Министерство образования Республики Беларусь

Учреждение образования

БелорусскиЙ государственный университет

информатики и радиоэлектроники

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

Программное игровое средство «3d shooter»

|  |  |  |
| --- | --- | --- |
| Студент |  | Е. Н. Траханов |
| Руководитель |  | Д.С. Шулицкий |

Минск 2019

СОДЕРЖАНИЕ

[Введение 5](#_Toc513665878)

[1 Анализ предметной области 7](#_Toc513665879)

[1.1 Обзор аналогов 7](#_Toc513665880)

[1.2 Постановка задачи 9](#_Toc513665881)

[2 Разработка программного средства 10](#_Toc513665882)

[2.1 Структура программы 10](#_Toc513665883)

[2.2 Интерфейс программного средства 10](#_Toc513665884)

[2.3 Работа со звуком 11](#_Toc513665885)

[2.4 Прорисовка карты и других объектов 1](#_Toc513665888)2

[2.5 Взаимодействие с объектами и игровая логика 16](#_Toc513665889)

[2.6 Работа с таблицей рекордов 1](#_Toc513665889)7

[3 Тестирование программного средства 20](#_Toc513665890)

[4 Руководство пользователя 22](#_Toc513665891)

[4.1 Правила игры 22](#_Toc513665892)

[4.2 Интерфейс программы 22](#_Toc513665893)

[Заключение 24](#_Toc513665894)

[Список использованных источников 25](#_Toc513665895)

[Приложение А. Исходный код программы 26](#_Toc513665896)

ВВЕДЕНИЕ

На сегодняшний день наблюдается бурное развитие компьютерной техники и информационным технологий в целом. В современном мире информационные технологии распространены повсеместно, что позволяет работать с большими объёма информации за относительно малое время, делая жизнь современного человека более удобной. Немалую роль в этом сыграли компьютерные игры, которые привили интерес к вычислительной технике более широкой группе людей, что стимулировало продажи, и, следовательно, и развитие компьютерной техники. Нередко компьютерные игры становились целью, на пути к достижении которой разработчики создавали новые технологии, использование которых выходило за рамки данной компьютерной игры. Так, например, Операционная система UNIX — прародитель многих современных операционных систем, таких как Linux, Android, Mac OS X была создана для запуска игры  «Space Travel», написанной Кеном Томпсоном.

Попытки создать игры на цифровых устройствах предпринимались ещё до начала Второй Мировой войны (а в 1947 уже была запрограммирована первая электронная игра, монитором для которой служил экран военного радара - это был симулятор вражеских ракет) - однако считается, что первой компьютерной игрой стала "ОХО" ("Крестики нолики"), в одиночку сделанная А.С. Дугласом в далёком 1952 году.

Первой трёхмерной игрой для персональных компьютеров считается 3D Monster Maze. Стоит отметить, что графическое изображение в игре имеет низкое [разрешение](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D0%BA%D0%B0)) и выполнено символами [псевдографики](https://ru.wikipedia.org/wiki/%D0%9F%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D0%BA%D0%B0).

Наиболее популярной на трёхмерной игрой стала Wolfenstein 3D.

Для отрисовки игрового мира использовался алгоритм ray-casting, для рендеринга трёхмерных сцен на двухмерный экран с помощью проекционных лучей, которые проецируются от глаз наблюдателя к [источнику света](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA_%D1%81%D0%B2%D0%B5%D1%82%D0%B0).. Первый алгоритм рейкастинга (не рейтрейсинга), используемый для рендеринга, был представлен Артуром Аппелем в 1968 году. В основе рейкастинга стои́т идея испускать лучи из «глаз» наблюдателя, один луч на [пиксель](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D0%BA%D1%81%D0%B5%D0%BB%D1%8C), и находить самый близкий объект, который блокирует путь распространения этого луча.

Целью данного курсового проекта является разработка игрового программного средства «3D Shooter», использующий технологию ray-casting и, подобно 3D Monster Maze, представляющий изображение с помощью символов псевдографики. Стоит также отметить, что данная программа будет полностью в консольном режиме, при этом используя только некоторые функции WinAPI. Таким образом данное ПО не имеет сильной привязки к конкретной библиотеке, что позволяет сосредоточиться на самом изучении принципов трёхмерной графики, а не на её реализации в какой-либо графической библиотеке, также это делает проект более гибким в плане визуализации.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Обзор аналогов

Пожалуй, наиболее популярными играми с использованием псевдотрёхмерных движков являются Wolfenstein 3D и Doom, разработанные студией id Software.

