Оглавление

I. **Игровой процесс**
   - A. *Суть игры*
   - B. *Управление*
   - C. *Игровые объекты*

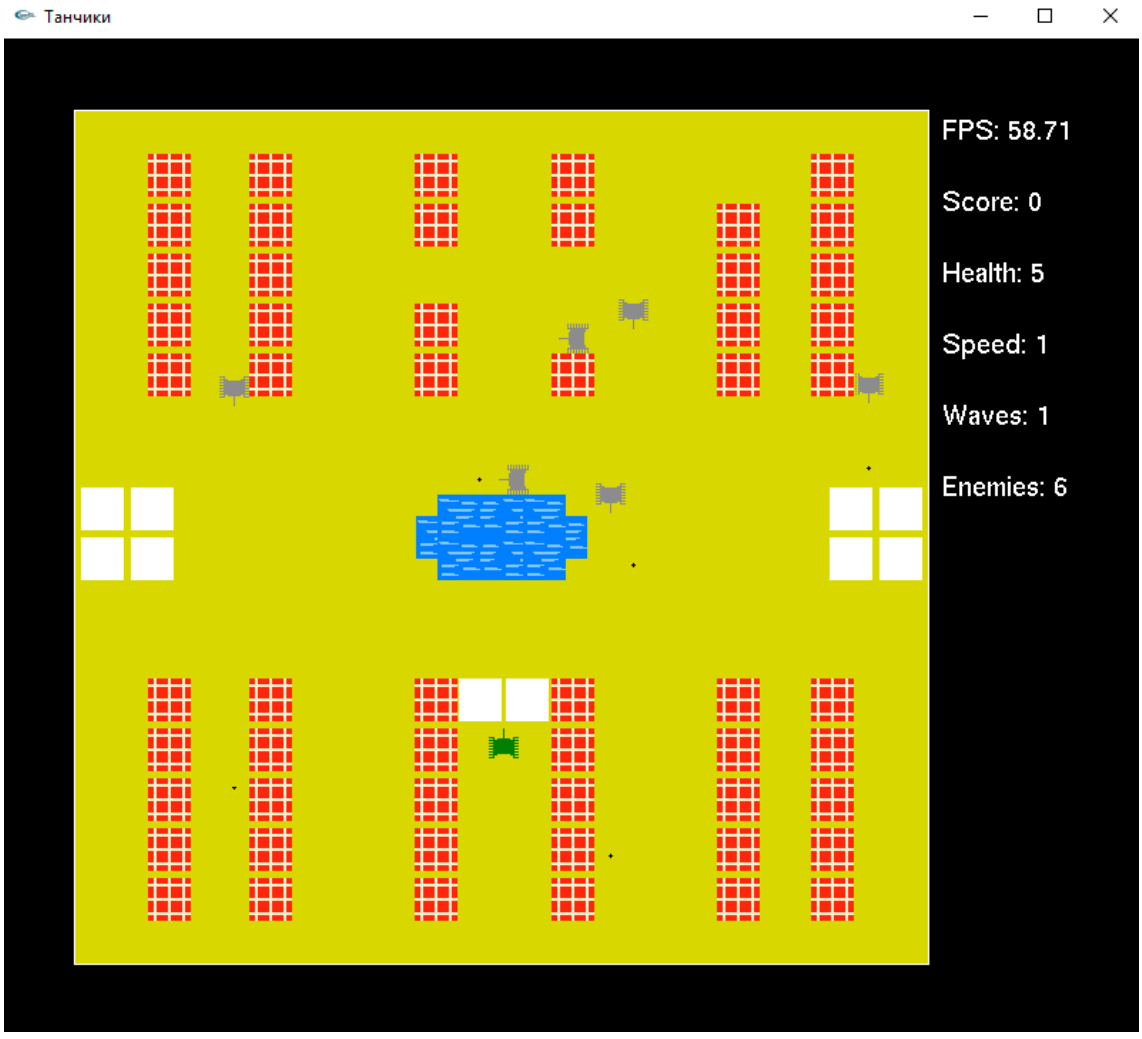
II. **Описание разработанной программы**
   - A. *Использование сторонних библиотек*
     1. Библиотека glut.h
     2. Библиотека STL
   - B. *Схема классов программы*
   - C. *Разбор классов*
     1. Point
     2. Wall
     3. Bullet
     4. Tank
     5. BoxCollider
     6. Painter
     7. Controls
     8. Game
   - D. *Принцип работы ИИ*
   - E. *Принцип работы BoxCollider*
   - F. *Разбор функции main*

