**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ**

**Кафедра веб-технологий и компьютерного моделирования**

**СОЗДАНИЕ 2D-ВИДЕОИГРЫ С ПОМОЩЬЮ ИГРОВОГО ДВИЖКА UNITY**

Курсовая работа

Логвиненко Антона Викторовича

студента 2 курса, специальность 1-31 03 08-01 Математика и информационные технологии.

Научный руководитель:  
кандидат физ.-мат. наук,  
доцент А. И. Кравчук

Минск, 2023

**Оглавление**

[Введение 3](#_Toc134632466)

[Глава 1 Обзор предметной области, Выбор технологий разработки и сравнительный Анализ игр-аналогов 4](#_Toc134632467)

[1.1 Предметная область: видеоигровой жанр «платформер» и его поджанр «бесконечный раннер» 4](#_Toc134632468)

[1.2 Сравнительный анализ игр-аналогов 4](#_Toc134632469)

[1.3 Выбор платформы для разработки 2D-игры 6](#_Toc134632470)

[1.4 Обзор платформы для создания видеоигр Unity 8](#_Toc134632471)

[1.4.1 Интерфейс редактора Unity Editor 8](#_Toc134632472)

[1.4.2 Базовые компоненты объектов и сценарии. Торговая площадка Unity Asset Store 11](#_Toc134632474)

[1.4.3 Анимации и контроллеры анимаций 13](#_Toc134632475)

[Глава 2 Проектирование и программная реализация видеоигры 15](#_Toc134632476)

[2.1 Концепция игры 15](#_Toc134632477)

[2.2 Программная реализация действий игрового персонажа 17](#_Toc134632478)

[2.3 Создание игрового уровня 20](#_Toc134632479)

[2.4 Система здоровья и неуязвимости 21](#_Toc134632480)

[2.5 Реализация рукопашного боя игрового персонажа 23](#_Toc134632481)

[2.6 Программная реализация препятствий и противников 24](#_Toc134632491)

[2.7 Система создания игровых объектов 26](#_Toc134632492)

[2.8 Реализация объектов-бонусов 28](#_Toc134632493)

[2.9 Проектирование пользовательского интерфейса 29](#_Toc134632494)

[2.10 Выбор компонентов игры из Unity Asset Store. Создание оригинальной анимации 32](#_Toc134632495)

[Заключение 33](#_Toc134632496)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 34](#_Toc134632497)

[Приложения 35](#_Toc134632498)

Введение

Видеоигры – это не только популярное развлечение для детей и взрослых, но и масштабная современная индустрия. Согласно журналу Forbes, ее прибыль исчисляется миллиардами долларов, а количество игроков по всему миру составляет более 2.5 миллиардов [1]. Разрабатывают и продают видеоигры как крупнейшие студии, так и независимые разработчики. Технологии разработки, такие как Unity и Unreal Engine, доступны бесплатно для обучения и коммерческого использования. Распространение собственных видеоигр на крупных торговых площадках, например, Steam или Epic Games Store, доступно любым разработчикам без участия издателей.

Видеоигровая индустрия быстро развивается и является чрезвычайно востребованной, что обуславливает ценность знаний о технологиях разработки видеоигр и обеспечивает актуальность работы.

Существует большое количество жанров видеоигр, однако одними из старейших и простых в реализации являются «игры-платформеры», в частности, особую популярность получил поджанр «бесконечный раннер»(или «бесконечный бег»).

Целью курсового проекта является изучение технологий для разработки компьютерных игр и создание «игры-раннера» с их помощью.

Для достижения указанной цели были сформулированы следующие задачи:

* Провести сравнительный анализ компьютерных игр жанра «бесконечный раннер».
* Изучить платформы для разработки компьютерных игр Unreal Engine и Unity, провести сравнение и выбрать подходящую для разработки компьютерной 2D-игры.
* Спроектировать правила игры.
* Реализовать спроектированные правила игры с помощью выбранных технологий разработки.
* Разработать пользовательский интерфейс игры, оформить визуальное представление и анимации с использованием готовых компонентов и собственных.

Глава 1 Обзор предметной области, Выбор технологий разработки и сравнительный Анализ игр-аналогов

1. Предметная область: видеоигровой жанр «платформер» и его поджанр «бесконечный раннер»

Предметной областью курсового проекта является 2D-игра жанра «платформер», поджанра «бесконечный раннер».

Жанры видеоигр — это категории, на которые они подразделяются в зависимости от основных игровых механик. Одними из самых старейших и простых в реализации являются игры жанра «платформер». В играх этого жанра основной игровой процесс составляет преодоление различных препятствий, например, прыжки по платформам, откуда и берет название жанр. Целью является либо сбор очков, либо прохождение определенного набора уровней, либо победа над противниками. Среди поджанров «платформера» выделяется «бесконечный раннер» (от англ. «runner» – «бегун»).

Отличительными особенностями «игр-раннеров» являются бесконечные уровни и случайно генерируемые препятствия. Сбор очков является главной целью игры – получить их можно пробежав наибольшее расстояние или преодолевая препятствия. Сложность игры со временем возрастает, и после того, как персонаж теряет все очки жизни, она заканчивается.

1. Сравнительный анализ игр-аналогов

Рассмотрим игры-аналоги: браузерная игра Dinosaur T-Rex Game и мобильная игра Subway Surfers.

Dinosaur T-Rex Game – видеоигра, встроенная в браузер Google Chrome. Она «спрятана» в браузере на экране ошибки при отсутствии соединения с интернетом. Игрок берет на себя управление бегущим динозавром. С помощью клавиш Вверх и Вниз, он может перепрыгивать низкие препятствия («кактусы») или уклоняться от препятствий в воздухе («птеродактили»). На экране отображается счетчик очков, которые увеличиваются на протяжении игровой сессии до поражения. Со временем скорость динозавра и частота появления препятствий увеличивается, что делает игру сложнее. Игра заканчивается, если игрок соприкасается с любым из препятствий или игровое время достигает 17 миллионов лет.

Достоинствами данной игры являются простое управление, возрастающая сложность и простая система очков, позволяющая игроку сохранять интерес к игре на продолжительное время с целью увеличения своего лучшего результата, соревнуясь с другими людьми. Среди недостатков можно отметить примитивный визуальный стиль, малое разнообразие среди видов препятствий.

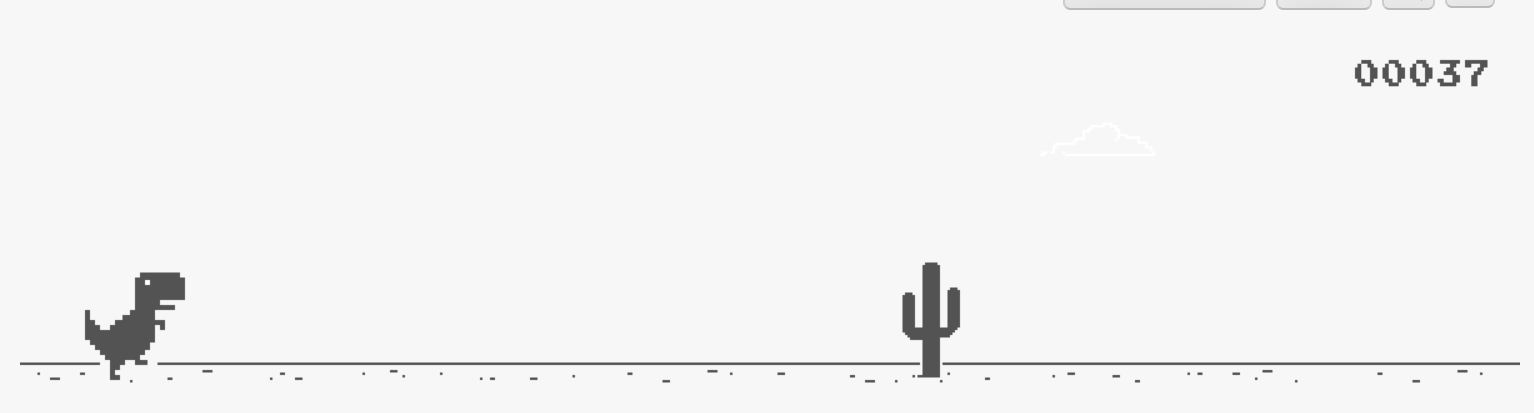
На изображении ниже (Рисунок 1.1) представлен экран игры.

Рисунок 1.1 – Экран браузерной игры Dinosaur T-Rex Game

В качестве следующего аналога рассмотрим 3D-игру Subway Surfers. По правилам игры, главному персонажу необходимо убегать от охранника по железнодорожным путям, избегая столкновений с поездами и дорожными ограждениями. Игрок может направлять персонажа по одному из трех железнодорожных путей. Задача игрока: вовремя менять путь, а также использовать прыжки и кувырки, чтобы уклоняться от препятствий. Отличительной особенностью правил игры Subway Surfers является система бонусов, которые игрок может получать в процессе игры. Чтобы получить бонус игроку необходимо направлять персонажа на определенные пути. Это усложняет игру, но в качестве награды игрок получает помощь в преодолении препятствий или дополнительные очки для улучшения результата. В качестве примеров можно привести: бонус «Ракетный Ранец». Он позволяет игроку на некоторое время подняться в воздух, где столкновение с объектами ему не грозит и он может получить увеличенное количество очков; бонус «Пружинные Сапоги», который предоставляет возможность перепрыгивать через высокие препятствия.

Среди преимуществ игры Subway Surfers: привлекательный визуальный стиль, разнообразие препятствий и способов их преодоления и система бонусов, поощряющая более успешных игроков. Главным недостатком игры можно назвать одну из особенностей системы бонусов: невозможно выбрать в какой момент применить определенный предмет. Поэтому в большинстве случаев игрок не получает выгоды от их использования.

