Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Факультет компьютерных систем и сетей
Кафедра информатики

Курсовой проект по дисциплине:
«Программирование»

Пояснительная записка к курсовой работе

Тема работы:
«Игра Galaxian»

Исполнитель
студент гр. 653501 _____ Севкович К.А.
(подпись дата)

Руководитель
_____ Козуб В.Н.
(подпись дата)

_____
(оценка)

Минск
2017 год

Содержание

1. Введение

1.1. Выбор темы

Неужели есть хоть кто-то из парней (а может даже и из девушек), чье, скажем так, «знакомство» с компьютерным миром началось не с игр? Возможно примитивных по своей механике, но всё же игр.

Сначала вся игровая механика воспринималось как что-то естественное: если какое-то событие происходит в реальном мире определённым образом, то так же оно должно происходить и в игре, и не может быть никак иначе. Но в какой-то момент пришло понимание, что компьютеру просто так не скажешь: «Сделай как в жизни». Мне стало интересно, как же все эти игры работают внутри? Как игра понимает, когда ты попал во врага, где находится яма, в которую персонаж обязательно упадёт и так далее.

Получив за два семестра знания в области программировании, достаточные для написания простой двумерной игры, я задался целью написать свой клон какой-нибудь известной и не сложной по своей механике игре. Выбор был огромный: Pacman, Space Invaders, Galaxian и много других. Но наиболее интересной темой для курсовой работы мне показалась именно игра Galaxian, в которой противники имели хоть и небольшую, но всё же индивидуальность поведения. Игра не обладает сложной механикой, но в ней есть свои интересные задачи, которые необходимо решить человеку, решившему самому с нуля написать её свою версию.

1.2. Выбор средств разработки

Для создания игры было решено использовать объектно-ориентированный язык программирования (ООЯП) C#, так как методология объектно-ориентированного программирования идеально подходит как для данной игры, так и для игр вообще. Важным преимуществом перед другим ООЯП C++ стало наличие в C# автоматического сборщика мусора, что избавляет разработчика от необходимости самому следить за расходом памяти. Крупные игры как раз для оптимизации потребления ресурсов принято писать на C++, но для такой небольшой игры уменьшение количества необходимых ресурсов не приведёт к какому-либо ощутимому приросту производительности.

В качестве интегрированной среды разработки (ИСР) был выбран уже знакомый по работе в семестре Visual Studio 2017.

Последним важным выбором на этапе планирования стал выбор игрового движка (ИД). Поскольку разработку решено было проводить на C#, то выбор стоял между Unity и Monogame. Оба ИД известны, на обоих было создано множество прекрасных игр (например, Fez и Bastion на Monogame и Hearthstone и Ori and the Blind Forest на Unity). Прочитав обсуждение на нескольких форумах, я пришёл к выводу, что Monogame лучше подходит для работы с ним людей, ранее не знакомых с программированием игр. Сделав выбор в пользу Monogame, я приступил к прочтению обучающих материалов по этому ИД.

2. Разработка программы

2.1. Методы программы

Итак, мы создали проект Monogame, в Visual Studio. Будет создана заготовка проекта по умолчанию, в которой будет создан класс с названием вашего проекта, который наследуется от класса `Game`. Просмотрев код этого класса, в нём увидим следующие пять методов, минимум 4 из которых предстоит реализовать. Это переопределённые методы класса Game: `Initialize()`,`LoadContent()`, `UnloadContent()`, `Update(GameTime gameTime)` и `Draw(GameTime gameTime)`. Все они имеют тип возвращаемого значения void. Разберём, для чего нужны все эти методы, кроме метода `UnloadContent()`, который нам не нужен, из-за отсутствия необходимости следить за производительностью и убирать ненужные материалы из памяти вручную.

Метод `Initialize()`

Этот метод позволяет до запуска игры настроить всё, что не связанно с непосредственно содержимым игры (именно настроить, а не загрузить их извне). Запускается один раз самым первым из методов.

Метод `LoadContent()`

Этот метод позволяет загрузить всё необходимое содержимое игры из вне: текстуры, звуки и шрифты. Запускается один раз после метода `Initialize()`.

