**Реферат**

ТРЕХМЕРНАЯИГРА *«MYTHICAL* *TOWERS*» ОТ ТРЕТЬЕГО ЛИЦА В ЖАНРЕ ЗАЩИТА БАШНИ С ЭЛЕМЕНТАМИ RPG НА ПЛАТФОРМЕ *UNITY* ПОД ОС *ANDROID*: дипломная работа / Е. С. Паньков – Гомель : ГГТУ им. П. О. Сухого, 2022. – Дипломная работа: 93 страницы, 42 рисунка, 15 таблиц, 13 источников, шесть приложений.

Ключевые слова: игра, приложения, мобильная, защита, башня.

Объектом разработки является мобильное игровое приложение в жанре «защиты башни».

Целью работы является разработка игрового приложения в жанре «защита башни».

Характеристика проделанной работы: при выполнении работы рассмотрены и проанализированы имеющиеся игровые приложения. Проведен обзор подобных приложений, после чего сформирован список требований, которым должен соответствовать разрабатываемый программный продукт. При разработке использованы объектно-ориентированный и структурный подходы программирования, паттерны проектирования. В результате разработки спроектировано и реализовано мобильное игровое приложение в жанре защита башни.

Дипломная работа выполнена самостоятельно, приведенный в дипломной работе материал объективно отражает состояние разрабатываемого объекта, пояснительная записка проверена в системе «Антиплагиат.ру» (ссылка на систему: *https://www.antiplagiat.ru/*). Процент оригинальности составляет 90. Все заимствованные из литературных и других источников теоретические и методологические положения и концепции сопровождаются ссылками на источники, которые указаны в «Списке использованных источников».

**Резюме**

Темой работы является «Трехмернаяигра *«Mythical* *Towers*» от третьего лица в жанре «защита башни» с элементами *RPG* на платформе *Unity* под ОС *Android*».

Объектом исследования является мобильное игровое приложения.

Целью работы является разработка игрового мобильного приложения в жанре защита башни с элементами *RPG*.

Основным результатом работы является, мобильное игровое приложение приложение.

**Рэзюме**

Тэмай працы з'яўляецца "Трохмерная гульня "Mythical Towers" ад трэцяй асобы ў жанры абарона вежы з элементамі RPG на платформе Unity пад АС Android".

Аб'ектам даследавання з'яўляецца задача стварэння мабільнага прыкладання.

Мэтай працы з'яўляецца распрацоўка гульнявога мабільнага прыкладання ў жанры абарона вежы.

Асноўным вынікам працы з'яўляецца справаздача аб праведзенай працы, мабільнае прыкладанне.

**Abstract**

The theme of the work is "Three-dimensional game "Mythical Towers" from a third person in the genre of tower defense with RPG elements on the Unity platform for Android OS".

The object of research is the task of creating a mobile application.

The aim of the work is to develop a gaming mobile application in the tower defense genre.

The main result of the work is a report on the work done, a mobile application.

**ОГЛАВЛЕНИЕ**

[Введение 7](#_Toc106303913)

[1 Средства разработки интерактивных приложений, жанры компьютерных игр 8](#_Toc106303914)

[1.1 Обзор игрового движка *Unity* 8](#_Toc106303915)

[1.2 Обзор игрового движка *Unreal* *Engine* 11](#_Toc106303916)

[1.3 Сравнение игровых движков *Unity* и *Unreal* *Engine* 4 13](#_Toc106303917)

[1.4 Обзор игр жанра «*Tower* *Defense*» 16](#_Toc106303918)

[2 Внутренняя архитектура и информационная модель данных   
разрабатываемого игрового приложения «*Mythical* *Towers*» 21](#_Toc106303919)

[2.1 Архитектура и взаимосвязь модулей игрового приложения «*Mythical* *Towers*» 21](#_Toc106303920)

[2.2 Информационная модель данный игрового приложения «*Mythical* *Towers*» 24](#_Toc106303921)

[3 Структура игрового приложения «*Mythical* *Towers*» 31](#_Toc106303922)

[3.1 Игровые механики, реализованные в игровом приложении «*Mythical* *Towers*» 31](#_Toc106303923)

[3.2 Структура классов игровой логики игрового приложения «*Mythical* *Towers*» 34](#_Toc106303924)

[3.3 Игровой интерфейс игрового приложения «*Mythical* *Towers*» 39](#_Toc106303925)

[4 Верификация и опытная эксплуатация игрового приложения «*Mythical* *Towers*» 41](#_Toc106303926)

[4.1 Принципы работы пользовательского графического интерфейса и основные механики игрового приложения «*Mythical* *Towers*» 41](#_Toc106303927)

[4.2 Модульное тестирование ключевых компонентов игрового   
приложения «*Mythical* *Towers*» 45](#_Toc106303928)

[5 Экономическое обоснование дипломной работы 51](#_Toc106303929)

[5.1 Обоснование целесообразности разработанного программного продукта 51](#_Toc106303930)

[5.2 Оценка конкурентоспособности программного продукта 51](#_Toc106303931)

[5.3 Оценка трудоемкости работ по созданию программного продукта 52](#_Toc106303932)

[5.4 Расчет затрат на разработку программного продукта 55](#_Toc106303933)

[5.5 Расчет частных экономических эффектов от производства и 61](#_Toc106303934)

[использования программного продукта 61](#_Toc106303935)

[6 Охрана труда и техника безопасности 62](#_Toc106303936)

[6.1 Факторы, влияющие на тяжесть поражения электрическим током 62](#_Toc106303937)

[7 Энергосбережение и ресурсосбережение при эксплуатации программного обеспечения 66](#_Toc106303938)

[7.1 Энергосбережение и ресурсосбережение при внедрении и   
эксплуатации программного обеспечения 66](#_Toc106303939)

[7.2 Вопросы энерго- и ресурсосбережения, связанные с эксплуатацией](#_Toc106303940) [интерактивной системы 67](#_Toc106303941)

[Заключение 69](#_Toc106303942)

[Список Использованных Источников 70](#_Toc106303943)

[Приложение А Листинг программы 71](#_Toc106303944)

[Приложение Б Руководство системного программиста 85](#_Toc106303945)

[Приложение В Руководство программиста 86](#_Toc106303946)

[Приложение Г Руководство пользователя 87](#_Toc106303947)

[Приложение Д Результаты расчетов экономического обоснования 89](#_Toc106303948)

[Приложение Е Список публикаций 93](#_Toc106303949)

# **ВВЕДЕНИЕ**

Игровая индустрия не стоит на месте и постоянно развивается. Очень много людей играет в видеоигры по всему миру. Существует огромное множество разновидностей жанров видеоигр, поэтому очень много людей в той или иной игре находят что-то для себя интересное. Особенно сильно влияние игр на молодое поколение. Пока человек развивается, он ищет для себя что-то новое. В настоящее время широкую популярность приобрели игры на мобильных платформах. Сейчас практически у каждого человека есть телефон или планшет.

Начало компьютерных игр можно отнести к 1950-м годам, когда разрабатывались простые игры в рамках диссертаций. Так в 1952 году А. Дуглас написал игру «крестики-нолики». Но первая коммерческая игра была разработана в 1970-х годах под названием «*Pong*». В скором времени она заполучила огромную популярность.

Самым главным смыслом создания компьютерной игры является получение возможности опробовать разные характеристики, которые игрок в идеале хотел бы иметь. Хорошо разработанная компьютерная игра может способствовать вовлеченности и создавать эффективную среду обучения. В некоторых отношениях компьютерные игры могут преподавать ценные уроки и могут даже помочь улучшить умственные способности игрока.

Игры отражают то, как человеческий разум был разработан, чтобы учиться. Они мотивируют игроков принимать риски и действия, упорствовать через неудачи, набор и достижения более сложных целей, и уделять внимание, время и усилия для приобретения знаний и навыков. Все это в то время, как игра отслеживает действия игрока и оценивает достижения игрока и навыки.

Целью дипломной работы является разработка игрового приложения в жанре «*Tower* *defense* ». Игра будет реализована от третьего лица. Игра измеряет мастерство игрока, а также уровень его сообразительности.

Актуальность жанра «*Tower* *defense»* заключается в том, что игры подразумевают простое, интуитивно-понятное управление игровыми объектами. Пользователю нет необходимости настраивать и изучать управление игры, что позволяет практически сразу приступить к игровому процессу.

# **СРЕДСТВА РАЗРАБОТКИ ИНТЕРАКТИВНЫХ ПРИЛОЖЕНИЙ, ЖАНРЫ КОМПЬЮТЕРНЫХ ИГР**

* 1. **Обзор игрового движка *Unity***

*Unity* – это кроссплатформенный игровой движок, разработанный *Unity* *Technologies*, впервые анонсированный и выпущенный в июне 2005 года на Всемирной конференции разработчиков *Apple* *Inc*. в качестве эксклюзивного игрового движка для *Mac* *OS* *X* [1, с. 10]. С тех пор движок постепенно расширялся для поддержки различных настольных, мобильных, консольных платформ и платформ виртуальной реальности. *Unity* особенно популярен для разработки мобильных игр для *iOS* и *Android* и используется для таких игр, как *Pokémon* *Go*, *Monument* *Valley*, *Call* *of* *Duty*: *Mobile*, *Beat* *Saber* и *Cuphead*. *Unity* считается простым в использовании для начинающих разработчиков и популярен для разработки инди-игр (рисунок 1.1).



Рисунок 1.1 – Среда разработки *Unity*

Движок можно использовать для создания трехмерных (3*D*) и двумерных (2*D*) игр, а также интерактивных симуляций и других впечатлений. Движок был принят на вооружение отраслями, не относящимися к видеоиграм, такими как кино, автомобилестроение, архитектура, машиностроение, строительство и Вооруженные силы Соединенных Штатов.

Исходный код *Unity* на *C*# был опубликован под лицензией «только для ссылок» в марте 2018 года, которая запрещает повторное использование и модификацию.

По состоянию на 2020 год программное обеспечение, созданное с использованием игрового движка *Unity*, работало на более чем 1,5 миллиардах устройств. По данным *Unity*, приложения, созданные на их игровом движке, составляют 50 процентов всех мобильных игр и загружаются более 3 миллиардов раз в месяц, а с помощью их программного обеспечения ежедневно запускается около 15 000 новых проектов. *Financial* *Times* сообщила, что движок *Unity* «поддерживает некоторые из самых прибыльных мобильных игр в мире», такие как *Pokémon* *Go* и *Activision* *Call* *of* *Duty* *Mobile*.

В июне 2020 года *Unity* представила студию смешанной и дополненной реальности (*MARS*), которая предоставляет разработчикам дополнительные функциональные возможности для создания приложений дополненной реальности (*AR*) на основе правил. 9 декабря 2020 года *Unity* выпустила *Unity* *Forma*, инструмент для решения автомобильных и розничных задач.

*Unity* приобрела *Finger* *Food* *Advanced* *Technology* *Group* в 2020 году, поскольку стремилась расширить свое использование закусок, не связанных с видеоиграми, и предложить клиентам дополнительную помощь в разработке дизайна. Компания стала публичной в сентябре 2020 года, чтобы еще больше расширить использование своего игрового движка в отраслях, не связанных с играми.

*Unity* 2021 принесла множество новых функций, таких как официальный инструмент визуальных сценариев в *Unity*, новая многопользовательская библиотека для поддержки многопользовательских игр, улучшенная производительность *Il*2*cpp* во время выполнения, объемные облака для конвейера рендеринга высокой четкости. Кэширование теней и глобальная подсветка экранного пространства для *HDRP*. Для универсального конвейера рендеринга были добавлены новые функции, такие как точечные световые тени, Отложенный рендеринг и общие улучшения, и исправления основного движка.

В декабре 2021 года *Unity* приобрела инструменты, конвейер, технологии и инженерные таланты *Weta* *Digital* Питера Джексона за 1,625 миллиарда долларов США наличными и акциями.

В 2022 году *Unity* купила *Ziva* *Dynamics*, технологическую компанию, специализирующуюся на сложных симуляциях и создании персонажей в реальном времени [6, с. 123]. О приобретении было объявлено цифровым человеком, работающим в движке.

Основными преимуществами *Unity* являются наличие визуальной среды разработки, межплатформенной поддержки и модульной системы компонентов. Визуальный рабочий процесс – достаточно уникальная вещь, выделяющая *Unity* из большинства сред разработки игр. Альтернативные инструменты разработки зачастую представляют собой набор разрозненных фрагментов, требующих контроля, а в некоторых случаях библиотеки, для работы с которой нужно настраивать собственную интегрированную среду разработки (*Integrated* *Development* *Environment*, *IDE*), цепочку сборки и прочее в этом роде. В *Unity* же рабочий процесс привязан к тщательно продуманному визуальному редактору. Именно в нем будут компоноваться сцены будущей игры, связывая игровые ресурсы и код в интерактивные объекты. Он позволяет быстро и рационально создавать профессиональные игры, обеспечивая невиданную продуктивность разработчиков и предоставляя в их распоряжение исчерпывающий список самых современных технологий в области видеоигр.

К недостаткам относят появление сложностей при работе с многокомпонентными схемами и затруднения при подключении внешних библиотек. Все библиотеки, могут быть задействованы, вручную копируются в проект, вместо того чтобы дать ссылку на одну папку общего доступа. Отсутствие единой папки с библиотеками затрудняет коллективное использование функционала разными проектами [1, с. 20-21].

На *Unity* написаны тысячи игр, приложений и симуляций, которые охватывают множество платформ и жанров. При этом *Unity* используется как крупными разработчиками, так и независимыми студиями

Редактор *Unity* имеет простой *Drag*&*Drop* интерфейс, который легко настраивать, состоящий из различных окон, благодаря чему можно производить отладку игры прямо в редакторе (рисунок 1.2). Движок поддерживает два скриптовых языка: *C*#, *JavaScript* (модификация). Ранее была поддержка *Boo* (диалект *Python*), но его убрали в 5-й версии. Расчёты физики производит физический движок *PhysX* от *NVIDIA*.

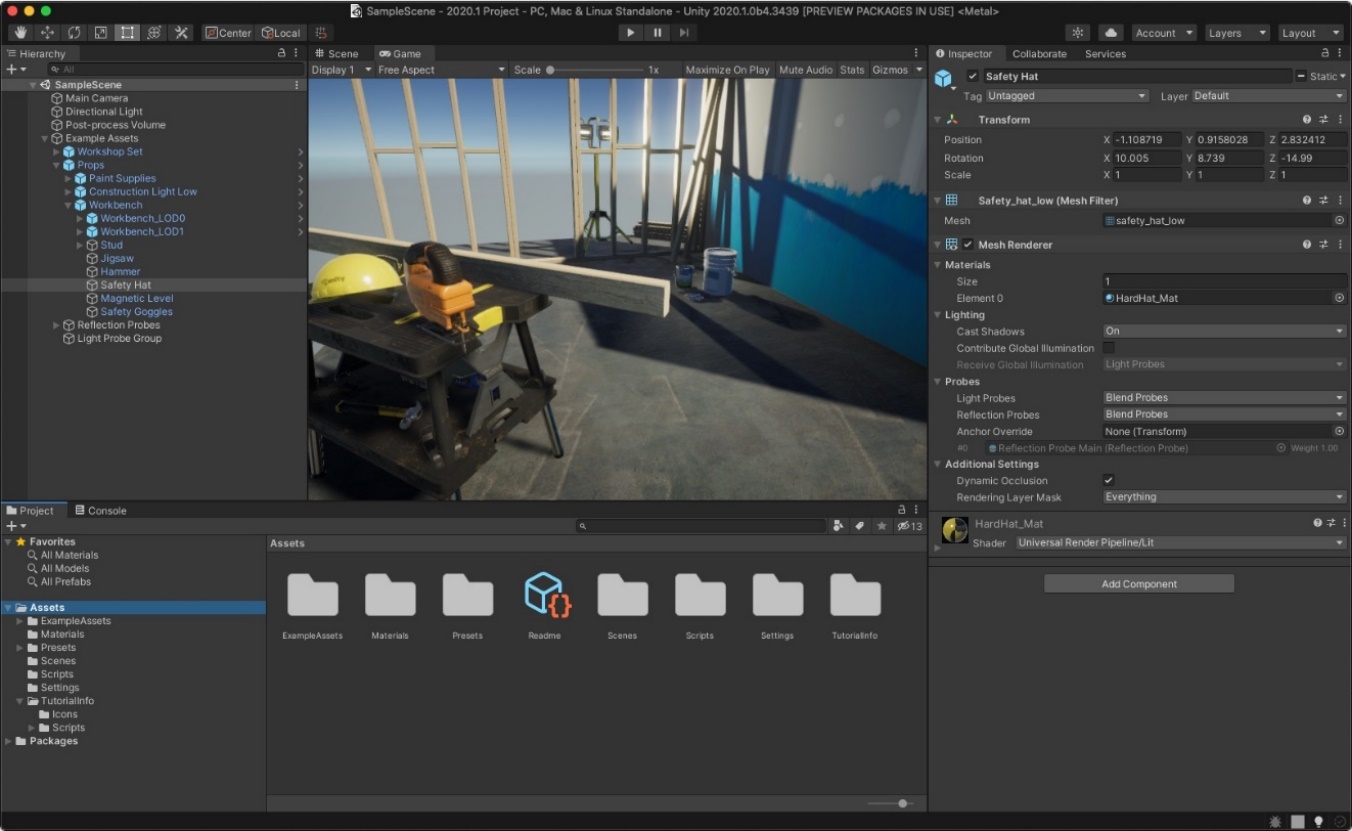


Рисунок 1.2 – Интерфейс *Unity*

Проект в *Unity* делится на сцены (уровни) – отдельные файлы, содержащие свои игровые миры со своим набором объектов, сценариев, и настроек. Сцены могут содержать в себе как, собственно, объекты (модели), так и пустые игровые объекты – объекты, которые не имеют модели («пустышки»). Объекты, в свою очередь содержат наборы компонентов, с которыми и взаимодействуют скрипты. Также у объектов есть название (в *Unity* допускается наличие двух и более объектов с одинаковыми названиями), может быть тег (метка) и слой, на котором он должен отображаться. Так, у любого объекта на сцене обязательно присутствует компонент *Transform* – который хранит в себе координаты местоположения, поворота и размеров объекта по всем трём осям. У объектов с видимой геометрией также по умолчанию присутствует компонент *Mesh* *Renderer*, делающий модель объекта видимой.