На рисунке 1.2 представлен пример экрана игры Subway Surfers.

Рисунок 1.2 – Основной игровой экран мобильной игры Subway Surfers

При создании игры будут учтены достоинства и недостатки рассмотренных игр-аналогов. Из обеих игр будут позаимствованы возрастающая сложность, ведение счета очков и простота управления. Из игры Subway Surfers – система бонусов. При этом будут учтены упомянутые недочеты: однообразный визуальный стиль, отсутствие гибкой системы применения бонусов.

1. Выбор платформы для разработки 2D-игры

При создании видеоигр используются так называемые игровые движки. Игровой движок – базовое программное обеспечение компьютерной игры, пригодное для повторного использования и расширения. Они содержат общие технологии, необходимые при создании каждой игры. Благодаря движкам разработчику не нужно заново реализовывать для каждой игры базовые системы, например расчет поведения физических тел и их отображение. Самыми популярными платформами для создания игр являются Unity и Unreal Engine.

Unity — кроссплатформенная среда разработки компьютерных игр, разработанная компанией Unity Technologies. Она снабжена движками для расчета физики объектов как в 3D, так и в 2D проектах. Для создания уровней Unity Editor использует сцены – файлы, содержащие информацию об игровых объектах внутри игровой среды. Они могут содержать как объекты, обладающие визуальным представлением в мире игры, так и объекты без отображения. Каждый из них является совокупностью компонентов, с помощью которых происходит взаимодействие скриптов (сценариев) с ним. Для того, чтобы, например, применить силу гравитации на объект, необходимо добавить компонент RigidBody2D.

Для написания скриптов используется язык программирования C#, который обладает С-подобным синтаксисом и наиболее близок к языкам программирования С++ и Java. Он является статически типизированным, реализует механизмы и принципы ООП, при этом избавившись от проблематичных моделей своих предшественников, например, множественного наследования в языке С++.

Ключевыми возможностями Unity являются наличие визуальной среды разработки Unity Editor, что упрощает прототипирование, разработку и тестирование игры; наличие большого количества материалов для обучения; простота интерфейса редактора и магазин Unity Asset Store, где разработчики могут продавать различные компоненты игр, например, модели, аудиофайлы, библиотеки и другое.

Версия Personal платформы Unity является бесплатным для разработчиков, которые зарабатывают меньше 100 тысяч долларов за год.

Среди недостатков можно отметить ограниченность визуального редактора при работе со сложной иерархией объектов, из-за чего затрудняется взаимодействие со сценой; сложная настройка работы с внешними библиотеками.

Другой программной средой разработки игр является Unreal Engine от компании Epic Games. Благодаря тому, что программный код пишется на языке C++, игры, сделанные на Unreal Engine, работают быстрее и эффективнее. Он предоставляет широкие графические возможности, гибкие настройки звука и поведения искусственного интеллекта. Среди других достоинств платформы есть специализированный визуальный язык программирования Blueprints, который упрощает прототипирование и создание простейших сценариев игры. Недостатками Unreal Engine является отсутствие официальной поддержки средств для создания 2D-игр, а также высокие системные требования. Unreal Engine не подходит для разработки двумерных игр, поэтому выбор был сделан в пользу платформы Unity.

Unity не ограничивает в выборе среды разработки для языка программирования C#, однако наиболее популярной является Visual Studio от Microsoft. При создании данного проекта использовалась Visual Studio Community 2022 с установленными инструментами для Unity.

1. Обзор платформы для создания видеоигр Unity
2. Интерфейс редактора Unity Editor

Новый проект Unity содержит сцену, где находится единственный объект – камера. Камера служит для задания области игрового мира, которая будет видна на экране монитора.

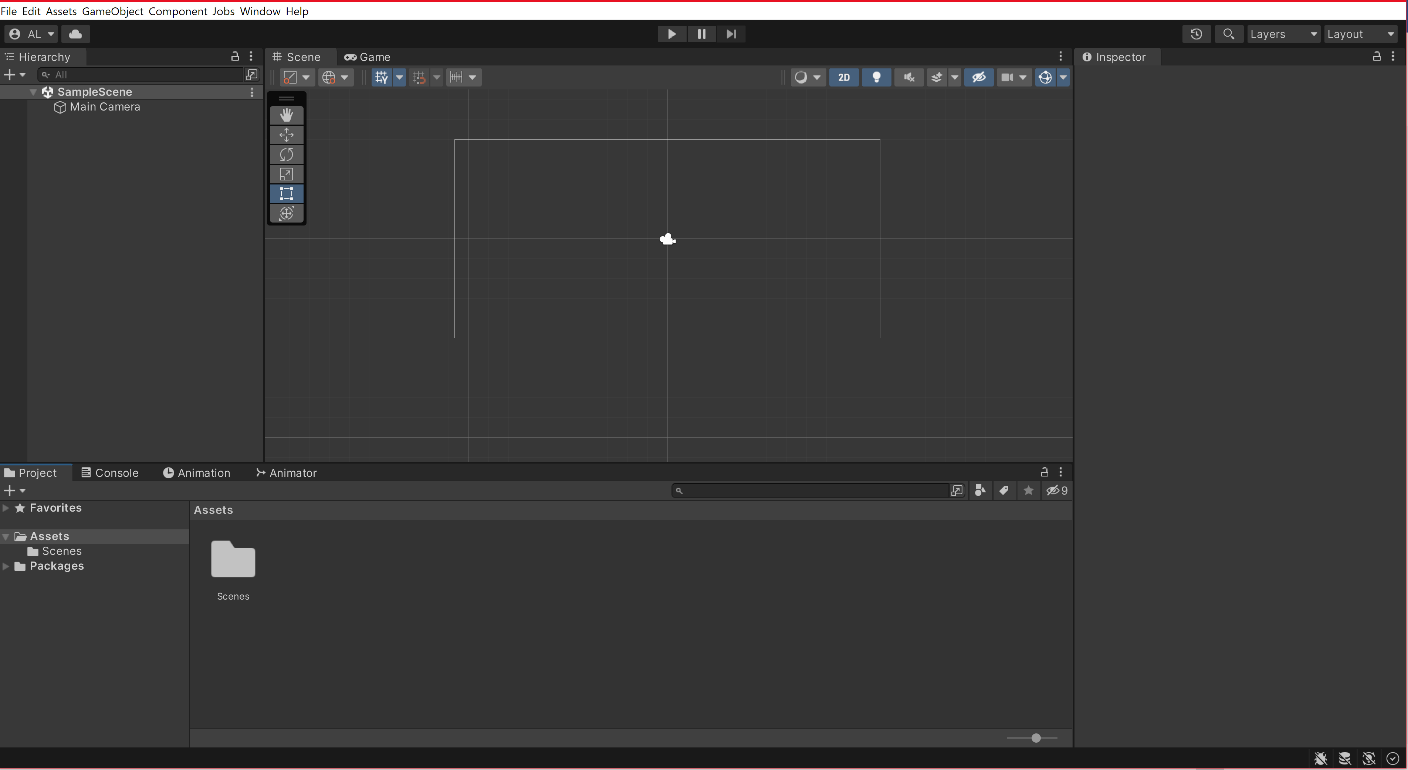
На рисунке 1.3 представлено окно Unity Editor, как оно выглядит после создания нового проекта.

Рисунок 1.3 – Интерфейс редактора Unity

Рассмотрим представленный интерфейс редактора подробнее. Базовый интерфейс состоит из панелей Hierarchy, Inspector, Console, Scene, Game и Project. Панели можно перемещать и менять местами для удобства пользования, так что у разных пользователей интерфейс может отличаться. На рисунке выше представлен интерфейс по умолчанию.

Панели также можно скрывать, если в них нет необходимости. Отобразить скрытые панели можно с помощью меню Window.

В панели Hierarchy находится иерархия объектов внутри сцены, здесь же можно создавать, выбирать и удалять объекты сцены. Объекты отображаются в виде древовидного меню.

Каждый объект в игре по умолчанию создается с присвоенным компонентом Transform. Этот компонент используется для хранения позиции, угла поворота, масштаба объекта внутри игрового мира. Каждый Transform может являться потомком другого компонента Transform. В таком случае его размеры и положение будут определяться относительно своего родителя. Зависимость одних компонентов от других отражается в древовидной структуре панели Hierarchy.

Чтобы получить информацию о компонентах объекта, необходимо выбрать нужный в окне иерархии – информация об объекте отобразиться в панели Inspector, где можно добавлять и удалять компоненты, изменять их свойства.

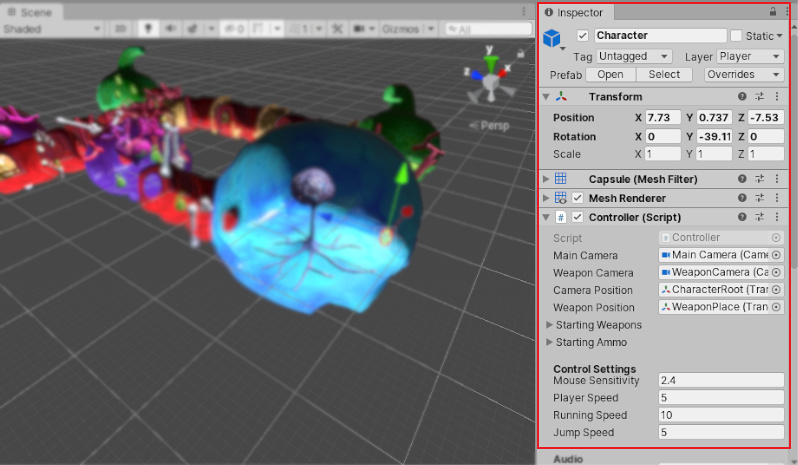
Кроме компонентов каждый игровой объект имеет тег и слой. Теги нужны для разбиения объектов на группы, присваивая им строку-идентификатор, что упрощает написание сценариев, если нужно применить какое-то действие на несколько сущностей. Слои, в свою очередь, нужны для определения взаимодействий объектов друг с другом внутри сцены, например, скрыть от камеры определенные объекты при отображении. На рисунке 1.4 представлен пример панели Inspector.