По умолчанию в Monogame предусмотрен *конвейер содержимого*, позволяющий загружать все данные, которые разработчик посчитает нужными. Однако я узнал о нестабильности работы этой программы, поэтому для загрузки был создан и описан отдельный статический класс `ContentManager`, но об этом будет написано позже.

Метод `Update(GameTime gameTime)`

Этот метод позволяет игре обновлять своё состояние, производя расчёты положения объектов на экране, реагировать на нажатия клавиш на клавиатуре и осуществлять смену спрайтов на экране для создания анимации. По умолчанию метод вызывается 60 раз в секунду. Объект `gameTime` содержит информацию о времени игры, ног нам понадобится только следующая информация:

время которое запущена игра, и время, прошедшее с последнего вызова метода `Update` или `Draw` (в зависимости от того, в каком методе вызывается). Этот объект нужен для работы с временными задержками при анимации и при нажатии клавиш.

Метод `Draw(GameTime gameTime)`

Этот метод позволяет выводить спрайты на экран. По умолчанию он вызывается так же 60 раз в секунду после метода `Update()`, и это значит, что значение кадров в секунду так же равно 60, чего более чем хватает для комфортной игры. Поскольку игра не требует много ресурсов, значение кадров в секунду не проседает. Поэтому менять это значение смысла не имеет.

В итоге мы имеем *игровую петлю* такого вида:

2.2. Структура проекта

Всего в рабочем проекте курсовой работы присутствует более 50 различных файлов игры, в зависимости от их предназначения распределённых по папкам. Каждый проект содержит свой отдельный класс или перечисление.

Все текстуры были взяты из открытых источников в интернете и выглядят так же, как текстуры из оригинальной игры, а звуки были сгенерированы в программе «sfxr», созданной для применения на хакатонах, когда нет времени искать в интернете подходящие звуки. Программа содержит множество параметров и позволяет случайным образов генерировать звук, исходя из того, что вам надо: звук взрыва, лазерного выстрела, подбирания монетки и так далее.