III. **Исходный код программы**

Игровой процесс

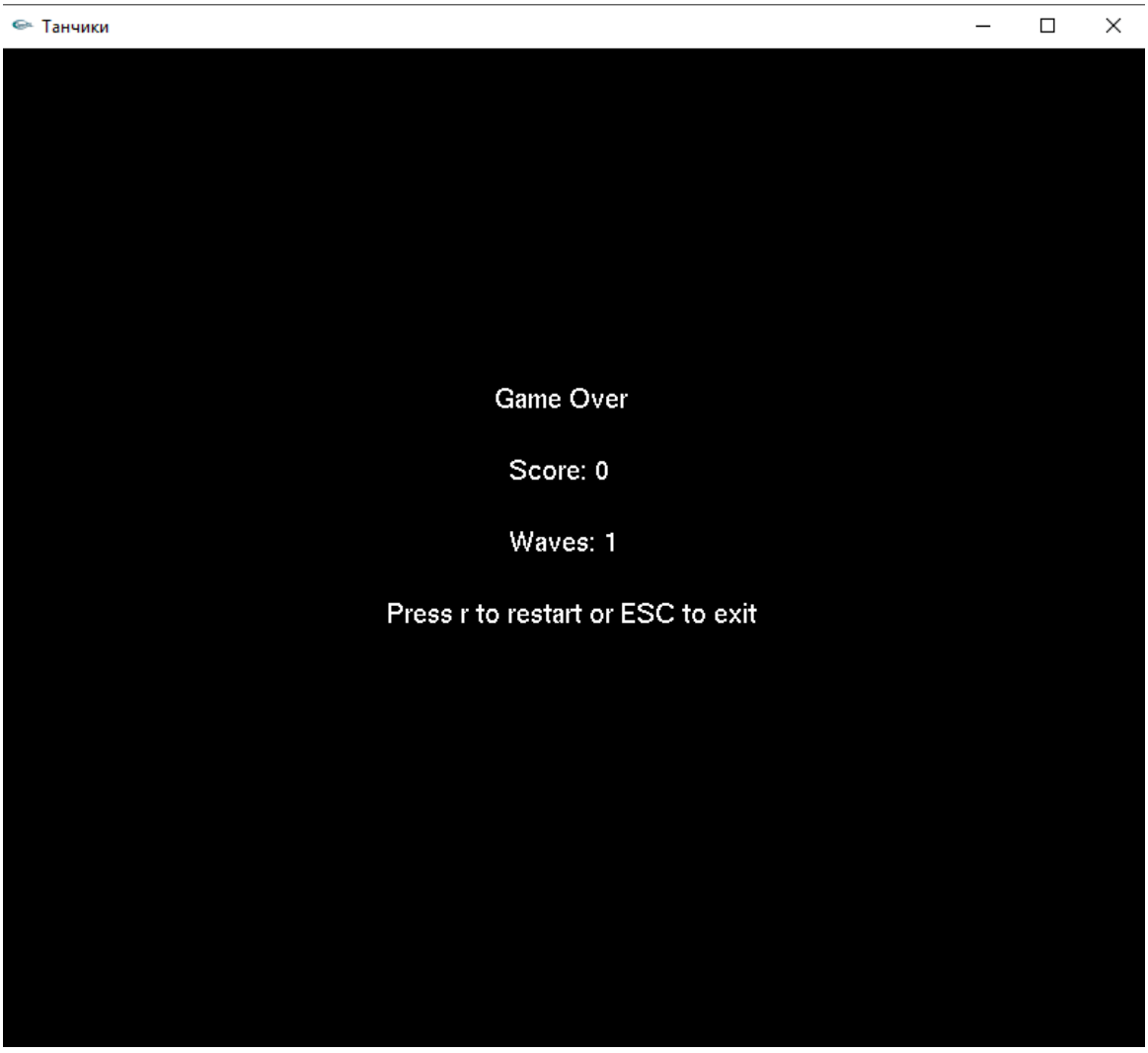
Суть игры

Игроку под руководство даётся танк зелёного цвета. Цель игры пережить как можно больше волн врагов(серые танки), которые будут появляться до самого конца игры, пока игрок не потеряет все жизни или не решит выйти из программы принудительно. Боты, благодаря специальному алгоритму, будут стараться уничтожить вас, поэтому не стоит надеяться на успех с первого раза. Игрок может использовать на карте укрытия, часть из которых разрушаема, а часть нет. Интерфейс справа от карты покажет вам кадровую частоту, количество заработанных вами очков, количество здоровья, скорость, номер волны и количество не уничтоженных противников до новой волны.



Если игрок теряет 1 жизнь, то он возрождается на том же месте, что и в самом начале игры.

Если игрок растерял все свои жизни, то выводится экран окончания игры(Game Over). Экран Game Over даст игроку исчерпывающие данные о его игровом опыте и предложит выйти из игры или переиграть.



Управление

- Движение вверх(w, ↑)
- Движение вправо(d, →)
- Движение влево(a, ←)
- Движение вниз(s, ↓)
- Стрельба(l)
- Выход из игры(ESC)
- Перезапуск в Экране Game Over(r)

Игровые объекты

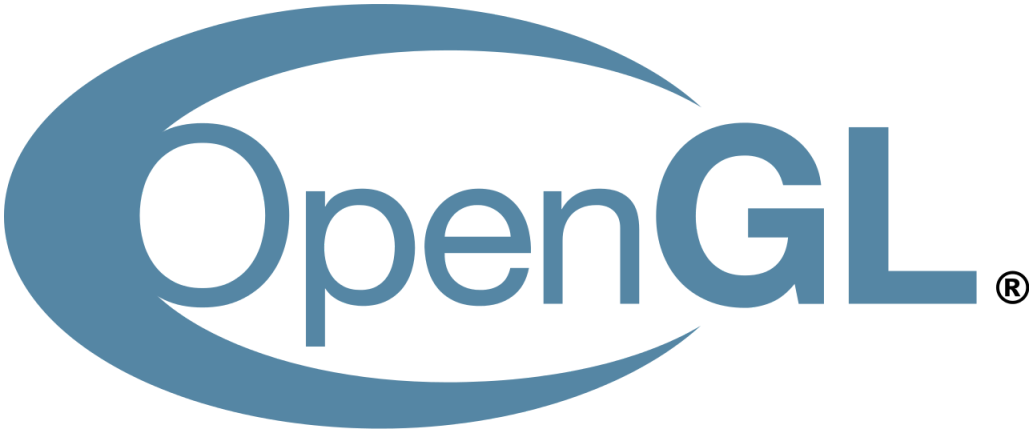
- Не разрушаемая стена(белая)
- Разрушаемая кирпичная стена(оранжевая)
- Вода
- Танк игрока(зелёный)
- Танки ботов(серые)