Рисунок 1.4 – Панель Inspector

В верхней части панели представлено имя объекта (на изображении – «Character»). В полях «Tag» и «Layer» – тег («Untagged») и слой («Layer»). Объекту на изображении выше присвоены компоненты «Transform», «Capsule (Mesh Filter)» и «Mesh Renderer».

С помощью панели Project можно взаимодействовать с файлами проекта и библиотеками. Панель представляет собой упрощенный файловый менеджер, и в работе аналогичен файловым менеджерам, встроенным в систему.

Панель Console используется для отображения ошибок, появляющихся при компиляции скриптов и во время игры, а также для вывода пользовательской информации во время выполнения скриптов.

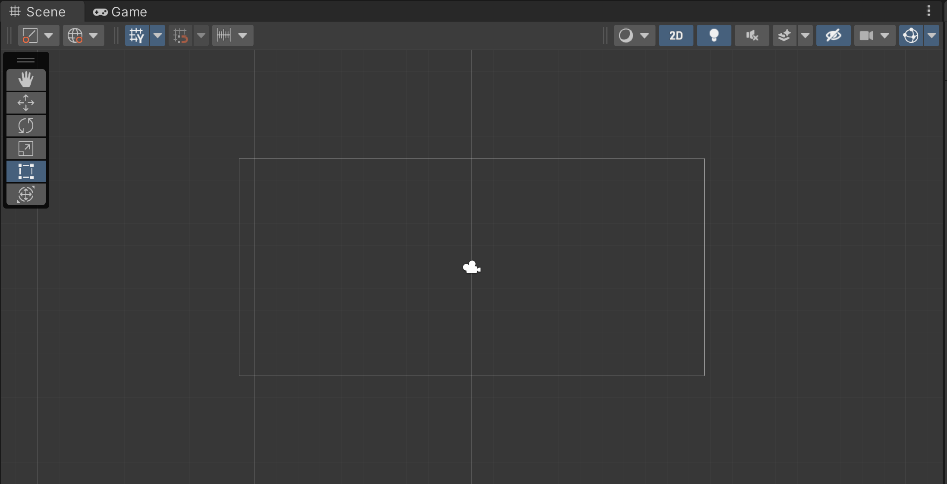
Отображение сцены со всеми объектами, находящимися в ней, расположено в панели Scene, где можно манипулировать объектами с помощью инструментов перемещения, поворота, масштабирования и другие. То, как игра будет выглядеть для конечного пользователя, находится в окне Game, где происходит сам игровой процесс при запуске. На рисунке 1.5 представлен внешний вид панели Scene нового проекта Unity.

Рисунок 1.5 – Окно сцены

На изображении выше в левой части панели расположены следующие инструменты: Move Tool – для изменения положения игровых объектов; Rotate Tool, предназначенный для изменения угла поворота объектов; Scale Tool, используемый при масштабировании; Transform Tool, объединяющий в себе функционал упомянутых выше инструментов; Rect Tool для манипуляции над элементами пользовательского интерфейса и View Tool, используемый для навигации внутри сцены;

1. Базовые компоненты объектов и сценарии. Торговая площадка Unity Asset Store

Редактор Unity дает возможность переключаться на разные формы представления игры: 2D и 3D.

В 2D режиме игра представляется в виде плоских изображений (спрайтов). Для того, чтобы присвоить изображение определенному объекту внутри сцены, необходимо добавить компонент Sprite Renderer, и указать в поле Sprite нужную картинку.

Для расчета поведения объектов при столкновении используется компонент Collider2D. Он позволяет определить геометрическую форму объекта для регистрации столкновений, отличную от отображаемой формы. Дочерний компонент BoxCollider2D позволяет определить форму как прямоугольник. Внутри компонента есть свойства, позволяющие настроить поведение объекта: проинформировать скрипты про наличие столкновения, позволяя другой сущности пройти насквозь; просчитать поведение объектов с помощью физических формул, чтобы объекты отскочили друг от друга или остановились после удара.

RigidBody2D – компонент, контролирующий поведение объекта с помощью физического движка, то есть применяющий силы гравитации, вес, скорость, коэффициент трения.

Для определения пользовательского поведения объекта или для программирования событий внутри игры используются скрипты, которые добавляют к компонентам объектов. По умолчанию Unity Editor создает внутри скрипта класс, наследующийся от MonoBehaviour – базовый класс для всех сценариев Unity. Движок передает контроль скриптам поочередно, вызывая определенные функции внутри сценария. После выполнения функции контроль возвращается обратно к Unity. Такие функции известны как функции событий, так как они вызываются при реагировании платформы на некоторые события внутри игры.

Одной из таких функций является метод Update. Игра похожа на анимацию в том смысле, что новые положения и поведения игровых сущностей отображаются в виде кадров. Каждое изменение рассчитывается непосредственно перед новым кадром и для этой цели используется функция Update, которая вызывается каждый раз перед отображением кадра и расчетом анимаций.

Если необходимо вызвать код инициализации, то применяются методы Start и Awake, которые срабатывают перед первым вызовом Update и при создании объекта соответственно.

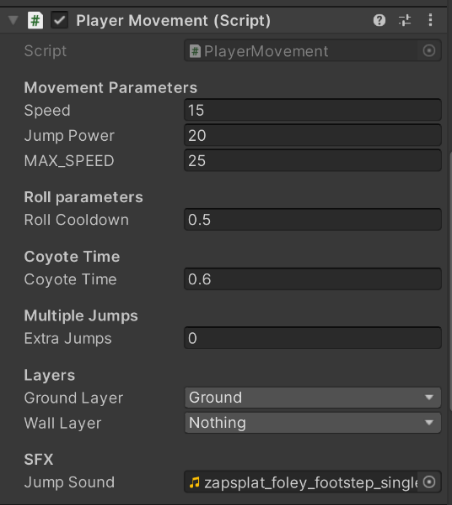
Для полей класса сценария могут задаваться модификаторы доступа: при указанном модификаторе public доступ к полу будет возможен из любого другого кода в той же сборке; при private – только из кода того же объекта или структуры. Для полей можно указывать аннотации, например, [SerializeField]. В таком случае поле называется сериализуемым. Эта аннотация, а также модификатор public позволяют редактировать значение поля напрямую из Unity Editor в компоненте соответствующего сценария, но при этом для сериализуемого поля класса можно настроить различные модификаторы доступа. Члены, доступные для редактирования в Unity Editor, отображаются в виде полей для ввода в панели Inspector у соответствующего компонента, как на рисунке 1.6.

Рисунок 1.6 – Компонент Script, с полями для редактирования

Для того, чтобы добавить аудио-сопровождение к игре, Unity предоставляет компонент AudioListener, который играет роль микрофона в 3D-пространстве. Этот компонент присваивается объекту, который будет записывать звуки вокруг себя и проигрывать их игроку через аудиоустройства компьютера. В сцене может быть только один слушатель и его обычно размещают на главной камере.

Для воспроизведения аудиодорожки на источнике звука должен находится компонент AudioSource. В поле AudioClip помещают файл звука; с помощью других свойств можно настроить громкость, устанавливать режим циклического воспроизведения и другое.

Unity Asset Store – это площадка, на которой разработчики делятся компонентами игр, такими как картинки, модели, аудиофайлы, анимации для использования в проектах Unity. Редактор позволяет импортировать купленные пакеты в проект, добавлять их в файловую систему и пользоваться ими внутри редактора.

1. Анимации и контроллеры анимаций

Анимация – представление движения статических изображений путем отображения последовательности кадров с высокой частотой. Unity Editor, помимо возможности импортировать записи анимаций из сторонних приложений, предоставляет разработчику встроенные инструменты для создания собственных анимаций. Для этого в редакторе существует Animator Controller (контроллер анимаций), использующий для управления переходами между ними механизм State Machine (машину состояний). Unity реализует одноименный паттерн объектно-ориентированного программирования, объединяющий достоинства централизации логики переходов и паттерна State (локализация кода, зависящего от состояния в отдельных классах) [2]. Главная идея State Machine заключается в том, что анимируемый объект находится в определенном состоянии в любой момент времени. Такими состояниями могут быть, например, положение покоя, бег, прыжок. Чтобы перейти от одного состояния в другое, необходимо указать условие перехода с помощью изменения значений переменных состояния. В совокупности наборы состояний, переходы между ними и переменные состояния образуют State Machine.

Чтобы анимировать игровой объект, необходимо добавить компонент Animator, а затем в поле Animator Controller поместить контроллер, который будет управлять анимациями данной сущности. Анимации в Unity могут быть применены для изменения положения, поворота, масштабирования, внешнего вида объекта и переменных скриптов, примененных к объекту.

Для настройки этих состояний и переходов между ними используется окно Animator. Здесь визуализирован граф всех связей внутри контроллера анимаций, а также условия переходов между разными записями, скорость перехода и другое.

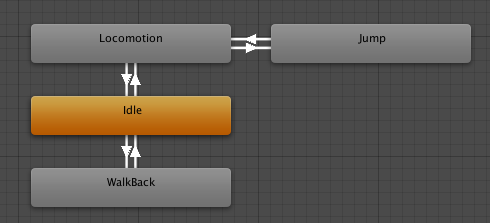
На рисунке 1.7 приведен пример связей внутри контроллера анимаций, визуализированных внутри панели Animator.