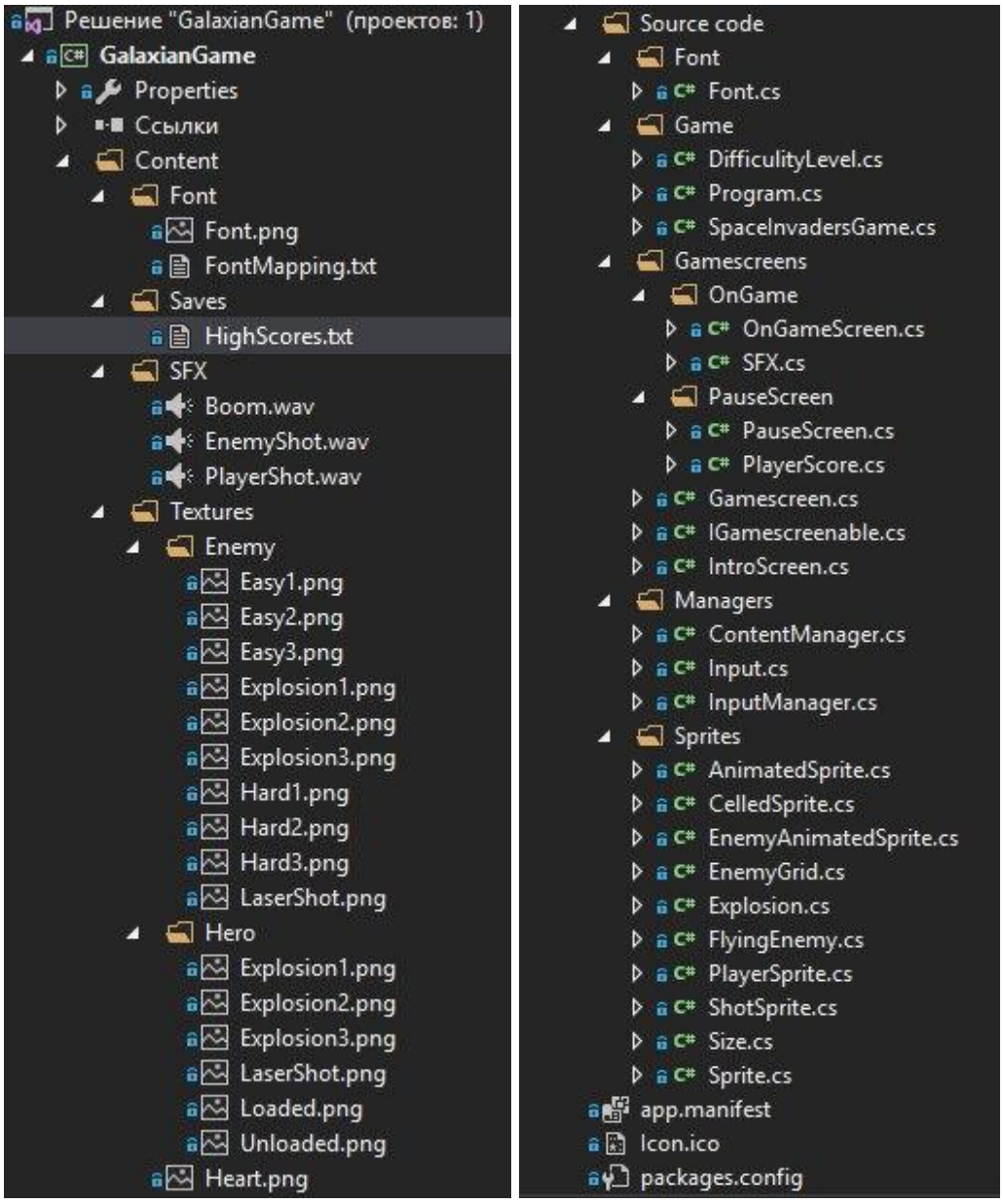

Рис. 1 и 2 «Обозреватель решений»

2.3. Класс `ContentManager`

Из-за нестабильной работы встроенного *конвейера содержимого*, пришлось искать другой способ загрузки необходимых материалов. Решением стало создание статического класса `ContentManager`, в котором определены функции `GetTexture2D` (загрузка текстур), `GetSoundEffect` (загрузка звуков) и `GetFontMapping` (загрузка маппинга шрифта).

Подробно работа со шрифтом будет описана ниже.

2.4. Работа со спрайтами

Также для отображения объектов игры на экране обязательно нужен объект типа `SpriteBatch` – главный объект для рисования объектов в 2D. Объект этого типа содержит методы для рисования *спрайтов* на экране.

Спрайт - растровый объект, который можно отобразить на экране.

В шаблоне классе игры уже есть объект класса `SpriteBatch`. Но нам нужен спрайт. Для этого создаём класс `Sprite`, хранящий информацию о текстуре спрайта, его позиции на экране и его размеры. Последнее нам пригодится при обработке столкновений объектов.

В самом классе `Sprite` создали метод `Draw`, принимающий в качестве параметров объект класса `SpriteBatch` и дробное значение, позволяющее увеличивать или уменьшать изображение. В результате своей работы метод `Draw` выводит спрайт на экран.

2.5. Работа со шрифтами

Для работы со шрифтами использовалась следующая схема:

1. Был создан класс `Font` и его объект для работы со шрифтами
2. Был создан класс `CelledSprite`, для работы с текстурой, состоящей из множества текстур.
3. Загружалась текстура шрифта типа `CelledSprite`, состоящая из всех букв латинского алфавита, цифр и некоторых знаков препинания.
4. Символы в таблице нумеруются от 0 до (количество символов – 1).
5. Вызывая функцию рисования, определённую в `Font`, для каждого символа строки мы получаем соответствующий номер в таблице.
6. По номеру в таблице мы получаем текстуру символа, и выводим её на экран при помощи объекта класса `SpriteBatch`.

2.6. Игрок, враги и выстрелы

Для описания всех трёх объектов, упомянутых в заголовке подраздела, были созданы отдельные классы:

1. Класс `PlayerSprite`, наследник класса `Sprite`. Предназначен для описания и управления спрайтом-кораблём главного героя.
2. Класс `ShotSprite`, наследник класса `Sprite`. Предназначен для описания спрайта выстрела.
3. Класс `AnimatedSprite` для работы со спрайтами, в которых есть анимация. В отличии от класса `Sprite`, содержит не одну-единственную текстуру, а массив текстур.
   Для описания врагов был создан класс `EnemyAnimatedSprite`, наследник `AnimatedSprite`.
   Для упрощения передвижения всех находящихся на экране врагов и упрощения проверки все ли враги мертвы, был создан класс `EnemyGrid`, тоже наследник `AnimatedSprite`.
4. Класс `FlyingEnemy`, позволяющий описывать поведение врага, пикирующего как камикадзе на главного героя. Если он вас достанет, то лишит одной жизни. Дойдя же до нижней границы экрана он телепортируется на своё исконное место.

