**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](#_Toc134463235)

[Глава 1 Обзор игровой платформы и Анализ аналогов 4](#_Toc134463236)

[1.1 Выбор платформы для разработки игр 4](#_Toc134463237)

[1.2 Обзор жанра игры 5](#_Toc134463238)

[1.3 Анализ игр аналогов 6](#_Toc134463239)

[1.4 Обзор платформы Unity 8](#_Toc134463240)

[Глава 2 Проектирование и программная реализация видеоигры 15](#_Toc134463241)

[2.1 Концепция игры 15](#_Toc134463242)

[2.2 Программная реализация игрового персонажа 17](#_Toc134463243)

[2.3 Создание игрового уровня 19](#_Toc134463244)

[2.4 Система здоровья и неуязвимости 21](#_Toc134463245)

[2.5 Реализация рукопашного боя игрового персонажа 22](#_Toc134463246)

[2.6 Программная реализация препятствий и противников 23](#_Toc134463247)

[2.7 Создание игровых систем 25](#_Toc134463248)

[2.8 Разработка бонусов 27](#_Toc134463249)

[2.9 Проектирование пользовательского интерфейса 28](#_Toc134463250)

[2.10 Создание анимации 31](#_Toc134463251)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 34](#_Toc134463252)

[Приложения 35](#_Toc134463253)

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. 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 снискал 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 жанра бесконечный бег.
* Спроектировать правила игры.
* Реализовать спроектированные правила игры с помощью выбранных технологий разработки.
* Разработать пользовательский интерфейс игры, оформить визуальное представление игры и анимации с использованием готовых компонентов и собственных.

Глава 1 Обзор игровой платформы и Анализ аналогов

1. Выбор платформы для разработки игр

При создании видеоигр используются так называемые игровые движки. Игровой движок – базовое программное обеспечение компьютерной игры, пригодное для повторного использования и расширения. Общие технологии, необходимые при создании каждой игры, уже встроены в них, и разработчику, например, не нужно заново реализовывать физику тел и их отображение для каждой игры. Самыми популярными платформами для создания игр на данный момент являются Unity и Unreal Engine.

Unity — кроссплатформенная среда разработки компьютерных игр, разработанная американской 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 – отдельные 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 объект 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, который отвечает за физику тел.

Для написания скриптов используется язык C# – язык с С-подобным синтаксисом, наиболее близкий к языкам С++ и Java. Он является статически типизированным, реализует механизмы и принципы ООП, при этом избавившись от проблематичных моделей своих предшественников, такие как множественное наследование С++.

Ключевыми возможностями Unity являются наличие визуальной среды разработки, что упрощает прототипирование и тестирование игры; наличие большого количества обучающего контента; простота интерфейса редактора и магазин Unity Asset Store, где разработчики могут продавать различные компоненты, используемые при создании игр, например, модели, аудио-файлы, библиотеки и другое. Так же 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 простейшей логики игры. Недостатками Unreal Engine являются отсутствие 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.

Для работы с кодом проекта подойдет любая среда разработки, однако больше всего инструментов для работы с платформой Unity предоставляют Rider от компании JetBrains и Visual Studio от Microsoft. При создании данного проекта использовалась бесплатная версия Visual Studio Community 2022 с установленными инструментами для Unity.

1. Обзор жанра игры

В зависимости от основных игровых механик видеоигры подразделяются на различные категории и подкатегории. Одними из самых старейших и простых в реализации являются игры жанра «платформер». В таких играх основной игровой процесс составляют прыжки по платформам и сбор предметов для победы над противниками. Среди поджанров жанра «платформер» выделяется «бесконечный раннер»(или «бесконечный бег»).

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

1. Анализ игр аналогов

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

Dinosaur T-Rex Game – встроенная в браузер Google Chrome игра, спрятанная на странице ошибки при отсутствии соединения с интернетом. Игрок берет на себя управление бегущего динозавра с помощью клавиш вверх и вниз, чтобы перепрыгивать или пригибаться перед препятствиями. На экране отображается счетчик очков, которые увеличиваются во время хода игры. 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.

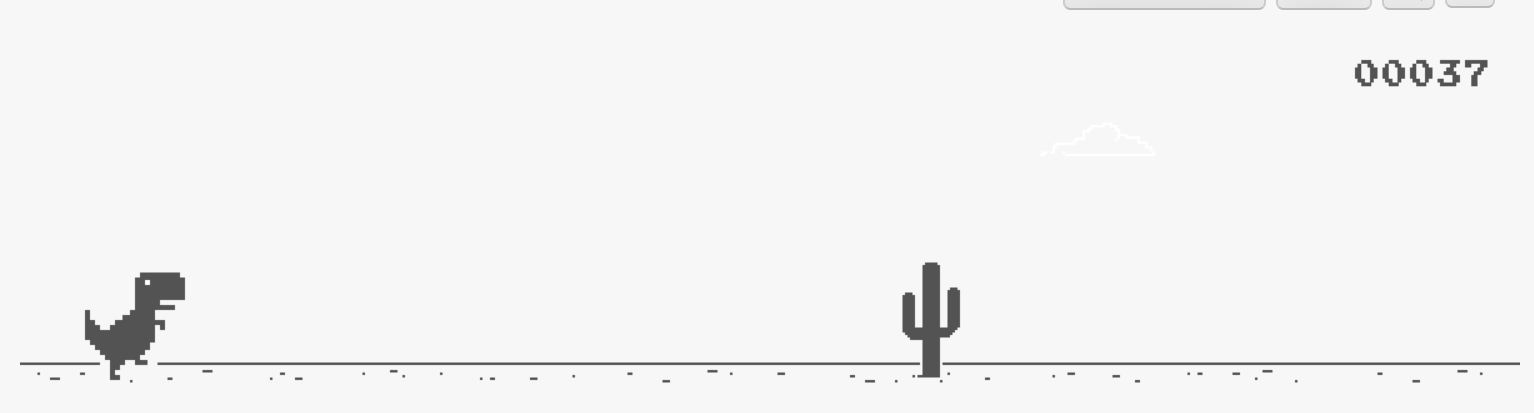
Достоинствами данной игры является простое управление, возрастающая сложность и система очков, позволяющая игроку сохранять интерес к игре на продолжительное время с целью увеличения своего лучшего результата, соревнуясь с другими людьми. Среди недостатков выделяется однообразный визуальный стиль, малое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

Рисунок 1.1 – Dinosaur T-Rex Game

Другим аналогом является мобильная 3D-игра Subway Surfers. Главному персонажу нужно убегать от охранника по железнодорожным путям, уворачиваясь от поездов и минуя другие препятствия, встречающиеся на его пути. Игроку предоставлены три пути передвижения, во время игрового процесса ему будет необходимо своевременно сменять пути движения для предотвращения столкновений, а также использовать прыжки и кувырки, чтобы уклоняться от некоторых препятствий, когда смена дорожки невозможна. Одной из главных игровых механик Subway Surfers является система бонусов, которые игрок подбирает во время своего пути. Они помогают игроку преодолевать препятствия, добавляют мультипликатор счета для улучшения результата, помогают собирать монеты и так далее. Например, ракетный ранец позволяет игроку на некоторое время подняться в воздух, где столкновение с объектами ему не грозит, или специальные сапоги, которые позволяют перепрыгивать через поезда.

