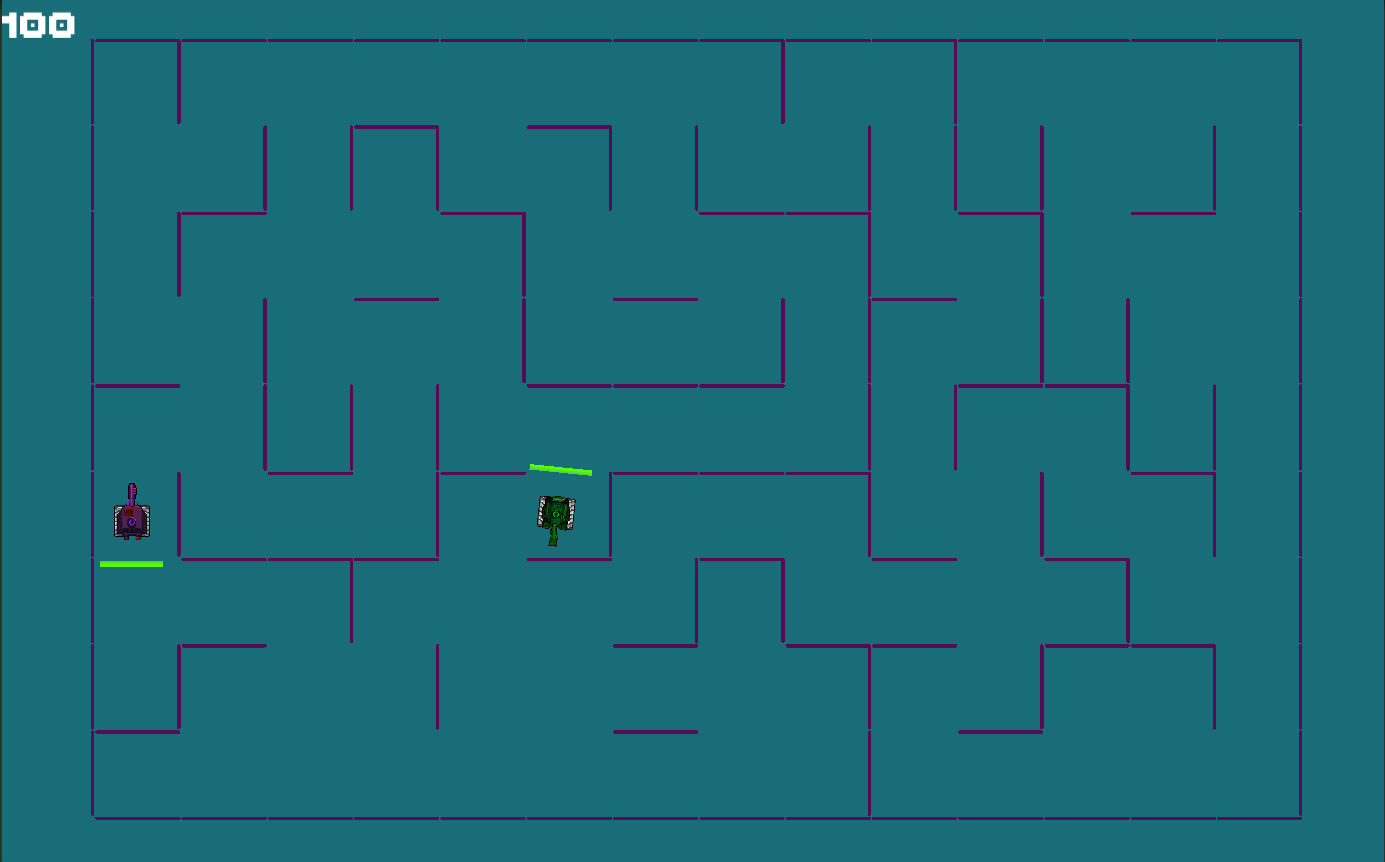
 На скриншоте ниже показано, как выглядит игра 1 против 1

Рис. 3

### Класс AITank

Класс наследуется от класса Tank и отвечает за часть управления ис-

кусственным интеллектом бота.

#### Ориентирование по лабиринту

Для того, чтобы танк мог ориентироваться по графу, он использует матрицу достижимости, которая строится на основе матрицы смежности. Для этого представляем наш лабиринт в виде графа.

Матрицу достижимости получает с помощью алгоритма Форда-Уоршелла

#### Описание алгоритма Форда-Уоршелла:

Ключевая идея алгоритма — разбиение процесса поиска кратчайших путей на **фазы**.

Перед k-ой фазой (k = 1 \ldots n) считается, что в матрице расстояний d[][] сохранены длины таких кратчайших путей, которые содержат в качестве внутренних вершин только вершины из множества \{ 1, 2, \ldots, k-1 \} (вершины графа мы нумеруем, начиная с единицы).