Рисунок 1.7 – Контроллер анимации

Чтобы добавить условие для перехода к новому состоянию, в окне Animator необходимо добавить новый параметр, имеющий один из следующих типов: Int, Float, Bool и Trigger. Параметры типа Int и Float, хранящие целые числа и числа с плавающей точкой соответственно, вызывают переход, только если значение параметра стало больше или меньше определенного значения. Условие типа Bool (принимает значения «правда-ложь») срабатывает при значении равном либо «true» («правда»), либо «false» («ложь»). Trigger позволяет сразу активировать переход при вызове функции SetTrigger внутри скрипта.

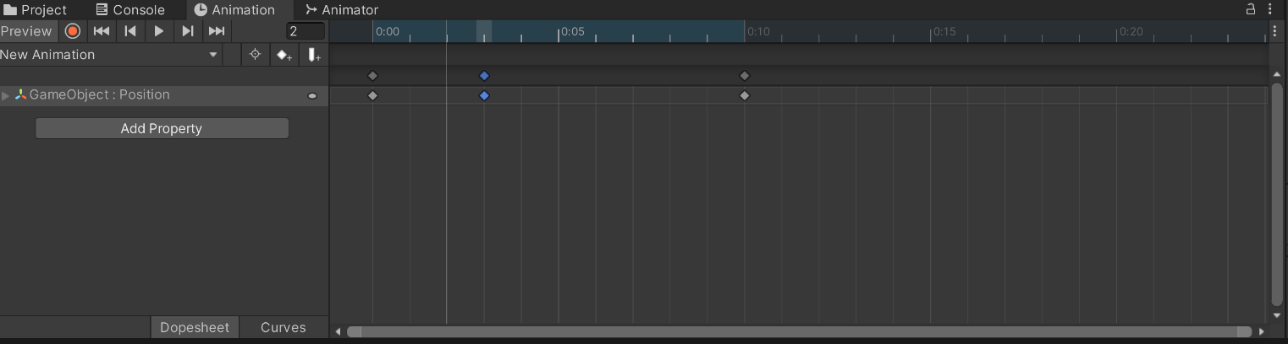
Для записи анимации внутри редактора Unity, используется окно Animation. При этом необходимо добавить только ключевые кадры анимации, остальные кадры будут рассчитаны автоматически. Кадры анимации располагаются на временной шкале, определяющей в какой момент времени будет отображено новое состояние, как это продемонстрировано на рисунке 1.8. Таким образом возможно контролировать продолжительность записей.

Рисунок 1.8 – Окно Animation

Unity предоставляет возможность вызывать функции в любой момент анимации, например, в анимации атаки персонажа – в момент удара мечом. Выделив мышкой место на временной шкале и нажав на кнопку «Add event», создается событие, которое будет вызывать указанные функции в момент воспроизведения анимации. Эти функции должны быть определены с модификатором public и должны принадлежать сценарию того же игрового объекта.

Согласно анализу технологий разработки видеоигр и аналогичных продуктов, описанному в этой главе, при создании игры будет использоваться платформа для разработки видеоигр Unity с использованием изображений, анимаций и аудиофайлов из магазина Unity Asset Store. В игре нужно будет преодолевать препятствия, появляющиеся случайно. Сложность игры будет возрастать со временем за счет увеличения скорости движения игрока и частоты появления преград. Во время игры будет происходить расчет очков и сохранение лучшего результата.

Глава 2 Проектирование и программная реализация видеоигры

1. Концепция игры

Подлежащая разработке видеоигра была названа «Running Knight».

«Running Knight» – это «игра-раннер» с двумерной графикой, в которой игроку нужно преодолевать препятствия и противников, набирая при этом очки. Персонаж игрока движется с увеличивающейся со временем скоростью, при этом на его пути встречаются ловушки и противники, которые при соприкосновении с игроком наносят ему урон – отнимают одну единицу жизни. Когда количество жизней иссякает, игра заканчивается, и игроку необходимо начинать сначала. Препятствия случайного типа появляются в случайный момент времени, что создает множество уникальных ситуаций, проверяющих мастерство и реакцию. Игрок соревнуется с самим собой на установление нового лучшего счета.

Задачей разработки является создание двумерной бесконечной «игры-раннера» c помощью инструментов платформы Unity и компонентов из Unity Asset Store, которая должна содержать три типа препятствий (две ловушки, одного противника), три бонуса, одну оригинальную анимацию.

Игра будет адаптирована под персональные компьютеры с горизонтальной ориентацией экрана. Минимальная поддерживаемая ширина экрана – 1920px, Максимальная – 3840px. Приложение будет разработано для операционной системы Windows 10.

В игре главный персонаж движется автоматически в направлении оси X, при этом главная камера сцены следует за ним. Чтобы справляться с вражескими рыцарями, попадающимися у игрока на пути, персонаж может атаковать мечом. При столкновении с ловушкой или при получении удара противника главный герой теряет одну жизнь. Когда количество жизней игрока доходит до нуля, персонаж погибает, и игра начинается заново.

Чтобы помочь игроку восстановиться после получения урона, была создана механика неуязвимости на короткое время после удара. Игрок не будет получать урон от последующих попаданий игрока в ловушки и ударов противников на заданный промежуток времени. Неуязвимость демонстрируется несколькими «вспышками» – изображение героя становится красным, после чего возвращает исходный цвет.

Основными препятствиями в «Running Knight» являются ловушки и противники. Разные виды преград должны требовать от игрока разного поведения, что делает способы взаимодействия игрока с игровым миром более разнообразным.

Ловушка Spikes (шипы) – это препятствие, которое располагается непосредственно на земле. Шипы необходимо перепрыгнуть, иначе игрок получит урон при касании ловушки. При этом игрок может беспрепятственно проходить ловушку насквозь, не прекращая движения. Внешний вид ловушки Spikes находится в Приложении Г.



Другим видом ловушки является летящая стрела: она двигается в воздухе на уровне игрока или чуть выше него (см. рисунок 2.1), что ограничивает мобильность персонажа. Чтобы избежать получения урона, необходимо либо перепрыгнуть через стрелу, либо сделать кувырок.

Разный уровень полета стрел

Противником в игре являются вражеские рыцари, которые нападают на героя, когда тот находится в зоне атаки. Игрок может либо перепрыгнуть через противника, либо атаковать его в ответ, поразив его перед тем, как тот нанесет ему урон. Внешний вид противника и кадры анимаций представлены в Приложении Б.

Бонусы, помогающие игроку во время игрового процесса, появляются в мире игры на месте поверженного врага. Эти предметы можно разделить на защитные и атакующие. Поднятые бонусы либо применяются мгновенно, либо помещаются в «инвентарь» (хранилище для предметов, принадлежащих игроку), хранящий только первые три поднятых предмета. Остальные бонусы будут пропадать, если инвентарь заполнен.

Поднимаемый предмет DestructionCollectible уничтожает все препятствия, находящиеся на данный момент в игровой сцене. После применения бонуса SteelShoesCollectible игрок перестает получать урон от шипов на некоторое время.

Внешний вид упомянутых выше бонусов продемонстрирован на рисунке 2.2.

Рисунок 2.2 – Изображения бонусов (слева направо: DestructionCollectible, HealthCollectible, SteelShoesCollectible)

Получение очков является главной мотивацией игрока становиться лучше и продолжать играть, постепенно улучшая свой результат. Основную массу очков игрок получает за расстояние, которое он смог пробежать. Кроме того за каждого поверженного противника начисляется фиксированное число баллов. Если в забеге игрок набрал результат, превышающий свой предыдущий рекорд, то после перезапуска игры новый лучший счет перезапишет старый.

Во время игрового процесса скорость передвижения главного героя увеличивается на одну десятую от времени, прошедшего с прошлого кадра, до тех пор, пока она не достигнет максимального заданного значения. При этом частота появления новых препятствий будет становится все больше за счет уменьшения времени ожидания между созданием новых преград. Все это в совокупности будет увеличивать сложность игры с течением времени.

1. Программная реализация действий игрового персонажа

Движение персонажа происходит следующим образом: при создании объекта игрока в сцене внутри функции Awake скрипта PlayerMovement, отвечающем за перемещение, берутся ссылки на другие компоненты персонажа, такие как RigidBody2D для симуляции движения, Animator для воспроизведения анимаций бега, кувырка и прыжка, BoxCollider2D для определения положения игрока относительно пола.

Пример кода для взятия ссылок на компоненты объекта представлен на листинге 2.1.

Листинг 2.1 – Получение ссылок на компоненты объекта

body = GetComponent<Rigidbody2D>();

anim = GetComponent<Animator>();

boxCollider = GetComponent<BoxCollider2D>();

Скорость тела внутри компонента RigidBody2D задается двумерным вектором. В функции событий Update на тело накладывается скорость, что приводит тело в движение в направлении оси X, как это продемонстрировано на листинге 2.2.

Листинг 2.2 – Назначение скорости персонажу

body.velocity = new Vector2(speed, body.velocity.y);

Значение скорости увеличивается с каждым кадром, пока не достигнет максимального значения (см. листинг 2.3).

Листинг 2.3 – Увеличение скорости персонажа

if (speed < MAX\_SPEED)

{

speed += Time.deltaTime/10;

}

Для определения положения персонажа относительно пола был создан метод isGrounded. С помощью функции BoxCast класса Physics2D создается виртуальный прямоугольник ниже объекта с такой же шириной, что и BoxCollider2D игрока, и высотой равной 0.1 метра. Если объект, у которого значение слоя – groundLayer, попадает внутрь прямоугольника, то будет возвращена информация об этом объекте посредством RaycastHit2D, в противном случае – будет возвращено значение null. Сравнением полученного объекта с null можно определить, находится игрок на земле или нет.