Особенностями игры Wolfenstein 3D является то, что один из её создателей, Джон Кармак, использовал необычный в то время подход с отображением графики с помощью [метода бросания лучей](https://ru.wikipedia.org/wiki/Ray_casting) (Ray casting, алгоритм которого будет рассмотрен далее в разделе 2.4), при котором требуется рассчитывать только поверхности, видимые игроку, а не всё окружающее его пространство.



Рисунок 1.1 – «Wolfenstein 3D», аналог игры «Shooter 3D»

Игра Doom (1993 г.) также создавалась под предводительством Джона Кармака. Движок создавался на [Си](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) на рабочих станциях [NeXT](https://ru.wikipedia.org/wiki/NeXT) на операционной системе [NEXTSTEP](https://ru.wikipedia.org/wiki/NEXTSTEP).



Рисунок 1.2 – «Doom», аналог «Shooter 3D»

Самой первой 3D игрой для персональных компьютеров было 3D Monster Maze, выпущенная в 1981 году.

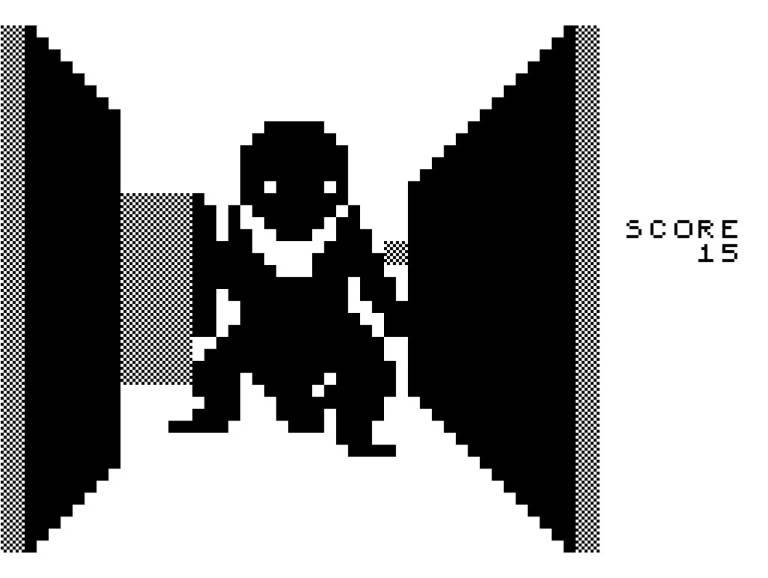


Рисунок 1.3 – “3D Monster Maze”, первая 3D игра.

1.2 Постановка задачи

В рамках данного курсового проекта планируется разработка игрового программного средства «3D Shooter», адаптированного под запуск на современных операционных системах Windows.

В процессе реализации будут разработаны алгоритмы отрисовки карты и объектов на ней в трёхмерном представлении, а также игровой логики, а именно взаимодействие игрока с объектами карты и с самой картой.

В ходе разработки данного программного средства планируется решить следующие задачи:

* выбор уровней карты;
* отрисовка двумерной карты в трёхмерном представлении;
* отрисовка противников и других объектов;
* взаимодействие с другими объектами на карте;
* стрельба по противникам;
* поведение противников;
* изменение уровня сложности;
* меню с возможностью выбора уровней и настроек;
* изменение уровня сложности;
* визуализация таблицы рекордов.

Для разработки программного средства будет использоваться язык программирования C++ и функции WinAPI.

2 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

2.1 Структура программы

В данном приложении использовались следующие классы и файлы:

–Enemy – класс, описывающий поведение и состояние базовых противников;

– EnemySoldier – класс, который расширяет класс Enemy;

– Bullet – класс, отвечающий за реализацию снарядов;

– EventHandler – класс, позволяющий отследить нажатие и момент поднятия клавиши;

– Menu – класс, содержащий модель главного меню;

– Pause – класс, содержащий модель меню паузы;

– Levels – класс, содержащий модель меню выбора уровней и саму информацию об уровнях;

– Sound – класс, обеспечивающий работу со звуком;

-EDS – класс, использующийся для подписи таблицы рекаордов;

– Engine.h – файл, содержащий основные константы и структуру, описывающую состояния игрока;

– Source.cpp – основной файл, в котором осуществляется прорисовка и всё остальное.