Среди преимуществ игры Subway Surfers имеются привлекающий визуальный стиль, наличие нескольких способов для решения одной и той же ситуации, наличие системы бонусов, помогающих игроку в прохождении. Недостатком же можно назвать то, что бонусы применяются сразу после подъема, хотя нужды в этом в данный момент может и не быть.

Рисунок 1.2 –Subway Surfers

При создании игры будут учтены достоинства и недостатки игр-аналогов. Из них будут позаимствованы возрастающая сложность, ведение счета очков, простота управления, система бонусов для помощи игроку. При этом будут исправлены упомянутые недочеты: однообразный визуальный стиль, отсутствие возможности отложить применение бонусов для более подходящих ситуаций.

1. Обзор платформы Unity

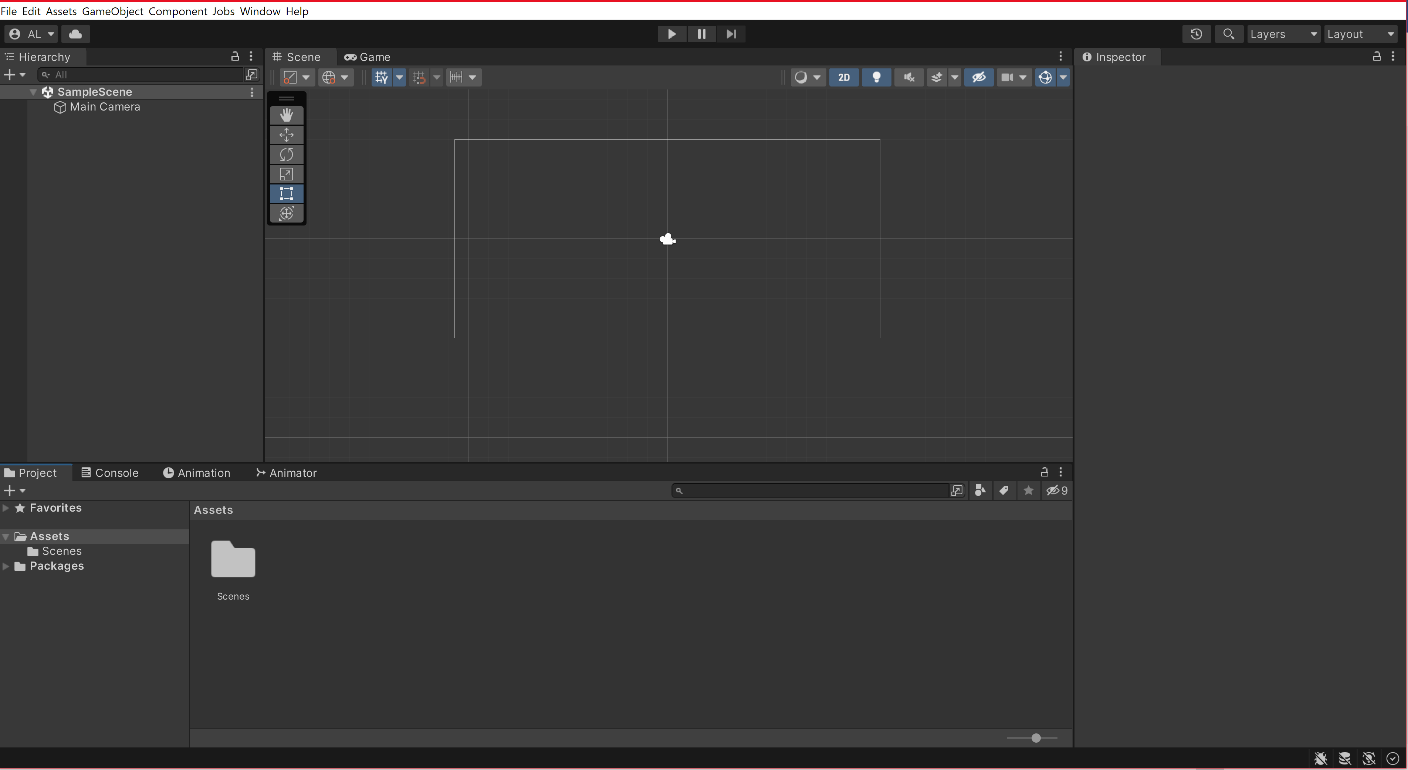
При создании проекта в Unity открывается редактор Unity Editor с открытой сценой, где находится единственный объект – камера. Камера служит для задания области игрового мира, которая будет видна на экране монитора.

Рисунок 1.3 – Интерфейс редактора Unity

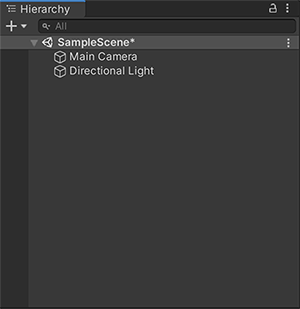
В окне Heirarchy находится иерархия объектов внутри сцены, здесь же можно создавать, выбирать и удалять объекты сцены.

Рисунок 1.4 – Окно иерархии

Каждый объект в игре создается с уже добавленным компонентом Transform. Этот компонент используется для хранения позиции, угла поворота, масштаба объекта внутри игрового мира. Положение объектов определяется координатами x, y, z внутри сцены. Каждый Transform может являться потомком другого компонента Transform, и его размеры и положение будут определяться относительно своего родителя. Это и позволяет Unity создавать иерархию объектов, которую мы видим в окне Heirarchy.