2.7. Взаимодействие объектов

В классах `Sprite` и `AnimatedSprite` существует поле имеющее тип `Rectangle` и позволяющее хранить в нём координаты и размеры места, которое объект занимает на экране, а именно координаты верхнего левого угла, а так же ширину и высоту объекта.
В игре проверяются столкновения трёх типов: столкновение противников с выстрелом игрока, столкновение игрока с выстрелом противника и столкновение игрока с противником. Для удобства проверок создаётся список, состоящий из пар выстрел-стрелок. При выстреле туда заносятся объекты типа `ShotSprite` и `EnemyAnimatedSprite` (последний пригодится при определении координат выстрела). При проверке на столкновение проверяются абсолютно все существующие объекты обоих типов списка.

К сожалению, такая схема не позволяет с абсолютной точностью установить, столкнулся один объект с другим или нет, так как текстуры объектов (за имключением объектов `ShotSprite`) не являются

идеальными прямоугольниками. Но для данной игры этот факт не важен, и никаких проблем из-за этого замечено не было.

2.8. Игровые экраны

Всего в игре есть три игровых экрана: *вводный экран*, *экран игры* и *экран паузы*.

На первом экране игрок может ознакомиться с управлением, а так же выбрать уровень сложности (на лёгком уровне враги имеют одну жизнь, а персонаж имеет две, на сложном же уровне у персонажа всего одна жизнь, а то время как у врагов – две):