2.2 Интерфейс программного средства

Внешний вид и удобность в использовании являются одними из главных критериев качества программного средства. Поэтому взаимодействие приложения с пользователем необходимо организовать максимально интуитивно и просто.

Для реализации удобного меню была выбрана парадигма MVC. В данной реализации модель представляется с помощью отдельного класса, а контроль и отображение осуществляются с помощью функций в файле Source.cpp, при этом контроллер изменяет только состояние модели, а отображение обращается только к модели

Класс главного меню Menu имеет поле itemPointer перечислимого типа ItemPointer, который представляет собой указатель на конкретный пункт меню. Все значения перечислимого типа соответствуют пунктам меню. Далее представлено описание этого типа:

enum ItemPointer {ipStart, ipLevels, ipRecords, ipQuit, ipAmount};

Соответствующие пункты меню:

.– «Start»;

– «Levels»;

– «Records»

-«Quit».

Последнее значение перенумерованного типа ItemPointer используется для обозначения количества пунктов меню.

Отображение данного меню представлено на рисунке 2.1.



Рисунок 2.1 – Главное меню

С помощью класса EventHandler фиксируется поднятие клавиши “W”, “S” или пробела и вызывается функция Move(int direction) класса Menu с соответствующим параметром. Данная функция изменяет указатель на пункт меню и возвращает значение перенумерованного типа GameState, которое определяет настоящее состояние игры. Описание типа GameState приведено ниже:

enum GameState {gsMainMenu, gsRun, gsPause, gsQuit, gsChooseLevel,

gsLoadLevel, gsShowRecords, gsGameOver};

При выборе пункта “start” выбирается карта, на которой враги генерируются случайно, при этом при убийстве одного врага генерируются дополнительные двое.

При выборе пункта “levels” переменная gameState, отвечающая за состояние игры, меняет своё значение на gsChooseLevel, и, соответственно далее идёт работа с классом Levels, который устроен аналогично классу Menu. Однако, данный класс также содержит в себе описание уровней. Каждый уровень описывается следующей структурой:

typedef struct \_Level

{

int ObjectsNum;//кол-во объектов на уровне

Object\* objects;//казатель на массив объектов

int MapWidth, MapHeight;//высота и ширина карты

std::string map;//сама карта

float PlayerX, PlayerY;//координаты, в которых появляется игрок при загрузке карты

} Level;

Object является структурой, которая содержит информацию о том, к какому типу принадлежит данный объект и его координаты на данной карте. Тип, который указывает, к какому виду принадлежит объект, является перенумерованным:

enum ObjectType {otEnemy, otEnemySoldier};

typedef struct \_Object

{

ObjectType type;

float x, y;

} Object;

При выборе пункта “records” аналогичным образом осуществляется показ таблицы рекордов. Более подробно работа с таблицей рекордов будет рассмотрена в пункте 2.6.

При выборе пункта “Quit” переменная состояния принимает значение gsQuit, для которого не предусмотрено никаких действий, кроме выхода из главного цикла игры, в ходе чего программа завершает работу.

Также стоит упомянуть о состояниях паузы и завершении игры.

При значении gsPause переменной состояния повляется меню, из которого можно выйти в главное, прдолжить играть, или выйти из игры вовсе.

В состоянии gsGameOver показывается результат игрока за последнюю игру, который включает в себя время игры и количество уничтоженных врагов. Из данного меню можно выйти в главное меню или из самой программы.