* 1. **Обзор игрового движка *Unreal* *Engine***

*Unreal* *Engine* (*UE*) – игровой движок, разработанный *Epic* *Games*, профессиональный набор инструментов для разработки видеоигр, который содержит в себе современный рендер физически корректных материалов в реальном времени, отражений и освещения с мощными инструментами для создания непередаваемого интерактивного опыта [5, с. 20-21]. Инструменты включают в себя симуляторы физики, освещения и теней, пользовательских интерфейсов, листвы и рендеринга, массивных террейнов, сложных материалов, визуальных скриптов, анимации персонажа, моделирование системы частиц, кинематики, мультиплеера.

Впервые представленный в шутере от первого лица *Unreal* 1998 года (рисунок 1.3). Первоначально разработанный для компьютерных шутеров от первого лица, он с тех пор использовался в различных жанрах трехмерных (3*D*) игр и был принят другими отраслями, в первую очередь киноиндустрией и телевизионной индустрией. Написанный на *C*++ движок *Unreal* *Engine* отличается высокой степенью переносимости, поддерживая широкий спектр настольных, мобильных, консольных платформ и платформ виртуальной реальности.

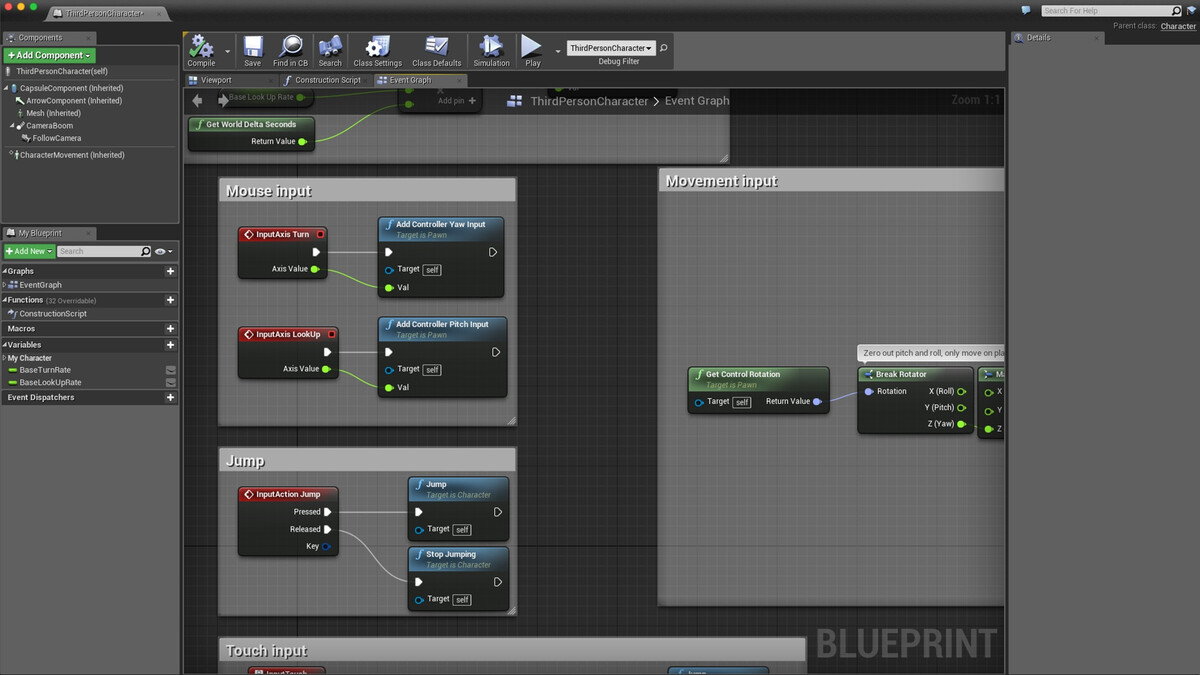


Рисунок 1.3 – Среда разработки *Unreal* *Engine*

Система визуальных сценариев *Blueprint* в *Unreal* *Engine* представляет собой полную систему сценариев игрового процесса, основанная на концепции использования интерфейса на основе узлов для создания элементов игрового процесса из *Unreal* *Editor*. Как и во многих общих языках сценариев, он используется для определения объектно-ориентированных (OO) классов или объектов в двигателе.

Эта система чрезвычайно гибкая и мощная, поскольку дает дизайнерам возможность использовать практически полный спектр концепций и инструментов, как правило, доступны только для программистов. Кроме того, *Blueprint*, доступный в реализации *Unreal* *Engine* *C* ++, позволяет программистам создавать базовые системы, которые могут быть расширены дизайнерами. Классы *Blueprint* идеально подходят для создания интерактивных активов, таких как двери, коммутаторы, коллекционные предметы и разрушаемые пейзажи.

*Unreal* *Engine* 5 был представлен 13 мая 2020 года и поддерживает все существующие системы, включая консоли следующего поколения *PlayStation* 5 и *Xbox* *Series* *X*/*S*. Работа над движком началась примерно за два года до его анонса. Он был выпущен в раннем доступе 26 мая 2021 года и официально запущен для разработчиков 5 апреля 2022 года.

Одной из его основных функций является *Nanite*, движок, который позволяет импортировать в игры исходный фотографический материал с высокой детализацией. Технология *Nanite* *virtualized* *geometry* позволяет *Epic* воспользоваться своим прошлым приобретением *Quixel*, крупнейшей в мире библиотеки фотограмметрии по состоянию на 2019 год.

Целью *Unreal* *Engine* 5 было максимально упростить для разработчиков создание детализированных игровых миров без необходимости тратить чрезмерное время на создание новых детализированных ресурсов. *Nanite* может импортировать практически любое другое ранее существовавшее трехмерное представление объектов и сред, включая модели *ZBrush* и *CAD*, обеспечение возможности использования ресурсов пленочного качества. *Nanite* автоматически обрабатывает уровни детализации (*LOD*) этих импортированных объектов, соответствующие целевой платформе и расстоянию отрисовки., задача, которую в противном случае пришлось бы выполнять художнику.

*Lumen* – это еще один компонент, описываемый как «полностью динамичное глобальное решение для освещения, которое немедленно реагирует на изменения сцены и освещения». *Lumen* устраняет необходимость для художников и разработчиков создавать световую карту для заданной сцены, но вместо этого вычисляет отражения света и тени на лету, тем самым обеспечивая поведение источников света в реальном времени.

Виртуальные карты теней – компонент, добавленный в *Unreal* *Engine* 5, описываемый как «новый метод отображения теней, используемый для обеспечения согласованного, затенение с высоким разрешением, которое работает с объектами качества фильма и большими, динамически освещенными открытыми мирами». Виртуальные карты теней отличаются от обычной реализации карты теней чрезвычайно высоким разрешением, более детализированными тенями и отсутствием появляющихся и исчезающих теней, которые можно найти в более распространенных техника теневых карт благодаря теневым каскадам. Дополнительные компоненты включают *Niagara* для динамики жидкостей и частиц и *Chaos* для физического движка.

*Epic* разработала *Unreal* *Engine* 5 для работы с десятками миллиардов полигонов, присутствующих на одном экране с разрешением 4*K*, чтобы воспользоваться преимуществами будущих высокоскоростных решений для хранения данных с консольным оборудованием следующего поколения, которое будет использовать сочетание оперативной памяти и пользовательских твердотельных накопителей. *Epic* тесно сотрудничает с *Sony* в оптимизации *Unreal* *Engine* 5 для *PlayStation* 5, также *Epic* сотрудничал с *Sony* над архитектурой хранилища консоли. Чтобы продемонстрировать легкость создания детализированного мира с минимальными усилиями, на презентации движка в мае 2020 года была продемонстрирована демонстрация под названием «*Lumen* *in* *the* *Land* *of* *Nanite*», работающая на *PlayStation* 5, которая была создана в основном за счет извлечения ресурсов из библиотеки *Quixel* и использования *Nanite*, *Lumen* и других компонентов *Unreal* *Engine* 5 для создания фотореалистичной обстановки пещеры, которую можно исследовать.

*Epic* подтвердила, что *Unreal* *Engine* 5 также будет полностью поддерживаться на *Xbox* *Series* *X*, но во время анонса они были сосредоточены на *PlayStation* 5 в результате их работы с *Sony* в предыдущие годы. *Epic* использует Fortnite в качестве испытательного стенда для *Unreal* *Engine* 5, чтобы продемонстрировать, что движок может сделать для индустрии, с игрой, в которой будет использоваться *Unreal* *Engine* 5 в декабре 2021 года. *Senua's* *Saga* от *Ninja* *Theory*: *Hellblade* II также одна из первых игр, использующих *Unreal* *Engine* 5. *The* *Matrix* *Awakens*, объединяющий опыт перед выпуском *The* *Matrix* *Resurrections*, был разработан *Epic* для дальнейшей демонстрации *Unreal* *Engine* 5 наряду с другими технологиями, которые они приобрели в течение 2020 и 2021 годов, включая их создателя-метачеловека, разработанного и интегрированного в *Unreal* *Engine* 5 с использованием технологий 3*Lateral*, *Cubic* *Motion* и *Quixel*.

* 1. **Сравнение игровых движков *Unity* и *Unreal* *Engine* 4**

Разница между *Unity* и *Unreal* *Engine* фундаментально огромная. И в техническом плане, и в лицензионном, и в финансовом, и в сообществе.

*Unreal* *Engine* появился в 1998 году. В те времена еще не было даже понятия «движок». *Epic Games* делал игру, которая должна была использовать самые современные возможности компьютерного железа. В этом плане философия остается такой до сих пор – выжать максимум из железа.

*Unity* в этом плане появился в 2005 году – чтобы делать игры на компьютерах от *Apple* и для компьютеров от *Apple*. И на волне смартфонов *Unity* быстро оседлали волну технологического прогресса под лозунгом «чем больше платформ, тем лучше».

Компания *Epic* *Games* зарабатывает тем, что делает и продает игры. Компания *Unity* *Technologies* зарабатывает тем, что делает и продает игровой движок. Это объясняет и разницу между экосистемами.

*Unity* – движок для всех. Компания предоставляет кучу сопутствующих услуг: формальное обучение и сертификаты, соответственную аналитику и рекламу.

*Epic* *Games* – движок для выжимания технологического максимума. С продвижением в массы у них поскромнее, но зато есть гранты. Они просто дают денег тем, кто им понравится.

В плане работы *Unity* сделан простым, а *Unreal* *Engine* – эффективным. Это прослеживается в работе с файлами, в том, как пишется игровая логика, даже в том, как выглядит стартовый проект.

По функционалу они очень схожи. Оба движка оборудуются браузерами для ассетов, обработки скриптов и остальных типов файлов. Разрабатываемые объекты просто перетаскиваются к сцене, и они вносятся в общую иерархию.

Для влияния на объекты, расположенные внутри редактора, используются встроенные инструменты: масштаб, поворот, отражение, изменение позиции и т.д. По функциям редактора колоссальных отличий не наблюдается. Все свойства объектов, разработанных с помощью *Unity*, отражаются в «Инспекторе», а в случае с *Unreal* – всё размещено в «Деталях».

Обеим движкам характерно наличие *static* *meshes*, которые можно передвигать, вертеть, масштабировать и т. д. Скелетные меши также присущи обоим вариантам – геометрические сущности, связывающиеся со скелетными костями, их задача сводится к анимированию персонажей.

Для работы с *Unity* требуется две небольшие вещи: язык программирования *C*# и сам движок. *С*# нужен для создания скриптов [7, с. 519] (рисунок 1.4).

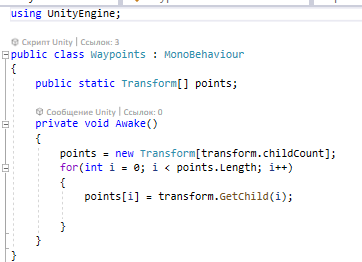


Рисунок 1.4 – Пример скрипта в *Unity*

Обратная сторона медали в том, что без программирования в *Unreal* можно сделать достаточно много. Скриптинг в *Unreal* делается с помощью их визуального языка – *Blueprints* (рисунок 1.5). Система визуальных сценариев *Blueprint* в *Unreal* *Engine* представляет собой полную систему сценариев игрового процесса, основанная на концепции использования интерфейса на основе узлов для создания элементов игрового процесса из *Unreal* *Editor*. Как и во многих общих языках сценариев, он используется для определения объектно-ориентированных классов или объектов в двигателе. Но на *С*++ можно писать игровую логику, и расширения для движка.

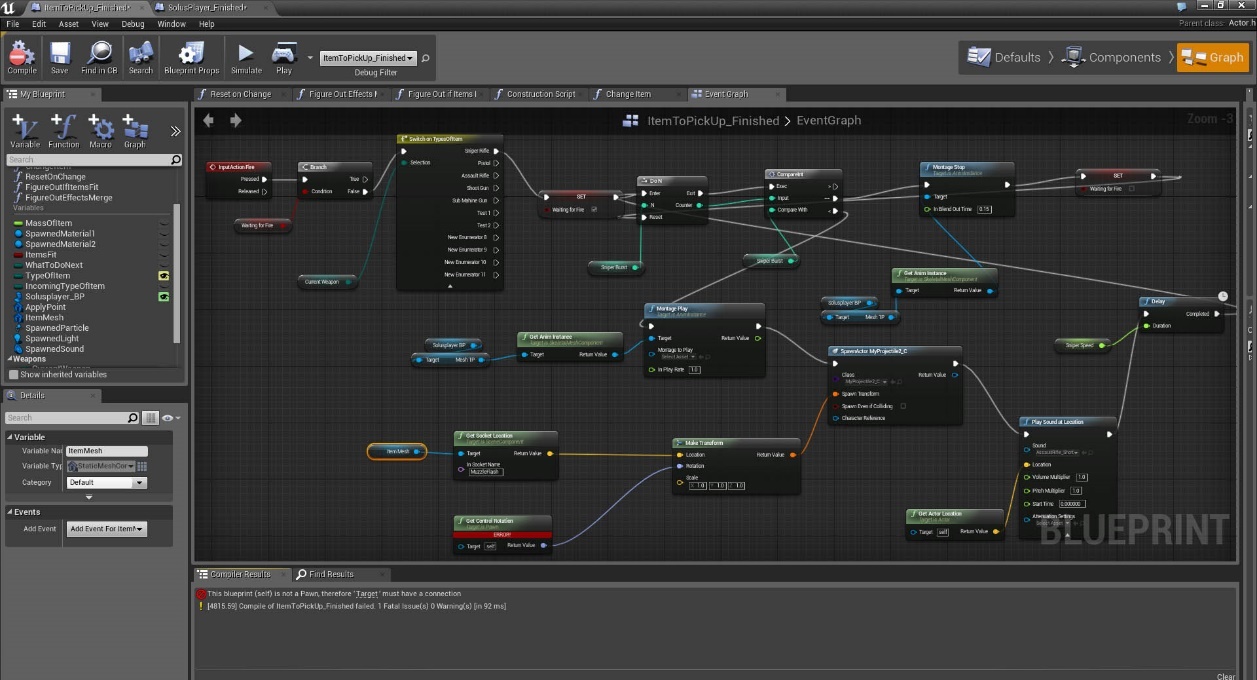


Рисунок 1.5 – Визуальный язык *Blueprints*

В том же *Fortnite*, к примеру 80% игровой логики написано на *Blueprints*, и оставшиеся 20% на *С*++. В *Unity* же 100% игровой логики пишется на *C*#. И из этого вытекает второе различие.

В индустрии часто случается так, что если команда разрабатывает на *Unity*, то с движком работают только программисты. Художники и геймдизайнеры предоставляют им контент и сами движок вообще не трогают. Иногда даже доходит до такого, что в отделе появляется «специалист по вставлению контента в движок».

С *Unreal* же работает вся команда – художники, которые сами вставляют в движок контент, и геймдизайнеры, которые сами пишут правила игры на блюпринтах. Так происходит не во всех студиях, но движок способствует такому подходу.

После изучения всей информации было принято решения для реализации игрового приложения использовать *Unity.* *Unity* занимает на устройстве гораздо меньше места, меньше сказывается на работе конечного пользователя. Компактный размер особенно важен в *Google* *Play* *Store*, где *APK* приходится делить на части, если этот файл оказывается крупнее 50*mb*. Длительность итерации в *Unity* гораздо меньше. *Unity* визуально выглядит более согласованно на всех устройствах.

* 1. **Обзор игр жанра «*Tower* *Defense*»**

*Tower* *defense* (*TD*) – это поджанр стратегических игр, цель которых состоит в том, чтобы защитить территории или владения игрока, препятствуя вражеским нападающим или мешая врагам добраться до выходов, обычно это достигается путем размещения оборонительных сооружений на пути их атаки или вдоль него (рисунок 1.6). Обычно это означает создание множества различных структур, которые служат для автоматического блокирования, препятствия, атаки или уничтожения врагов. *Tower* *defense* рассматривается как поджанр стратегических видеоигр в реальном времени из-за своего происхождения в реальном времени, хотя многие современные игры *tower* *defense* включают аспекты пошаговой стратегии. Стратегический выбор и расположение защитных элементов – важнейшая стратегия жанра.



Рисунок 1.6 – Игра *Tower* *Defense*

Жанр *Tower* *Defense* восходит к золотому веку аркадных видеоигр 1980-х годов. Целью аркадной игры *Space* *Invaders*, выпущенной в 1978 году, была защита территории игрока от волн приближающихся врагов. В игре были представлены щиты, которые можно было использовать, чтобы стратегически препятствовать вражеским атакам на игрока и помогать игроку защищать свою территорию, хотя и не для явной защиты территории. Игра 1980 года *Missile* *Command* изменила это, придав щитам более стратегическую роль. В игре игроки могли препятствовать приближающимся ракетам, и в каждой волне атаки было несколько путей атаки. *Missile* *Command* также была первой в своем роде, которая использовала устройство наведения, трекбол, позволяющий игрокам использовать перекрестие прицела. Это нововведение опередило свое время и предвосхитило последующий бум жанра, который был вызван широким распространением компьютерной мыши. Кроме того, в *Missile* *Command* единственной целью атакующих является база, а не конкретный персонаж игрока (рисунок 1.7). По этим причинам некоторые считают ее первой настоящей игрой в этом жанре.