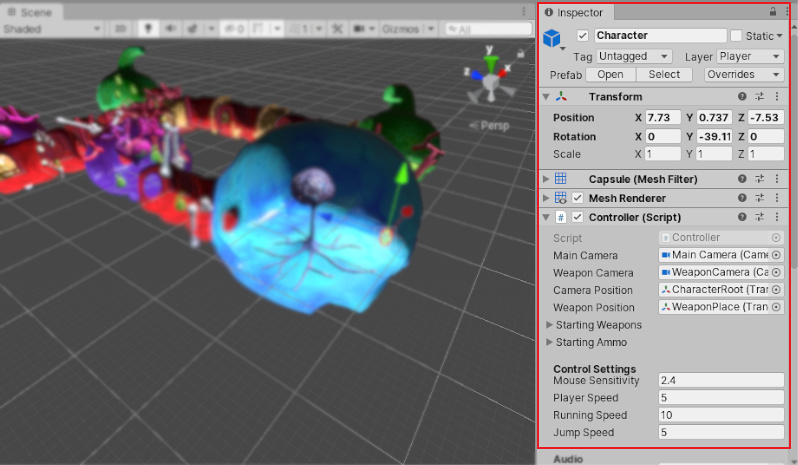
Чтобы посмотреть компоненты объекта, нужно выбрать этот объект в окне иерархии или внутри сцены простым нажатием на объект Левой кнопки мыши. Вся информация появится в окне Inspector, где можно добавлять компоненты, удалять и изменять свойства объекта. Каждый игровой объект имеет тег и слой. Теги нужны для разбиения объектов на группы, присваивая им строку-идентификатор, что упрощает написание сценариев, если нужно применить какое-то действие на несколько объектов. Слои, в свою очередь, нужны для определения взаимодействий объектов друг с другом внутри сцены, например, скрыть от камеры определенные объекты при отображении.

Рисунок 1.5 – Окно инспектора

С помощью окна Project можно взаимодействовать с файлами проекта и библиотеками. Console используется для отображения ошибок, появляющихся при компиляции скриптов и во время игры, а также для вывода пользовательской информации во время выполнения скриптов.

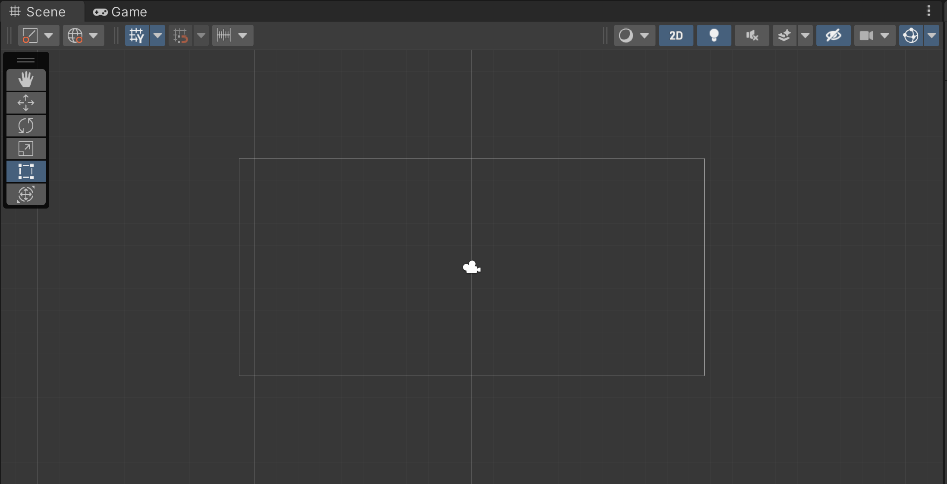
Отображение сцены со всеми объектами, находящимися в ней, находится в окне Scene, где можно манипулировать объектами с помощью инструментов перемещения, поворота, масштабирования и другие. То, как игра будет выглядеть для конечного пользователя, находится в окне Game, где происходит сам игровой процесс при запуске игры.

Рисунок 1.6 – Окно сцены

Игровой движок Unity дает возможность переключаться на разные способы отображения игры: 2D и 3D. В 2D режиме вся графика будет отображаться в виде плоских картинок, именуемых спрайтами, в то время как трехмерные модели требуют 3D-отображение. Для того, чтобы присвоить изображение определенному объект внутри сцены, необходимо добавить ему компонент Sprite Renderer, и перетаскиванием(или выбором в выпадающем меню) добавить в поле Sprite нужную картинку.

Для расчета поведения объектов при столкновении используется встроенный компонент Unity. Collider2D позволяет разработчику определить форму объекта для определения столкновения с другими объектами при пересечении данной формы. Дочерний компонент BoxCollider2D позволяет определить форму как прямоугольник. Внутри него есть свойства, позволяющие настроить поведения объекта при столкновении: просто проинформировать скрипты про наличие столкновения, позволяя другой сущности пройти насквозь, или просчитать поведение объектов с помощью физических формул, чтобы объекты оттолкнулись друг от друга или остановились после столкновения.

RigidBody2D – компонент, накладывающий контроль физического движка над поведением объекта, то есть к сущности будут применяться силы гравитации, она будет обладать своим весом, скоростью, коэффициентом трения и прочими свойствами, необходимыми для применения силы по направлению к объекту.

Для определения пользовательского поведения объекта или для программирования событий внутри игры используются скрипты(сценарии), которые находятся среди компонентов объектов. По умолчанию Unity Editor создает внутри скрипта класс, наследуемый от MonoBehaviour – базовый класс для всех сценариев Unity, которые впоследствии станут компонентами игровых сущностей. Сценарий в Unity не является традиционной идеей программы, где код выполняется последовательно, пока задача не будет выполнена. Вместо этого Unity передает контроль скрипту 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 функция 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 всякий раз перед отрисовкой кадра и перед расчетом анимаций.

Если необходимо вызвать код инициализации перед началом обновлений, то применяются методы Start и Awake, которые срабатывают при включении скрипта и при создании объекта соответственно.

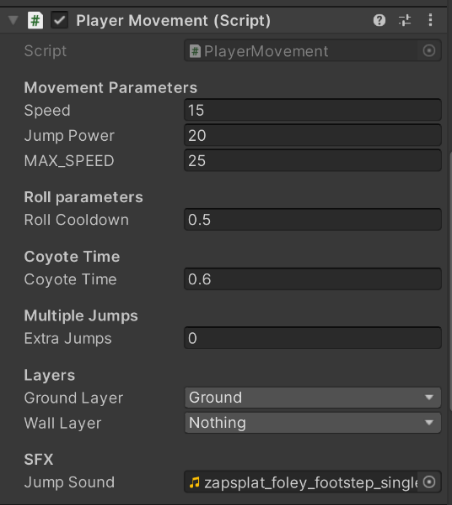
Поля класса сценария, помимо встроенных в язык программирования C# модификаторов доступа, могут принимать модификатор типа [SerializeField]. В таком случае поле называется сериализуемым. Этот вид доступа, а также модификатор public позволяют редактировать значение поля напрямую из Unity Editor в компоненте соответствующего сценария, но в отличие от последнего, сериализуемое поле класса будет недоступно для изменения другим сценариям проекта.