2.3 Работа со звуком

Работу со звуком обеспечивает класс Sound. При этом при запуске программы происходит загрузка аудиофайлов в динамическую память для более быстрого и удобного доступа к ним в дальнейшем. Для удобства работы с данными аудиофайлов была определена следующая структура:

typedef struct \_SoundData

{

char \*sndBuf;//буфер, куда записываются данные аудифайла

int size;//размер буфера

bool isLoaded;//флаг, который показывает загружен ли аудифайл в динамичсекую память

} SoundData;

Также для описания того, какой именно звук содержит данная структура был создан перенумерованный тип SoundName:

При создании объекта данного класса в конструкторе Sound происходит загрузка аудиофайлов в динамическую память, при этом, если загрузка не удалась, флаг isLoaded принимает значение false, иначе - true. Далее представлен код процедуры, реализующей эту загрузку:

Sound::Sound()

{

int size;

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

{

std::ifstream f;

switch (i)

{

case sndShoot:

f.open(FileShoot, std::ios::binary);

break;

case sndGameOver:

f.open(FileGameOver, std::ios::binary);

break;

case sndMain:

f.open(FileMain, std::ios::binary);

break;

}

if (f.is\_open())

{

f.seekg(0, std::ios\_base::end);

size = f.tellg();

sounds[i].size = size;

sounds[i].sndBuf = new char[size];

f.seekg(0, std::ios\_base::beg);

f.read(sounds[i].sndBuf, size);

f.close();

sounds[i].isLoaded = true;

}

else

sounds[i].isLoaded = false;

}

}

Проигрывание загруженных аудиофайлов осуществляется в функции Play с помощью API функции PlaySound. Данная функция принимает в качестве параметра значение перенумерованного типа, которое определяет какой звук надо проиграть.

void Sound::Play(SoundName Name)

{

if (sounds[Name].isLoaded)

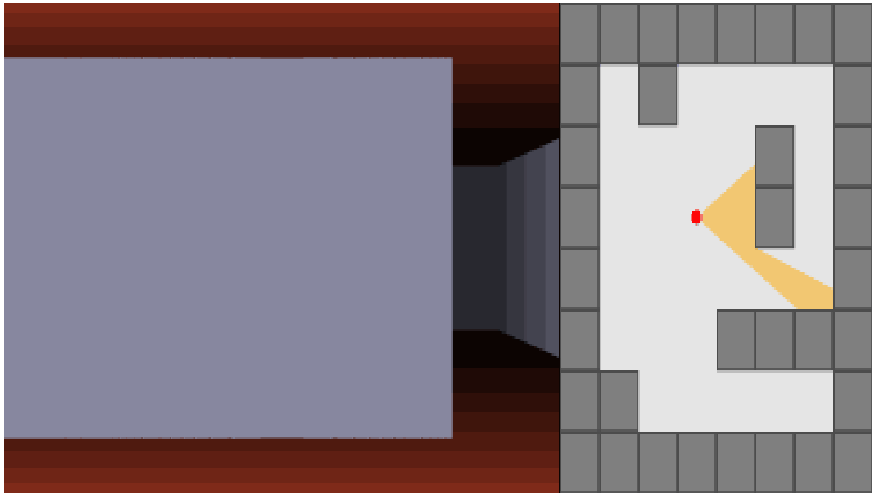
sndPlaySound((LPCSTR)sounds[Name].sndBuf, SND\_MEMORY |SND\_ASYNC);

}

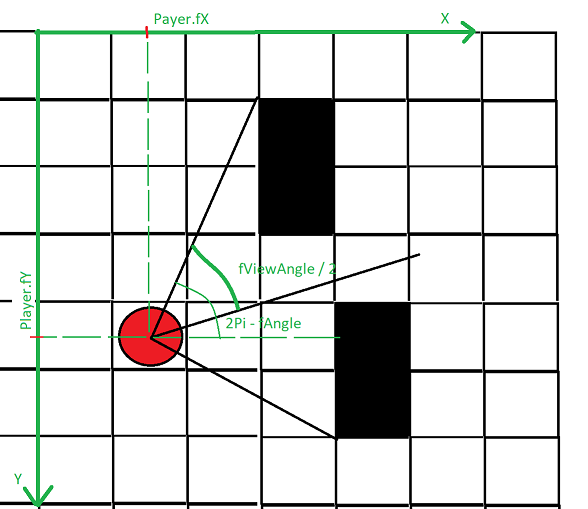
2.4 Прорисовка карты и других объектов

Для прорисовки карты используется такой метод рендеринга, как ray casting (метод “бросания лучей”), при котором изображение строится на основе замеров пересечения лучей с визуализируемой поверхностью.