Рисунок 1.7 – Игра *Missile* *Command*

В то время, как более поздним аркадным играм, таким как *Defender* (1981) и *Choplifter* (1982), не хватало стратегического элемента управления ракетами, они положили начало тенденции игр, которые сместили основную цель на защиту неигровых предметов. В этих играх защита неигровых игроков от волн нападающих является ключом к прогрессу. Игра *Parker Brothers* 1982 года «Звездные войны: Империя наносит ответный удар» для *Atari* 2600 была одним из первых дополнений, популяризировавших стиль защиты базы. Концепция волн врагов, атакующих базу гуськом (в данном случае *AT*-*ATs*), доказала формулу, которая впоследствии была скопирована многими играми, когда начался переход от аркадных игр к компьютерным. Игроки теперь могли выбирать из различных методов, препятствующих продвижению нападающих. «Ученик чародея» для *Atari* 2600 с изображением Микки Мауса был впервые опубликован в 1983 году.

В популярных ручных играх *Nintendo* *Game* & *Watch* 1980-х годов было много популярных предшественников. *Vermin* (1980), одна из первых игр, в которой игроки защищали сад (тема, за которой последовали многие последующие игры) от безжалостной орды кротов. В последующие годы появилось множество подобных названий, в том числе «Люк» (1981), «Парашют» (1981) и «Попай» (1981).

В 1982 году появилось несколько игр, основной целью которых была защита зданий от горения: «Пожарная атака», «Нефтяная паника» и «Микки и Дональд». В более поздних играх использовались несколько артикулирующих экранов, чтобы увеличить сложность для игроков. С двумя экранами эти игры вводили базовое управление ресурсами (например, нефтью и водой), заставляя игроков выполнять несколько задач. *Green* *House* (1982) – еще одна популярная игра с двумя экранами, в которой игроки используют облака пестицидных аэрозолей для защиты цветов от волн атакующих насекомых (рисунок 1.8).



Рисунок 1.8 – Игра *Green* *House*

Несмотря на ранний всплеск архетипических названий, в конечном счете произошел общий спад в играх с фиксированной ячейкой из-за их технических ограничений, упрощенного игрового процесса и появления персональных компьютеров и карманных компьютеров *Game* *Boy*; соответственно, этот жанр также пришел в упадок. Редким исключением был *Safebuster* (мульти экранный фильм 1988 года), в котором игрок защищает сейф от вора, пытающегося его взорвать (рисунок 1.9).



Рисунок 1.9 – Пример игры *Safebuster*

К середине 1980-х годов элементы стратегии начали развиваться дальше. Ранние примеры компьютерных игр включают игры *Commodore* 64 1984 года *Gandalf* *the* *Sorcerer*, шутер с элементами *tower* *defense* и игру *Imagine* *Software* 1984 года выпуска *Pedro*. *Pedro*, игра для защиты сада, представила новые элементы геймплея, включая различные типы врагов, а также возможность устанавливать фиксированные препятствия, а также строить и ремонтировать территорию игрока.

Имеются несущественные различия между разными играми данного типа. Например, в большинстве версий, когда башня модернизируется, её радиус действия, уровень и мощность увеличиваются одновременно. Однако в версии игры под названием *Onslaught* *Defence*, каждый параметр может быть улучшен отдельно. В некоторых играх враг может обороняться (рисунок 1.10). В отдельных версиях маршрут продвижения врага не ограничен какими-либо стенками. Волны наступления могут начинаться по команде игрока (тем самым давая возможность спокойно подготовиться), либо с определённым промежутком времени. Могут присутствовать элементы экономической стратегии (например, можно построить банк, который будет в геометрической прогрессии увеличивать денежные средства). Противники в разных играх могут различаться по своим свойствам – например, некоторые из них могут быть летающими и для их поражения требуются специальные «противовоздушные» башни.



Рисунок 1.10 – Пример игры *Onslaught* *Defence*

В игре *Gemcraft* разделили башни (и ловушки) от их активных способностей по атаке (которые обеспечиваются драгоценными камнями). Камни могут выниматься из башен и объединяться, усиливаясь при этом. Дополнительно, появились характеристики игрока, развивающиеся от карты к карте.

# **2** **ВНУТРЕННЯЯ АРХИТЕКТУРА И ИНФОРМАЦИОННАЯ МОДЕЛЬ ДАННЫХ РАЗРАБАТЫВАЕМОГО ИГРОВОГО ПРИЛОЖЕНИЯ «*MYTHICAL* *TOWERS*»**

**2.1 Архитектура и взаимосвязь модулей игрового приложения «*Mythical* *Towers*»**

## 

**Игра** – это программа, организующая игровой процесс на мобильном устройстве, которая имеет правила, героев, декорации и цели. Таким образом, возникает необходимость разработки внешнего вида и функциональной части пользовательского графического интерфейса.

Исходя из лучших практик проектирования архитектуры программного обеспечения, важно выделить несколько ключевых факторов, которые объединяют хорошо спроектированное программное обеспечение: минимизация связности написанного кода, разграничение компонентов программного обеспечения по типам решаемых задач и целям изменения, разграничение деталей и бизнес правил.

Для выполнения первого правила, из перечисления представленного выше, следует ввести и активно использовать в программном коде интерфейсы, а также направить все имеющиеся зависимости в сторону созданных интерфейсов.

Выполнение второго правила преследует цель уменьшить количество затронутых функций для внесения каких-либо дополнений в будущем.

Для выполнения третьего правила следует выделить алгоритмы, непосредственно решающие задачу от стороннего программного обеспечения, такого как пользовательский интерфейс, ввод, вывод данных, хранение данных, сетевое взаимодействие.

При проектировании игрового приложения, возникает необходимость определения основных структурных элементов приложения. Весь процесс взаимодействия пользователя с программой можно разделить на три независимых структурных элемента, представленных на рисунке 2.1.

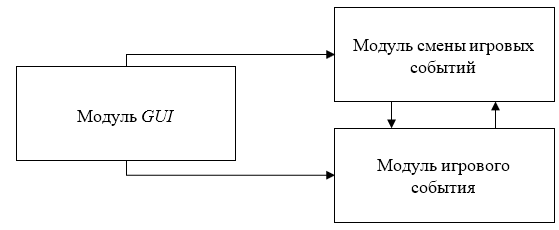


Рисунок 2.1 – Схема взаимодействия структурных элементов программы

Интерактивная система включает следующие модули:

– блок пользовательского графического интерфейса, взаимодействующий со всеми остальными компонентами программы и служащий инструментом интерактивного взаимодействия пользователя с программным продуктом;

– блок переключения игровых событий, отвечающий за процесс переключения пользователя между отдельными игровыми ситуациями (уровнями игры);

– блок игровых событий, отвечающий за отдельные игровые ситуации и являющийся основным игровым модулем в приложении.

Так как игровое приложение не является крупным проектом, то графический интерфейс, также не должен занимать большую часть приложения, а являться скорее дополнением и удобным для пользователя интерфейсом, позволяющим осуществлять дополнительные манипуляции с приложением. Также, на первом этапе проектирования интерфейса не стоит большое внимание уделять цветовой палитре и лишним деталям, так как в первую очередь важен общий вид и заложенная в интерфейс базовая функциональность. Далее рассматриваются служебные компоненты пользовательского интерфейса, не оказывающие влияния на сам игровой процесс.

Модуль пользовательского графического интерфейса является единственным модулем приложения, взаимодействующим со всеми остальными модулями, так как отвечает за перемещение игрока в приложении, запуске и остановке процессов и так далее. На рисунке 2.2 представлена схема функционала данного игрового модуля.

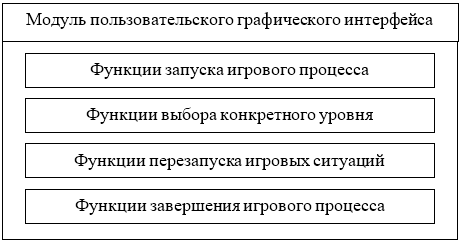


Рисунок 2.2 – Задачи модуля пользовательского интерфейса

Данный модуль выполняет ряд служебных задач и должен:

– запускать игровой процесс: функции запуска начала игры, активация функций перехода между игровыми ситуациями;

– запускать отдельный игровой уровень;

– прекращать игровой процесс: завершение игрового события, закрытие игрового приложения;

– перезапускать игровой процесс: перезапуск конкретного игрового события для повторного прохождения.

Таким образом, описанные функции целиком и полностью решают необходимые задачи данного модуля интерактивной системы.

Подмодуль сборки игровой сцены является важным компонентом представленного блока, так как способствует дальнейшему внедрению новых игровых ситуаций без необходимости нарушения целостности структуры кода приложения. Эта задача решается преимущественно благодаря системе сцен в *Unity*3*d*.

Подмодуль игрового процесса решает следующие задачи:

– инициализация компонентов на сцене: загрузка необходимых игровых объектов, загрузка текста для отображения информации о конкретном игровом уровне;

– логика игрового процесса конкретного уровня: основные игровые механики, позволяющие осуществлять непосредственно игровой процесс конкретном игровом уровне;

– отслеживание завершенности игрового уровня: проверка состояния игрового процесса на завершенность;

– логика завершения игровой ситуации: набор дополнительных действий, выполняемых после прохождения определенного игрового уровня игроком. На рисунке 2.3 представлена схема модуля игровых уровней.

Подмодуль сборки игровой сцены

Функции получения данных о сцене

Функции загрузки игрового уровня

Подмодуль игрового процесса

Функции инициализации сцены

Функции игрового процесса

Функции завершения игрового процесса

Рисунок 2.3 – Схема модуля игровых уровней

Описанные функции модуля позволяют осуществлять непосредственно игровой процесс выбранного игрового уровня.

Таким образом, представленные игровые модули формируют общую архитектуру интерактивного игрового приложения. Представленная архитектура также способствует в дальнейшем простому внедрению новых функциональных возможностей в приложение по добавлению новых элементов пользовательского графического интерфейса.

## **2.2 Информационная модель данный игрового приложения «*Mythical* *Towers*»**

Приложение условно включает в себя четыре независимых блока с различными элементами пользовательского интерфейса на каждом рисунок 2.4:

– начальная сцена приложения;

– сцена переключения уровней;

– сцена уровня;

– сцены обучения.

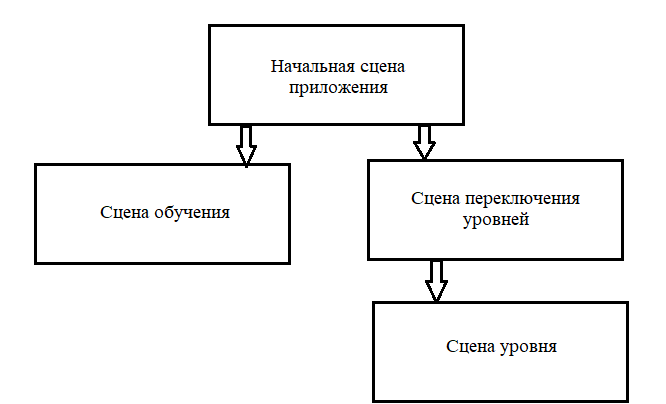


Рисунок 2.4 – Взаимосвязь с различными элементами пользовательского интерфейса

Начальная сцена является стартовой точкой в приложении и должна включать в себя следующие элементы:

– кнопка старта игры, позволяющая перейти на сцену выбора уровня;

– кнопка для прохождения обучения, позволяющая пользователю пройти обучения;

– кнопка выхода из игры, позволяющая пользователю закрыть приложение.

Экран выбора уровня является переходным между различными уровнями игрового приложения и позволяет пользователю выбрать интересующий для прохождения уровень (каждый уровень отличается сложностью его прохождения) или уровень необходимый для открытия новый уровней (рисунок 2.5). Данный экран должен включать:

– кнопку старта уровня, позволяющую запускать выбранный уровень;

– кнопку перехода на предыдущую сцену в игровом приложении.

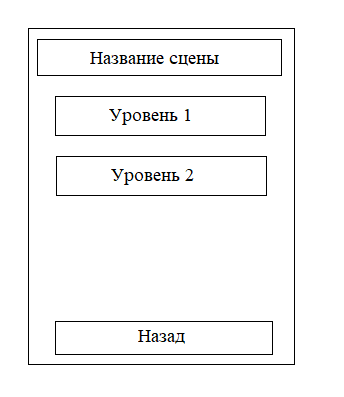


Рисунок 2.5 – Схема экрана выбора уровня

Экран игрового уровня – игровая сцена, меняющаяся в зависимости от выбранного пользователем уровня. Пользовательский интерфейс на данном экране содержит следующие элементы:

– кнопки выбора используемой башни;

– кнопку постановки игры на паузу;

– элементы пользовательского интерфейса, содержащие дополнительную информацию на уровне такую как: количество оставшихся денег на уровне, а также счетчик времени, который показывает сколько времени осталось у пользователя до следующей волны.

Все описанные элементы пользовательского интерфейса имеют схожие черты и в большинстве своем выполняют однотипные функции, что позволит в дальнейшем повторно использовать уже готовые элементы интерфейса.

Большую часть элементов пользовательского интерфейса составляют простые кнопки, а также текстовые поля. На рисунке 2.6 представлена диаграмма использования элементов пользовательского интерфейса.

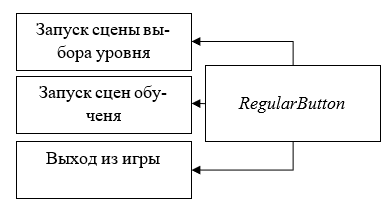


Рисунок 2.6 – Диаграмма использования элементов интерфейса

С текстовыми полями в дальнейшем будут взаимодействовать различные блоки кода, не относящиеся напрямую к пользовательскому интерфейсу.

Все кнопки в приложении имеют однотипный функционал и структуру и могут быть описаны одним блоком кода. На рисунке 2.7 представлена схема главного меню.

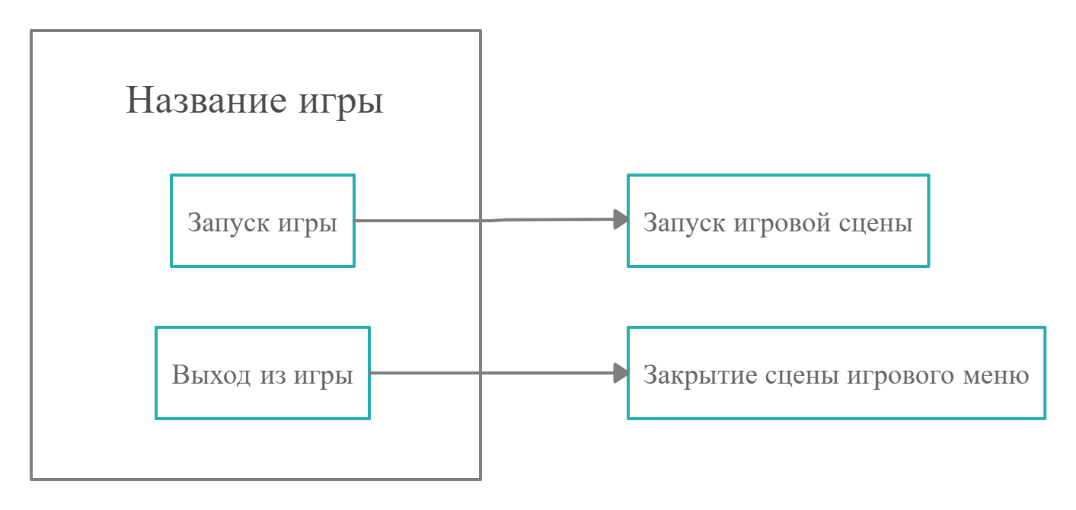


Рисунок 2.7 – Схема главного меню

Компоненты игровой логики составляют основную часть «*Mythical* *Towers*» и являются основными элементами, позволяющими пользователю осуществлять непосредственно игровой процесс. Как было описано ранее, вся структура приложения представляет собой набор из нескольких независимых игровых сцен, связанных между собой пользовательский интерфейсом. Тем не менее, сцены игрового процесса являются уникальными и содержат в себе отличные друг от друга игровые элементы.

Разрабатываемое игровое приложения является игрой в жанре «защиты башни*»*, целью которой является не пропустить врагов к главное башни с помощью расстановки башен. Кроме того, помимо самой игровой механики расстановки башен, на экране пользователя, во время игрового процесса, приводятся сведения о количестве денег на уровне, времени до следующей волны, доступных башнях и их стоимости. На рисунке 2.8 приведено схематическое отображение игрового приложения во время игрового процесса.



Рисунок 2.8 – Схематическое отображение игрового приложения

После описания необходимых компонентов, отвечающих за логику переключения пользователя между отдельными уровнями, возникает необходимость описания иерархии классов, отвечающих непосредственно за игровой процесс отдельного уровня в приложении.

Взаимодействие отдельных модулей игрового приложения происходит посредством запуска служебных компонентов приложения, которые осуществляют выбор необходимой игровой сцены на основе полученных игровых данных: название сцены, название текущей сцены, название предыдущей сцены и так далее.

Все уровни должны иметь один общий вид приложения, для этого интерфейс имеет общую структуру, представленную на рисунке 2.9

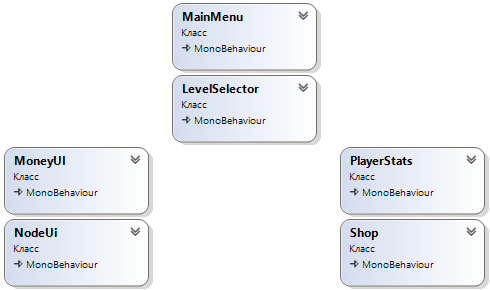


Рисунок 2.9 – Диаграмма классов, отвечающих за реализацию графического интерфейса

В дынном модуле представлены классы, выполняющие роль графического интерфейса игрового приложения. Иерархия классов включает в себя:

– публичный класс *MainMenu*, отвечающий за игровую сцену главного меню;

– класс *MoneyUI*, отвечает за графическое отображение денег на игровой сцене;

– класс *NodeUi*, отвечает за графическое отображение продажи и улучшения построенных башен;

– класс *PlayerStats*, отвечает за графическое отображения жизней игрока;

– классы *Shop*, отвечает за покупку башен на игровом поле.

После описания необходимых компонентов, отвечающих за графический интерфейс, возникает необходимость описания иерархии классов, отвечающих непосредственно за игровой процесс отдельного игрового уровня в приложении.

Каждый отдельный уровень должен иметь уникальный набор компонентов. Кроме того, каждый из видов игровых ситуаций должен иметь некоторую общую структуру и общий код, позволяющий интегрировать новые уровни, новых врагов и новые башни в проект. Данная структура представлена в модуле, диаграмма которого изображена на рисунке 2.10.