Описание разработанной программы

Использование сторонних библиотек

Библиотека glut.h

OpenGL Utility Toolkit (GLUT) — библиотека утилит для приложений под OpenGL, которая в основном отвечает за системный уровень операций ввода-вывода при работе с операционной системой. Из функций можно привести следующие: создание окна, управление окном, мониторинг за вводом с клавиатуры и событий мыши. Она также включает функции для рисования ряда геометрических примитивов: куб, сфера, чайник. GLUT даже включает возможность создания несложных всплывающих меню.



При разработки программы данная библиотека использовалась для создания окна с палитрой RGB и двойным буфером. При помощи инструментария данной библиотеки нарисованы все игровые объекты и производится мониторинг за вводом с клавиатуры.

Библиотека STL

Библиотека стандартных шаблонов (STL) (англ. *Standard Template Library*) — набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.
Данная библиотека использовалась для организации игровых объектов в **vector**(реализация динамического массива переменного размера) для более удобной работы и высокой производительности, нежели обычные динамические массивы.

Схема классов

Схема классов не отображает полной сигнатуры, лишь поля и методы классов без типов и параметров. Более подробно будет в разборе каждого класса в отдельности.

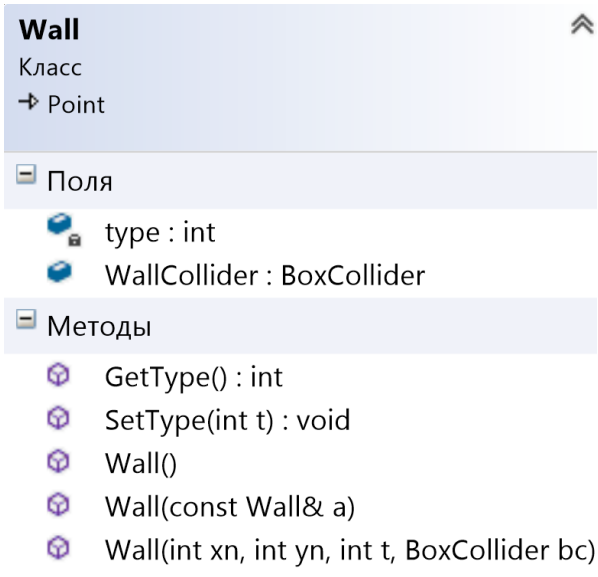
Разбор классов

Теперь пройдёмся по всем классам в отдельности.

Рассмотрим класс **Point**, который согласно схеме является базовым для **Wall**, **Bullet** и **Tank.** Характеризует центр объекта всех производных классов.

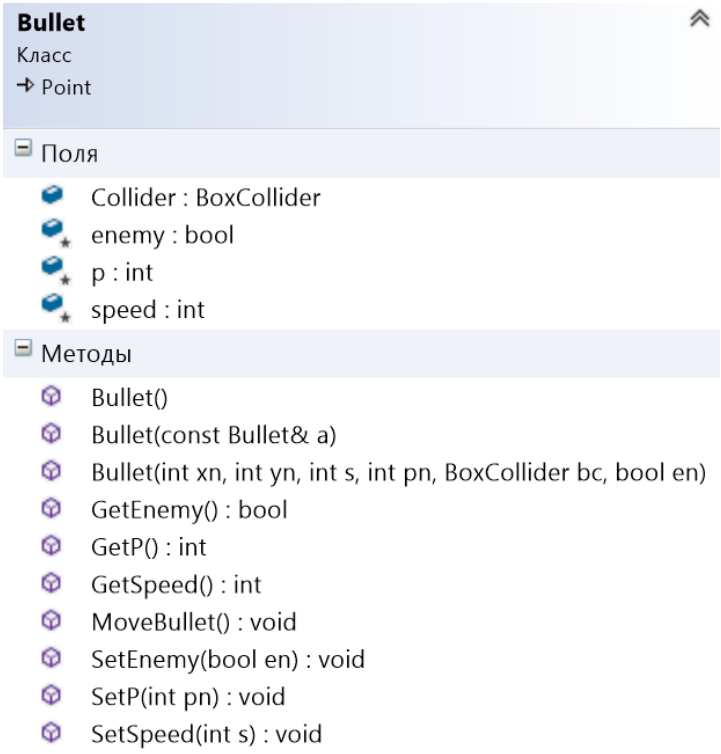
- **Поля x, y**, несут в себе информацию о нахождении игрового объекта на карте(его координаты). Для всех производных классов эти 2 поля будут являться центром объекта.
- Класс **Point** имеет 3 конструктора(по умолчанию, с параметрами и копирования).
- Геттеры, сеттеры для доступа к полям с уровнем доступа *protected.*
- У класса переопределён поток ввода, для чтения данных из файла. Это нужно для загрузки уровня из файла **LoadLevel** в классе **Game**.