Рисунок 1.7 – Сериализуемые поля внутри редактора Unity

Для того, чтобы добавить аудио-сопровождение к игре, Unity предоставляет компонент AudioListener(слушатель аудио), который играет роль микрофона в 3D пространстве. Этот компонент размещается на объекте, который будет записывать звуки вокруг себя и проигрывать их игроку через аудио-устройства компьютера. В сцене может быть только один слушатель и его обычно размещают на главной камере, что является выбором по умолчанию.

Для воспроизведения аудиодорожки на источнике звука должен находится компонент AudioSource(источник звука). В поле AudioClip помещают файл звука, с помощью других свойств можно настроить громкость, зацикленность звука и другое.

Анимация – представление движения статических картинок путем отображения последовательности кадров с высокой частотой. Unity Editor, помимо возможности импортировать записи анимаций сторонних приложений, предоставляет разработчику встроенные инструменты, чтобы анимировать игровые сущности. Для этого в редакторе существует Animator Controller – контроллер анимаций, использующий для управления переходами между ними механизм Unity State Machine(машину состояний). Она реализует одноименный паттерн объекто-ориентированного программирования, объединяющий достоинства централизации логики переходов и паттерна State (локализация кода, зависящего от состояния в отдельных классах). Главная идея State Machine заключается в том, что анимируемый объект находится в определенном состоянии в любой момент времени. Такими состояниями могут быть положение покоя, бег, прыжок и другие действия. Чтобы перейти от одного состояния в другое, необходимо указать условие перехода посредством 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.

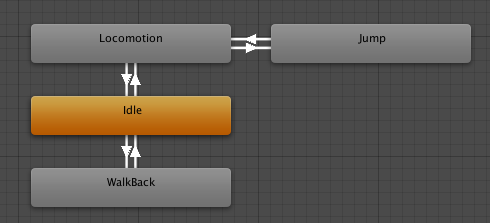
Для настройки этих состояний и переходов между ними используется окно Animator. Здесь визуализирован граф всех связей внутри контроллера анимаций, а также условия переходов между разными записями, скорость перехода и другое.

Рисунок 1.8 – 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(тип для «правда-ложь») срабатывает при значении равном либо true, либо false, в то время как Trigger позволяет сразу активировать переход при вызове функции SetTrigger внутри скрипта.

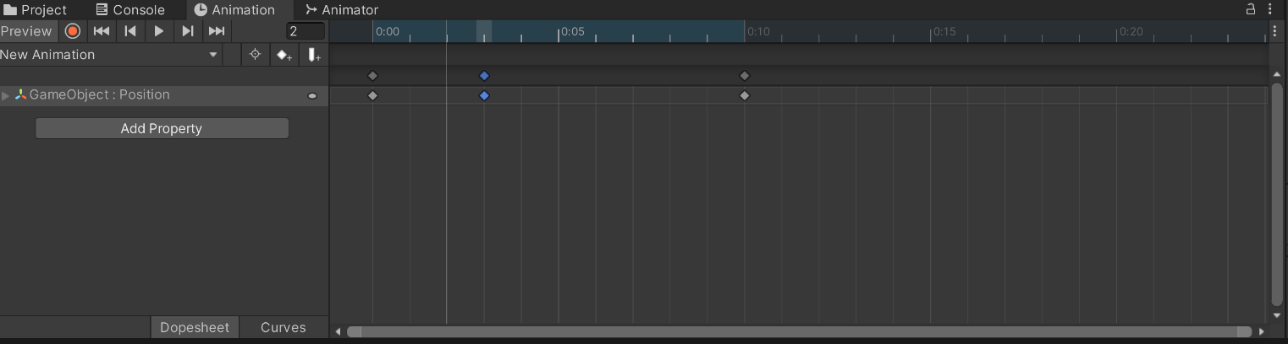
Для записи анимации внутри редактора Unity, используется окно Animation. При этом необходимо добавить только ключевые кадры анимации, все остальное будет просчитано и добавлено платформой Unity автоматически. Кадры анимации располагаются на временной шкале, определяющей, в какой момент времени будет отображено новое состояние. Таким образом возможно контролировать протяженность записей.

Рисунок 1.9 – Окно Animator

Unity предоставляет возможность вызывать функции в любой момент анимации, например, в момент удара мечом. Выделив мышкой место на временной шкале и нажав на кнопку «Add event», создается событие, которое будет вызывать указанные функции, примененных к анимируемому объекту, в момент проигрывания анимации. Функции, которые упомянуты выше, должны быть публичными и принадлежать сценарию того же игрового объекта, чтобы все работало правильно.

Unity Asset Store – это площадка, где разработчики делятся компонентами игр, такими как картинки, модели, аудио-файлы, анимации и многое другое для использования в других проектах Unity. Unity Editor позволяет импортировать купленные пакеты в проект, добавляя их в файловую систему. После этого содержимым пакета можно воспользоваться внутри редактора.

При создании игры будет использоваться платформа для разработки видеоигр Unity в сочетании с магазином Unity Asset Store, откуда будут взяты изображения, анимации и аудио-файлы. 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.

Вывод из теоретической части какие технологии будешь использовать (Unity, Asset Store); какой будет игра (в кратце)  
  
В начале практической части: постановка задачи (сделать игру с 3-мя препятсвиями, 3-мя бонусами, 1-ой своей анимацией и т.д); системные требования, написать что десктопная игра;

Отдельно написать про правила игры как можно подробнее: какие бонусы, что делают, на сколько увеличивается скорость игрока со временем и т.п. Чтобы разные части игры не возникали внезапно и не начинались с реализации. Мои замечания насчет этого в других частях работы можно вынести сюда.

Глава 2 Проектирование и программная реализация видеоигры

1. Концепция игры

«Running Knight» – это 2D-игра, в которой игроку нужно преодолевать препятствия и противников, набирая при этом очки. Персонаж игрока движется с увеличивающейся со временем скоростью вправо, 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 появляются со случайным промежутком времени и случайным образом, создавая таким образом множество уникальных ситуаций, проверяющие мастерство и реакцию пользователя. Игрок соревнуется с самим собой на установление нового лучшего счета.

Были поставлены следующие задачи: с помощью инструментов игрового движка Unity и компонентов из Unity Asset Store создать двумерную бесконечную видеоигру с 2-мя ловушками, 1-им противником, 3-мя бонусами; нарисовать одну 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 10x64.