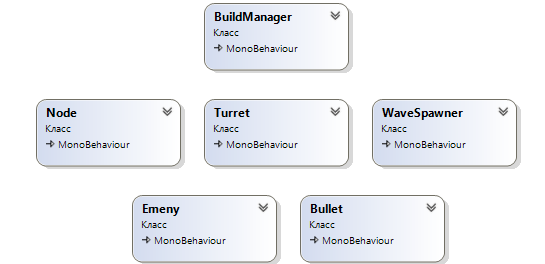


Рисунок 2.10 – Диаграмма классов, отвечающих за реализацию игры

В данном модуле представлены классы, выполняющие роль загрузчиков игровых уровней. Иерархия классов включает в себя:

– публичный класс *BuildManager*, описывающий общую логику и функции;

– класс *WaveSpawner*, осуществляет логику появления различных волн на игровом уровне. На рисунке 2.11 представлена схема настройки волн;

– класс *Node*, осуществляет логику расстановки башен на карте;

– класс *Turret*, осуществляет логику создания башен;

– классы *Emeny* и *Bullet*, хранящие информацию о врагах и снарядах.

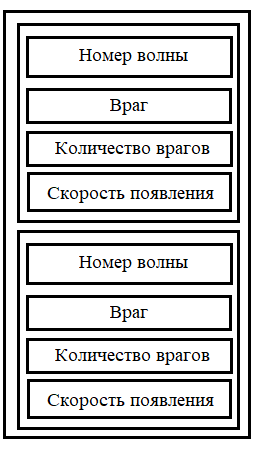


Рисунок 2.11 – Схема настройки волн

Описанная иерархия классов позволяет реализовать задуманную вариацию игровых уровней. Кроме того, благодаря общим классам, данная реализация позволяет в дальнейшем заново использовать уже созданный компонент для изменения варианта конкретного уровня.

Компонентный подход *Unity*3*d* позволяет избежать создания больших классовых иерархий, что делает конечный программный код менее сложным и устраняет проблему его читаемости. Компоненты представляют собой автономный код (подсистему), который может быть повторно использован за счет его независимого развертывания. Компоненты не обязательно должны быть большими по объему кода, но в основном они больше по объему отдельного класса или группы слабо связанных классов. Таким образом, описанные программные модули формируют независимые иерархии игровых классов, которые в последствии будут использоваться в качестве компонентов игрового движка *Unity*3*d* и собраны в отдельные модули (*prefabs*) для дальнейшего внедрения в приложение. А система игровых сцен в *Unity*3*d* и общая структура приложения позволяет легко интегрировать новые уровни в уже существующий программный код.

# **3 СТРУКТУРА ИГРОВОГО ПРИЛОЖЕНИЯ «*MYTHICAL* *TOWERS*»**

## **3.1 Игровые механики, реализованные в игровом приложении «*Mythical* *Towers*»**

Все игровые сущности имеют свои модели. Вся визуальная составляющая для игрового приложения моделировалась в программном обеспечении для создания трехмерной компьютерной графики *Blender*. Все созданные трехмерные модели для сущностей импортируются в проект в формате .*fbx*. Модель игровой карты, на которой проходит весь игровой процесс представлена на рисунке 3.1.

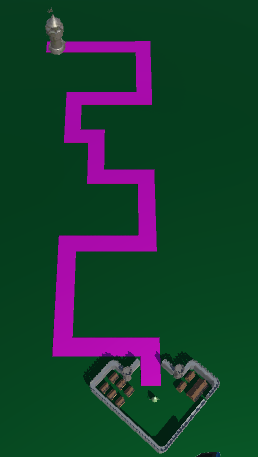


Рисунок 3.1 – Игровое поле

На модели игрового поля видно, что карта условно разделена на вражескую дорогу и зону, которую защищает игрок. Враги будут двигаться строго по дороге не останавливаясь. Главным объектом карты для игрока является замок. Для врага соответственно – это башня.

На протяжении все игры пользователь защищает башню и ее властителя Авокадо. К сожалению для игрока, ворот при входе в крепость нет, и все враги могут подойти близко к башне и нанести ей сокрушительный урон.

Коллизиями оформлена стена, защищающая башню и ее правителя. Для того чтобы попасть в застенки владений царя Авокадо нужно пройти через главный вход.

Все действия игрок осуществляет от третьего лица как бы располагаясь над картой. Механики сделаны в соответствии с таким видом для пользователя. Игрок учувствует в сражении посредством своих орудий. У него есть возможность поставить на поле три вида вооружения. Все модели для объектов орудий спроектированы согласно их характеристикам. Смотря на изображение, пользователю, становится интуитивно понятно какую огневую мощь имеет то или иное оружие. Модели орудий представлены на рисунке 3.2.

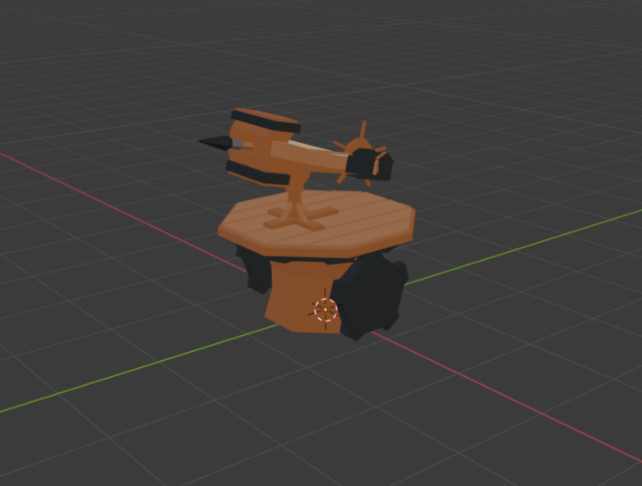
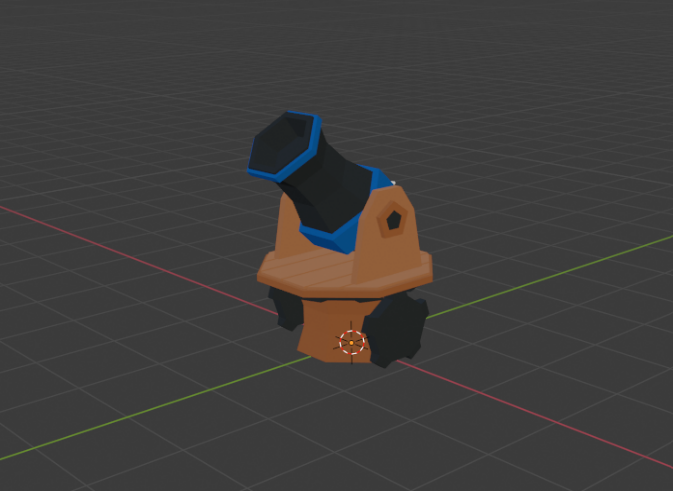


Рисунок 3.2 – Модели оружий игрока

Расставление оружия осуществляет с помощью нажатия на экран.

Все орудия стоят своей игровой валюты. В начале игры пользователю дается некоторое количество монет, которое он может потратить на оружие. Цена у каждого орудия разная и пользователю нужно выбрать с помощью нажатия на кнопку интерфейса надлежащую защиту. Для более интересного геймплея в игре реализована механика прокачки и продажи уже установленных башен. Продавая башню, игрок получает меньше монет чем потратил на покупку.

Такие механики игры заставляют пользователя думать на перспективу и принимать взвешенное решение относительно покупки той или иной огневой мощи.

Что касается врагов то их модели были спроектированы согласно задумке относительно сюжета. Модели врагов изображена на рисунке 3.3.





Рисунок 3.3 – Модели врагов

Игровая сцена является трехмерной в перспективной проекции. Перспективная проекция подразумевает собой проекцию на плоскость прямыми лучами, проходящими через центр проецирования. Один из проецирующих лучей перпендикулярен к плоскости проецирования и называется главным. Точка пересечения этого луча и плоскости проекции – главная точка картины. Особенностью центральной проекции является существенное различие в масштабах изображения предметов, находящихся на различных расстояниях от центра проектирования. Это связано с уменьшением угловых размеров предмета (и соответственно с уменьшением линейных размеров в плоскости изображения) при удалении от съемочной сцены. Изменение ширины полосы и размеров прямоугольников создает ощущение глубины пространства.

Игровой интерфейс приложения выполнен в формате 2*D*. Пользователю предоставлено два вида интерфейса. Это меню и внутриигровой интерфейс.

## **3.2 Структура классов игровой логики игрового приложения «*Mythical* *Towers*»**

При проектировании игрового приложения в жанре «защита башни» реализованы различные события и эффекты в игре. Непосредственно механики в игре реализованы в соответствии с жанром. Ими являются размещение орудий, перемещения врагов, выбор орудий и передвижение по игровому пространству. Визуальными и звуковыми эффектами были сопровождены смерть врагов, выстрелы и установку орудий. Во время игры звучит фоновая музыка, создающая надлежащую атмосферу рисунок 3.4.

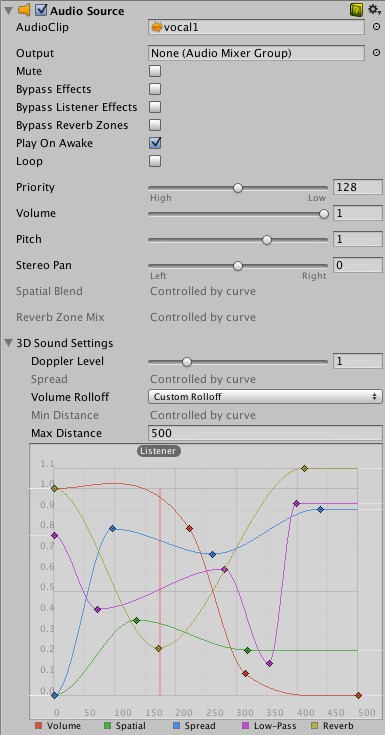


Рисунок 3.4 – Добавление фоновой музыки

Для того, чтобы проект был грамотно реализован и выполнял все свои функции правильно, в ходе разработки приложения был применен принцип разделения крупных элементов проекта на более мелкие и легко управляемые задачи – принцип декомпозиции. С использованием методов декомпозиции определяется так называемая иерархическая структура работ. Данная структура разделяет проект на иерархически связанные, управляемые и контролируемые подзадачи. Архитектура скриптов представлена на рисунке 3.5.

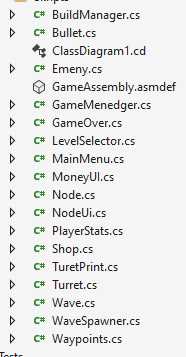
****

Рисунок 3.5 – Архитектура скриптов

Для реализации волн в игре используется интерфейс *IEnumerator*. Итератор – это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления. Идея паттерна Итератор состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс. Объект-итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом.

Первым делом в проекте реализованы площадки, на которые в будущем будут ставятся башни и будут передвигаться враги. Класс *Node*.*cs* (Приложение А, код программы для класса *Node*.*cs*) хранит переменные расположения турелей. В методе *GetBuildPossition*() происходит проверка позиции площадки для постройки башни. Метод *OnMouseDown*() проверяет нажатие на экран. Метод *BuildTurret*() строит башни в необходимое поле. Методы *UpgradeTurret*() и *SellTurret*() добавляют прокачку и продажу башен. Подробное описание класса *Node* представлено в таблице 3.1.

Таблица 3.1 – Описание класса *Node*

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Вид элемента | Тип | Описание |
| *Start* | Метод | *void* | Активация контроллеров |
| *GetBuildPossition* | Метод | *Vector*3 | Определяет позицию для постройки башен |
| *OnMouseDown* | Метод | *void* | Добавляет управление |
| *BuildTurret* | Метод | *void* | Строит башни |
| *UpgradeTurret* | Метод | *void* | Улучшает башню |
| *SellTurret* | Метод | *void* | Продает башни |
| *turret* | Поле | *GameObject* | Объект башни на сцене |
| *isUpgrate* | Поле | *bool* | Используеться для улучшения башни |

Для создания управления и использования турелей используется класс *Turret*.*cs* (Приложение А, код программы для класса *Turret*.*cs*). Класс *Turret*.*cs* (Приложение А, код программы для класса *Turret*.*cs*) хранит переменные скорости стрельбы, дальности стрельбы. А также содержит методы обнаружения врага и выстрела. Подробное описание класса *Turret* представлено в таблице 3.2.

Таблица 3.2 – Описание класса *Turret*

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Вид элемента | Тип | Описание |
| *range* | Поле | *float* | Дальность стрельбы |
| *fireRate* | Поле | *float* | Частота стрельбы |
| *ternSpeed* | Поле | *float* | Скорость стрельбы |
| *Update* | Метод | *void* | Обновление сцены |
| *Shoot* | Метод | *void* | Выстрел |
| *UpdateTarget* | Метод | *void* | Отвечает за обновление цели |
| *bulletPrefab* | Поле | *GameObject* | Объект пули |
| *firePoint* | Поле | *Transform* | Точка стрельбы |
| *enemyTag* | Поле | *string* | Для поиска врага |

Для создания врага используется класс *Emeny*.*cs* (Приложение А, код программы для класса *Emeny*.*cs*) который хранит такую переменную, как скорость врага. Класс *Emeny*.*cs* (Приложение А, код программы для класса *Emeny*.*cs*) содержит метод *Update* отвечающий за перемещение врага и *GetNextWaypoint* отвечающий за поиск следующей точки траектории движения. Точки движения хранятся в классе *Waypoints*.*cs* (Приложение А, код программы для класса *Waypoints*.*cs*). Подробное описание класса *Emeny* представлено в таблице 3.3.

Таблица 3.3 – Описание класса *Emeny*

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Вид элемента | Тип | Описание |
| *speed* | Поле | *float* | Скорость врага |
| *startHealth* | Поле | *int* | Стартовое здоровье врага |
| *healthsBar* | Поле | *Image* | Графическое отображения здоровья |
| moneyUp | Поле | *int* | Количество денег получаемые за убийство врага |
| *TakeDamage* | Метод | *void* | Высчитывает полученный урон у врагов |
| *Die* | Метод | *void* | Уничтожает врага, добавляет деньги |
| *Update* | Метод | *void* | Обновление игровой сцены |
| *GetNextWaypoint* | Метод | *void* | Движение врага по заданным точкам |
| *EndPath* | Метод | *void* | Конец пути врага на игровой сцене |
| *waypointIndex* | Поле | *int* | Индекс точек движения |

Для стрельбы по врагам используется класс *Bullet*.*cs* (Приложение А, код программы для класса *Bullet*.*cs*) содержащий переменную скорость пули. В классе *Bullet*.*cs* (Приложение А, код программы для класса *Bullet*.*cs*) существуют методы: *Seek* который отвечает за поиск пулей врага на игровой сцене, *Update* отвечающий за расположение пули на экране, *HitTarget* отвечающий за нанесения урона врагу, *Damage* наносит урон врагу. Подробное описание класса *Bullet* представлено в таблице 3.4.

Таблица 3.4 – Описание класса *Bullet*

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Вид элемента | Тип | Описание |
| 1 | 2 | 3 | 4 |
| *damage* | Поле | *int* | Наносимый урон |
| *impactEffect* | Поле | *GameObject* | Эффект попадания во врага |
| *target* | Поле | *Transform* | Мишень |
| *speed* | Поле | *float* | Скорость |
| *Seek* | Метод | *void* | Поиск врагов |
| *Update* | Метод | *void* | Обновление сцены |

Продолжение таблицы 3.4

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | *3* | 4 |
| *HitTarget* | Метод | *void* | Высчитывает попадание во врага |
| *Damage* | Метод | *void* | Наносит урон врагу на игровой сцене |

Для покупки турелей существует класс *Shop*.*cs* (Приложение А, код программы для класса *Shop*.*cs*) который содержит методы покупки турелей *BuySimpleGun* и *BuyStrongGun*.

Для слежения за всем происходящим используется класс *BuildManager*.*cs* (Приложение А, код программы для класса *BuildManager*.*cs*) содержащий переменные такие как: *selectNode* отвечает за выбор позиции для постройки башни, *turretToBuild* отвечающий за постройку башни. В классе *BuildManager*.*cs* (Приложение А, код программы для класса *BuildManager*.*cs*) располагаются такие методы как: *BuildTurretOn* который позволяет построить башню если хватает финансов на это, *SelectNode* отвечающий за выбор куда поставить башню. Подробное описание класса *BuildManager* представлено в таблице 3.5.

Таблица 3.5 – Описание класса *Node*

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Вид элемента | Тип | Описание |
| *simpleTurretPrefab* | Поле | *GameObject* | Объект простой башни |
| *strongTurretPrefab* | Поле | *GameObject* | Объект сильной башни |
| *fastTurretPrefab* | Поле | *GameObject* | Объект быстрой башни |
| *nodeUi* | Поле | *NodeUi* | Добавляет интерфейс |
| *HasMoney* | Свойство | *bool* | Проверяет хватает ли игроку денег на постройку башни |
| *CanBuild* | Свойство | *bool* | Проверяет построена ли башня в данном месте |
| *SelectNode* | Метод | *void* | Выбор позиции для постройки башни |
| *BuildTurretOn* | Метод | *void* | Позволяет построить башню |

Классы являются скриптовыми компонентами объектов, созданных в сцене игрового «движка» *Unity*, обеспечивают обработку входных данных и отвечают за интерактивную составляющую приложения.

## **3.3 Игровой интерфейс игрового приложения «*Mythical* *Towers*»**

Игровой интерфейс – это первое что увидит пользователь, а значит он должен завлекать внимание, но в тоже время быть прост и незамысловат. Элементы игрового интерфейса должны содержать в себе информацию важную для игрового процесса, но не перетягивать внимания с самой игры. Общий пользовательский интерфейс представляет собой набор графических кнопок, позволяющих выполнять пользователю определенные действия, и имеет одинаковый внешний вид и функциональность на большинстве сцен. Однако некоторые сцены (стартовая сцене) имеют отличный от других пользовательский графический интерфейс.

Игровое меню позволяет понять пользователю с чем он имеет дело. Перед нажатием клавиши «*Play*» пользователь должен приблизительно понять, что он запустит. Меню содержит элементы, которые дают возможность понять, что из себя представляет игра. Главное меню представлена на рисунке (рисунок 3.6).



Рисунок 3.6 – Главное меню приложения «*Mythical Towers*»

Для удобства использования был разработан интуитивно понятный интерфейс. В интерфейсе существует несколько основных элементов. Самое основное это информация о оставшихся деньгах у игрока (рисунок 3.7.)