Иными словами, перед k-ой фазой величина d[i][j] равна длине кратчайшего пути из вершины i в вершину j, если этому пути разрешается заходить только в вершины с номерами, меньшими k (начало и конец пути не считаются).

Легко убедиться, что чтобы это свойство выполнилось для первой фазы, достаточно в матрицу расстояний d[][] записать матрицу смежности графа: d[i][j] = g[i][j] — стоимости ребра из вершины i в вершину j. При этом, если между какими-то вершинами ребра нет, то записать следует величину "бесконечность" \infty. Из вершины в саму себя всегда следует записывать величину 0, это критично для алгоритма.

Пусть теперь мы находимся на k-ой фазе, и хотим **пересчитать** матрицу d[][] таким образом, чтобы она соответствовала требованиям уже для k+1-ой фазы. Зафиксируем какие-то вершины i и j. У нас возникает два принципиально разных случая:

* Кратчайший путь из вершины i в вершину j, которому разрешено дополнительно проходить через вершины \{ 1, 2, \ldots, k \}, **совпадает** с кратчайшим путём, которому разрешено проходить через вершины множества \{ 1, 2, \ldots, k-1 \}.

В этом случае величина d[i][j] не изменится при переходе с k-ой на k+1-ую фазу.

* "Новый" кратчайший путь стал **лучше** "старого" пути.

Это означает, что "новый" кратчайший путь проходит через вершину k. Сразу отметим, что мы не потеряем общности, рассматривая далее только простые пути (т.е. пути, не проходящие по какой-то вершине дважды).

Тогда заметим, что если мы разобьём этот "новый" путь вершиной k на две половинки (одна идущая i \Rightarrow k, а другая — k \Rightarrow j), то каждая из этих половинок уже не заходит в вершину k. Но тогда получается, что длина каждой из этих половинок была посчитана ещё на k-1-ой фазе или ещё раньше, и нам достаточно взять просто сумму d[i][k] + d[k][j], она и даст длину "нового" кратчайшего пути.

**Объединяя** эти два случая, получаем, что на k-ой фазе требуется пересчитать длины кратчайших путей между всеми парами вершин i и j следующим образом:

new\_d[i][j] = min (d[i][j], d[i][k] + d[k][j]);

Таким образом, вся работа, которую требуется произвести на k-ой фазе — это перебрать все пары вершин и пересчитать длину кратчайшего пути между ними. В результате после выполнения n-ой фазы в матрице расстояний d[i][j] будет записана длина кратчайшего пути между i и j, либо \infty, если пути между этими вершинами не существует.

Последнее замечание, которое следует сделать, — то, что можно **не создавать отдельную матрицу** \rm new\_d[][] для временной матрицы кратчайших путей на k-ой фазе: все изменения можно делать сразу в матрице d[][]. В самом деле, если мы улучшили (уменьшили) какое-то значение в матрице расстояний, мы не могли ухудшить тем самым длину кратчайшего пути для каких-то других пар вершин, обработанных позднее.

**Асимптотика** алгоритма, очевидно, составляет O (n^3).

Несмотря на плохую асимптотику, алгоритм позволяет один раз посчитать матрицу достижимости и дальше, на ее основе находит кратчайший путь от одной вершины графа до другой за O(k\*n) и даже быстрее.

Поиском матрицы достижимости и восстановлением кратчайшего пути занимается класс **Graph**.

#### Нахождение кратчайшего пути

Нахождение кратчайшего пути осуществляется с помощью матрицы достижимости.

while (length != 0 && c != area)

{

for (int i = 0; i < size; i++)

{

if (i == ind || i == start)

{

continue;

}

if (matrix[ind,i] && (waysMatrix[start, i] + waysMatrix[ind, i] == length))

{

ind = i;

length = waysMatrix[start, i];

way.Add(ind);

break;

}

}

c++; //спасательный круг от зацикливания

}

Суть кода выше в том, чтобы бежать в матрице смежности по строке, которой соответствует вершина А, которой мы ищем смежную вершину Б, принадлежащую кратчайшему пути.

В данной строке мы просматриваем каждую вершину Б, если вершины А и Б смежны и сумма пути от начала до вершины Б и пути от Б до А равна оставшейся длине, то мы нашли следующую вершину и теперь Б становится А.