В игре главный персонаж движется автоматически по направлению оси X, при этом главная камера сцены следует за ним, чтобы игрок мог отслеживать его перемещение. 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. Плюс к этому за каждого поверженного противника начисляется фиксированное число баллов. Если в забеге игрок набрал результат, превышающий свой предыдущий рекорд, то после перезапуска игры новый лучший результат перезапишет старый.

Чтобы справляться с вражескими рыцарями, попадающимися у игрока на пути, персонаж имеет возможность атаковать противника обратно. При столкновении с ловушкой или при попадании удара противника теряет одну жизнь. Когда количество жизней игрока дойдет до нуля, персонаж погибает, и игра начинается заново.

Чтобы помочь игроку восстановиться после получения урона, была создана механика неуязвимости на короткое время после удара. Это значит, что 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.

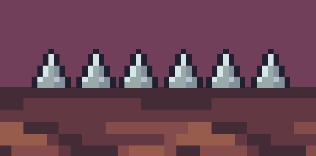
Ловушка Spikes(шипы) – это препятствие, которое располагается непосредственно на земле. Шипы необходимо перепрыгнуть, в противном случае игрок получает урон при касании к любому месту ловушки. При этом игрок может беспрепятственно проходить ловушку насквозь, не прекращая движения.

Рисунок 2.1 – Ловушка Spikes(шипы)

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

Противником в игре являются вражеские рыцари, которые нападают на героя, когда тот попадает в зону атаки. Игрок может либо перепрыгнуть через противника, либо атаковать его в ответ, поразив его перед тем, как тот нанесет ему урон.

Рисунок 2.2 –Вражеский рыцарь

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 3 поднятых предмета. При этом остальные бонусы будут пропадать, если инвентарь заполнен.

Поднимаемый предмет DestructionCollectible уничтожает все препятствия, находящиеся в данный момент в игровой сцене. После применения SteelShoesCollectible игрок перестает получать урон от шипов на некоторое время. HealthCollectible добавляет одно очко жизни игроку, применятся автоматически.

Рисунок 2.3 – Изображения бонусов(слева направо: DestructionCollectible, HealthCollectible, SteelShoesCollectible)

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 максимального 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(движение игрока) берутся ссылки на другие компоненты персонажа, такие как RigidBody2D для симуляции движения, Animator для проигрывания анимаций бега, кувырка и прыжка и BoxCollider2D для определения положения игрока относительно пола.

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, заданная заранее, что приводит тело в движение в направлении оси X:

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

Значение скорости увеличивается с каждым кадром, пока не достигнет максимального значения:

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. 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. Затем сравнением полученного объекта с null можно определить, находится ли игрок на земле или нет:

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, и, если в этот момент управляемый объект находится на земле, скорость тела по координатам y становится равным заданному значению числа jumpPower(сила прыжка). При этом играет анимация прыжка персонажа путем вызова функции SetTrigger("Jump") на контроллере анимаций:

private void Jump()

{

if (isGrounded())

{

anim.SetTrigger("Jump");

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

}

}

Высота прыжка будет зависеть от продолжительности нажатия Пробела игроком: если отпустить клавишу еще до того, как тело достигнет максимальной точки, то его скорость уменьшиться наполовину, чтобы игрок начал процесс падения быстрее.

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 Пробел в пределах небольшого временного диапазона после последнего совершенного прыжка.

Для совершения кувырка, применяемый для уворота от стрел, летящих слишком низко, необходимо нажать кнопку C клавиатуры. При этом размер BoxCollider2D игрока уменьшится вдвое по оси Y, а также запуститься соответствующая запись анимации кувырка. На последнем кадре анимации вызывается функция, возвращающая размеры в исходное состояние.

1. Создание игрового уровня

Для создания уровня был использован компонент Tilemap, встроенный в Unity Editor. Компонент Tilemap позволяет размещать изображения внутри сетки, которая создается компонентом Grid. Grid 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. Картинки, из которых будет состоять Tilemap, помещаются в Tile Pallette – инструмент, схожий по смыслу с палитрой и кисточкой. С помощью кисточки можно выбрать спрайт, находящийся в палитре, и начать размещать картинки внутри сетки. Такой способ разметки упрощает создание уровней и позволяет быстро создавать прототипы при создании 2D-миров внутри игры.

Чтобы движок рассчитывал поведение объектов при столкновении с плитками Tilemap, на карту необходимо добавить компонент Tilemap Collider 2D в сочетании 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 игрок передвигается. Земля в игре создается с помощью компонента Tilemap повторяющимися изображениями. Создать карту большой длины было бы непроизводительно и неэффективно, потому что отображаться в это время на экране будет малая часть от всей сетки.

Данная проблема решается созданием нового спрайта 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, впоследствии удаляются, установив нулевое значение на месте ненужных спрайтов.

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 другого метода, потому что провалы в земле, которые игроку необходимо перепрыгнуть, можно было заменить ловушкой Spikes.

1. Система здоровья и неуязвимости

Базовый абстрактный класс Health(здоровье) ответственен за систему здоровья внутри игры, а сценарии HealthPlayer(здоровье игрока) и HealthEnemy(здоровье противника) 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, которые являются сериализуемыми. Базовый класс объявляет, но не определяет метод TakeDamage(получить урон), так как реакция на получение урона у игрока и противника будет разной: например, при смерти игрока должны запускаться сценарии конца игры, в то время как смерть противника приводит к падению бонуса, который поможет при прохождении, и начислению очков игроку. Класс Health определяет метод AddHealth(добавить здоровье), 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 основном, возвращается объект, который отсчитывает время ожидания для выполнения остальной части кода после операторов yield return, например:

yield return new WaitForSeconds(5);

Затем сценарий выполняет остальной код после StartCoroutine. Когда проходит время ожидания для выполнения отложенной части программы, управление возвращается к следующей строке после оператора yield. На листинге продемонстрировано, как это было реализовано для создания неуязвимости в игре:

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 проигрываться запись анимации атаки игрока. В середине кадра в момент удара мечом происходит проверка, находится ли противник в пределах некоторой зоны перед игроком. Для этого перед игроком создается виртуальный прямоугольник, длину и отдаленность от игрока которого определяется разработчиком предварительно с помощью сериализованных полей range и colliderDistance внутри Unity Editor.

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 было необходимо было визуальное представление зоны атаки игрока, поэтому была применена функция OnDrawGizmos, которая выполнила это требование. Gizmos – это инструмент Unity Editor, предоставляющий возможность разработчику визуализировать компоненты объекта, такие как BoxCollider2D, во время запуска игры. Это помогло при определении оптимального расстояния, в пределах которых противник будет получать урон.