Пример кода для определения положения персонажа относительно пола предоставлен в листинге 2.4.

Листинг 2.4 – Определение положения персонажа относительно пола

private bool isGrounded()

{

RaycastHit2D raycastHit = Physics2D.BoxCast(

boxCollider.bounds.center,

boxCollider.bounds.size,

0, Vector2.down, 0.1f,

groundLayer

);

return raycastHit.collider != null;

}

Прыжок происходит после нажатия клавишы Пробел на клавиатуре (полная схема управления находится в Приложении Е), и, если в этот момент персонаж находится на земле, скорость тела по координате Y становится равной заданному значению числа jumpPower (сила прыжка). Воспроизводится анимация прыжка персонажа с помощью вызова функции SetTrigger("Jump") контроллера анимаций, как это показано в листинге 2.5.

Листинг 2.5 – Функция Jump

private void Jump()

{

if (isGrounded())

{

anim.SetTrigger("Jump");

body.velocity = new Vector2(body.velocity.x, jumpPower);

}

}

Высота прыжка будет зависеть от продолжительности нажатия клавишы Пробел: если отпустить клавишу до того, как тело достигнет максимальной точки, то его скорость подъема уменьшиться наполовину, чтобы падение началось быстрее (см. листинг 2.6).

Листинг 2.6 – Контроль высоты прыжка

if(Input.GetKeyUp(KeyCode.Space) && body.velocity.y > 0)

{

body.velocity = new Vector2(

body.velocity.x,

body.velocity.y / 2

);

}

Персонаж может совершать несколько прыжков подряд без касания земли, если нажать клавишу Пробел сразу после последнего совершенного прыжка.

Для совершения кувырка для уклонения от стрел, летящих слишком низко, необходимо нажать клавишу C клавиатуры. Размер BoxCollider2D игрока уменьшится вдвое по оси Y, а также запустится соответствующая запись анимации кувырка. На последнем кадре анимации вызывается функция, возвращающая размеры BoxCollider2D в исходное состояние.

1. Создание игрового уровня

Для создания уровня был использован компонент Tilemap, встроенный в Unity Editor. Tilemap позволяет размещать изображения внутри сетки, которая создается компонентом Grid. Он разбивает экран на сетку, состоящую из точек и линий, где каждая ячейка имеет одинаковый размер. Ячейки могут обладать различными формами, например, квадраты, шестиугольники или параллелограммы. Изображения, из которых будет состоять Tilemap, помещаются в Tile Palette – инструмент, схожий по смыслу с палитрой и кисточкой. С помощью кисточки можно выбрать спрайт, находящийся в палитре, и начать размещать его внутри сетки. Такой способ разметки упрощает создание уровней и позволяет быстро создавать прототипы при проектировании 2D-миров внутри игры.

Чтобы движок рассчитывал поведение объектов при столкновении с плитками Tilemap, необходимо добавить компонент Tilemap Collider 2D с Composite Collider 2D, который объединит все картинки в одну форму, если спрайты находятся в соседних ячейках сетки.

Для бесконечного бега персонажа в направлении оси X требуется наличие бесконечно длинного пола, по которому он будет передвигаться. Земля в игре создается с помощью компонента Tilemap повторяющимися изображениями.

Создать карту большой длины – непроизводительно и неэффективно, потому что отображаться в это время на экране будет малая часть от всей сетки. Данная проблема решается созданием новых спрайтов земли на некотором расстоянии перед игроком при каждой смене кадра и удалении ненужных, вышедших за пределы камеры. Для этого берется положение правой границы видимости главной камеры, следящей за движением игрока. Затем прибавляется некоторое расстояние по координате X к координатам границы, чтобы появление новых спрайтов происходило за кадром. И, наконец, переведя полученный двумерный вектор в координаты сетки, устанавливается новый спрайт земли. Все спрайты, вышедшие за левую границу области видимости удаляются: устанавливается нулевое значение на месте ненужных спрайтов.

Пример создания новых спрайтов земли и удаление ненужных продемонстрировано листингом 2.7.

Листинг 2.7 – Создание новых спрайтов земли и удаление ненужных

int cameraLeftPosition = (int)Camera.main.ViewportToWorldPoint(Vector3Int.left).x - 3;

cameraRightPosition = (int)Camera.main.ViewportToWorldPoint(Vector3Int.right).x + 2;

for (int i = -2; i < cameraTopPosition; i++)

{

tilemap.SetTile(tilemap.WorldToCell(

new Vector3Int(cameraLeftPosition, i, 0)

), null);

}

tilemap.SetTile(tilemap.WorldToCell(

new Vector3Int(cameraRightPosition, -2, 0)

), tile);

Другой вариант сценария бесконечной генерации земли использовал случайные числа, то есть для каждого блока делался выбор: будет ли на его месте установлена земля или останется пустое место – «пропасть». Однако от этого варианта пришлось отказаться в пользу описанного выше метода, потому что провалы в земле, которые игроку необходимо перепрыгнуть, можно было заменить ловушкой Spikes.

1. Система здоровья и неуязвимости

Базовый абстрактный класс Health ответственен за систему здоровья внутри игры, а сценарии HealthPlayer и HealthEnemy наследуются от этого класса для определения того, как будет работать здоровье непосредственно для игрока и его противников. Изначальное число жизней, которые определяют сколько ударов может выдержать персонаж перед тем, как он будет деактивирован, задается предварительно либо внутри кода, либо с помощью изменения полей внутри Unity Editor. Базовый класс объявляет, но не определяет метод TakeDamage, так как реакция на получение урона у игрока и противника будет разной: например, при смерти игрока должны запускаться сценарии конца игры, а смерть противника – дает бонус и начисляет очки. Класс Health определяет метод AddHealth, который восстанавливает утраченные очки здоровья на заданное значение.

Неуязвимость персонажа после получения урона реализована с помощью сопрограммы, также называемой Coroutine. Coroutine – это объект Unity, который позволяет выполнять действия параллельно. Используются сопрограммы в Unity в тех случаях, когда код необходимо выполнить в течение нескольких кадров, а не одного. Когда внутри сценария запускается функция StartCoroutine, создается экземпляр сопрограммы, который выполняет некоторые действия до тех пор, пока не доходит до строки yield return, в которой возвращается IEnumerator. В большинстве случаев возвращается объект, который отсчитывает время ожидания для выполнения остальной части кода после операторов yield return, как на листинге 2.8.

Листинг 2.8 – Пример задания времени ожидания

yield return new WaitForSeconds(5);

После выхода из сопрограммы с помощью оператора yield return выполняется код находящийся после вызова StartCoroutine. Когда проходит время ожидания для выполнения отложенной части программы, управление возвращается к следующей строке после оператора yield. На листинге 2.9 продемонстрировано, как это было реализовано для создания неуязвимости в игре.

Листинг 2.9 – Неуязвимость

StartCoroutine(Invulnerability());

protected IEnumerator Invulnerability()

{

invulnerability= true;

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

{

spriteRend.color = new Color(1, 0, 0, 0.5f);

yield return new WaitForSeconds(

iframeDuration/(numberOfFlashes \* 2)

);

spriteRend.color = Color.white;

yield return new WaitForSeconds(

iframeDuration / (numberOfFlashes \* 2)

);

}

invulnerability = false;

}

Функция Invulnerability начинается с установления флага invulnerability в значение true, что предотвращает снятие очков здоровья игрока при выполнении функции TakeDamage. Далее запускается цикл, где в зависимости от количества «вспышек» изменяется цвет изображения игрока с помощью свойства color компонента SpriteRenderer. Между сменами цветов устанавливается время ожидания, и после выхода из цикла неуязвимость заканчивается.

1. Реализация рукопашного боя игрового персонажа

Атака происходит при нажатии Левой кнопки мыши, и если после прошлой атаки прошло заданное количество времени, то начинает воспроизводиться запись анимации атаки игрока. В середине кадра в момент удара мечом происходит проверка, находится ли противник в пределах определенной зоны перед игроком. Для этого перед игроком создается виртуальный прямоугольник, длину и расстояние от игрока которого определяется разработчиком предварительно с помощью полей range и colliderDistance внутри Unity Editor.

Код проверки на наличие противника в зоне атаки можно увидеть на листинге 2.10 ниже.

Листинг 2.10 – Проверка наличие противника в зоне атаки

private bool EnemyInSight()

{

RaycastHit2D hit = Physics2D.BoxCast(

boxCollider.bounds.center + transform.right

\* range \* transform.localScale.x \* colliderDistance,

new Vector3(

boxCollider.bounds.size.x

\* range,boxCollider.bounds.size.y,

boxCollider.bounds.size.z

),

0, Vector2.left, 0, EnemyLayer

);

if (hit.collider != null){

enemyHealth = hit.transform.GetComponent<Health>();

}

return hit.collider != null;

}

Если во время удара мечом в этом прямоугольнике находился противник, то мы получаем доступ к его здоровью и наносим ему урон.

Во время создания прототипов и итерирования механики атаки необходимо было визуальное представление зоны атаки игрока. Для этого была использована функция OnDrawGizmos.

Код представлен на листинге 2.11; визуальное представление продемонстрировано в Приложении Д.