Игровой мир построен на квадратной сетке из стен однородной высоты и ровных сплошных полов и потолков. Для отрисовки игрового уровня из области наблюдения «выпускается» луч, который проходит через каждый столбец символов на экране. Далее алгоритм проверяет, не пересёк ли луч стену, и если пересёк, то в соответствии с этим выбирает и масштабирует текстуру стены согласно тому, где на уровне луч «столкнулся» со стеной и насколько далеко он распространился до этого.



Для реализации данного алгоритма в структуре Player есть поле fAngle, которое обозначает угол камеры с положительным направлением оси 0X и принимает значения от 0 до 2π. Данное значение изменяется при нажатии клавиш “A” или “W”, т.е. при нажатии данных клавиш происходит поврот камеры вправо или влево.



Далее в цикле для всех столбцов символов “выпускаются” лучи и производятся необходимые измерения. Ниже приведена часть кода, реализующая основную часть алгоритма “бросания” лучей:

for (x = 0; x < ScreenWidth; x++)

{

/\*вычисляем угол луча относительно основной системы координат\*/

float fRayAngle = (player.fAngle - fViewAngle / 2.0f)

+ ((float)x / ScreenWidth) \* fViewAngle;

float fWallDistance = 0;

bool isWall = false;

while (!isWall && fWallDistance < player.fViewRad)

{

fWallDistance += 0.1f;//"наращиваем" луч

/\*вычисляем координаты конца отрезка луча \*/

int TestX = (int)((player.fX + cosf(fRayAngle) \* fWallDistance) + MapWidth) % MapWidth;

int TestY = (int)((player.fY + sinf(fRayAngle) \* fWallDistance) + MapHeigth) % MapHeigth;

/\* если луч вышел за рамки карты, останавливаем вычисление \*/

if (TestX < 0 || TestX >= MapWidth || TestY < 0 || TestY >= MapHeigth)

{

isWall = true;

fWallDistance = fDepth;

}

else

{

/\* фиксируем столкновение луча со стеной и останавливаем вычисления\*/

if (map[TestY \* MapWidth + TestX] == '#')

isWall = true;

}

}

DepthBuf[x] = fWallDistance;//заносим значение расстояния до стены в буфер глубины

/\*вычисляются координаты верхней границы стены в символах\*/

int Ceiling = (float)(ScreenHeight / 2.0) - ScreenHeight / ((float)fWallDistance);

/\*вычисляются координтаты нижней границы стены в символах \*/

int Floor = ScreenHeight - Ceiling;

…

}

Для того, чтобы изображение выглядело более трёхмерным, для отрисовки стен используются разные текстуры, в зависимости от расстояния до них:

if (fWallDistance <= 5)

WallShade = 178;

else

if (fWallDistance <= 10)

WallShade = 177;

else

WallShade = 176;

Отрисовка противников и других объектов происходит немного иначе. Сначала проверяется, попадает ли противник в сектор обзора игрока и, если попадает, вычисляем расстояние до него и выполняем соответствующую отрисовку. Далее приведён код, реализующий данную идею:

/\*вычисляем вектор от игрока до противника\*/

float alpha = atan2(enemy->getY() - player.fY, enemy->getX() - player.fX);

/\*проверяем, попадает ли противник в сектор обзора\*/

if ((alpha > CheckLow && alpha < CheckHigh) || isVisual)

{

int Ceiling = ScreenHeight / 2.0f - ScreenHeight / (fEnemyDistance);

int Floor = ScreenHeight - Ceiling;

int Height = Floor - Ceiling;

…

}

2.5 Взаимодействие с объектами и игровая логика

Все игровые события и отрисовка происходят внутри главного игрового цикла, выход из которого осуществляется только в том случае, если переменная состояния приняла значение gsQuit.

Основной игровой цикл выполняется при значении переменной состояния gsRun. В данном цикле происходит обработка нажатий клавиш, вызываются функции, которые инициируют поведение противников, полёт пуль, отрисовка карты и интерфейса, на котором изображено количество кадров в секунду, координаты игрока, угол поворота камеры и запас здоровья. В конце цикла происходит проверка запаса здоровья игрока, и если оно меньше либо равно нулю, то переменная gameState принимает значение gsGameOver и происходит отрисовка соответствующего меню и запись в таблицу рекордов.

Для хранения объектов карты используется класс-контейнер list, представляющий двусвязный список. Класс-контейнер list был выбран по той причине, что в ходе игры будет часто происходить удаление объектов, а данная структура позволяет выполнить данную операцию за O(1).