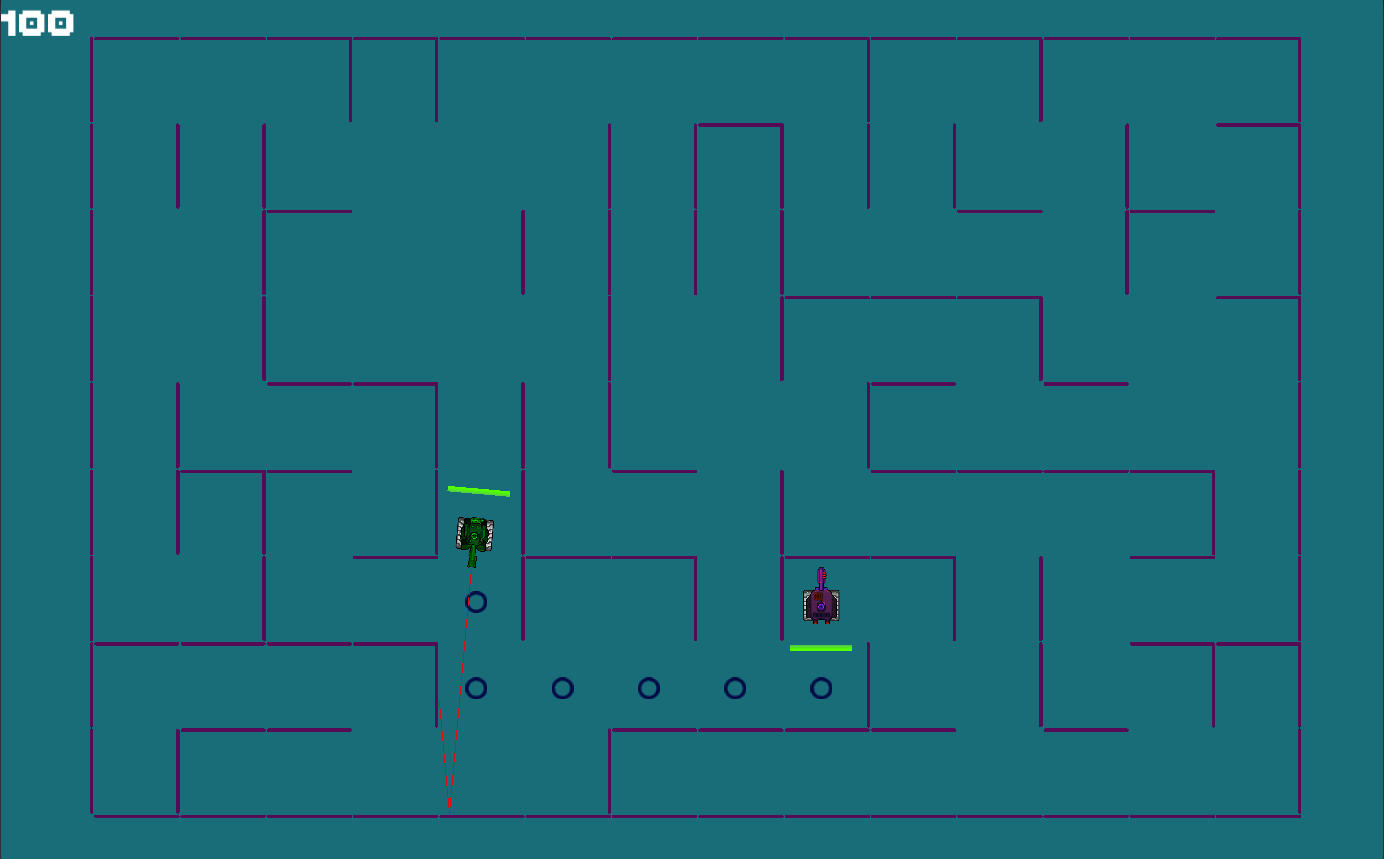
В игре данный путь отображается кругами на скриншоте ниже:

Рис. 4

### AITankController

У танка есть 3 состояния(поворот, ехать вперед, поиск цели), что представляет собой паттерн State.

Принцип механики бота сводится к следующему:

* Танк запрашивает кратчайший путь до врага;
* Танк генерирует waypoint в рандомном месте в пределе следующей ячейки лабиринта, куда ему надо добраться, поворачивается в сторону waypoint и едет туда. Рандом реализован для того, чтобы танк ехал не просто прямо, а как бы немного имитируя змею;
* когда танк "нащупывает" лазером цель, он также сразу выстреливает и переходит в режим прицеливания на время;
* таймер обновляется, когда танк находит цель;
* при истечении таймера, танк снова начинает движение.

На скриншоте видно, что именно имеется ввиду под waypoint(слева):

Рис. 5

## Разработка лазера

За обработку лазера отвечает класс **LaserManager**.

Класс не ограничивается обработкой одного лазера бота. При создании объекта, его оружие заносится в статический список всех лазеров с помощью статического метода **AddLaser**. Каждый кадр класс вызывает фунцию **CalcLaserLine**, которая отвечает за вычисления отражения и вызова функции **RedrawLine** для прорисовки линии лазера.