Рассмотрим класс **Wall**, который согласно схеме является производным от класса **Point.**



- Поля и методы от базового класса смотреть в описании класса **Point.**
- Поле **type** определяет тип стены
    - 1 - не разрушаемая стена(белая)
    - 2 - разрушаемаемая кирпичная стена(оранжевая)
    - 3 - вода
- Поле **WallCollider** определяют форму объекта для обработки физического столкновения. Подробнее в разборе класса **BoxCollider**.
- Класс **Wall** имеет 3 конструктора(по умолчанию, с параметрами и копирования).
- Геттеры, сеттеры для доступа к полям с уровнем доступа *private*.
- У класса переопределён поток ввода, для чтения данных из файла. Это нужно для загрузки уровня из файла **LoadLevel** в классе **Game**.

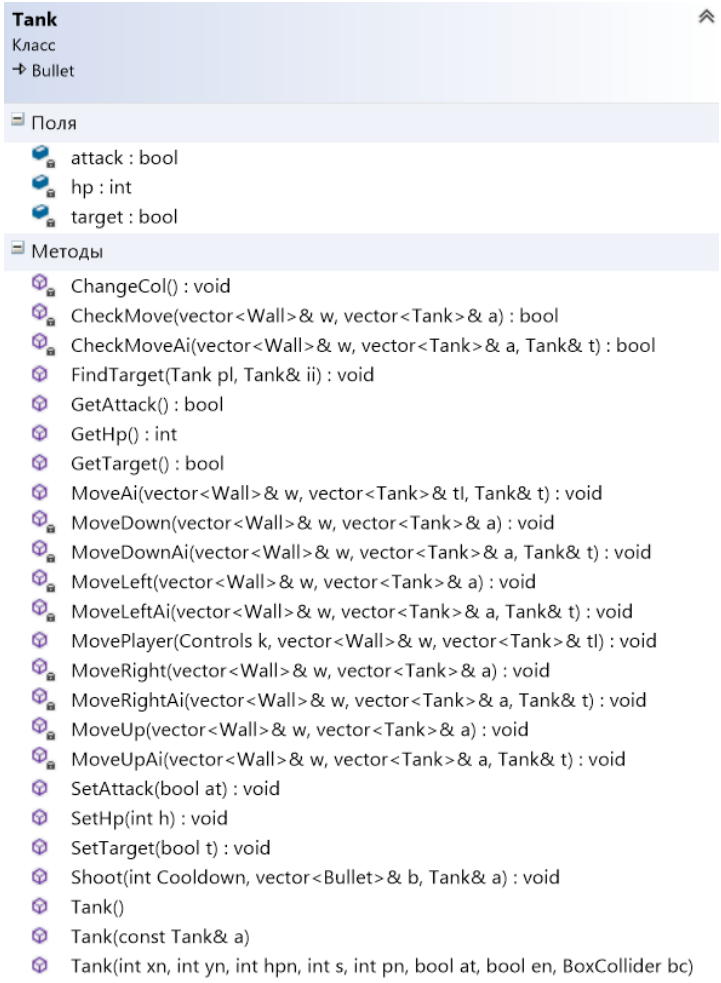
Рассмотрим класс **Bullet**, который согласно схеме является производным от **Point** и базовым для **Tank.**



- Поля и методы от базового класса смотреть в описании класса **Point.**
- Поле **Collider** определяют форму объекта для обработки физического столкновения. Подробнее в разборе класса **BoxCollider**.
- Поле **enemy** позволяет определить принадлежность пули(вражеская или игрока). В производных классах аналогичный принцип.
    - true - вражеская
    - false - игрока
- Поле **p** определяет направление движения пули. В производных классах тоже отвечает за направление.
    - 1-вверх
    - 2-вправо

    - 3-влево
    - 4-вниз
- Поле **speed** определяет скорость движения пули. В производных классах тоже отвечает за скорость.
- Класс **Bullet** имеет 3 конструктора(по умолчанию, с параметрами и копирования).
- Геттеры, сеттеры для доступа к полям с уровнем доступа *protected.*
- Метод **MoveBullet** выполняет перемещение пуль по игровому полю согласно их направлению(поле **p**).
- У класса переопределён поток ввода, для чтения данных из файла. Это нужно для загрузки уровня из файла **LoadLevel** в классе **Game**.

Рассмотрим класс **Tank**, который согласно схеме является производным от **Bullet** и **Point**.

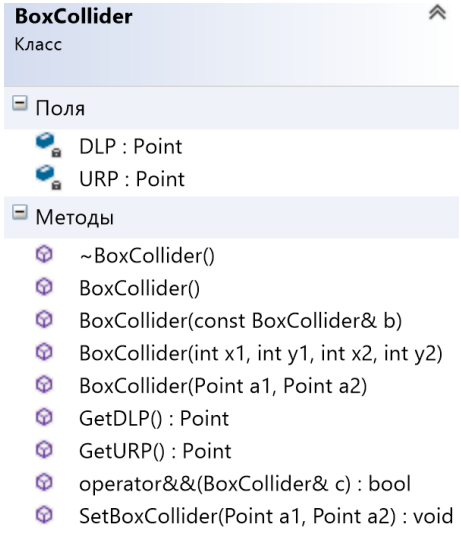


- Поля и методы от базовых классов смотреть в соответствующих описаниях классов **Bullet** и **Point.**
- Поле **attack** является своего рода переключателем, можно ли уже стрелять или нет.
    - true - уже можно стрелять
    - false - нельзя стрелять
- Поле **hp** отвечает за очки здоровья.
- Поле **target** является своего рода переключателем, можно ли уже ботам искать цель или нет. Подробно об ИИ в части про **принцип работы ИИ**