Рисунок 3.7 – Оставшиеся финансы у игрока

Также для игрока выводиться информация о времени между волнами (рисунок 3.8). Сделано это для того, чтобы пользователь мог определить сколько у него осталось времени на подготовку перед следующей волной.

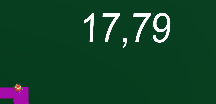


Рисунок 3.8 – Оставшиеся время до следующей волны

Также у игрока внизу экрана располагаются башни для выбора. Возле каждой башни указана стоимость, которая будет отниматься при покупке башни (рисунок 3.9).



Рисунок 3.9 – Башни на выбор для игрока

После выбора башни игрок может располагать их на карте.

В итоге получился простой интерфейс, отражающий общий стиль игры и дающий максимум полезной для пользователя информации.

**4 ВЕРИФИКАЦИЯ И ОПЫТНАЯ ЭКСПЛУАТАЦИЯ ИГРОВОГО ПРИЛОЖЕНИЯ «*MYTHICAL* *TOWERS*»**

**4.1 Принципы работы пользовательского графического интерфейса и основные механики игрового приложения «*Mythical* *Towers*»**

При эксплуатации игры было проанализировано соответствие *Tower* *defense*. Так же была оценена целостность геймплея и удобство в управлении.

Пользовательский графический интерфейс является основным инструментом пользователя, позволяющий:

– переключаться между игровыми сценами: начальный экран, экран выбор уровня, обучение.

– ставить игру на паузу;

– запускать игру;

На рисунке 4.1 представлен начальный экран приложения.

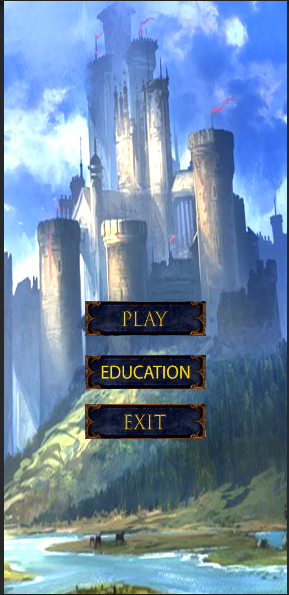


Рисунок 4.1 – Начальный экран приложения

Так же в функции пользовательского интерфейса входят непосредственно механики игровых событий, взаимодействие пользователя с игровым полем, взаимодействие с башнями, прокачка башен.

Функционал пользовательского интерфейса делится на такие части как:

– общий графический интерфейс, присутствующий на всех игровых сценах;

– интерфейс взаимодействия пользователя с элементами управления при выборе в меню игры.

Общий пользовательский интерфейс представляет собой набор графических кнопок, позволяющий выполнять пользователю определенные действия, и имеет одинаковый внешний вид и функциональность на большинстве сцен. Но стартовый интерфейс (меню игры) и меню выбора уровня имеют отличный от других пользовательский графический интерфейс.

На экране пользователю предлагается запустить игру нажав кнопку «*Play»* или выйти по нажатию кнопки «*Exit»*. По кнопке «*Education»* пользователь может пройти обучение. Все нажатия в данном меню осуществляются при наведении курсора и по нажатию левой клавиши мыши.

Экран переключения уровней игрового приложения предназначен для удобного перемещения между уровнями, а также для понимания пользователя сколько он прошел уровней и сколько ему осталось пройти. Находясь на данном экране, пользователь имеет возможность выбирать интересующий его уровень, из списка представленных на игровой сцене. На рисунке 4.2 представлен экран переключения уровней.



Рисунок 4.2 – Сцена переключения уровней игрового приложения

Интерфейс игровой сцены включает в себя следующие элементы:

* панель отображения текстовой информации о количестве оставшихся денег у пользователя;
* панель выбора башни, а также стоимости башен;
* панель отображения текстовой информации о оставшемся времени до следующей волны.

На рисунке 4.3 представлен скриншот игровой сцены.

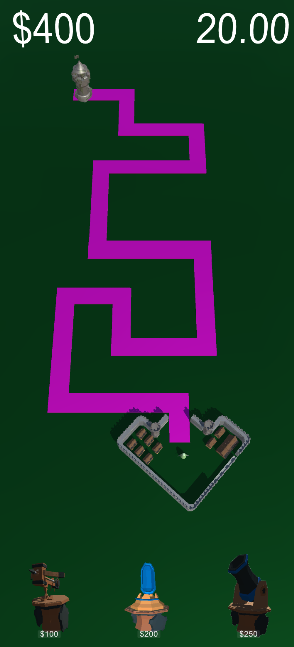


Рисунок 4.3 – Игровая сцена

В начале игры пользователь начинает свое погружение в мир от третьего лица. Игрок как бы парит над картой. По сюжету в данном мире находится раса похожая на человеческие овощи. Племя овощей решилось на войну против царя Авокадо. Они всю жизнь были бедны и жили в своих грядках в то время, как их царствующий правитель отстроил себе башню и дома для приближенных. Но последней каплей стало возведение стены. Царь хотел оградиться от простого народа. И вот армия овощей решилась на отчаянный шаг. Вылезая из грядок, они идут прямо на царя. У них есть шансы на победу пока стена не достроена. Но Авокадо не так прост. Его защищают орудия и овощам нужно проявить немалую стойкость чтобы их преодолеть.

Задачей игрока является сохранение жизни царя. Ведь банда овощей пытается осуществить экспансию покушаясь на владения царя Авокадо.

При запуске игры у пользователя перед глазами открывается игровое поле. На этой местности начинаются боевые действия и нужно срочно принимать меры так как овощи уже восстают из башни.

Игроку сразу предлагают поставить первую пушку. При каждом виде оружия указана его цена. Пример выбора пушки представлен на рисунке 4.4.



Рисунок 4.4 – Выбор башни

Выбрав пушку, игрок располагает ее в любом доступном месте. Конечно, это все возможно, если у игрока хватает монет. На старте игры ему дается некоторое количество монет, зависящее от уровня сложности, которые он может потратить по своему усмотрению. За каждое убийство противника начисляются монеты. У каждого противника своя цена.

Врагами в игре являются овощи и их механика передвижения очень просты. Но стоит учитывать, что пушка выцеливает одного врага и может не успеть добить его, пока не подъехал следующий враг. Для того чтобы избежать проблем следует сохранить баланс между видами пушек. Существует всего 3 вида башен. Самая дешевая имеет обычный урон и скорость стрельбы. Вторая башня – скорострельная, имеет меньший урон, но стреляет чаще и на более далекие дистанции. И самая тяжелая башня, которая имеет самый большой урон среднюю скорость стрельбы и больший радиус стрельбы.

Размещение башен на игровом поле показано на рисунке 4.5.

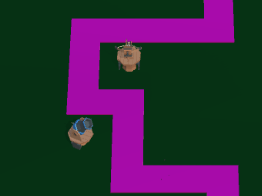
.

Рисунок 4.5 – Пушки на игровом поле

В игровом приложении представлено несколько врагов. Все враги отличаются своими характеристиками. Характеристики врагов в игре:

* количество жизней;
* скорость передвижения;
* количество денег, которое получит игрок после уничтожения врага.

Враги представлены на рисунке 4.6.

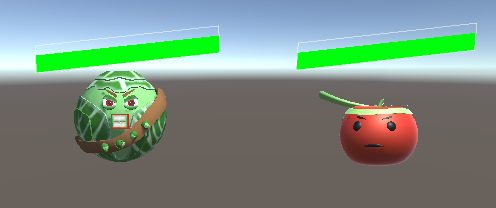


Рисунок 4.6 – Враги на игровом поле

Во время игры пользователь может отслеживать свои деньги при помощи внутриигрового интерфейса. Интерфейс помогает отслеживать как протекает ход игры.

**4.2 Модульное тестирование ключевых компонентов игрового приложения «*Mythical* *Towers*»**

В процессе разработки программного обеспечения ключевую роль играет тестирование разрабатываемых программных модулей. В качестве инструмента тестирования интерактивной системы было взято модульное тестирование. Данный вид тестирования позволяет протестировать основные функции игровых компонентов и избежать последующих ошибок в работе приложения.

*Unity*3*d* обладает встроенным функционалом, позволяющим интегрировать модульные тесты *C*# с компонентами игрового движка. Однако, тестирование сложных объектов на *Unity*3*d* является трудоемкой задачей.

Задачи, которые решались в ходе разработки и тестирования игрового приложения:

* оперативное устранение сбоев и ошибок, возникающих в ходе работы игрового приложения;
* подтверждение стабильности работы разработанного игрового приложения;
* проверка корректности реализации функционала игрового приложения;
* подтверждение соответствия функционала заданию;
* проверка удобства интерфейса.

Для проверки корректности работы приложения было проведено мануальное тестирование.

В ходе тестирования правильности работы приложения были выделены слабые места приложения, к ним подобраны всевозможные кейсы и протестированы вручную.

Список возможных кейсов:

1. нажатие кнопки «*Play*»;
2. при нажатии, не попадает в меню выбора уровня;
3. нажатие кнопки «*Education*»;
4. не начинается обучение;
5. игровая сцена;
6. попытка поставить в одно место две башни;
7. попытка прокачать одну башню два раза;
8. попытка потратить больше денег чем имеется;
9. попытка поставить башню на маршруте врагов;
10. нажатие кнопка выхода.

Классы отдельных игровых компонентов являются важными частями приложения и нуждаются в тестировании, так как в случае некорректной работы функций данных классов, игрок может потерять контроль над элементами управления на игровой сцене, в следствии чего дальнейшее осуществление игрового процесса станет невозможным.

У интерфейсов мобильных устройств есть ряд особенностей, которые существенно отличают их от классических интерфейсов настольного ПО. В первую очередь это конечно же размер устройства, а следовательно, и количество информации, которую можно отобразить на его экране без ущерба для ее восприятия. Во-вторых, это метод взаимодействия пользователя с интерфейсом посредством пальцев, которые в точности значительно уступают привычной мыши или тачпаду

При проведении юзабилити-тестирования был проверен интерфейс игрового приложения. В главном меню располагаются три кнопки: играть, обучение, выход. Кнопки визуально выделены, пользователю удобно нажимать.

Масштаб приложения удобен для использования, все элементы располагаются на экране.

По нажатию на клавишу играть пользователь попадает в меню выбора уровня, что оправдывает ожидание пользователя. Не заставляет его догадываться, как поведет себя интерфейс и совершать неестественные действия.

В меню игровой сцены важная информация для игрока располагается именно там, где пользователь ожидает ее увидеть, время и деньги сверху экрана, а выбор башни снизу экрана.

В сцене обучения пользователю рассказывается, где какой находиться элемент, как ставить башни.

В данном модуле теста объектами тестирования были выбраны следующие методы:

– *SpawnWave*, отвечающий за логику появления волн на игровой сцене;

– *SpawnEmeny*, отвечающий за логику появления врагов на игровой сцене.

На рисунке 4.7 представлен результат тестирования основных компонентов появления врагов.

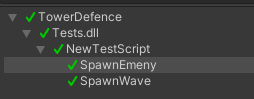


Рисунок 4.7 – Тестирование методов появления врагов

В данном модуле теста объектами тестирование были выбраны методы врага:

– *speed*, отвечающий за скорость перемещения врагов на игровой сцене;

– *startHealth*, отвечающий за начальное здоровье врага;

– *healthsBar*, отвечающий за логику отображения полосы здоровья;

– *moneyUp*, отвечающий за логику получения денег после смерти врага;

– *TakeDamage*, отвечающий за логику получения урона;

– *Die*, отвечающий за смерть врага;

– *GetNextWaypoint*, отвечающий за логику перемещения врагов на игровом поле;

– *EndPath*, отвечающий за снятие здоровья башни в случае прохождения всего маршрута.

На рисунке 4.8 представлен результат тестирования основных компонентов врага.

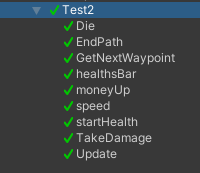


Рисунок 4.8 – Тестирование методов врагов

В данном модуле теста объектами тестирование были выбраны методы площадок для постройки башен:

– *BuildTurret*, отвечающий за постройку башен на игровой сцене;

– *GetBuildPossition*, отвечающий позицию для постройки башни на игровой сцене;

– *OnMouseDown*, отвечающий управление;

– *SellTurret*, отвечающий за логику продажи башни;

– *UpgradeTurret*, отвечающий за логику улучшения башни.

На рисунке 4.9 представлен результат тестирования основных компонентов площадок для постройки башен.

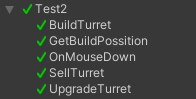


Рисунок 4.9 – Тестирование методов площадок для постройки башен

В данном модуле теста объектами тестирование были выбраны методы башен:

– *Shoot*, отвечающий за логику выстрела башни;

– *Update*, отвечающий за обновление сцены;

– *UpdateTarget*, отвечающий за логику обнаружения врагов на игровой сцене.

На рисунке 4.10 представлен результат тестирования основных компонентов башен.

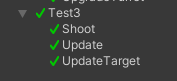


Рисунок 4.10 – Тестирование методов башен

В данном модуле теста объектами тестирование были выбраны методы выстрела:

– *Damage*, отвечающий за логику нанесения урона по врагам;

– *HitTarget*, отвечающий за логику попадание пули в врага;

– *Seek*, отвечающий за логику поиска врага;

– *Update*, отвечающий за обновление игровой сцены.

На рисунке 4.11 представлен результат тестирования основных компонентов башен.



Рисунок 4.11 – Тестирование методов пули

Результаты анализа утилиты *Code Coverage* представлены на рисунке 4.8.

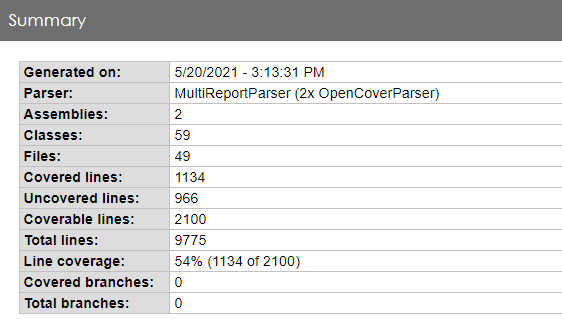


Рисунок 4.9 – Результаты анализа *Code Coverage*

Общий процент покрытия исходного кода модульными тестами составляет 54%.

Результат тестирования игрового приложения «*Mythical Towers*» – тестирование было проведено на нескольких мобильных устройствах, которые имеют минимальные технические характеристики, требуемые для работы игрового приложения. На каждом из устройств игровое приложение работало стабильно и корректно, установка и удаление не вызвало проблем, объекты отображались нормально, с ними можно было взаимодействовать.

Результаты модульного тестирования – каждый модуль (скрипт) тестировался отдельно. Работоспособность модулей корректна.

Результаты функционального тестирования – каждая функция тестировалась согласно функциональным требованиям приложения. Все функции работают исправно.

1. **ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ДИПЛОМНОЙ РАБОТЫ**

## **5.1 О****боснование целесообразности разработанного программного продукта**

Игровое приложение «*Mythical Towers*», разработанное на *Unity*3*D* предназначено для игрового процесса на платформе *Android*. Сделано в жанре «защита башни» имеет интересные игровые механики и геймплей, что выделит ее на рынке среди других игровых приложений.

## **5.2 Оценка конкурентоспособности программного продукта**

Техническая прогрессивность разрабатываемого программного продукта определяется коэффициентом эквивалентности (). Расчет этого коэффициента осуществляется путем сравнения технического уровня товара-конкурента и разрабатываемого программного продукта по отношению к эталонному уровню программного продукта данного направления с использованием формулы (5.1):

|  |  |
| --- | --- |
|  | (5.1) |
|  |  |

где ,– коэффициенты технического уровня нового и базисного программного продукта, которые можно рассчитать по формуле (5.2):

|  |  |
| --- | --- |
| , | (5.2) |
|  |  |

где β – коэффициенты весомости *i*-го технического параметра;

*n* – число параметров;

*Pi* – численное значение *i*-го технического параметра, сравниваемого программного продукта;

*P*э – численное значение *i*-го технического параметра эталона.

Расчет коэффициента эквивалентности приведен в таблице Д.1.

Полученное значение коэффициента эквивалентности больше 1, следовательно, разрабатываемый программный продукт является технически прогрессивным.

Далее рассчитывается коэффициент изменения функциональных возможностей () нового программного продукта по формуле (5.3):

|  |  |
| --- | --- |
|  | (5.3) |
|  |  |

где , – балльная оценка неизмеримых показателей нового и базового изделия соответственно.

Расчет коэффициента изменения функциональных возможностей нового программного продукта приведен в таблице Д.2.

Коэффициент функциональных возможностей превышает единицу т.е. новый программный продукт превосходит по своим функциональным возможностям базовый в 1,17 раза.

Конкурентоспособность нового программного продукта по отношению к базовому можно оценить с помощью интегрального коэффициента конкурентоспособности, по формуле (5.4), учитывающего все ранее рассчитанные показатели.

|  |  |
| --- | --- |
| , | (5.4) |
|  |  |

где Кн – коэффициент соответствия нового программного продукта нормативам (Кн = 1);

Кц – коэффициент цены потребления (= 0,94).

Расчет уровня конкурентоспособности нового программного продукта приведен в таблице Д.3.

Коэффициент цены потребления рассчитывается как отношение договорной цены нового программного продукта к договорной цене базового (= 0,94).

Ки = (1,131,171)/0,94 = 1,4.

Интегральный коэффициент конкурентоспособности () больше 1, т.е. новый программный продукт является более конкурентоспособным, чем базовый.

**5.3 Оценка трудоемкости работ по созданию программного продукта**

В качестве единицы измерения объема ПО может быть использована строка исходного кода (*LOC*). Общий объем ПО (*V0*) определяется исходя из количества и объема функций, реализуемых программой, по каталогу функций ПО по формуле (5.5):

, (5.5)

где *Vi* – объем отдельной функции ПО;

*n* – общее число функций.

Уточненный объем ПО () определяется по формуле (5.6):

, (5.6)

где – уточненный объем отдельной функции ПО в строках исходного кода.

Результаты расчетов представлены в приложении Д, таблица Д.4.

Рассчитаем поправочные коэффициенты, учитывающие организационно-технические условия разработки ПО.

Разработанное в ходе выполнения дипломной работы программное обеспечение по своим характеристикам относится ко второй категории сложности.

На основании принятого к расчету (уточненного) объема и категории сложности ПО принимаем нормативную трудоемкость ПО выполняемых работ Тн = 207 чел.-дн.