Дополнительно реализован пул объектов для линий лазера для уменьшения нагрузки. Задача его в том, чтобы не пересоздавать линии, удаляя и создавая их каждый кадр, а просто менять их положение и размер

## Разработка системы сохранения и загрузки

Из-за отсутствия необходимости сохранения большого количества информации не было смысла использовать базы данных, поэтому была применена сериализация данных.

Для этого в Unity3D необходимо создать отдельный тип данных с пометкой [System.Serializable], и записать в объект данного типа все необходимые данные, после чего записать в файл с помощью классов **FileStream** и **BinaryFormatter**.

## Перезагрузка статических элементов

В Unity3D статические элементы не принадлежат сценам. Это значит, что если у вас есть статические элементы, как в моем случае Graph, ConsoleController и LaserManager, то при перезагрузке сцены или загрузке другой могут возникнуть своего рода ошибки, потому что будут использоваться данные, которые были получены из прошлой сцены.

Для избежания ошибки выше, был добавлен статический класс с одним лишь статическим методом **Reload** для перезагрузки всех статических элементов, для этого каждый такой класс должен иметь свой локальный статический **Reload.**

## Анимация

Для анимации, как ранее упоминалось, использовалась программа DragonBonesPro. В ней были созданы 2 простых вида анимации:

* выстрел
* движение

### Анимация выстрела

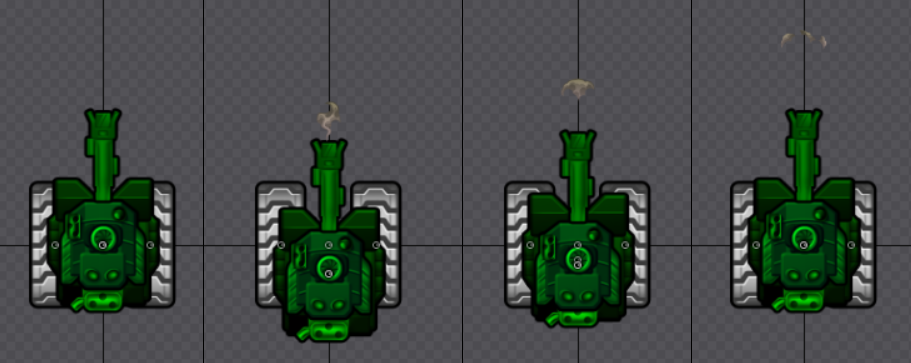


Рис. 6

### Анимация движения

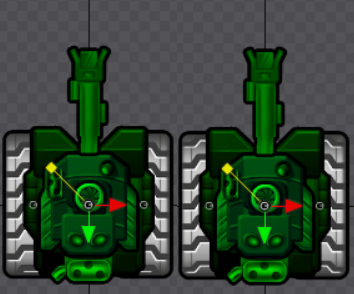
Анимация движения осуществляется лишь поворотом гусениц.

Рис. 7

## Меню

Рис.8

## GameOver

Рис. 9

# Заключение

В результате выполнения курсового проекта была разработана готовая игра, в которую можно играть как с другом, так и одному. В ходе разработки активно использовались преимущества объектно-ориентированного программирования на языке C# и среды разработки Unity3D

При написании проекта был получен значительный опыт в проектировании, построении диаграммы классов, использовании объектно-ориентированного подхода, создании анимации, элементам Unity3D и были углублены знания в языке программирования C#. В целом опыт, полученный при создании игры с помощью Unity3D дал представление, что такое разработка видеоигр, из каких этапов состоит и сложность в целом.

Говоря о разработке видеоигр в целом - это довольно интересная сфера, которая позволяет реализовать любые свои фантазии. Например, при желании можно рассказать какую-нибудь историю в жанре графической новеллы или как платформер, написать игру о выживании в лесу любого уровня реалистичности. Так же стоит сказать про симуляторы. К примеру, летчиков перед их первым вылетом могут посадить в дорогостоящий симулятор, который практически точно передаст все ощущения и события, которые могут возникнуть во время полета на реальном самолете.

# Список использованных источников

* WIKIPEDIA [Электронный ресурс] – WIKIPEDIA :

<https://ru.wikipedia.org>

* DOCS.MICROSOFT [Электронный ресурс] – DOCS.MICROSOFT: <https://docs.microsoft.com>
* DOCS.UNITY3D [Электронный ресурс] – DOCS.UNITY3D:

<https://docs.unity3d.com/Manual/index.html>

* DRAGONBONES [электронный ресурс] – DRAGONBONES:

<http://dragonbones.com/en/learn.html>

* Шаблоны игрового программирования:

<https://live13.livejournal.com/462582.html>

* Joseph Hocking - Unity в действии. Мультиплатформенная разработка на C# - 354c
* Торн Алан - Основы анимации в Unity – 423 c

# ПРИЛОЖЕНИЕ А. Диаграмма классов. Иерархия наследования классов