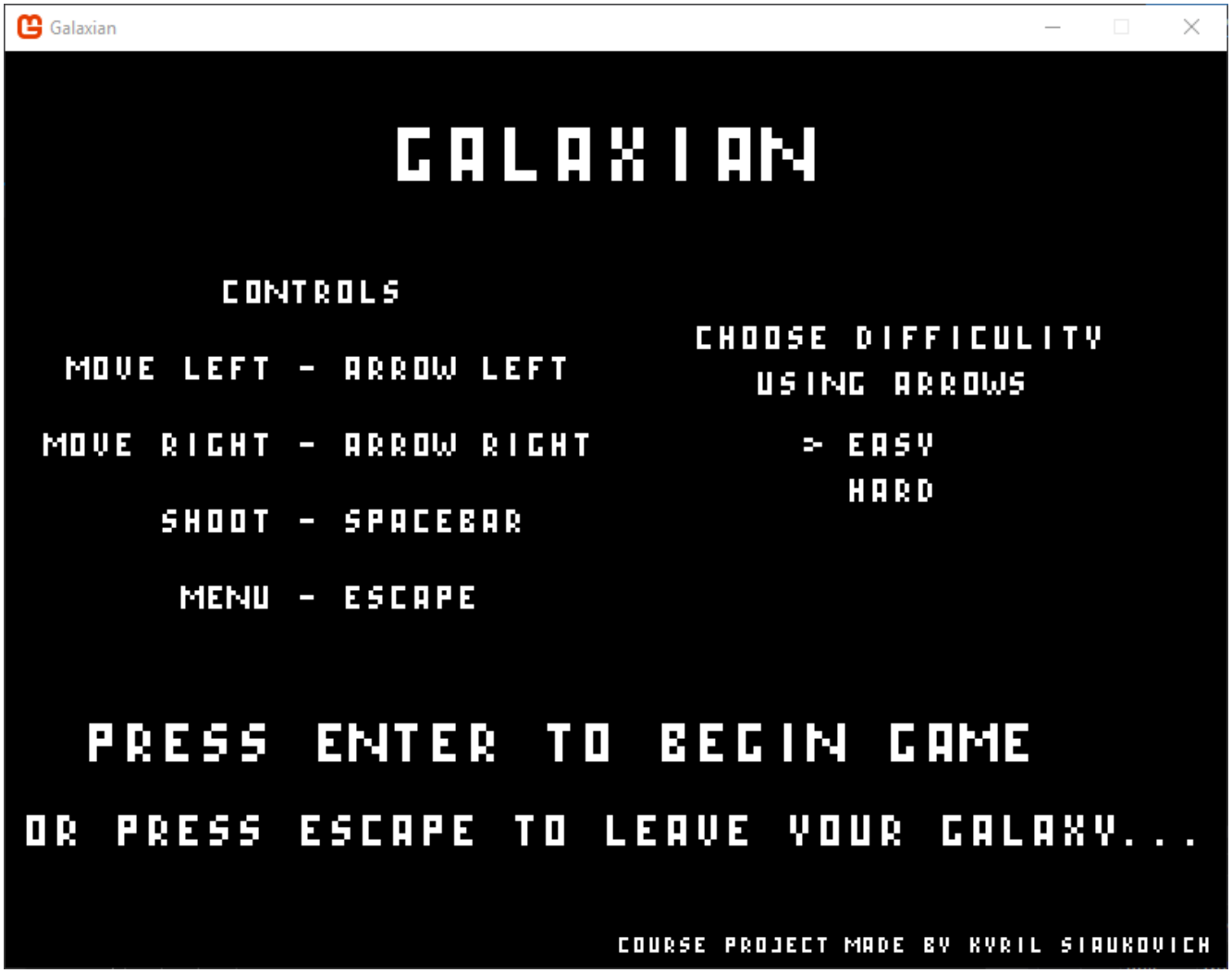

Рис. 3 Вводный экран

Второй экран представляет собой саму игру. Игрок отстреливается, старается не умирать, зарабатывает (и, возможно, теряет) очки, совершая различные действия:

Схема подсчёта очков:

1. По 10 за каждого убитого врага на лёгком уровне сложности и по 20 за каждого врага на высоком уровне сложности.
2. Игрок теряет 30 очков, если его убивают.
3. Игрок теряет 5 очков, если его выстрел не попадает ни в одного врага и вылетает за пределы экрана.
4. После победы количество очков увеличивается на значение оставшегося времени (поле TIME внизу экрана, в начале игры установлено на 200 и отсчёт идёт до нуля)