Дополнительные затраты труда, связанные с повышением сложности разрабатываемого ПО, учитываются посредством коэффициента повышения сложности ПО (Кс), который определяем по формуле (5.7):

, (5.7)

где – коэффициент, соответствующий степени повышения сложности;

– количество учитываемых характеристик.

Принимаем коэффициент повышения сложности ПО равным 0,06.

Влияние фактора новизны на трудоемкость учитывается путем умножения нормативной трудоемкости на соответствующий коэффициент, учитывающий новизну ПО (Кн). Разработанная программа обладает категорией новизны Б, а значение Кн = 0,63.

Степень использования в разрабатываемом ПО стандартных модулей определяется их удельным весом в общем объеме ПО. Коэффициент, учитывающий степень использования стандартных модулей в разработанном приложении равен Кт = 0,77.

Программный комплекс разработан на языке *C++*, что соответствует коэффициенту, учитывающему средства разработки ПО, Ку.р = 1,0.

Значение коэффициентов удельных весов трудоемкости стадий разработки ПО определяются с учетом установленной категории новизны ПО и приведены в таблице Д.5.

Нормативная трудоемкость ПО () выполняемых работ по стадиям разработки корректируется с учетом коэффициентов: повышения сложности ПО, учитывающих новизну ПО (Кн), учитывающих степень использования стандартных модулей (Кт), средства разработки ПО (Ку.р) и определяются по формулам:

* для стадии технического задания по формуле (5.8)

; (5.8)

* для стадии эскизного проекта формуле (5.9)

; (5.9)

* для стадии технического проекта формуле (5.10)

; (5.10)

* для стадии рабочего проекта формуле (5.11)

; (5.11)

* для стадии ввода в действие формуле (5.12)

, (5.12)

где Кт.з, Кэ.п, Кт.п, Кр.п и Кв.н – значения коэффициентов удельных весов трудоемкости стадий разработки ПО в общей трудоемкости ПО.

Общая трудоемкость разработки ПО () определяется суммированием нормативной (скорректированной) трудоемкости ПО по стадиям разработки по формуле (5.13):

, (5.13)

где *n* – количество стадий разработки;

– нормативная (скорректированная) трудоемкость разработки ПО на *i*-й стадии, чел.-дн.

чел.-дн.

Результаты расчетов по определению нормативной и скорректированной трудоемкости программного обеспечения по стадиям разработки и общую трудоемкость разработки ПО () представлены в таблице Д.6.

**5.4 Расчет затрат на разработку программного продукта**

В состав затрат на разработку ПП входят следующие статьи расходов:

* затраты труда на создание ПП (затраты по основной, дополнительной заработной плате и соответствующие отчисления) ();
* затраты на изготовление эталонного экземпляра ();
* затраты на технологию (затраты на приобретение и освоение программных средств, используемых при разработке ПП; затраты на ПО, используемое как эталон) ();
* затраты на машинное время (расходы на содержание и эксплуатацию технических средств разработки, эксплуатации и сопровождения) ();
* затраты на материалы (информационные носители) ();
* затраты на энергию, на использование каналов связи (для отдельных видов);
* общепроизводственные расходы (затраты на управленческий персонал, на содержание помещений) ();
* непроизводственные (коммерческие) расходы (затраты, связанные с рекламой, поиском заказчиков, поставками конкретных экземпляров) ().

Для расчета затрат на разработку (себестоимости) программного продукта определим необходимые параметры: тарифная ставка, ставка арендных платежей, стоимость ПК, стоимость кВт-час и занесем их в таблицу Д.7.

Суммарные затраты на разработку ПО определяются по формуле (5.14):

, (5.14)

Расходы на оплату труда разработчиков с отчислениями определяются по формуле (5.15):

, (5.15)

где – основная заработная плата разработчиков, руб.;

– дополнительная заработная плата разработчиков, руб.;

– сумма отчислений от заработной платы (социальные нужды, стра-хование от несчастных случаев), руб.

Основная ЗП разработчиков рассчитывается по формуле (5.16):

, (5.16)

где – средняя часовая тарифная ставка, руб./час;

– общая трудоемкость разработки, чел.-час;

– коэффициент, учитывающий доплаты стимулирующего характера, примем = 1,6.

Средняя часовая тарифная ставка определяется по формуле (5.17):

, (5.17)

где – часовая тарифная ставка разработчика *i*-й категории, руб./час;

*ni* – количество разработчиков *i*-й категории.

Часовая тарифная ставка определяется путем деления месячной тарифной ставки на установленный при восьмичасовом рабочем дне фонд рабочего времени () (5.18):

, (5.18)

где – базовая ставка специалиста;

– тарифный коэффициент.

руб./ч.

Следовательно, можно вывести значение основной ЗП разработчиков, используя полученные данные о средней часовой тарифной ставки, общей трудоёмкости разработки и коэффициент доплат стимулирующего характера:

руб.

Дополнительная заработная плата рассчитывается по формуле (5.19):

, (5.19)

где – норматив на дополнительную заработную плату разработчиков.

руб.

Отчисления от основной и дополнительной заработной платы (отчисления на социальные нужды и обязательное страхование) рассчитываются по формуле (5.20):

, (5.20)

где – процент отчислений на социальные нужды и обязательное страхование от суммы основной и дополнительной заработной платы (  = 34%).

руб,

руб.

Затраты машинного времени () определяются по формуле (5.21):

, (5.21)

где – стоимость 1 часа машинного времени, руб./ч;

– коэффициент мультипрограммности, показывающий распределение времени работы ЭВМ в зависимости от кол-ва пользователей ЭВМ; = 1;

– машинное время ЭВМ, необходимое для разработки и отладки проекта, ч.

Стоимость машино-часа определяется по формуле (5.22):

, (5.22)

где – амортизационные отчисления за год, руб./год;

– затраты на электроэнергию, руб./год;

– затраты на материалы, необходимые для обеспечения нормальной работы ПЭВМ (вспомогательные), руб./год;

– затраты на текущий и профилактический ремонт ЭВМ, руб./год;

– прочие затраты, связанные с эксплуатацией ПЭВМ, руб./год;

– действительный фонд времени работы ЭВМ, час/год.

Такие коэффициенты как (затраты на заработную плату обслуживающего персонала с учетом всех отчислений, руб./год) и (стоимость аренды помещения под размещение вычислительной техники, руб./год) не будут учитываться при расчёте стоимости машино-часа, так как для разработки данного проекта не требуется помещение и обслуживающий персонал. Сумма годовых амортизационных отчислений () определяется по формуле (5.24):

, (5.24)

где – затраты на приобретение *i*-го вида основных фондов, руб;

 – коэффициент, характеризующий дополнительные затраты, связанные с доставкой и наладкой оборудования, = 13% от ;

/ (1 + ) – балансовая стоимость ЭВМ, руб;

– норма амортизации, %.

руб.

Если за год амортизационные отчисления составляют 282,5 руб., то за 150 дней разработки составят 116,09руб.

Стоимость электроэнергии, потребляемой за год, () определяется по формуле (5.25):

, (5.25)

где – паспортная мощность ПЭВМ, кВт; = 0,41 кВт;

– стоимость одного кВт-часа электроэнергии, руб;

*А* – коэффициент интенсивного использования мощности, *А* = 0,98.

Действительный годовой фонд времени работы ПЭВМ () рассчитывается по формуле (5.26):

, (5.26)

где – общее количество дней в году; = 365 дней;

, – число выходных и праздничных дней в году, + = 112 дней;

– продолжительность 1 смены, = 8 часов;

– количество рабочих смен ЭВМ, = 1;

– коэффициент, учитывающий потери рабочего времени, связанные с профилактикой и ремонтом ЭВМ, примем = 0,2.

час в год.

С учётом, что срок разработки программного продукта составляет 150 дней, действительный фонд времени работы ПЭВМ составляет 1200 ч.

руб.

Следовательно, за 150 дней разработки расходуется 104,47 руб.

Затраты на материалы, необходимые для обеспечения нормальной работы ПЭВМ составляют около 1% от балансовой стоимости ЭВМ и определяются по формуле (5.27):

, (5.27)

где – затраты на приобретение (стоимость) ЭВМ, руб.;

– коэффициент, характеризующий доп. затраты, связанные с доставкой, монтажом и наладкой оборудования, = 12 – 13 % от ;

– коэффициент, характеризующий затраты на вспомогательные материалы ( = 0,01).

руб.

Затраты на текущий и профилактический ремонт () принимаются равными 5% от балансовой стоимости ЭВМ и вычисляются по формуле (5.28):

, (5.28)

где – коэффициент, характеризующий затраты на текущий и профилактический ремонт, = 0,05.

руб.

Прочие затраты на эксплуатацию ПК состоят из амортизационных отчислений на здания, стоимости услуг сторонних организаций и составляют 5% от балансовой стоимости. Вычисляются по формуле (5.29):

, (5.29)

где – коэффициент размера прочих затрат, связанных с эксплуатацией ЭВМ ( = 0,05).

руб.

Для расчета машинного времени ЭВМ ( в часах), необходимого для разработки и отладки проекта, следует использовать формулу (5.30):

, (5.30)

где – срок реализации стадии «Рабочий проект» (РП);

– срок реализации стадии «Ввод в действие» (ВП); + = 45;

– продолжительность рабочей смены, ч; = 8 ч;

– количество рабочих смен, = 1.

ч.

руб./ч.

руб.

Расчет затрат на изготовление эталонного экземпляра осуществляется по формуле (5.31):

, (5.31)

где – коэффициент затрат на изготовление эталонного ПП, = 0,05.

руб.

Затраты на материалы (носители информации и прочее), необходимые для обеспечения работы ПЭВМ, рассчитываются по формуле (5.32):

, (5.32)

где – затраты на приобретение ЭВМ, руб.;

– коэффициент, характеризующий доп. затраты, связанные с доставкой, монтажом и наладкой оборудования, = 12-13 % от ;

– коэффициент, характеризующий затраты на вспомогательные материалы ( = 0,01).

руб.

Общепроизводственные затраты () определяются по формуле (5.33):

, (5.33)

где – норматив общепроизводственных затрат.

руб.

Непроизводственные затраты рассчитываются по формуле (5.34):

, (5.34)

где – норматив непроизводственных затрат.

руб.

Итого получаем суммарные затраты на разработку:

руб.

## **5.5 Расчет частных экономических эффектов от производства и**

## **использования программного продукта**

В качестве модели распространения разрабатываемого ПО была принята модель *free-to-play*, следовательно цена на использование продукта составляет 0 руб. Для получения прибыли планируется использование так называемой рекламы в интернете. Средняя стоимость рекламы за 100 показов составляет 1ц,5 руб. Прибыль от перехода пользователя по баннеру варьируется в зависимости от региона: стоимость перехода от американского пользователя составляет 0,4 руб, стоимость же перехода в России и Беларуси составляет 0,09 руб. Следовательно, американские пользователи являются приоритетнее, так как разница в прибыли за переход по баннеру составляет примерно 0,31 руб.

Доход от реализации *free-to-play* игр рассчитывается по формуле:

руб, (5.30)

где – количество пользователей в день, равняется 150;

– средний ежедневных доход с пользователя, равняется 0,18 руб.;

– количество дней, равняется 30;

– количество месяцев, равняется 12.

Прибыль за реализацию за 12 месяцев составит:

Таким образом можно рассчитать сроки окупаемости проекта. Прибыль берется с учетом налога на доходы физических лиц (далее НДЛФ), который равняется 13% от прибыли.

лет с учетом НДФЛ.

Рентабельность проекта рассчитывается по формуле (5.31):

(5.31)

где П– показатель прибыли в год;

В – показатель выручки.

Таким образом, по результатам рассчитанных показателей установлено, что реализация дипломной работы является экономически целесообразной.

# **6 ОХРАНА ТРУДА И ТЕХНИКА БЕЗОПАСНОСТИ**

## **6.1 Факторы, влияющие на тяжесть поражения электрическим током**

Тяжесть поражения человека электрическим током зависит от совокупности многих факторов, и их можно разделить на группы: электрического, неэлектрического характера, факторы производственной среды (рисунок 6.1).

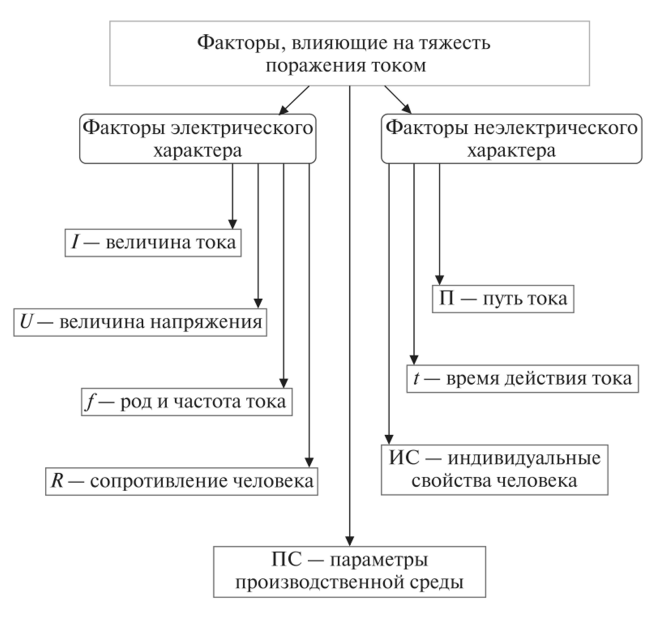


Рисунок 6.1 – Факторы, влияющие на тяжесть поражения электротоком

Рассмотрим факторы электрического характера. Основным поражающим фактором принято считать электрический ток. Но, согласно закону Ома, ток представляет собой производную от величины приложенного к человеку напряжения. Следовательно, рассматривать действие тока в отрыве от напряжения, сопротивления тела человека и других факторов не совсем корректно. Тем не менее рассмотрим эти факторы как самостоятельные, для наглядности их влияния на тяжесть травмирования.

Рассмотрим реакции человеческого организма, вызванные электрическим током различного вида и различной силы при прохождении тока в направлении «рука – рука» или «рука – нога».

Обычно человек начинает ощущать раздражающее действие переменного тока промышленной частоты 50 Гц при величине 0.6–1.5 мА и постоянного тока 5–7 мА. Эти токи называются ощутимыми пороговыми токами. Они не представляют опасности для человека, и человек может самостоятельно отключиться от цепи.

При переменных токах 5–10 мА раздражающее действие электрического тока становится более сильным, появляется боль в мышцах и непроизвольное их сокращение.

При токах 10–15 мА боль в мышцах становится такой сильной, что человек уже не в состоянии самостоятельно освободиться от действия тока (не может разжать руку, отбросить от себя провод и т.д).

Переменные токи 10–15 мА и выше и постоянные токи 50–80 мА и выше называются неотпускающими токами.

Переменный ток 25 мА и выше (в зависимости от пути прохождения тока) воздействует на мышцы грудной клетки, что может привести к параличу дыхания и вызвать смерть человека.

Электрический ток около 100 мА и более при частоте 50 Гц и 300 мА и более при постоянном напряжении за короткое время (1–2 с) поражает мышцу сердца человека и вызывает его фибрилляцию. Эти токи называются фибрилляционными.

Токи более 5 А вызывают паралич сердца и дыхания, минуя стадию фибрилляции сердца. При длительном протекании тока (несколько секунд) – тяжёлые ожоги, разрушение тканей организма человека.

Длительностьпротеканиятока через тело человека влияет на исход поражения вследствие того, что со временем резко нарастает ток за счет уменьшения сопротивления тела и накапливаются отрицательные последствия воздействия тока на организм. Через 30 с сопротивление тела человека протеканию тока падает примерно на 20%, а через 90 с – на 70%.

Фактор, род и частота тока по-разному воздействует на человека в зависимости от переменный или постоянный ток. При прикосновении к токоведущим частям, находящимся под напряжением, переменный ток, протекающий через человека, приводит к судорожным сокращениям мышц руки, в которой зажат проводник, при этом пострадавший самостоятельно не может освободиться от действия тока.

При переменном токе опасность поражения растет вместе с ростом тока, проходящего через человека; поэтому следует ожидать, что увеличение частоты ведет к повышению этой опасности. Действительность показывает, что это предположение справедливо лишь в диапазоне частот от 0 до 50 Гц; дальнейшее же повышение частоты, несмотря на рост тока, проходящего через человека, сопровождается снижением опасности поражения, которая полностью исчезает при частоте 450–500 кГц, т. е. такие токи не могут поразить человека. Правда, эти токи сохраняют опасность ожогов как при возникновении электрической дуги, так и при прохождении тока непосредственно через человека.

Постоянный ток приводит к отбросу пострадавшего от токоведущих частей, что может привести к механическим повреждениям (вывихи, ушибы, переломы ит.п.). Если напряжение протекающего тока не превышает 500 В, то воздействие постоянного тока на организм человека меньше, чем переменного тока. А если напряжение выше 500 В, то постоянный ток становится опаснее переменного.

Чем больше частота переменного тока превышает 50 Гц, тем меньше последствия электротравмы.

Сопротивление тела человека – величина не постоянная, ее значение зависит от многих факторов: от состояния человека на момент контакта (психического и физического), от параметров замкнутой цепи, от внешних условий среды, в которой человек на момент удара находится.

Тело человека состоит из различных тканей, и каждый вид тканей обладает своим сопротивлением. Так например, сухожилия, кожа, жировая ткань, хрящи и кости имеют удельное сопротивление порядка 3–20 кОм/м. Кровь, мышцы, лимфа, головной и спинной мозг – всего от 0,5 до 1 Ом/м. Из всех этих тканей наибольшим сопротивлением отличается кожа, поэтому именно кожа в значительной степени определяет сопротивление человеческого тела электрическому току. Электрическое сопротивление тела человека зависит от следующих пяти факторов:

* от общего психологического и физиологического состояния (индивидуальные особенности);
* от пола – от толщины кожи (у мужчин сопротивление выше, чем у женщин);
* от возраста – от грубости кожи (у взрослых сопротивление выше, чем у детей);
* от внешних условий (температура, давление, влажность, плотность);
* от общего состояния кожи (раны, грязь, увлажненность и т. д.);
* от внешних раздражителей (внезапные удар, укол, свет или звук), способных снизить сопротивление на 20–50 % за несколько минут.

Существенное значение имеет путь протекания тока через тело человека. Наиболее часто встречающиеся пути протекания тока через организм человека: «правая рука – ноги», «левая рука – ноги», «рука – рука», «нога – нога» (рисунок 6.2).