Листинг 2.11 – Применение OnDrawGizmos

private void OnDrawGizmos()

{

Gizmos.color = Color.red;

Gizmos.DrawWireCube(

boxCollider.bounds.center + transform.right

\* range \* transform.localScale.x \* colliderDistance,

new Vector3(

boxCollider.bounds.size.x \* range,

boxCollider.bounds.size.y, boxCollider.bounds.size.z

)

);

}

Gizmos – это инструмент Unity Editor, предоставляющий возможность разработчику визуализировать компоненты объекта, такие как BoxCollider2D, внутри сцены игры. Это помогло при определении оптимального расстояния, в пределах которых противник будет получать урон.

1. Программная реализация препятствий и противников

Нанесение урона игроку посредством препятствий реализовано с помощью сценария EnemyDamage. Базовый класс сценариев MonoBehaviour объявляет метод OnTriggerEnter2D, который вызывается при регистрации любого столкновения объекта с другими сущностями, передавая в качестве аргумента информацию о столкновении. При этом у одного из объектов столкновения должно быть отмечено свойство IsTrigger внутри компонента BoxCollider2D, которое позволит другим телам проходить объект насквозь. Регистрация столкновений будет происходить как и раньше.

Внутри функции OnTriggerEnter2D, которая была определена в сценарии EnemyDamage, происходит проверка тега объекта столкновения: если тег имеет значение Player, то персонажу наносится некоторый урон. Ловушка Spikes наносит урон игроку именно таким образом, при этом функция OnTriggerEnter2D вызывается только в случае вхождения в ловушку, то есть пока игрок передвигается внутри ловушки и при выходе из нее урон наноситься не будет.

Ловушка со стрелой состоит из двух частей: источник выстрела и стрела. Ловушка не имеет визуального представления, она служит как источник полета стрелы. Сразу же после создания объекта в мире игры, она выпускает снаряд через функцию Attack сценария ArrowTrap. Внутри метода генерируется случайное целое число с помощью функции Range класса Random, которое определяет, на какой высоте полетит стрела. В конце мы обращаемся к методу ActivateProjectile объекта стрелы для выпуска снаряда.

Пример кода функции Attack ловушки со стрелами предоставлен в листинге 2.12:

Листинг 2.12 – Функция Attack сценария ArrowTrap

private void Attack()

{

int random = Mathf.RoundToInt(Random.Range(2f, 3f));

arrow.transform.position = new Vector2(

Camera.main.ViewportToWorldPoint(Vector3Int.right).x + 3,

random

);

arrow.GetComponent<EnemyProjectile>().ActivateProjectile();

}

К стреле применен сценарий EnemyProjectile, который наследуется от EnemyDamage, чтобы впоследствии наносить урон игроку при попадании. Метод ActivateProjectile активирует объект вместе с регистрацией столкновений. В функции событий Update стреле присваивается скорость для движения против оси X, а также отсчитывается время после запуска снаряда. Если время станет больше некоторой заданной величины, то стрела будет деактивирована. Сделано это для повышения производительности, так как стрела потенциально может продолжать лететь, хотя игрок уже давно ее преодолел. В случае, если стрела столкнется на своем пути с любым объектом, то она деактивируется и передает классу-родителю информацию о столкновении.

Противники для нанесения урона игроку используют сценарий MeleeEnemy, который функционирует схожим образом, как и сценарий атаки игрока. Вместо нажатия Левой кнопки мыши, началом атаки является вход игрока внутрь виртуального прямоугольника перед противником. То есть враги будут реагировать на появление игрока в «зоне видимости», чтобы начать нападение.

1. Игровые объекты, управляющие системами игры

За управление системой создания новых препятствий перед игроком ответственен объект SpawnManager, который является пустым объектом с единственным компонентом с таким же названием. Функционирует он следующим образом: внутри функции Update запускается таймер. Когда проходит время ожидания, то в игре создается новый случайный объект из списка, который заполнен экземплярами каждого из видов препятствий. Максимальное количество одновременно существующих в игровом мире препятствий ограничено сверху заданным числом. Новая сущность помещается правее крайней границы камеры на высоте, которая вычисляется в зависимости от вида преграды. Частота появления новых препятствий будет случайной внутри диапазона, верхняя граница которого будет уменьшаться на одну сотую от времени, прошедшего с последнего кадра игры.

Для проигрывания всех звуков был создан специальный объект с компонентом AudioSource. Этот объект, под названием SoundManager, станет источником всех звуков в игре. Сценарий SoundManager предоставляет публичный метод для воспроизведения звуков – PlaySound, принимающий аудиодорожку и тут же ее воспроизводящий. Объект для управления звуком игры должен быть один, поэтому SoundManager реализует паттерн Singleton. Singleton – паттерн программирования, при котором существует гарантия того, что будет создан единственный объект класса и что к этому экземпляру будет предоставлен доступ.

Воспроизвести звук в других объектах можно следующим образом: сначала мы получаем ссылку на экземпляр источника звука внутри других скриптов, а затем на этом экземпляре вызываются все необходимые звуки. При этом сам SoundManager не будет уничтожен после загрузки новой сцены, чтобы все настройки уровня громкости сохранялись при переходе между игрой и главным меню.

Объект GameManager отвечает за конец и перезапуск игры, которые происходят после смерти игрока. Фоновая музыка обрывается, а далее происходит отложенный вызов функции, которая показывает экран проигрыша. Отложенный вызов достигнут использованием встроенной функции Invoke, первым аргументом которой передается строка, содержащая название функции, которую необходимо запустить, а вторым – время ожидания.

После этого сцена загружается заново, перед этим сохранив лучший счет, если таковой был достигнут в этом сеансе. Все объекты принимают такое же состояние, как и в начале игры, что создает зацикленность игрового процесса.

Для сохранения данных игрока необходим специальный обработчик, который будет способен записывать и считывать информацию из файла, чтобы хранить лучший счет игрока даже после выключения игры. В сценарии DataManager контейнером для данных служит объект внутреннего класса Data, в полях которого в свою очередь хранится информация, которую необходимо сохранить: в нашем случае это лучший счет. В сценарии также хранится путь к файлу расширения .json, в котором будут находится данные игры. При первом запуске приложения лучший счет равен нулю. Для того, чтобы установить новое значение поля bestScore и получить текущее состояние объекта типа Data существуют соответствующие функции SetBestScore и GetGameData.

Фукнция SetBestScore получает новый лучший счет и устанавливает полю объекта класса Data bestScore новое значение. Если объекта еще не существует, то метод его создает. GetGameData также создает объект для хранения данных, если это не было сделано ранее, и возвращает его. Реализация функций SetBestScore и GetGameData представлены на листинге 2.13.

Листинг 2.13 – Функции SetBestScore и GetGameData

public void SetBestScore(int bestScore)

{

if (gameData == null)

{

gameData = new Data();

}

gameData.bestScore = bestScore;

}

public Data GetGameData()

{

if (gameData == null)

{

gameData = new Data();

gameData.bestScore = 0;

}

return gameData;

}

Методы Load и Save нужны для чтения и записи объекта gameData в файл. Чтобы загрузить данные из файла, создается экземпляр класса StreamReader, встроенный в C#, затем все содержимое файла помещается в строку, из которой потом извлекается объект класса Data. В случае сохранения информации необходимо сначала перевести объект в строку в формате JSON, а затем записать ее содержимое в файл посредством StreamWriter.

1. Реализация объектов-бонусов

Объекты предметов, «поднимаемых» при касании персонажа игрока, должны либо моментально примениться, либо добавиться в инвентарь игрока. Общим поведением, которым обладают все поднимаемые предметы, является уничтожение объекта, если его не подняли в течении некоторого времени. Эта логика находится внутри скрипта Collectible, от которого будут наследоваться сценарии всех видов бонусов.

Бонусы, которые можно будет применить самим игроком, хранятся в компоненте Bonuses. Все виды поднимаемых предметов находятся в перечислении CollectibleTypes, доступ к которому есть у всех сценариев приложения. При создании сцены берутся ссылки на компоненты GameManager для запуска анимаций внутри пользовательского интерфейса, SpriteRenderer игрока, чтобы визуально продемонстрировать применение некоторых бонусов, числовое значение слоев Player и Spike, и ссылку на элемент интерфейса, отображающий бонусы, которые игрок может использовать.

В каждом кадре сценарий ждет нажатия на клавишу F (или А при русской раскладке) на клавиатуре, отвечающую за применение бонуса. Все предметы находятся в очереди Queue<CollectibleTypes> collected, ограниченного сверху заданным числом. Структура данных Очередь, или еще называемая Queue, реализует принцип «первый вошел - первый вышел» . Новые элементы очереди добавляются в «хвост» списка, при этом взять можно только первый элемент, в данный момент находящийся в «голове». Примером очередей могут служить очереди в магазине, в которых первым обслуживается покупатель во главе.

Первый попавший в список предмет будет использован в первую очередь. Поэтому после нажатия кнопки, switch-выражение проверяет значение элемента списка, и в зависимости от его вида, выполняет различные действия, в которых и будет заключаться функция каждого из бонусов.

После применения бонуса DestructionCollectible необходимо было реализовать уничтожение всех объектов препятствий, созданных SpawnManager. Для начала берется ссылка на экземпляр SpawnManager, чтобы затем вызвать на нем функцию emptySpawnedList, уничтожающую все объекты преград. На экране при этом проигрывается анимация удара молнии на белом фоне, чтобы преграды не исчезали на глазах у игрока.

В конце из списка доступных бонусов убирается первый элемент. При этом содержимое «инвентаря» обновляется на экране с помощью кода на листинге 2.14.