    - true - цель найдена(ещё нельзя искать)
    - false - цель не найдена(уже можно искать)
- Класс **Tank** имеет 3 конструктора(по умолчанию, с параметрами и копирования).
- Геттеры, сеттеры для доступа к полям с уровнем доступа *private.*
- Метод **ChangeCol** выполняет корректировку Collider в связи с изменением положения объекта на игровом поле.
- Метод **FindTarget** создаёт поток thread(C++11), в котором осуществляется поиск игрока на игровом поле. На входе в поток значение поля target устанавливается true(цель найдена), на выходе из потока после задержки Sleep значение становится false(цель не найдена). Это позволяет обнаруживать игрока 1 раз в единицу времени. Подробнее о ИИ в соответствующей главе.
- Метод **Shoot** производит выстрел танком. Создаётся поток thread(C++11), в котором создаётся объект класса Bullet, он помещается в вектор всех пуль в игре ArrayOfBullet. На входе в поток, значение поля attack устанавливается false(стрелять нельзя), на выходе из потока после задержки Sleep(имитация некой задержки, перезарядки) значение становится true(можно стрелять). Это позволяет вести стрельбу с некой задержкой(перезарядкой).
- Методы движения для игрока
    - **CheckMove** - проверяет можно ли вообще двигаться, нет ли столкновения с другими объектами на игровом поле(стены, боты, вылет за пределы игрового поля). Аргументы метода: вектор всех стен и ботов в игре.
        - true - можно двигаться
        - false - нельзя двигаться
    - **MoveUp** - осуществляет движение вверх, предварительно произведя **CheckMove**, в который передаёт свои аргументы. Аргументы метода: вектор всех стен и ботов в игре. Перед завершением работы для корректировки положения **Collider** вызывает **ChangeCol.**
    - **MoveRight** - осуществляет движение вправо, предварительно произведя **CheckMove**, в который передаёт свои аргументы. Аргументы метода: вектор всех стен и ботов в игре. Перед завершением работы для корректировки положения **Collider** вызывает **ChangeCol.**
    - **MoveLeft** - осуществляет движение влево, предварительно произведя **CheckMove**, в который передаёт свои аргументы. Аргументы метода: вектор всех стен и ботов в игре. Перед завершением работы для корректировки положения **Collider** вызывает **ChangeCol.**
    - **MoveDown** - осуществляет движение вниз, предварительно произведя **CheckMove**, в который передаёт свои аргументы. Аргументы метода: вектор всех стен и ботов в игре. Перед завершением работы для корректировки положения **Collider** вызывает **ChangeCol.**

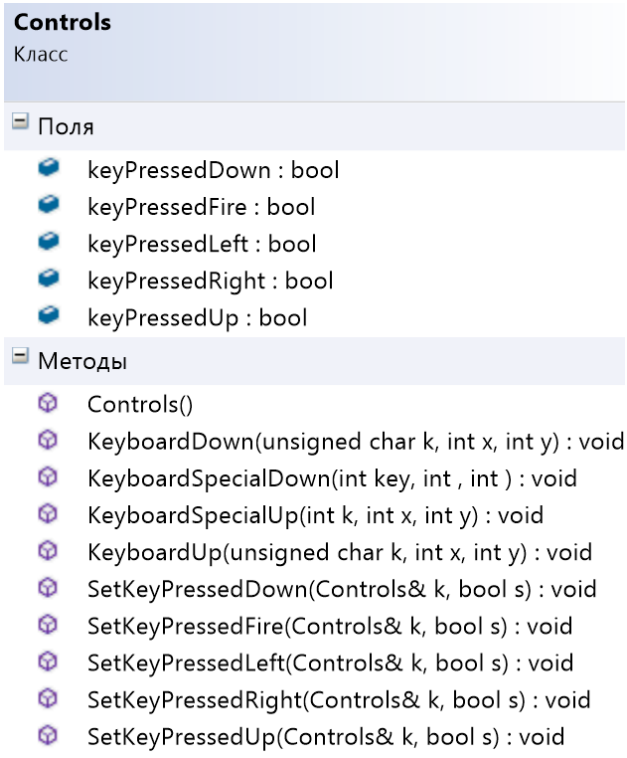
    - **MovePlayer** - соотносит логическое значение полей класса **Controls** с направлением движения и вызывает в зависимости от его поля и значения **MoveUp, MoveRight, MoveLeft или MoveDown.** Аргументы метода: вектор всех стен и ботов в игре, объект класса **Controls**(подробнее в описании класса **Controls**).
- Методы движения для ботов
    - **CheckMoveAi** - проверяет можно ли вообще двигаться, нет ли столкновения с другими объектами на игровом поле(стены, другие боты, игрок, вылет за пределы игрового поля). Аргументы метода: вектор всех стен и ботов в игре, танк игрока.
        - true - можно двигаться
        - false - нельзя двигаться
    - **MoveUpAi** - осуществляет движение вверх, предварительно произведя **CheckMoveAi**, в который передаёт свои аргументы. Аргументы метода: вектор всех стен и ботов в игре, танк игрока. Перед завершением работы для корректировки положения **Collider** вызывает **ChangeCol.**
    - **MoveRightAi** - осуществляет движение вправо, предварительно произведя **CheckMoveAi**, в который передаёт свои аргументы. Аргументы метода: вектор всех стен и ботов в игре, танк игрока. Перед завершением работы для корректировки положения **Collider** вызывает **ChangeCol.**
    - **MoveLeftAi** - осуществляет движение влево, предварительно произведя **CheckMoveAi**, в который передаёт свои аргументы. Аргументы метода: вектор всех стен и ботов в игре, танк игрока. Перед завершением работы для корректировки положения **Collider** вызывает **ChangeCol.**
    - **MoveDownAi** - осуществляет движение вниз, предварительно произведя **CheckMoveAi**, в который передаёт свои аргументы. Аргументы метода: вектор всех стен и ботов в игре, танк игрока. Перед завершением работы для корректировки положения **Collider** вызывает **ChangeCol.**
    - **MoveAi** - соотносит значение поля p (см. описание класса Bullet) с направлением движения и вызывает в зависимости от его значения **MoveUpAi, MoveRightAi, MoveLeftAi или MoveDownAi,** в которые передаёт свои аргументы. Аргументы метода: вектор всех стен и ботов в игре, танк игрока.
- У класса переопределён поток ввода, для чтения данных из файла. Это нужно для загрузки уровня из файла **LoadLevel** в классе **Game**.