В файле Source объявлены список объектов класса Bullet и список указателей на объекты класса Enemy. Для противников использовался список указателей на объекты, а не список самих объектов для того, чтобы можно было работать с экземплярами классов, производных от класса Enemy. Наиболее важным методом класса Enemy и его производных является метод void Move(float time), который обеспечивает поведение противников. Т.к. интерфейс данного метода не отличается от интерфейса в производных классах и данный метод надо будет вызывать для экземпляров разных классов, то удобно объявить метод Move как виртуальную функцию.

Как объекты класса Bullet, так и объекты Enemy и его производных имеют поле isDestroy, которое определяет, нужно ли удалять объект из списка или нет. В функциях UpdateBullet проверяется. Попала ли пуля в стену или в противника, и, если попала, то поле isDestroy принимает значение true. Функция UpdateEnemyes() проверяет, попала ли пуля в противника и также выставляет значение поля isDestroy объекта класса Enemy и его производных в зависимости от этого. Данная функция представлена ниже:

void UpdateEnemyes()

{

enemyes.sort([](const Enemy\* en1, const Enemy\* en2) {return \*en1 < \*en2; });

//SortEnemyes();

float size;

for (Enemy\* pEnemy : enemyes)

{

for (Bullet &bullet : bullets)

{

float size = pEnemy->getSize();

if (bullet.getX() > pEnemy->getX() - size

&& bullet.getX() < pEnemy->getX() + size

&& bullet.getY() < pEnemy->getY() + size

&& bullet.getY() > pEnemy->getY() - size)

{

KillsCounter++;

pEnemy->setDestroy(true);

bullet.setDestroy(true);

}

}

}

enemyes.remove\_if([](Enemy\* en) {

if (en->getDestroy())

{

delete(en);

return true;

}

return false; });

}

2.6 Работа с таблицей рекордов

Работа с таблицей рекордов реализована с помощью класса Records.

Данный класс содержит структуру Record, которая хранит в себе результат игрока для определённого уровня. Как было указано ранее, результат состоит из двух элементов: времени и количества убитых врагов. Следовательно, структура имеет следующий вид:

typedef struct \_Record

{

int kills;

int time;

} Record;

Сами рекорды хранятся в массиве values. Также определено значение результата по умолчанию, на случай если рекорд для данного уровня ещё не поставлен.

Также для проверки подлинности таблицы рекордов при загрузке их из файла используется цифровая подпись, работу с которой обеспечивает класс EDS. В данном классе реализована хеш-функция, которая в дальнейшем в качестве своих аргументов будет принимать указатель на массив рекордов и размер этого массива в байтах. Таким образом, в общем случае для разных значений рекордов значение хеш-функции будет различно, однако не стоит забывать про вероятность коллизии. Для быстроты вычислений используется довольно простая хеш-функция, код которой приведён ниже:

#define sqr(a) a \* a

int EDS::GetHash(char\* data, int size)

{

int h0 = SecretStartVaue;

int h;

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

{

h = (sqr(h0 + (unsigned char) data[i])) % R;

h0 = h;

}

return h0;

}

Сама цифровая подпись будет представлять значение данной хеш-функции. Также для определения факта наличия подписи непосредственно перед самой подписью будет ставиться заранее определённый непечатный символ.

При запуске программы происходит попытка чтения рекордов из файла. При этом проверяется наличие специального непечатного символа, который определяет, существует ли цифровая подпись. Если файл успешно прочитан и цифровая подпись найдена, то на основании прочитанных рекордов вычисляется значение хеш-функции, которое сравнивается с прочитанным из файла значением и, если они равны, подпись признаётся действительной. Если хотя бы один из этих этапов прошёл некорректно (не удалось обнаружить файл, нет цифровой подписи и т.д.), то рекорды заполняются значением по умолчанию. Далtе приведён код, реализующий данные действия:

void LoadMyRecords()

{

bool isCorrect = true;

char temp;

int size;

EDS eds;

Record tempRec;

Record tempRecs[4];

std::ifstream fin(RecordsFileName);

if (fin.is\_open())

{

for (int i = 0; i < 4; i++)

{

tempRec = records.GetRecord(0);

if (fin >> tempRec.kills && fin >> tempRec.time)

records.SetRecord(i, tempRec);

else

isCorrect = false;