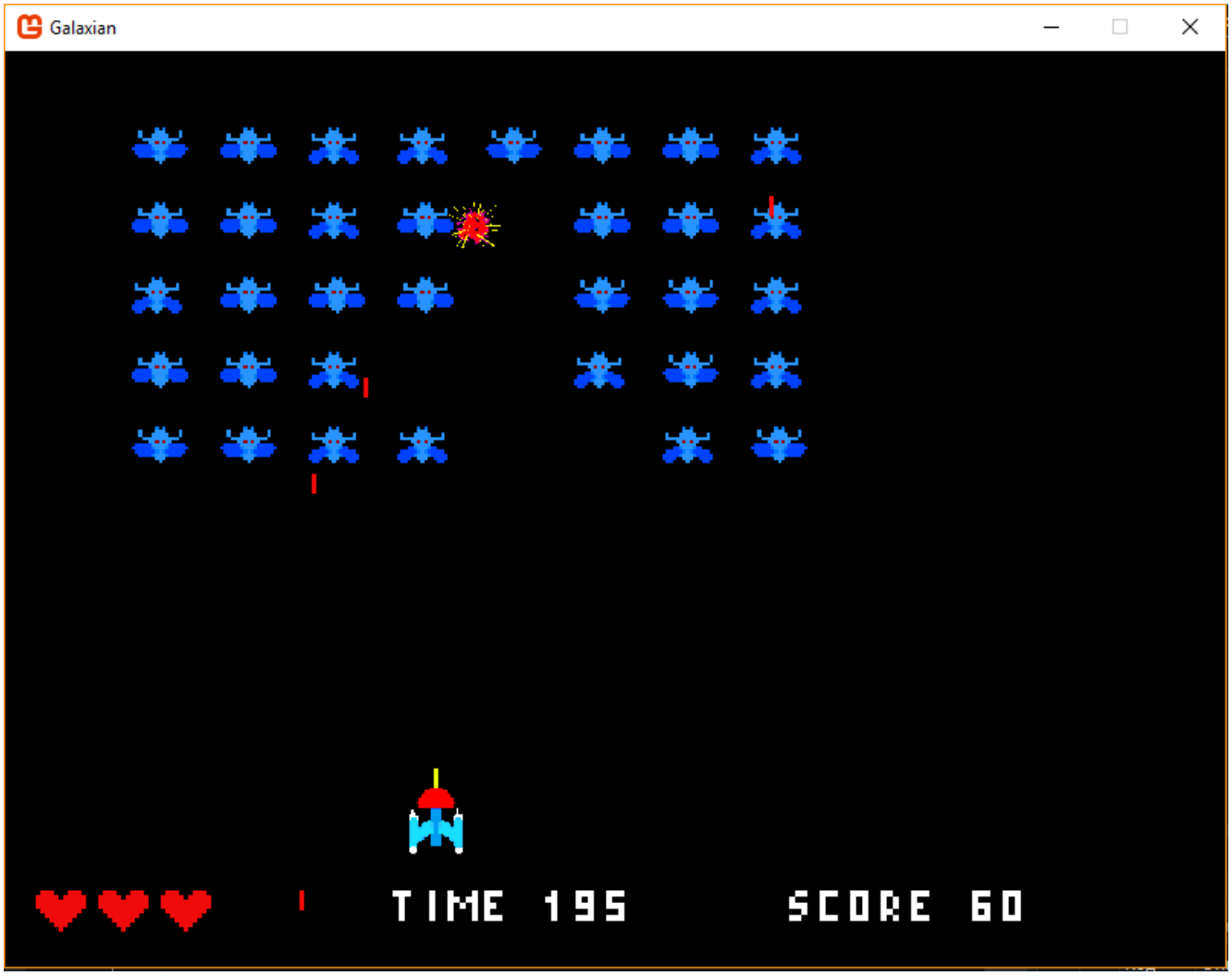

Рис. 4 «Экран игры»

После успешного прохождения уровня (то есть убийства всех врагов) изображение на экране меняется на значение очков и уведомление, если произошло обновление максимального рекорда.