Рассмотрим класс **BoxCollider.**



- Поле **DLP** (Down Left Point) нижняя левая точка.
- Поле **URP** (Up Right Point) правая верхняя точка.
- Класс **BoxCollider** имеет 4 конструктора(по умолчанию, 2 с параметрами и копирования).
- Геттеры, сеттеры для доступа к полям с уровнем доступа private.
- Перегрузка логической операции **&&** “И”. Самая важная операция в игре, позволяет определять есть ли столкновение с другими **BoxCollider**. Это основа физического взаимодействия. Подробнее в разборе **принципа разбора BoxCollider.**
    - true - есть столкновение
    - false - нет столкновения

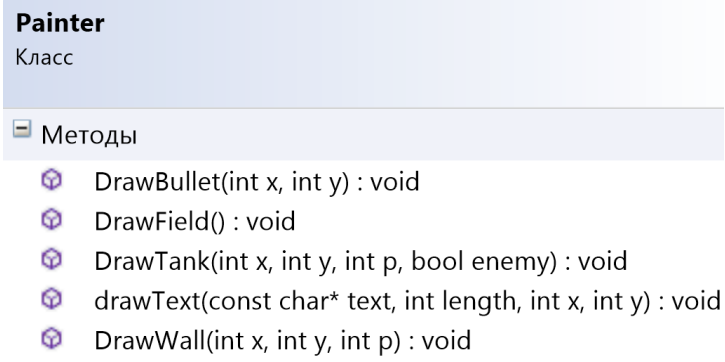
Рассмотрим класс **Controls**.



- Поле **keyPressedUp** нажата ли кнопка, отвечающая за движение вверх.

    - true - нажата
    - false - не нажата
- Поле **keyPressedDown** нажата ли кнопка, отвечающая за движение вниз.
    - true - нажата
    - false - не нажата
- Поле **keyPressedRight** нажата ли кнопка, отвечающая за движение вправо.
    - true - нажата
    - false - не нажата
- Поле **keyPressedLeft** нажата ли кнопка, отвечающая за движение влево.
    - true - нажата
    - false - не нажата
- Поле **keyPressedFire** нажата ли кнопка, отвечающая за выстрел.
    - true - нажата
    - false - не нажата
- Конструктор по умолчанию
- статический метод **KeyboardDown** обрабатывает нажатия клавиш на клавиатуре.
- статический метод **KeyboardUp** обрабатывает отжатие клавиш клавиатуры.
- статический метод **KeyboardSpecialDown** обрабатывает нажатия специальных клавиш на клавиатуре.
- статический метод **KeyboardSpecialUp** обрабатывает отжатие специальных клавиш клавиатуры.
- Сеттеры устанавливают значение публичных переменных в статических методах класса **Controls.**

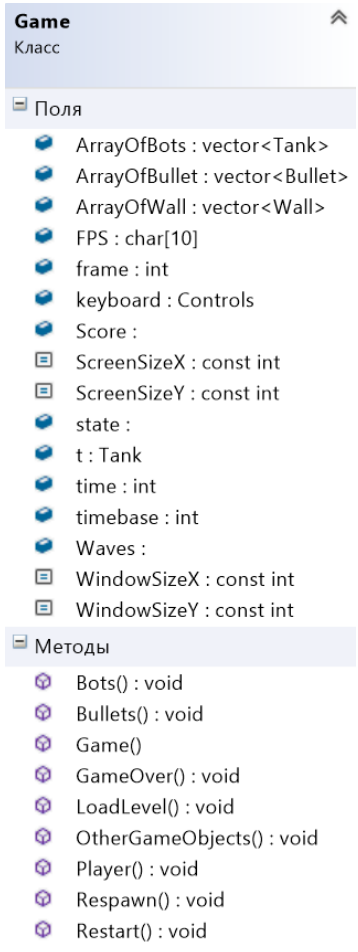
Рассмотрим класс **Painter**.