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)

);

}

1. Программная реализация препятствий и противников

Нанесение урона игроку посредством препятствий реализовано с помощью сценария EnemyDamage(урон противника). 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(является переключателем) внутри компонента BoxCollider2D, которое не будет препятствовать прохождению других тел через объект. 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(Игрок), 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: сама ловушка и стрела. К стреле применен сценарий EnemyProjectile(вражеский снаряд), который наследуется от EnemyDamage, чтобы в последствии наносить урон игроку при попадании. Метод ActivateProjectile(активировать снаряд) 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 об столкновении.

Ловушка не имеет визуального представления в мире игры, она служит как источник выстрела стрелы. Сразу же после создания объекта в мире игры, она выпускает стрелу через функцию Attack(атаковать) сценария ArrowTrap(ловушка со стрелой). 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.

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();

}

Противники для нанесения урона игроку используют сценарий MeleeEnemy(рукопашный бой противника), который функционирует схожим образом, как и сценарий атаки игрока. Вместо нажатия кнопки Левой кнопки мыши, катализатором для начала атаки для противника является вход игрока внутрь виртуального прямоугольника перед противником. То есть противник будет реагировать на появление игрока в «зоне видимости», чтобы начать нападение.

1. Создание игровых систем

За управление системой создания новых препятствий перед игроком ответственен объект SpawnManager(управляющий созданием), который является пустым объектом с единственным компонентом, называющимся идентично. Функционирует это следующим образом: внутри Update запускается таймер. Когда время превышает 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. Затем 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, прошедшую со времени последнего кадра игры.

Для проигрывания всех звуков был создан специальный объект с компонентом AudioSource. Этот объект, под названием SoundManager(управляющий звуками), станет источником всех звуков в игре. Сценарий SoundManager предоставляет публичный метод для воспроизведения звуков – PlaySound, принимающий аудиодорожку и тут же ее запускающий. 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 между переходом между игрой и главным меню.

Объект GameManager(управляющий игрой) 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 состояние, как 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 файла расширения json, в котором будут находится данные игры. При первом запуске игры 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 поле объекта класса Data bestScore на новое значение; если объекта еще не существует, то метод его создает. GetGameData тоже создает объект для хранения данных, если это не было сделано ранее, и возвращает его.

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 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. Разработка бонусов

Объекты поднимаемых предметов при касании с игроком, должны либо моментально примениться, либо добавиться в инвентарь игрока. Общим свойством, которым обладают все поднимаемые предметы, является уничтожение объекта, если его не подняли в течении некоторого времени. Это 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 кнопку F(или А при русской раскладке) на клавиатуре, отвечающая за применение бонуса. Все предметы находятся в очереди Queue<CollectibleTypes> collected, ограниченного сверху заданным числом. Структура данных Очередь, или еще называемое 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 очереди.

Первый попавший в список предмет будет использован в первую очередь. Поэтому после нажатия кнопки, switch-выражение проверяет значение первого 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 ней функцию emptySpawnedList(опустошить список), уничтожающая все объекты преград. На экране при этом проигрывается анимация удара молнии на белом фоне, чтобы преграды не исчезали на глазах у игрока. После этого из списка доступных бонусов убирается первый элемент. При этом содержимое «инвентаря» обновляется на экране.

case CollectibleTypes.Destruction:

SpawnManager spawnManager = (SpawnManager)FindAnyObjectByType(typeof(SpawnManager)); SoundManager.instance.PlaySound(lightningSound);

gameManager.PlayLightningAnim();

spawnManager.emptySpawnedList();

ui.RefreshUIList();

break;

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

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 и так далее, показать напрямую в игровом мире зачастую невозможно или неудобно. Для этой цели служит специальный экран, который накладывается поверх игрового процесса, называемый UI(user interface) или пользовательский интерфейс.

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, то есть чтобы разместить элемент в углу интерфейса, достаточно просто перетащить его туда. Это достигается тем, что каждый объект интерфейса Unity имеет компонент RectTransform, который имеет положение, масштаб и угол поворота, как и обычный Transform, но в дополнению к ним добавлены значения ширины и высоты, определяющие размерность прямоугольника, внутри которого будет находиться текст, изображения и так далее.

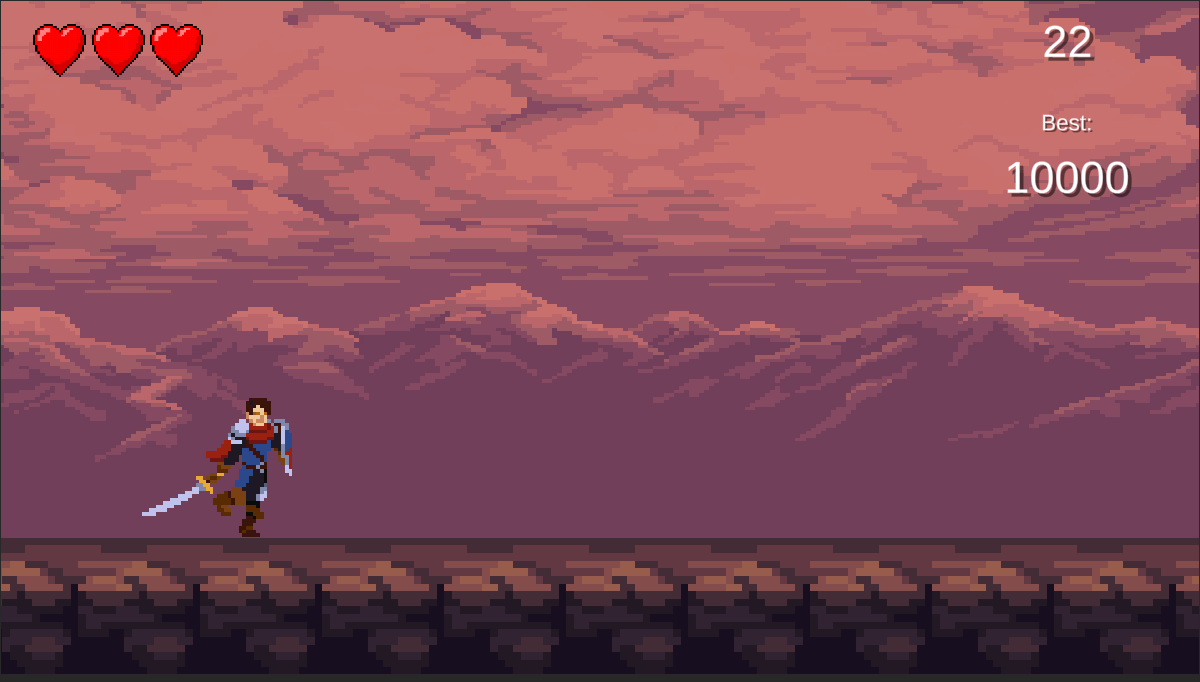
Интерфейс “Running Knight” во время игрового процесса выглядит следующим образом:

Рисунок 2.4 –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 картинок. Чтобы отобразить в этих элементах список бонусов, проходим циклом по всем доступным картинкам, для каждой картинки устанавливая изображение бонуса, который находится в “инвентаре” игрока. Если бонусов меньше, чем максимально возможное количество, то для картинок, которым не досталось своего бонуса, устанавливается пустое значение.