}

if (!(fin >> temp) && isCorrect)

isCorrect = false;

if (temp == EDSSign && isCorrect)

{

if (isCorrect && !(fin >> Siganture))

isCorrect = false;

for (int i = 0; i < 4; i++)

{

tempRecs[i] = records.GetRecord(i);

}

if (isCorrect && eds.GetHash((char\*)tempRecs, 4 \* sizeof(Record)) != Siganture)

isCorrect = false;

}

else

{

isCorrect = false;

}

}

else

{

isCorrect = false;

}

fin.close();

if (!isCorrect)

{

records.SetDefaultRecords();

}

}

По завершению уровня или смерти игрока сравниваются результаты игрока на данном уровне с предыдущими результатами на данном и, если новый результат оказался лучше, то происходит запись в таблицу рекордов. Из двух результатов лучшим оказывается тот, у которого больше значение убитых противников, или, если данные значения равны, тот, у которого меньше значение времени. Код, осуществляющий сранение:

Record tempRec = records.GetRecord(levels.GetLevelNum());

if (KillsCounter > tempRec.kills ||

(KillsCounter == tempRec.kills && TotalTime / 1000 < tempRec.time))

{

tempRec.kills = KillsCounter;

tempRec.time = TotalTime / 1000;

records.SetRecord(levels.GetLevelNum(), tempRec);

}

SaveRecords();

При записи рекордов в файл также осуществляется вызов метода GetHash объекта класса EDS, который генерирует значение подписи в зависимости от новых результатов. Блок-схема функции, выполняющей сохранение, приведена ниже:



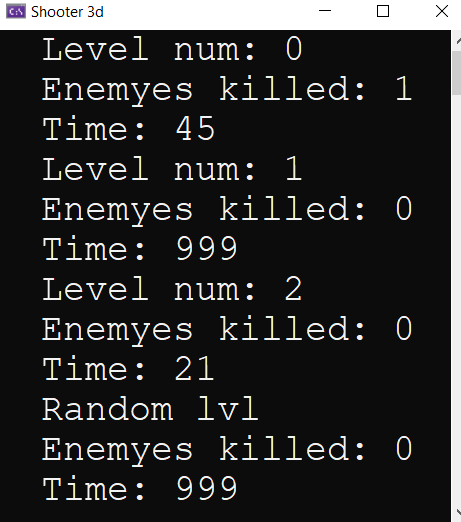


Рисунок 2.2 – Таблица рекордов

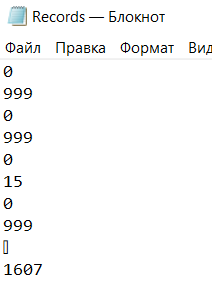


Рисунок 2.3 – Подписанный файл с таблицей рекордов

3 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

Тестирование программного обеспечения является неотъемлемой частью разработки любого проекта, которое напрямую влияет на его качество. В ходе тестирования проверяется соответствие между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов.

В ходе тестирования приложения были выявлены следующие недостатки данного программного средства.

Была обнаружена проблема при прорисовке противников, стоящих друг за другом. А именно, противник, который находиться дальше от игрока, может прорисовывается поверх противника, стоящего ближе к игроку. Для устранения данной проблемы был создан так называемый буфер глубины, хранящий расстояние до ближайшего объекта или стены для каждого столбца матрицы символов screen. Использование данного буфера также обеспечивает то, что противники и другие объекты не будут прорисовываться через стену. Ниже приведен код, описывающий принцип работы буфера глубины.

DepthBuf[x] = fWallDistance;

char Shade = enemy->texture[textureY][textureX];

if (Shade != EMPTY && DepthBuf[x] > fEnemyDistance)

{

screen[y \* ScreenWidth + x] = Shade;

DepthBuf[x] = fEnemyDistance;

}

Также были проблемы, связанные с удалением противников. Как было указано в разделе 2.5, для обеспечения функционирования врагов был создан список указателей на объекты классов, базовым для которых является Enemy. И при удалении из списка врагов, освобождалась память, которую занимает только указатель на объект, но не сам объект. Ниже приведён код, решающий данную проблему:

enemyes.remove\_if([](Enemy\* en) {

if (en->getDestroy())

{

delete(en);

return true;

}

return false; });