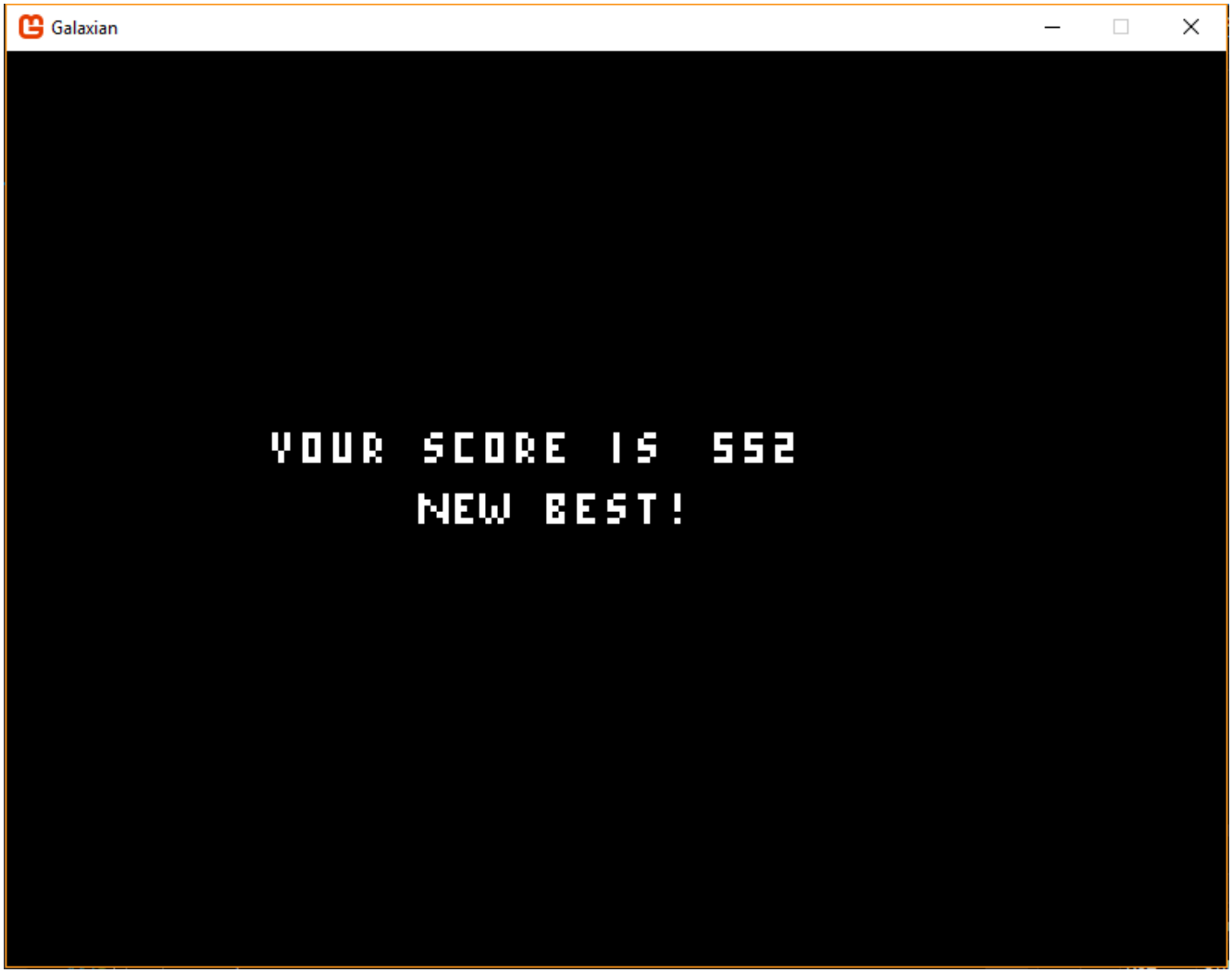

Рис. 5 «Экран игры с результатами»

На третьем экране игрок может увидеть список лучших результатов и выбрать, продолжить ли ему игру или выйти из неё.

Третий экран вызывается только из второго нажатием клавишу `Escape`. Вернуться из меню обратно в игру можно либо выбрав пункт BACK в меню, либо повторно нажав клавишу `Escape`.

Значения лучших результатов хранятся в отдельном файле `HighScores.txt`. Для удобства работы с результатами создан маленький класс `PlayerScore`, содержащий только два поля типа `string`: имя игрока и количество набранных им очков. Данные из файла считываются в список типа `List<PlayerScore>` (так удобнее, потому что нам заранее не

известно количество элементов там (известно только, что не больше трёх)).

Если игрок в меню выберет опцию QUIT, то игра сразу же закроется, без сохранения каких-либо результатов.

Рис. 6 «Экран паузы»

Все три экрана наследуют от интерфейса `IGamescreenable`, который обязывает эти классы определять в себе методы `LoadContent()`, `Update(GameTime pGameTime)` и `Draw(GameTime pGameTime)`.

Переключение между экранами происходит благодаря наличию булевых переменных, которые дают игре понять, когда нужно совершить переключение.

На вводном экране при нажатии клавиши `Enter` соответствующая переменная перейдёт в состояние `true`, а после этого будет загружаться экран игры. При нажатии же на `Escape` другая соответствующая будет иметь значение состояние `true` и приложение будет закрыто.

На экране игры по нажатию `Escape` нас переключит на экран паузы. Если умрут все враги – то покажется окно результатов. Если умрёт персонаж – игра закончится.

Принцип работы экрана паузы так же включает в себя булевы переменные, но его поведение уже было описано выше.

3. Заключение

Целью данной курсовой работы было создание игры-клона, в которой бы узнавался оригинал, а так же получение знаний того, как работают игры внутри.

Я считаю, что задача выполнена. Игра, кажется, получилась, а знания того, как работает механика игры – были получены. Но всё же видно, что это только первый мой проект такого рода, и есть места, где можно было сделать лучше. Например, можно было создать более сложную иерархию классов врагов и сделать их чуть более разнообразными, можно было улучшить иерархию классов спрайтов, улучшить дизайн игры. Где-то же мне просто не хватило уровня знаний языка и ИД.

В любом случае, разработка игры оказалась очень интересным занятием. Вряд ли я когда-нибудь всерьёз займусь этим, но как хобби, почему бы и нет?

4. Список использованных источников

1. Эндрю Троелсен «Язык программирования C# 6.0 и платформа .NET 4.6»
2. http://www.gamefromscratch.com/
3. https://metanit.com/sharp/monogame/
4. https://msdn.microsoft.com
5. https://stackoverflow.com/