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

{

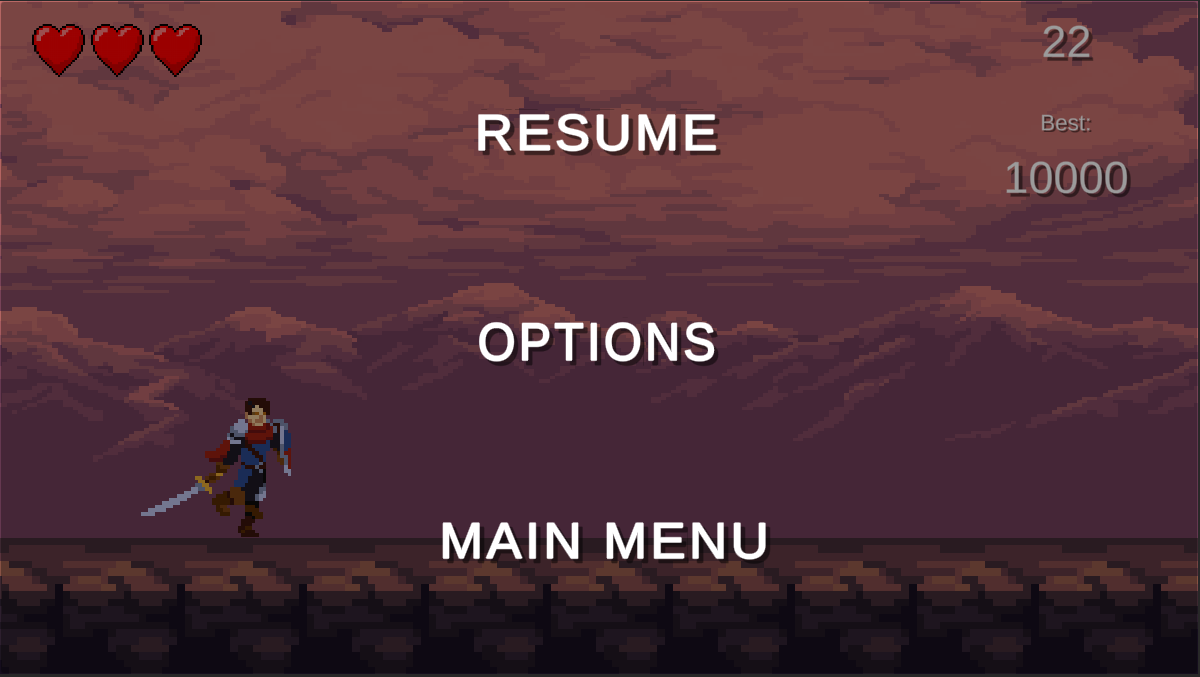
imageBuffer.enabled = false;

imageBuffer.sprite = null;

}

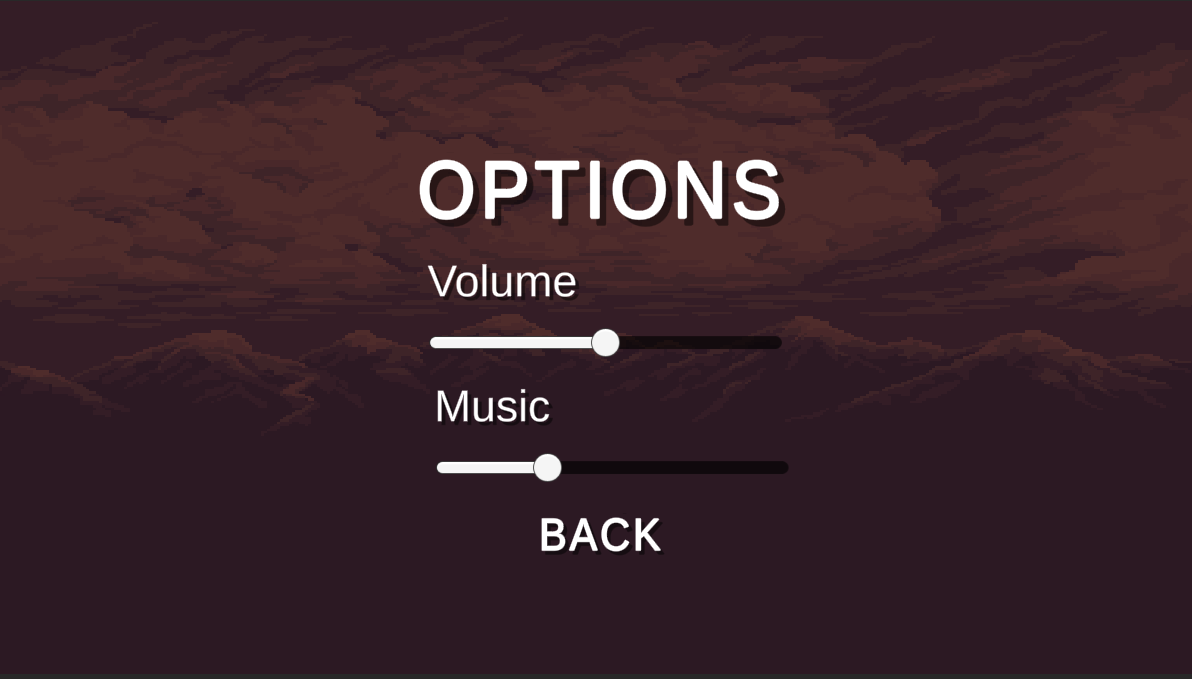
}

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

Рисунок 2.5 – Меню паузы

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.

Рисунок 2.6 – Главное меню

Рисунок 2.7 – Меню настроек звука

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

1. Создание анимации

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

Покадровая анимация удара молнии при использовании бонуса разрушения в игре была нарисована специально для игры посредством веб-приложения PixilArt. Этот сайт позиционируется как социальная платформа, где художники могли поделиться 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 все воедино. Можно выбрать различные цветовые палитры из встроенных или создать их самому. PixilArt дает возможность создавать анимации, позволяя нарисовать каждый кадр по отдельности, которые можно затем импортировать уже готовой записью в файле с расширением gif, или расположить каждый кадр друг за другом на одном изображении, что облегчает процесс переноса анимации для реализации внутри платформ для создания игр, например, Unity.