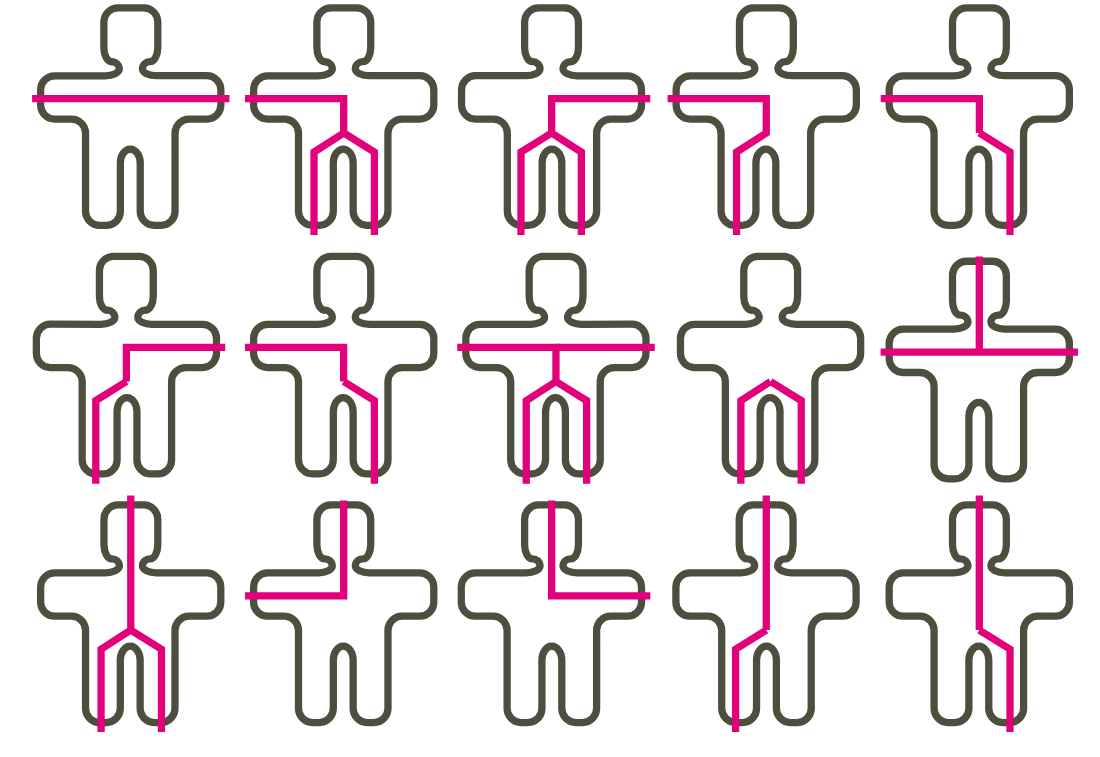


Рисунок 6.2 – Схема, пути протекания тока

Наибольшая опасность возникает при непосредственном прохождении тока через жизненно важные органы (сердце, лёгкие, головной мозг). Поэтому наиболее опасными следует признать пути протекания: «левая рука – ноги», «рука – рука», а также «голова – рука», «голова – ноги».

Наименее опасным путём тока (из наиболее часто встречающихся) является путь «нога – нога», когда человек попадает под шаговое напряжение.

Опасность поражения электрическим током зависит также от места контакта тела человека с токоведущей частью, то есть от места «входа тока» в организм. Например, при касании человека токоведущей части рукой, ток может входить через ладонь или тыльную часть руки, через пальцы или всю поверхность руки и т.д.

Наиболее опасными местами входа тока являются: тыльная сторона ладони, шея, голень, виски, грудь. Следует отметить, что данные места на теле человека обладают повышенной электропроводностью.

Имеют значение также индивидуальные особенности тела человека. Полностью здоровые люди во много раз выносливее, чем больные.

Чем дольше проходит ток через тело человека, тем больше снижается сопротивление организма, тем сильнее последствия, вызванные током.

# **7 ЭНЕРГОСБЕРЕЖЕНИЕ И РЕСУРСОСБЕРЕЖЕНИЕ ПРИ ЭКСПЛУАТАЦИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

## **7.1 Энергосбережение и ресурсосбережение при внедрении и эксплуатации программного обеспечения**

Энергосбережение – это комплекс мер по реализации правовых, организационных, научных, производственных, технических и экономических мер, направленных на эффективное (рациональное) использование (и экономное расходование) топливно-энергетических ресурсов и на вовлечение в хозяйственный оборот возобновляемых источников энергии [12, с. 9].

Актуальность энергосбережения растет во всех странах, особенно в странах с ограниченными энергоресурсами, в связи с опережающим ростом цен на основные традиционные виды энергоресурсов и постепенным истощением их мировых запасов.

Ресурсосбережение – это организационная, экономическая, техническая, научная, практическая, информационная деятельность, методы, процессы, комплекс организационно-технических мер и мероприятий, сопровождающие все стадии жизненного цикла изделий и направленные на рациональное использование и экономию ресурсов [12, с. 5].

В настоящее время ресурсосбережение – одна из приоритетных задач экономики Республики Беларусь. Это связано прежде всего с дефицитом многих видов ресурсов, ростом стоимости их добычи (подготовки), а также серьезными экологическими проблемами [13, с. 63].

К задачам ресурсосбережения можно отнести:

– сбережение топлива и энергии;

– рациональное использование и экономия материальных ресурсов;

– максимальное сохранение природных ресурсов;

– сохранение равновесия между развитием производств и потреблением вторичных материальных ресурсов с сохранением устойчивости окружающей техногенной среды;

– совершенствование систем управления качеством производства продукции, ее реализации и потребления, оказания услуг;

– обеспечение экономически эффективного и безопасного использования вторичных материальных ресурсов.

При разработке программного обеспечения требуется соблюдать основные принципы энерго- и ресурсосбережения. Экономия электроэнергии – крайне важный аспект жизни современного человеческого общества, затрагивающий и производственную сферу, и быт каждого отдельно взятого индивидуума. Неразумное потребление достаточно дорогостоящего вида энергии может привести к весьма значительным тратам, что может существенно сказаться как на благосостоянии человека, так и на развитии предприятия.

На сегодняшний день существуют самые разнообразные пути экономии электроэнергии, которые могут оказаться либо эффективными, либо не очень. Для уменьшения потребляемой электроэнергии необходимо проводить следующие процедуры:

– использовать персональные компьютеры лишь в дневное время, так как на многих предприятиях тариф электроэнергии в ночное время превышает тариф за неё в дневное время в несколько раз, что означает дополнительную трату средств;

– обновление всех ПК и периферийных устройств, обращая внимание на стандарт энергопотребления;

– заменить мониторы на аналоги, с меньшим энергопотреблением;

– не оставлять устройства включёнными на длительное время, если за ним никто не работает.

Неиспользуемый два часа компьютер даже в «спящем режиме» потребляет 200–300 Вт, за месяц порядка 12 кВт·ч. Периферийные устройства рекомендуется всегда выключать, если они не используются. Это еще позволит сэкономить порядка 2–3 кВт·ч за месяц.

## **7.2 Вопросы энерго- и ресурсосбережения, связанные с эксплуатацией**

## **интерактивной системы**

Разрабатываемая игровое приложение несет в себе задачу не снизить количество персонала на предприятии и т.д., а скорее развлекательную задачу и является автономным игровым приложением, не нуждающимся во внедрении. Следовательно, основные задачи сбережения ресурсов сводятся к следующим:

– снижение затрат на электричество при эксплуатации;

– снижение потребляемого объема памяти приложением.

Описанные задачи являются взаимосвязанными и для решения требуют:

– снизить объем ресурсов, используемых приложением: уменьшение размера текстур игровых моделей, оптимизация трехмерных моделей посредством уменьшения количества полигонов, что приедет к уменьшению размеров занимаемой моделями памяти;

– исключить процесс инициализации идентичных игровых ресурсов в процессе работы приложения, что также приведет к уменьшению объема занимаемой памяти, а также ускорит приложение и снизит затраты на использование электроэнергии.

Игровое приложение выполняет все перечисленные требования, и, как следствие, используем минимально возможное количество энергоресурсов при эксплуатации.

Человеческое здоровье – самый невосполнимый и дорогостоящий ресурс. С целью уменьшения излучение с экрана, графическое наполнение системы разработано в выдержанных тонах и не перегружено лишними элементами, что способствует снижению нагрузки на глаза пользователя при эксплуатации. Приложение также лишено моментов резкого изменения цветов и мерцания, что позволяет пользователям проводить больше времени в игровом приложение без появления усталости глаз.

Для уменьшения нагрузки на зрение пользователя, игровое приложение использует:

– выдержанную, неконтрастную цветовую гамму для элементов пользовательского графического интерфейса;

– плавные анимации изменения состояния отдельных игровых элементов без наличия резких переходов;

– наличие только самых необходимых элементов анимации, что также способствует снижению нагрузки на глаза пользователя.

Таким образом, разработанное приложение реализует большинство возможных подходов для снижения потребления ресурсов системой и психологической нагрузки пользователя при использовании. Кроме того, планируется доработка уже имеющихся графических ресурсов, что позволит еще больше сократить энергопотребление и занимаемый объем памяти.

# **ЗАКЛЮЧЕНИЕ**

Результатом дипломной работы является мобильное игровое приложение «*Mythical Towers*».

В ходе разработки решены следующие задачи:

– произведен аналитический обзор доступных программных средств разработки, позволяющих разработать мобильное игровое приложение;

– разработан пользовательский графический интерфейс пользователя, реализующий взаимодействие пользователя с программой;

– разработана общая игровая структура и механики, позволяющие осуществлять непосредственно игровой процесс;

– применены подходы проектирования на основе *Unity*3*d*, позволяющие разработать масштабируемое приложение;

– произведена верификация и апробация конечного программного продукта.

Функционал игрового приложения реализован с использованием среды разработки *Unity*3*d*.

Отличительной особенностью игрового приложения от аналогов является наличие разнообразия в игровых механиках.

Тестирование разработанного приложения показало, что игровое приложение выполняет свои функции, игровые механики функционируют должным образом. Кроме того, система является масштабируемой, что позволяет впоследствии внедрять новую функциональность без внесения изменений в уже существующий код программы.

# **Список использованных источников**

1. Хокинг, Д.: Unity в действии. Мультиплатформенная разработка на C#/ Д. Хокинг. – 2-е межд. изд. – СПб.: Питер, 2019. – 352 с.

2. Halpern, J.: Developing 2D Games with Unity: Independent Game Programming with C#/ J. Halpern. – New York, NY, USA, 2019. – 398 с.

3. Okita, A.: Learning C# Programming with Unity3D/ A.Okita. – Boca Raton, London, New York, 2015. – 686 с.

4. Куксон, А.: Разработка игр на Unreal Engine 4 за 24 часа / А. Куксон, Райан Даулингсока, Клинтон Крамплер; [перевод с анг. М. А. Райтман]. – Москва: Эксмо, 2019. – 528 с.

5. Shannon, T: Unreal Engine 4 for Design Visualization: Developing Stunning Interactive Visualizations, Animations, and Renderings/ T. Shannon. – Boston, 2017. – 260 с.

6. Горелик, А. Г.: Самоучитель 3ds Max 2018/ А. Г. Горелик. – СПб.: БХВ-Петербург, 2018. – 528 с.

7. Unity Real-Time Development Platform. – Электронные данные. – Unity.com. – Режим доступа: https://unity.com/. – Дата доступа: 04.05.2022.

8. The most powerful real-time 3D creation platform – Unreal Engine. – Электронные данные. – EnrealEngine.com. – Режим доступа: https://www.unrealengine.com/. – Дата доступа: 04.05.2022.

9. Кожевников, Е.А. Расчёт экономической эффективности разработки программных продуктов: метод. указания по подготовке организационно-эконо-мического раздела дипломных работ для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» дневной формы обучения / Е.А. Кожевников, Н.В. Ермалинская. – Гомель: ГГТУ им. П.О. Су-хого, 2012. – 68 с.

10. Факторы, влияющие на степень поражения электрическим током. – Электронные данные. – Studfile.net. – Режим доступа: https://studfile.net/preview/4080619/page:33/. – Дата доступа: 04.05.2022.

11. Факторы, влияющие на исход поражения электрическим током. – Электронные данные. – Yaklass.ru. – Режим доступа: http://surl.li/cdyjy. – Дата доступа: 04.05.2022.

12. Чернецова, А.А.: Энерго- и ресурсосбережение / А.А. Чернецова – Хабаровск: Изд-во Тихоокеан. гос. ун-та, 2018. – 125 с.

13. Самойлов, М.В.: Основы энергосбережения / М.В. Самойлов, В.В. Па-невчик, А.Н. Ковалёв. – Мн.: БГЭУ, 2002. – 198 с.

# **ПРИЛОЖЕНИЕ А**

(обязательное)

**Листинг программы**

**Листинг класса BuildManager.cs**

using UnityEngine;

public class BuildManager : MonoBehaviour

{

public static BuildManager instance;

void Awake()

{

if (instance != null)

{

return;

}

instance = this;

}

public GameObject simpleTurretPrefab;

public GameObject strongTurretPrefab;

public GameObject fastTurretPrefab;

private TurretPrint turretToBuild;

private Node selectNode;

public NodeUi nodeUi;

public bool CanBuild {get { return turretToBuild != null; } }

public bool HasMoney { get { return PlayerStats.Money >= turretToBuild.cost; } }

public void BuildTurretOn (Node node)

{

if(PlayerStats.Money < turretToBuild.cost)

{

Debug.Log("No Mony");

return;

}

PlayerStats.Money -= turretToBuild.cost;

GameObject turret = (GameObject)Instantiate(turretToBuild.prefab, node.GetBuildPossition(), Quaternion.identity);

node.turret = turret;

Debug.Log("Turret build! Money left " + PlayerStats.Money);

}

public void SelectNode (Node node)

{

if(selectNode == node)

{

DeselectNode();

return;

}

selectNode = node;

turretToBuild = null;

nodeUi.SetTarget(node);

}

public void DeselectNode()

{

selectNode = null;

nodeUi.Hide();

}

public void SetTurretBuild(TurretPrint turret)

{

turretToBuild = turret;

DeselectNode();

}

public TurretPrint GetTurretBuild ()

{

return turretToBuild;

}

}

**Листинг класса Bullet.cs**

using UnityEngine;

public class Bullet : MonoBehaviour

{

private Transform target;

public int damage = 30;

public float speed = 70f;

public GameObject impactEffect;

// поиск врага пулей

public void Seek(Transform \_target)

{

target = \_target;

}

void Update()

{

if (target == null)

{

Destroy(gameObject);

return;

}

Vector3 dir = target.position - transform.position;

float distanceThisFrame = speed \* Time.deltaTime;

if(dir.magnitude <= distanceThisFrame)

{

HitTarget();

return;

}

transform.Translate(dir.normalized \* distanceThisFrame, Space.World);

}

void HitTarget()

{

GameObject effectIns = Instantiate(impactEffect, transform.position, transform.rotation);

Destroy(effectIns, 2f);

//Destroy(target.gameObject);

Damage(target);

Destroy(gameObject);

}

void Damage(Transform enemy)

{

Emeny e = enemy.GetComponent<Emeny>();

if( e != null)

{

e.TakeDamage(damage);

}

}

}

**Листинг класса Emeny.cs**

using UnityEngine.UI;

using UnityEngine;

public class Emeny : MonoBehaviour

{

public float speed = 10f;

public int startHealth = 100;

private float health;

public int moneyUp = 20;

private Transform target;

private int waypointIndex = 0;

public Image healthsBar;

private void Start()

{

target = Waypoints.points[0];

health = startHealth;

}

public void TakeDamage(int amount)

{

health -= amount;

healthsBar.fillAmount = health / startHealth;

if (health <=0)

{

Die();

}

}

void Die()

{

PlayerStats.Money += moneyUp;

WaveSpawner.EnemyAlive--;

Destroy(gameObject);

}

void Update()

{

Vector3 dir = target.position - transform.position;

transform.Translate(dir.normalized \* speed \* Time.deltaTime, Space.World);

if (Vector3.Distance(transform.position, target.position) <= 0.4f)

{

GetNextWaypoint();

}

}

void GetNextWaypoint()

{

if(waypointIndex >= Waypoints.points.Length - 1)

{

EndPath();

return;

}

waypointIndex++;

target = Waypoints.points[waypointIndex];

}

void EndPath ()

{

PlayerStats.Lives--;

WaveSpawner.EnemyAlive--;

Destroy(gameObject);

}

}

**Листинг класса GameMenedger.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class GameMenedger : MonoBehaviour

{

private static bool gameEnded;

public GameObject gameOverUi;

void Start()

{

gameEnded = false;

}

void Update()

{

if (gameEnded)

return;

if (PlayerStats.Lives <=0)

{

EndGame();

}

}

void EndGame ()

{

gameEnded = true;

gameOverUi.SetActive(true);

}

}

**Листинг класса GameOver.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

public class GameOver : MonoBehaviour

{

public void Retry ()

{

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);

}

}

**Листинг класса LevelSelector.cs**

using UnityEngine.SceneManagement;

using UnityEngine;

public class LevelSelector : MonoBehaviour

{

public void Select( string levelName)

{

SceneManager.LoadScene(levelName);

}

}

**Листинг класса MainMenu.cs**

using UnityEngine;

using UnityEngine.SceneManagement;

public class MainMenu : MonoBehaviour

{

public string levelTolevel = "Main Scene";

public void Play()

{

SceneManager.LoadScene(levelTolevel);

}

public void Exit ()

{

Debug.Log("exit");

Application.Quit();

}

}

**Листинг класса MoneyUI.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class MoneyUI : MonoBehaviour

{

public Text moneyText;

void Update()

{

moneyText.text = "$" + PlayerStats.Money.ToString();

}

}

**Листинг класса Node.cs**

using UnityEngine;

using UnityEngine.EventSystems;

public class Node : MonoBehaviour

{

public Color hoverColor;

public Vector3 positionOffset;

public Quaternion rotationOffset ;

[HideInInspector]

public GameObject turret;

[HideInInspector]

public TurretPrint turretPrint;

[HideInInspector]

public bool isUpgrate = false;

private Renderer rend;

private Color startColor;

BuildManager buildManager;

void Start()

{

rend = GetComponent<Renderer>();

startColor = rend.material.color;

buildManager = BuildManager.instance;

}

public Vector3 GetBuildPossition()

{

return transform.position + positionOffset;

}

void OnMouseDown()

{

if (EventSystem.current.IsPointerOverGameObject())

return;

if(turret != null)

{

buildManager.SelectNode(this);

return;