Листинг 2.14 – Код, выполняемый при применении бонуса DestructionCollectible

case CollectibleTypes.Destruction:

SpawnManager spawnManager =

(SpawnManager)FindAnyObjectByType(typeof(SpawnManager));

SoundManager.instance.PlaySound(lightningSound);

gameManager.PlayLightningAnim();

spawnManager.emptySpawnedList();

ui.RefreshUIList();

break;

Если был применен бонус SteelShoesCollectible, то вызывается сопрограмма InvulnerabilityFromSpikes, которая работает схожим образом с функцией Invulnerability сценария Health, отключающая регистрацию столкновения слоев игрока и шипов, при этом изменяя цвет персонажа на серый.

Пример кода, выполняемого при использовании SteelShoesCollectible находится в листинге 2.15.

Листинг 2.15 – Код, выполняемый при применении бонуса SteelShoesCollectible

case CollectibleTypes.SteelShoes:

SoundManager.instance.PlaySound(steelShoeSound);

ui.RefreshUIList();

StartCoroutine(InvulnerabilityFromSpikes());

break;

1. Проектирование пользовательского интерфейса

Чтобы игрок обладал всей информацией, необходимой ему для прохождения игры, того, что происходит на экране, недостаточно. Информация о количестве здоровья, сколько было набрано очков, сколько он собрал бонусов – показать напрямую в игровом мире невозможно или неудобно. Для этой цели служит специальный экран, который накладывается поверх игрового процесса, называемый UI (user interface) или пользовательским интерфейсом.

В Unity все элементы интерфейса должны находиться внутри специальной зоны Canvas, играющую роль холста для всего интерфейса. Если создать элемент UI без существующего холста в сцене, то он будет автоматически добавлен вместо нас и изначальный элемент станет потомком Canvas. Отображаться пользовательский интерфейс может либо поверх всей сцены, либо прямо перед камерой игры или вообще являться таким же объектом мира, как и остальные сущности. В зависимости от предназначения интерфейса могут реализовываться разные варианты.

Элементы UI являются дочерними по отношению к холсту. Их положение на экране, угол поворота и размер меняется теми же инструментами, что остальные объекты в Unity. Это достигается тем, что каждый объект интерфейса Unity имеет компонент RectTransform, который имеет положение, масштаб и угол поворота, как и обычный Transform, но к ним добавлены значения ширины и высоты, определяющие размерность прямоугольника, внутри которого будет находиться текст, изображения и другие элементы.

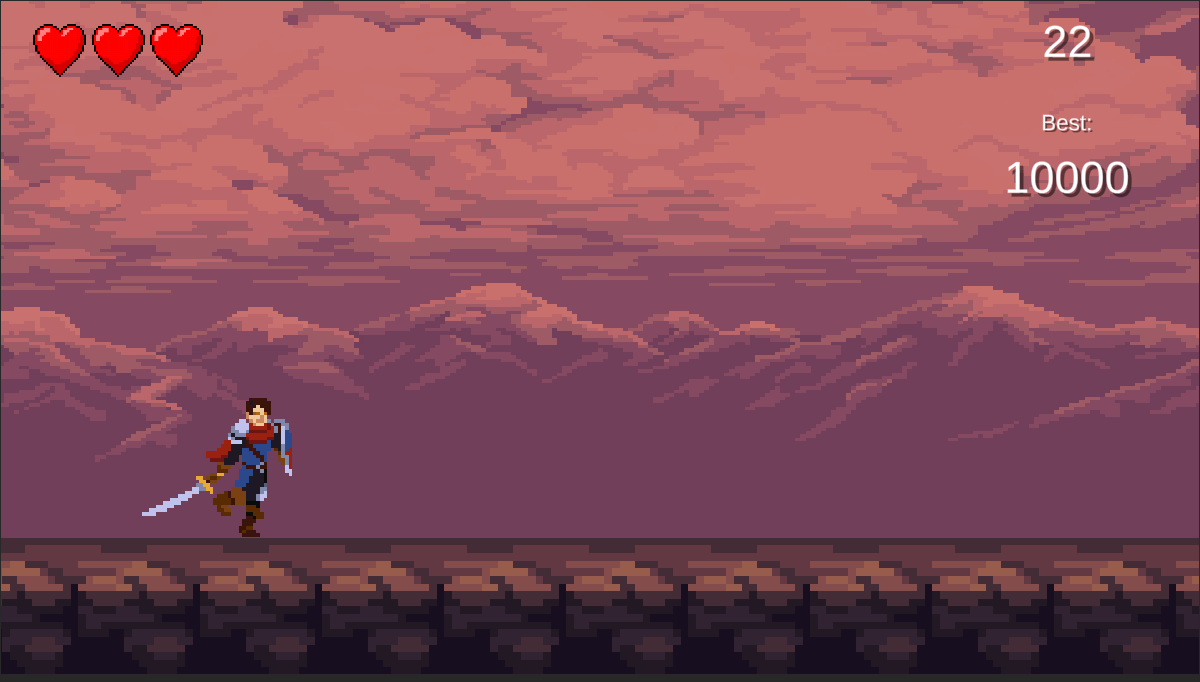
Интерфейс «Running Knight» во время игрового процесса выглядит как на рисунке 2.3.

Рисунок 2.3 –Интерфейс во время игрового процесса

Изображения сердец обозначают количество очков здоровья игрока, при получении урона одно из сердец закрашивается черным цветом, пока жизнь игрового персонажа не закончится. В правом углу сверху находится счетчик очков, полученных в этой сессии, в то время как ниже можно увидеть свой лучший результат за все время игры.

В нижнем левом углу экрана расположены бонусы, которые игрок может использовать. Они обновляются каждый раз, когда игрок применяет один из них, или подбирает новый. Для этого внутри холста интерфейса есть объект, содержащий элементы с компонентом для отображения изображений. Чтобы отобразить в этих элементах список бонусов, проходим с помощью цикла по всем доступным изображениям, для каждого устанавливая спрайт бонуса, который находится в «инвентаре» игрока. Если бонусов меньше, чем максимально возможное количество, то для картинок, которым не соответствует бонус, устанавливается пустое значение. Описанный код представлен на листинге 2.16.

Листинг 2.16 – Визуализация бонусов из «инвентаря»

toVisualize = bonuses.getList();

for(int i=0;i< imageList.Count;i++)

{

imageBuffer = imageList[i].GetComponent<Image>();

if (i< toVisualize.Count)

{

switch (toVisualize[i])

{

case CollectibleTypes.Destruction:

imageBuffer.enabled = true;

imageBuffer.sprite = destructImage;

break;

case CollectibleTypes.SteelShoes:

imageBuffer.enabled = true;

imageBuffer.sprite = steelShoesImage;

break;

case CollectibleTypes.Default:

imageBuffer.enabled = false;

imageBuffer.sprite = null;

break;

}

}

else

{

imageBuffer.enabled = false;

imageBuffer.sprite = null;

}

}

Чтобы приостановить игру на некоторое время, игрок может нажать клавишу Escape. В результате появится меню паузы, в котором игрок может продолжить игру, изменить настройки громкости звука и музыки игры или выйти в главное меню. При этом игра будет остановлена на фоне: значения скорости, с которой идет время внутри игры, установлено в 0. Выйти из меню можно нажав кнопку Escape еще раз.

При запуске приложения игры сначала загружается сцена главного меню, из которой можно перейти к игровому процессу, нажав кнопку Play. При нажатии на кнопку Options, появляется окно, где можно изменить громкость игры отдельно для звуков и музыки. Чтобы выйти из приложения, есть возможность нажать на Quit, что выключит игру.

Когда игрок погибает, прямо перед перезапуском уровня появляется экран, информирующий о конце игры.

1. Выбор компонентов игры из магазина Unity Asset Store. Создание оригинальной анимации

Изображения, необходимые для создания внешнего вида игры, были взяты из Unity Asset Store, где они распространялись бесплатно. При выборе спрайтов отдавалось предпочтение тем компонентам графики, в которых автор придерживался визуального стиля игр 70-80-х годов, называемый Pixel Art. Pixel Art – это стиль цифрового рисунка, при создании которого художник работает на уровне пикселей. Такой стиль широко использовался разработчиками видеоигр 70-80-х годов, что обусловлено ограничениями компьютеров того времени. После развития технологий и появления трехмерной графики интерес к Pixel Art не исчез, и до сих пор этот стиль применяется при создании видеоигр.

Покадровая анимация удара молнии при использовании бонуса разрушения была нарисована для игры с помощью веб-приложения PixilArt. Этот сайт позиционируется как социальная платформа, где художники могут делиться своими картинами, созданными в стиле пиксельной графики.

Помимо этого, веб-приложение предоставляет бесплатные инструменты для создания пиксельного изображения. Картины можно разбивать на слои, рисуя отдельные части на разных полотнах, в конце объединяя их воедино. Можно выбрать различные цветовые палитры из встроенных или создать их самому. PixilArt дает возможность создавать анимации, позволяя нарисовать каждый кадр по отдельности. Кадры затем можно импортировать в виде готовой записи в файле с расширением .gif, или расположить каждый кадр друг за другом на одном изображении. Последний формат облегчает процесс переноса анимации для реализации внутри платформ для создания игр, например, Unity.

В Приложении В представлена раскадровка созданной анимации для бонуса DestructionCollectible. Кадры анимации были созданы с помощью редактора PixilArt.

Заключение

Был проведен сравнительный анализ компьютерных игр жанра «бесконечный раннер», изучены основные составляющие жанра, рассмотрены достоинства и недостатки игр-аналогов.

Были сравнены платформы для разработки видеоигр Unreal Engine и Unity. Выбор был сделан в пользу игровой платформы Unity.

Были спроектированы правила игры, придуманы различные ловушки, противники; проработаны системы бонусов и система начисления очков. Реализовано три типа препятствий: ловушки Spikes и ArrowTrap, противник MeleeEnemy. Разработаны три бонуса: DestructionCollectible, HealthCollectible, SteelShoesCollectible. Создана одна оригинальная анимация применения бонуса DestructionCollectible.