- Статический метод **DrawBullet** рисует пулю. Разберём его аргументы:
    - **x, y** - центр пули, которую нужно нарисовать
- Статический метод **DrawField** рисует границы игрового поля и закрашивает его жёлтым цветом.
- Статический метод **DrawTank** рисует танк. Разберём его аргументы:
    - **x, y** - центр танка, который нужно нарисовать

    - **p** - направление движения (подробнее в классе **Tank**), чтобы орудие смотрело в правильную сторону
        - 1 - вверх
        - 2 - вправо
        - 3 - влево
        - 4 - вниз
    - **enemy** - вражеский танк или нет (подробнее в классе **Tank**), чтобы покрасить их в разный цвет
        - true - серый
        - false - зелёный
- Статический метод **drawText** рисует текст. Разберём его аргументы:
    - **text** - текст, который нужно нарисовать
    - **length** - длина текста, который нужно нарисовать
    - **x, y** - левая нижняя точка откуда начнёт рисоваться текст
- Статический метод **DrawWall** рисует блок. Разберём его аргументы:
    - **x, y** - центр стены, которую нужно нарисовать
    - **type** - тип стены, который нужно нарисовать
        - 1 - нарисуется не разрушаемая стена(белая)
        - 2 - нарисуется разрушаемая кирпичная стена(оранжевая)
        - 3 - нарисуется кубик воды

Рассмотрим класс **Game**.

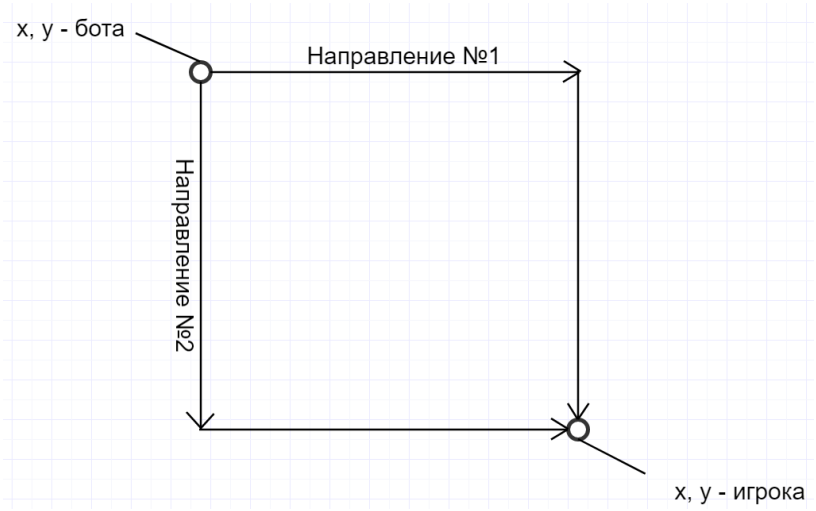


- Поле **ArrayOfBots** вектор всех ботов в игре.
- Поле **ArrayOfBullet** вектор всех пуль в игре.
- Поле **ArrayOfWall** вектор всех стен в игре.
- Поле **FPS** хранит в себе строку с информацией об FPS.
- Поле **frame** хранит в себе количество итераций игрового цикла(нужно для подсчёта FPS).
- Поле **keyboard** выполняет роль контролера, в котором хранятся все данные об управлении.
- Поле **Score** содержит в себе очки игрока.
- Поле **ScreenSizeX** содержит в себе ширину экрана пользователя. Нужно для центрирования окна программы.
- Поле **ScreenSizeY** содержит в себе высоту экрана пользователя. Нужно для центрирования окна программы.
- Поле **state** содержит в себе состояние, в котором находится игра
    - 0 - игра работает в обычном режиме, все объекты обрабатываются
    - 1 - игра окончена выводится экран Game Over
    - 2 - загрузка новой волны противников
- Поле **t -** танк игрока.
- Поле **time -** таймер, необходим для подсчёта FPS.
- Поле **TimeBase -** промежуточная переменная необходимая для подсчёта FPS.
- Поле **Waves** номер волны противников.
- Поле **WindowSizeX** ширина окна.
- Поле **WindowSizeY** высота окна.
- Конструктор по умолчанию.
- Метод **Bots** обрабатывает все действия, которые происходят с ботами(отрисовка, движение, стрельба).
- Метод **Bullets** обрабатывает все действия, которые происходят с пулями(отрисовка, движение, столкновение с объектами, уничтожение объектов, в которые попадает пуля).
- Метод **GameOver** подводит конечную статистику, уничтожает все игровые объекты и выводит экран Game Over.
- Метод **LoadLevel** осуществляет загрузку уровня, заполняя **ArrayOfBots** и **ArrayOfWall** данными из файлов.
- Метод **OtherGameObjects** обрабатывает все действия с другими игровыми объектами (отрисовывает интерфейс с игровым полем, отрисовывает карту, ведёт подсчёт и вывод FPS)
- Метод **Player** обрабатывает все действия, которые происходят с игроком (отрисовка, движение, стрельба)

- Метод **Respawn** осуществляет возрождение игрока в начальную точку после потери 1 жизни.
- Метод **Restart** перезапускает игру, заново загружая необходимые данные из файлов(вызывает **LoadLevel)**.

Принцип работы ИИ

После переключения поля **target** (см. разбор класса **Tank**) в положение false(цель можно искать) и вызова функции **FindTarget** (см. разбор класса **Tank**) в потоке будет происходить примерно следующее. **target** станет true (цель будет нельзя искать), программа рандомно выберет направление №1 или №2(то есть вправо, вниз пока не вызовется снова **Find Target** или не произойдёт столкновение с каким-либо игровым объектом).