Заключение. Ответ на задачи, поставленные во введении и в начале практической части. Объем >= 0.5 страницы Можно еще написать про даьнейшее развитие игры (новые бонусы/уровни, монетизация и т.д (не обязательно)) если про это напишешь здесь допиши это в основной текст работы, только подробнее

Приложения: ссылка на исходный код. Как можно больше скриншотов игры. Все визуальные составляющие поотдельности: фон, персонажи, но особенно важно то что делал сам (раскладка твоей анимации). Если не описываешь приложение в тексте,а просто «внешний вид персонажа можно увидеть здесь», выноси это в приложения. С кодом так же. И с таблицами

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.
12. Video Gaming Industry & Its Revenue Shift [Electronic resource] – Mode of access: <https://www.forbes.com/sites/ilkerkoksal/2019/11/08/video-gaming-industry--its-revenue-shift/> – Date of access: 13.03.2023.

Приложения

Приложение А

C:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Death.pngC:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Roll.pngC:\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-Run.pngC:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Idle.pngКадры анимации игрового персонажа

Рисунок А7 – анимация смерти

Рисунок А6 – анимация кувырка

Рисунок А5 – анимация падения

Рисунок А4 – анимация прыжка

Рисунок А2 – анимация бега

Рисунок А1 – анимация положения покоя

C:\Users\AdminNB\AppData\Local\Microsoft\Windows\INetCache\Content.Word\HeroKnight-Attack.png

Рисунок А3 – анимация удара

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

Рисунок А8 – анимация получения урона

Приложение Б

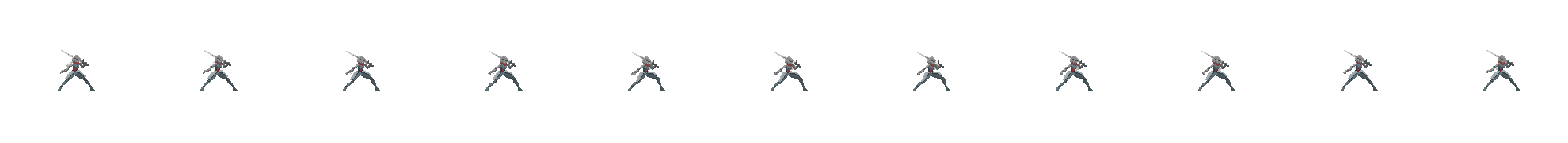
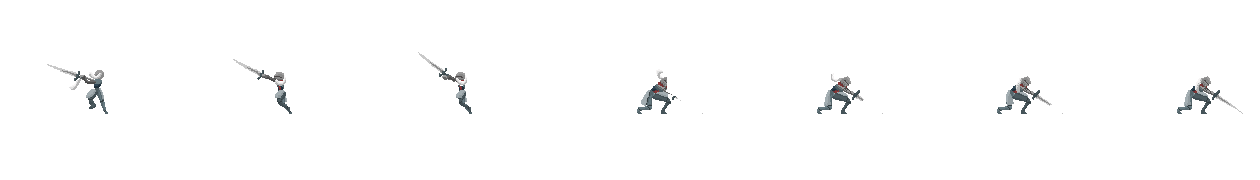
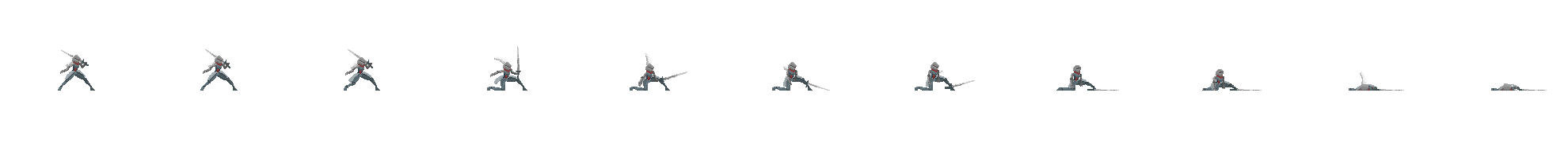
Изображения и анимации противника

Рисунок Б2 – анимация смерти

Рисунок Б2 – анимация удара

Рисунок Б1 – анимация положения покоя

Приложение B

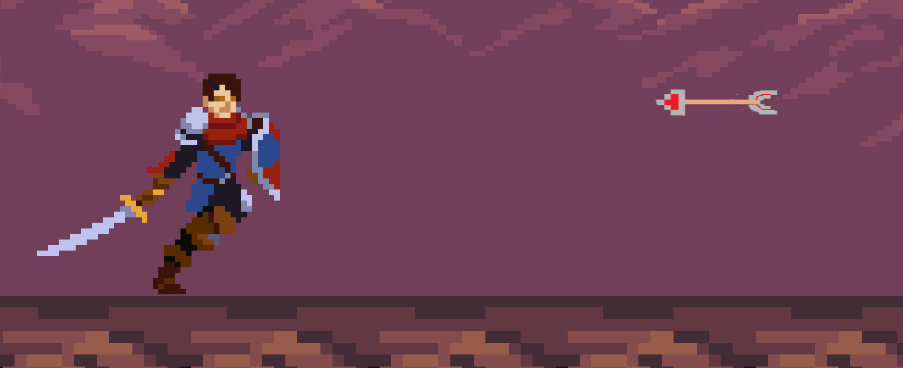
Изображения разных уровней полета стрелы

Рисунок В2 – полет стрелы на уровне с головой игрока

Рисунок В1 – полет стрелы над головой игрока

Приложение Г

Изображение зоны атаки персонажа и фоновое изображение уровня игры

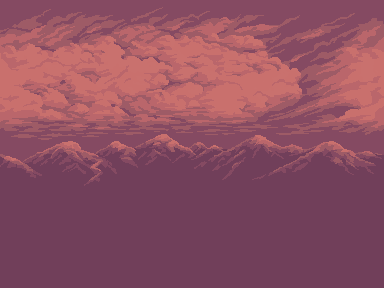


Рисунок Г2 – фоновое изображение уровня игры

Рисунок Г1 – зона атаки персонажа

Приложение Д

Схема управления

Таблица Д1 – схема управления

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

Приложение Е

Ссылка на исходный код проекта

https://github.com/SirGeckT0R/CourseworkUnity