Видеоигра разработана для операционной системы Windows с использованием платформы Unity и языка программирования C#, спрайтов и анимаций из магазина Unity Asset Store.

Игра предлагает игроку справляться с разнообразными препятствиями, преодоление которых усложняется с ходом игры. Система бонусов делает игру еще более разнообразной, предоставляя игроку выбор тактик прохождения. Созданная видеоигра позволяет провести в ней неограниченное количество времени, так как ее невозможно пройти, но при этом игрок мотивирован совершенствовать свое мастерство и ставить новые рекорды по количеству очков.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Н. Н. Шамгунов, Г. А. Корнеев, А. А. Шалыто. State Machine — новый паттерн объектно-ориентированного проектирования [Электронный ресурс] – Режим доступа: <http://www.softcraft.ru/auto/switch/statemachine/statemachine.pdf> – Дата доступа: 04.04.2023.
2. C# Documentation [Electronic resource] – Mode of access: <https://learn.microsoft.com/en-us/dotnet/csharp/> – Date of access: 16.03.2023.
3. RB Whitaker. The C# Player’s Guide Fifth edition– Starbound Software, 2022 – 495 p.
4. Dinosaur T-Rex Game [Electronic resource] – Mode of access: <https://trex-runner.com/> – Date of access: 12.03.2023.
5. Subway Surfers [Electronic resource] – Mode of access: <https://sybogames.com/subway-surfers/> – Date of access: 12.03.2023.
6. Unity Documentation [Electronic resource] – Mode of access: <https://docs.unity.com/> – Date of access: 13.03.2023.
7. Unity: Что представляет из себя Coroutine и зачем там Ienumerator [Электронный ресурс] – Режим доступа: <https://habr.com/ru/articles/682320/> – Дата доступа: 09.04.2023
8. Unity Asset Store [Electronic resource] – Mode of access: https://assetstore.unity.com/ – Date of access: 13.03.2023.
9. Unreal Engine 5 Documentation [Electronic resource] – Mode of access: <https://docs.unrealengine.com/5.0/en-US/> – Date of access: 13.03.2023.
10. Visual Studio documentation [Electronic resource] – Mode of access: <https://learn.microsoft.com/en-us/visualstudio/windows/?view=vs-2022> – Date of access: 23.02.2023.
11. PixilArt [Electronic resource] – Mode of access: <https://www.pixilart.com/> – Date of access: 11.04.2023.

Приложения

Приложение А

C:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Fall.pngC:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Jump.pngC:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Attack.pngC:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Attack.pngC:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Idle.pngC:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Run.pngКадры анимации игрового персонажа из Unity Asset Store

Рисунок А5 – анимация падения

Рисунок А4 – анимация прыжка

Рисунок А3 – анимация удара

Рисунок А1 – анимация положения покоя

Рисунок А2 – анимация бега

C:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-hurt.pngC:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Roll.pngC:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Roll.pngC:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Death.pngC:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Death.png

Рисунок А8 – анимация смерти

Рисунок А7 – анимация кувырка

Рисунок А6 – анимация получения урона

Приложение Б

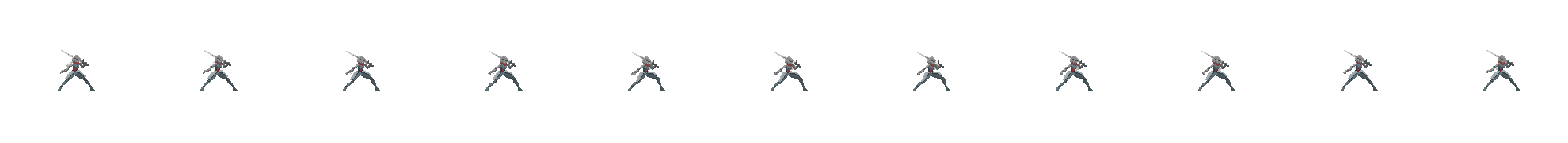
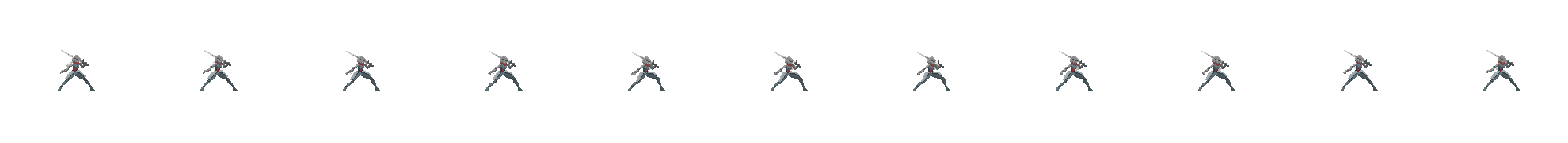
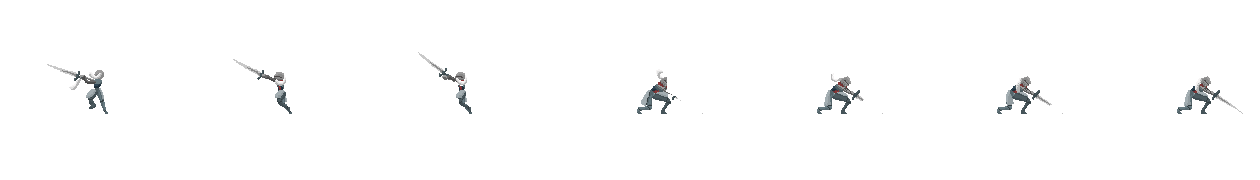
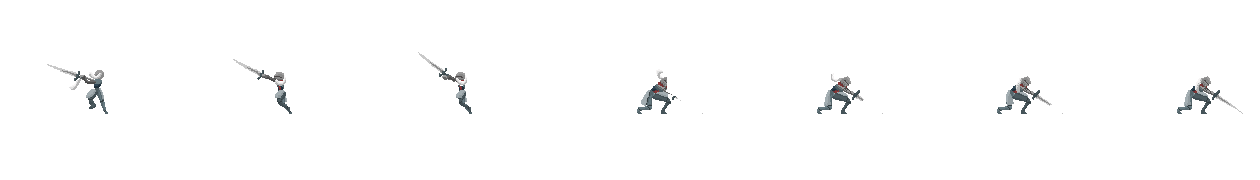
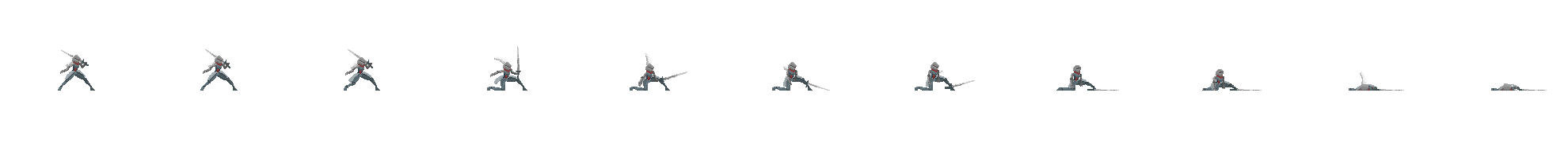
Кадры анимации противника из Unity Asset Store

Рисунок Б2 – анимация удара

Рисунок Б1 – анимация положения покоя

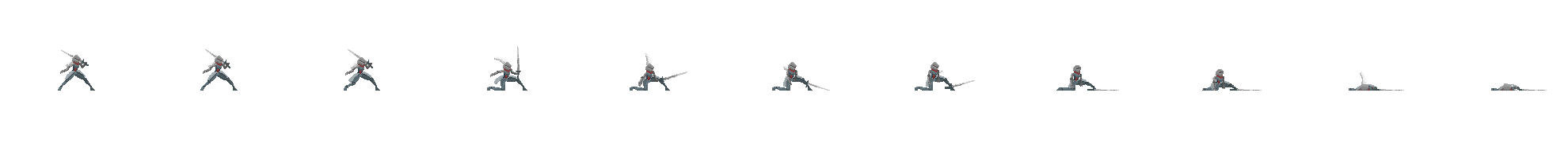


Рисунок Б3 – анимация смерти

ПРИЛОЖЕНИЕ В

Оригинальная анимация, созданная с помощью редактора PixilArt

Рисунок В1 - анимации удара молнии

Приложение Г

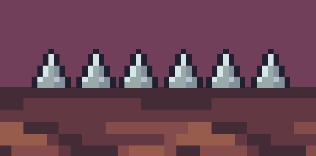
Изображение ловушки Spikes

Рисунок Г1 – Ловушка Spikes

Приложение Д

Изображение зоны атаки персонажа и фоновое изображение уровня игры

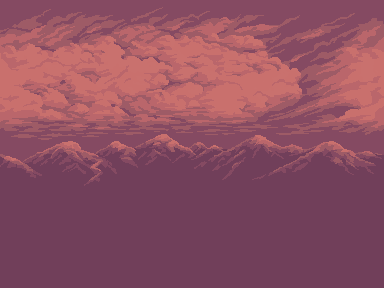


Рисунок Д2 – фоновое изображение уровня игры

Рисунок Д1 – зона атаки персонажа

Приложение Е

Схема управления

Таблица Е1 – схема управления

|  |  |
| --- | --- |
| **Клавиша** | **Функция** |
| Левая кнопка мыши | Атака |
| Space(Пробел) | Прыжок |
| С | Кувырок |
| F(А) | Применить бонус |

Приложение Ж

Ссылка на исходный код проекта

https://github.com/SirGeckT0R/CourseworkUnity