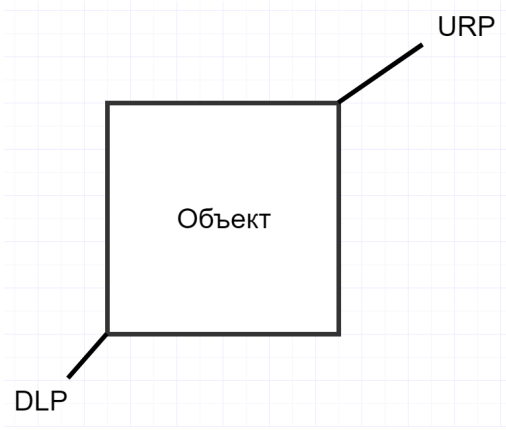


Затем будет задержка в 1 секунду, поле **target** станет снова false(цель можно будет искать). Всё снова повторится, в итоге боты будут получать данные о местоположение игрока 1 раз в 1 секунду и стараться двигаться к нему, каждый раз рандомно выбирая направление №1 или №2(зигзагообразно). Если на пути ботов встретится какое-то препятствия(в том числе сам игрок), то столкнувшись с ним они рандомно поменяют направление движения, немного отъедут, после чего снова сработает **FindTarget** и они направятся в сторону игрока.

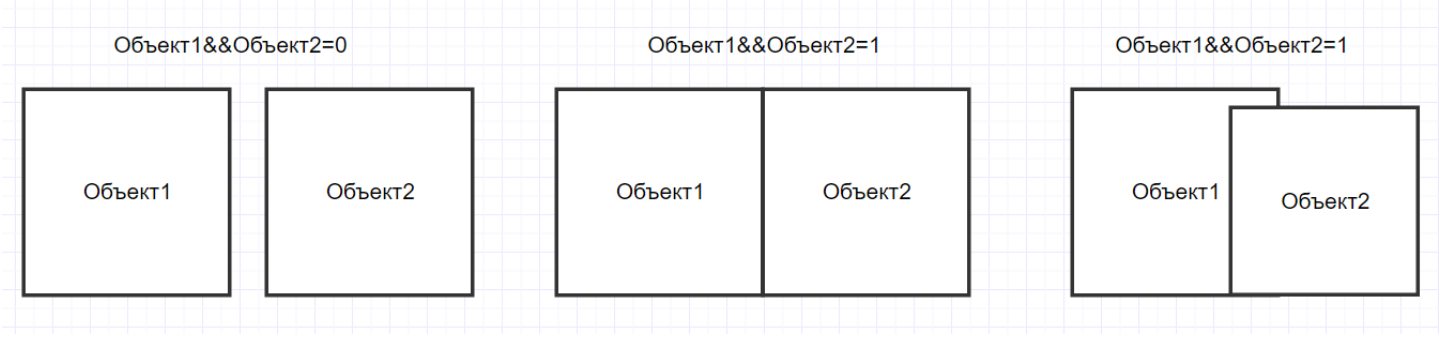
Алгоритм, конечно, не идеален, так как, 1 бот может долго наворачивать круги вокруг игрока, стараясь попасть в него, он попадёт, но будет это не очень скоро, только группой ботов удаётся загнать игрока в угол.

Принцип работы BoxCollider

У нас есть 2 точки(см. разбор класса **BoxCollider**), которые задают собой квадрат или прямоугольник



При перегрузке оператора **&&** “И” (см. разбор класса **BoxCollider**) мы проверяем на столкновение(вхождение 1-го **BoxCollider** в другой), вот результат работы всех случаев.



Это позволяет реализовать довольно примитивную физику в программе, чтобы например 1 объект(танк) не проезжал сквозь другой объект(стена).

Разбор функции main

Производится инициализация GLUT. Установка основных параметров окна и регистрация обратных вызовов(строка 45-48) которые вызываются при необходимости. После выполнения **glutMainLoop** постоянно, до условия выхода из программы, будет вызываться **glutDisplayFunc**, ну и зарегистрированные ранее обработчики вызовов при необходимости.

```
33  int main(int argc, char **argv)
34  {
35      glutInit(&argc, argv);  // Инициализация GLUT
36      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);//устанавливаем палитру цветов RGB и двойной буфер
37      glutInitWindowPosition(G1.ScreenSizeX/2- G1.WindowSizeX/2, G1.ScreenSizeY/2- G1.WindowSizeY/2); //устанавливаем позицию окна
38      glutInitWindowSize(G1.WindowSizeX, G1.WindowSizeY);//Устанавливает размер окна
39      glClearColor(0, 0, 0, 1.0);                     //задаём значение очистки цветом буфера цвета
40      glutCreateWindow("Танчики");                    //создание окна "Танчики"
41      glMatrixMode(GL_PROJECTION);                    //говорит о том, что команды относятся к проекту.
42      glLoadIdentity();                               //считывает текущую матрицу
43      glOrtho(0, 800, 0, 700, -1, 1);                 //установка ортогональной проекции
44      glutDisplayFunc(renderScene);                   //регистрация обратных вызовов
45      glutKeyboardFunc(&Controls::KeyboardDown);      //регистрация нажатий кнопок на клаве
46      glutKeyboardUpFunc(&Controls::KeyboardUp);      //регистрация отжатий кнопок на клаве
47      glutSpecialFunc(&Controls::KeyboardSpecialDown);//регистрация нажатий спец кнопок
48      glutSpecialUpFunc(&Controls::KeyboardSpecialUp);//регистрация отжатий спец кнопок
49      Timer(0);                                       //устанавливаем таймер
50      glutMainLoop();     // Основной цикл GLUT
51      return 0;
52  }
```