}

if (!buildManager.CanBuild)

return;

BuildTurret(buildManager.GetTurretBuild());

}

void BuildTurret (TurretPrint print)

{

if (PlayerStats.Money < print.cost)

{

Debug.Log("No Mony");

return;

}

PlayerStats.Money -= print.cost;

GameObject \_turret = (GameObject)Instantiate(print.prefab, GetBuildPossition(), Quaternion.identity \* rotationOffset);

turret = \_turret;

turretPrint = print;

Debug.Log("Turret build!");

}

public void UpgradeTurret()

{

if (PlayerStats.Money < turretPrint.upgrateCoast)

{

Debug.Log("No Mony");

return;

}

PlayerStats.Money -= turretPrint.upgrateCoast;

// удаление старой башни для создания прокаченой версии

Destroy(turret);

//постройка новой башни

GameObject \_turret = (GameObject)Instantiate(turretPrint.upgratePrefab, GetBuildPossition(), Quaternion.identity \* rotationOffset);

turret = \_turret;

isUpgrate = true;

Debug.Log("Turret build!");

}

public void SellTurret()

{

PlayerStats.Money += turretPrint.GetSellAmount();

Destroy(turret);

turretPrint = null;

}

//мышка

void OnMouseEnter()

{

if (EventSystem.current.IsPointerOverGameObject())

return;

if (!buildManager.CanBuild)

return;

}

void OnMouseExit()

{

rend.material.color = startColor;

}

}

**Листинг класса NodeUi.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class NodeUi : MonoBehaviour

{

public GameObject ui;

private Node target;

public Text updateCoast;

public Button updateButton;

public Text sellAmount;

public void SetTarget(Node \_target)

{

target = \_target;

transform.position = target.GetBuildPossition();

if (!target.isUpgrate)

{

updateCoast.text = "$" + target.turretPrint.upgrateCoast;

updateButton.interactable = true;

}else

{

updateCoast.text = "DODE";

updateButton.interactable = false;

}

sellAmount.text = "$" + target.turretPrint.GetSellAmount();

ui.SetActive(true);

}

public void Hide ()

{

ui.SetActive(false);

}

public void Upgrate()

{

target.UpgradeTurret();

BuildManager.instance.DeselectNode();

}

public void Sell()

{

target.SellTurret();

BuildManager.instance.DeselectNode();

}

}

**Листинг класса PlayerStats.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class PlayerStats : MonoBehaviour

{

public static int Money;

public int startMoney = 400;

public static int Lives;

public int startLives = 20;

private void Start()

{

Money = startMoney;

Lives = startLives;

}

}

**Листинг класса Shop.cs**

using UnityEngine;

public class Shop : MonoBehaviour

{

public TurretPrint simpleTurret;

public TurretPrint fastTurret;

public TurretPrint strongTurret;

BuildManager buildManager;

void Start()

{

buildManager = BuildManager.instance;

}

public void BuySimpleGun()

{

Debug.Log("Buy 1");

buildManager.SetTurretBuild(simpleTurret);

}

public void BuyFastGun()

{

Debug.Log("Buy 2");

buildManager.SetTurretBuild(fastTurret);

}

public void BuyStrongGun()

{

Debug.Log("Buy 3");

buildManager.SetTurretBuild(strongTurret);

}

}

**Листинг класса TurretPrint.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

[System.Serializable]

public class TurretPrint

{

public GameObject prefab;

public int cost;

public GameObject upgratePrefab;

public int upgrateCoast;

public int GetSellAmount()

{

return cost / 2;

}

}

**Листинг класса Turret.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Turret : MonoBehaviour

{

private Transform target;

[Header("Атрибуты")]

public float range = 15f;

public float fireRate = 1f;

private float fireCountdown = 0f;

[Header("Настройик врагов")]

public string enemyTag = "Enemy";

public Transform partToRotate;

public float ternSpeed = 10f;

public GameObject bulletPrefab;

public Transform firePoint;

// Start is called before the first frame update

void Start()

{

InvokeRepeating("UpdateTarget", 0f, 0.5f);// для оптимизации 0.5 увеличивать

}

void UpdateTarget()

{

GameObject[] enemies = GameObject.FindGameObjectsWithTag(enemyTag);

float shortesDistans = Mathf.Infinity;

GameObject nearestEmeny = null;

foreach (GameObject enemy in enemies)

{

float distanceToEmeny = Vector3.Distance(transform.position, enemy.transform.position);

if(distanceToEmeny < shortesDistans)

{

shortesDistans = distanceToEmeny;

nearestEmeny = enemy;

}

}

if( nearestEmeny != null && shortesDistans <= range)

{

target = nearestEmeny.transform;

} else

{

target = null;

}

}

// Update is called once per frame

void Update()

{

if (target == null)

return;

// поворот на ближайшего врага

Vector3 dir = target.position - transform.position;

Quaternion lookRotation = Quaternion.LookRotation(dir);

Vector3 rotation = Quaternion.Lerp(partToRotate.rotation, lookRotation, Time.deltaTime \* ternSpeed).eulerAngles;

partToRotate.rotation = Quaternion.Euler(-90f, rotation.y, 0f);

if(fireCountdown <= 0 )

{

Shoot();

fireCountdown = 1f / fireRate;

}

fireCountdown -= Time.deltaTime;

}

void Shoot ()

{

GameObject bulletGo = (GameObject)Instantiate(bulletPrefab, firePoint.position, firePoint.rotation);

Bullet bullet = bulletGo.GetComponent<Bullet>();

if (bullet != null)

bullet.Seek(target);

}

private void OnDrawGizmosSelected()

{

Gizmos.color = Color.red;

Gizmos.DrawWireSphere(transform.position, range);

}

}

**Листинг класса Wave.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

[System.Serializable]

public class Wave

{

public GameObject enemy;

// public GameObject enemy2;

public int count;

public float rate;

}

**Листинг класса WaveSpawner.cs**

using UnityEngine;

using System.Collections;

using UnityEngine.UI;

using UnityEngine.SceneManagement;

public class WaveSpawner : MonoBehaviour

{

public static int EnemyAlive = 0;

public string levelTolevel = "Main Scene";

public Wave[] waves;

public Transform SpawnPoint;

public float timeBetweenWaves = 5f;

private float countdown = 2f;

public Text waveCountdowntext;

private int WaveIndex = 0;

void Update()

{

if(EnemyAlive > 0 )

{

return;

}

if (countdown <= 0f)

{

StartCoroutine(SpawnWave());

// SpawnWave();

countdown = timeBetweenWaves;

return;

}

countdown -= Time.deltaTime;

countdown = Mathf.Clamp(countdown, 0f, Mathf.Infinity);

waveCountdowntext.text = string.Format("{0:00.00}", countdown);

}

IEnumerator SpawnWave()

{

Wave wave = waves[WaveIndex];

for (int i = 0; i < wave.count; i++)

{

SpawnEmeny(wave.enemy);

// SpawnEmeny(wave.enemy2);

yield return new WaitForSeconds(1f/ wave.rate);

}

WaveIndex++;

if (WaveIndex == waves.Length)

{

SceneManager.LoadScene(levelTolevel);

Debug.Log("Level wom!");

this.enabled = false;

}

}

void SpawnEmeny(GameObject enemy)

{

Instantiate(enemy, SpawnPoint.position, SpawnPoint.rotation);

EnemyAlive++;

}

}

**Листинг класса Waypoints.cs**

using UnityEngine;

public class Waypoints : MonoBehaviour

{

public static Transform[] points;

private void Awake()

{

points = new Transform[transform.childCount];

for(int i = 0; i < points.Length; i++)

{

points[i] = transform.GetChild(i);

}

}

}

**ПРИЛОЖЕНИЕ Б**

(обязательное)

**Руководство системного программиста**

1. Назначение и условия применения

Разработанное программное приложение предназначено для организации игрового процесса. Название игры – «*Mythical Towers*». Предполагается наличие игрока, который должен защитить башню от нападающих врагов. Управление игроком осуществляется с помощью нажатий на экран.

Для корректной работы программного средства необходимо соблюдение следующих требований:

* телефон на операционной системе *Android*.

1. Структура программы

Структурно приложение состоит из одной рабочей области –игрового окна.

1. Проверка программы

Программа была проверена при помощи модульного тестирования. При прохождении тестов ошибок не обнаружено. Отсутствие сообщений об ошибках при работе приложения также свидетельствует о корректной работе приложения.

1. Сообщения системному программисту

При появлении исключения требуется перезапустить игру.

# **ПРИЛОЖЕНИЕ В**

(обязательное)

**Руководство программиста**

1. Назначение и условия применения программы

Разработанное программное приложение предназначено для организации игрового процесса.

Для запуска и дальнейшей работы приложения необходимо наличие устройства на платформе Android 7.0 и выше.

2. Характеристика программы

Приложение начинает работу после запуска файла с расширением *.apk*. После запуска приложения на экране появится вступительная заставка *Unity*3*d* после чего, будет произведен запуск приложения.

3. Обращение к программе

Для обращения к программе необходимо запустить на мобильном устройстве файл с именем программы и расширением *.apk*.

4. Входные и выходные данные

Входными данными являются данные, полученные приложением после взаимодействия пользователя с сенсорным экраном мобильного устройства.

Выходными данными является графическая информация, выводимая на экран мобильного устройства.

5. Сообщения

Сообщения о несоответствии версии операционной системы.

# **ПРИЛОЖЕНИЕ Г**

(обязательное)

**Руководство пользователя**

1. Введение

Руководство пользователя обеспечивает получение пользователем базовых навыков по эксплуатации игрового приложения.

Разработанное программное приложение предназначено для организации игрового процесса. Предполагается наличие игрока, который должны одержать победу врагами. Управление игроком осуществляется с помощью нажатие на экран.

Программное средство обладает следующим функционалом:

* игровое поле;
* управление игроками при помощи нажатия на экран телефона;
* наличие игровых объектов.

Для использования программного средство пользователь должен быть ознакомлен с:

* настоящим руководством пользователя;
* правилами использования мобильного телефона.

1. Назначение и условия применения

Разработанное программное приложение предназначено для развлечения пользователя.

Для корректной работы программного средства необходимо соблюдение следующих требований:

* телефон на операционной системе *Android*.

1. Подготовка к работе

Для установки приложения необходимо загрузить на телефон установочное приложение *Build*.*apx*. После этого установить приложение на телефон, после установки запустить приложение *Tower* *defense*.

Если все инструкции соблюдены и приложение не выдает никаких сообщений об ошибках, значит, программа работает исправно.

1. Описание операций

Пользователю доступны следующие операции:

* расстановка башен при помощи нажатия на экран.

1. Аварийные ситуации

Чтобы избежать ошибок при использовании данного игрового приложения, необходимо соблюдать порядок действий и условия использования, описанные в главе 3 данного руководства пользователя.

В случае непредвиденного «зависания» программы рекомендуется завершить процесс в диспетчере задач и запустить снова.

1. Рекомендации по освоению

Приложение имеет простую реализацию, а все операции понятны на интуитивном уровне.

# **ПРИЛОЖЕНИЕ Д**

(справочное)

**Результаты расчета экономического обоснования**

Таблица Д.1 – Расчет коэффициента экономического обоснования

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Наименование  параметра | Вес  параметра, | Значение  параметра | | |  |  |  |  |
|  |  |  |
| Объём памяти | 0,3 | 9 | 8 | 6 | 1,5 | 1,33 | 0,45 | 0,4 |
| Время обработки  данных | 0,3 | 0,6 | 0,7 | 0,2 | 3 | 3,5 | 0,9 | 1,05 |
| Отказы | 0,6 | 1 | 2 | 1 | 1 | 2 | 0,6 | 1,2 |
| Итого | | | | | | | 1,95 | 2,65 |
| Коэффициент эквивалентности | | | | | | | 2,65/1,95=1,36 | |

Таблица Д.2 – Расчет коэффициента изменения функциональных возможностей

|  |  |  |
| --- | --- | --- |
| Наименование показателя | Балльная оценка  базового ПП | Балльная оценка  нового ПП |
| Объём памяти | 3 | 4 |
| Быстродействие | 4 | 4 |
| Удобство интерфейса | 3 | 5 |
| Степень утомляемости | 2 | 3 |
| Производительность труда | 3 | 4 |
| Итого | 15 | 20 |
| Коэффициент функциональных возможностей | 20/15 = 1,33 | |

Таблица Д.3 – Расчет уровня конкурентоспособности нового ПП

|  |  |
| --- | --- |
| Коэффициенты | Значение |
| Коэффициент эквивалентности () | 1,36 |
| Коэффициент изменения функциональных возможностей () | 1,33 |
| Коэффициент соответствия нормативам () | 1 |
| Коэффициент цены потребления () | 0,93 |
| Интегральный коэффициент конкурентоспособности | (1,33 1,36 1)/0,93 = 1,94 |

Таблица Д.4 – Перечень и объем функций ПО

|  |  |  |  |
| --- | --- | --- | --- |
| Код функций | Наименование (содержание) функций | Объем функции строк исходного кода ( *1000* ) | |
| по каталогу (*V*о) | уточненный (*V*y) |
| 101 | Организация ввода информации | 150 | 200 |
| 102 | Контроль, предварительная обработка и ввод информации | 550 | 500 |
| 305 | Формирование файла | 2460 | 400 |
| 303 | Обработка файлов | 1100 | 430 |
| 506 | Обработка ошибочных сбойных ситуаций | 1720 | 200 |
| 507 | Обеспечение интерфейса между компонентами | 1820 | 1000 |
| 702 | Расчетные задачи (расчет режимов обработки) | 1330 | 300 |
| 706 | Предварительная обработка, печать | 470 | 350 |
| 707 | Графический вывод результатов | 590 | 700 |
| Итого | | 10190 | 4080 |

Таблица Д.5 – Значения коэффициентов удельных весов трудоемкости стадий разработки ПО в общей трудоемкости

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Категория новизны ПО | Без применения CASE-технологий | | | | |
| Стадии разработки ПО | | | | |
| ТЗ | ЭП | ТП | РП | ВН |
| Значения коэффициентов | | | | |
|  |  |  |  |  |
| Б | 0,1 | 0,2 | 0,3 | 0,3 | 0,1 |

Таблица Д.6 – Расчет общей трудоемкости разработки ПО

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Показатели | Стадии разработки | | | | | Итого |
| ТЗ | ЭП | ТП | РП | ВН |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Общий объем ПО , кол-во строк (*LOC*) | – | – | – | – | – | 10190 |

Продолжение таблицы Д.6

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Общий объем ПО (*Vо*), кол-во строк (*LOC*) | – | – | – | – | – | 10190 |
| Общий уточненный объем ПО (*V*y), кол-во строк (*LOC*) | – | – | – | – | – | 4080 |
| Категория сложности разрабатываемого ПО | – | – | – | – | – | 2 |
| Нормативная трудоемкость разработки ПО (Тн), чел.-дн. | – | – | – | – | – | 213 |
| Коэффициент повышения сложности ПО (Кс) | 1,12 | 1,12 | 1,12 | 1,12 | 1,12 | – |
| Коэффициент, учитывающий новизну ПО (Кн) | 0,72 | 0,72 | 0,72 | 0,72 | 0,72 | – |
| Коэффициент, учитывающий степень использования стандартных модулей (Кт) | – | – | – | 0,77 | – | – |
| Коэффициент, учитывающий средства разработки ПО (Ку.р) | 0,55 | 0,55 | 0,55 | 0,55 | 0,55 | – |
| Коэффициенты удельных весов трудоемкости стадий разработки ПО (Ктз, Кэп, Ктп, Крп, Квн) | 0,1 | 0,2 | 0,3 | 0,3 | 0,1 | 1 |
| Распределение скорректированной (с учетом Кс, Кн, Ку.р) трудоемкости ПО по стадиям, чел.-дн. | 9 | 18 | 28 | 22 | 9 | – |
| Общая трудоемкость разработки ПО (То), чел.-дн. | – | – | – | – | – | 86 |

Таблица Д.7 – Параметры для расчета производственных затрат на разработку

|  |  |  |
| --- | --- | --- |
| Параметр | Единица измерения | Значение |
| 1 | 2 | 3 |
| Базовая ставка специалиста | руб. | 207 |
| Разряд разработчика | – | 12 |
| Тарифный коэффициент | – | 1,21 |
| Коэффициент Кув | – | 1,6 |
| Норматив отчислений на доп. зарплату разработчиков (Ндоп) | % | 20 |

Продолжение таблицы Д.7

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| Численность обслуживающего персонала | чел. | 1 |
| Разряд обслуживающего персонала | – | 8 |
| Тарифный коэффициент | – | 2,17 |
| Стоимость одного кВт-часа электроэнергии (СЭЛ) | руб. | 0,390852 |
| Коэффициент потерь рабочего времени (Кпот) | – | 0,2 |
| Премия | % | 5 |
| Доплата за стаж | руб. | 19,5 |

Таблица Д.8 – Расчет суммарных затрат на разработку ПО, руб

|  |  |
| --- | --- |
| Статья затрат | Итого |
| Затраты на оплату труда разработчиков (Зтр) | 4600,16 |
| Основная заработная плата разработчиков (ЗПосн) | 2860,8 |
| Дополнительная заработная плата разработчиков (ЗПдоп) | 572,16 |
| Отчисления от основной и дополнительной заработной платы (ОТЧс.н.) | 1167,20 |
| Затраты машинного времени (Змв) | 87,58 |
| Стоимость машино-часа, руб./час (Сч) | 0,39 |
| Стоимость электроэнергии, потребляемой за год (Зэ.п.) | 254,25 |
| Затраты на текущий и профилактический ремонт | 113 |
| Прочие затраты, связанные с эксплуатацией ЭВМ | 113 |
| Машинное время ЭВМ, час | 208 |
| Затраты на изготовление эталонного экземпляра (Зэт) | 130,98 |
| Затраты на технологию (Зтех) | 0 |
| Затраты на материалы (Змат) | 22,6 |
| Общепроизводственные затраты (Зобщ.пр) | 286,08 |
| Непроизводственные (коммерческие) затраты (Знепр) | 143,04 |
| Суммарные затраты на разработку ПО (Зр) | 5373,84 |

# **ПРИЛОЖЕНИЕ Е**

(справочное)

**Список публикаций**

1. Паньков Е.С. Использование динамической графики при создании игры в Unity / Е.С. Паньков // Новые математические методы и компьютерные технологии в проектировании, производстве и научных исследованиях, Гомель, 24 марта 2021 г. – Гомель, 2021. – C. 66–67.