# **Актуальность**

Видеоигра является [электронной игрой](https://en.wikipedia.org/wiki/electronic_game), которая базируется на взаимодействии человека и устройства посредством визуального [интерфейса](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81): [телевизора](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BB%D0%B5%D0%B2%D0%B8%D0%B7%D0%BE%D1%80), [монитора](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%BD%D0%B8%D1%82%D0%BE%D1%80_(%D1%83%D1%81%D1%82%D1%80%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE)) компьютера, телефона.

Кроссплатформенность видеоигры обуславливается наличием версий приложения, предназначенных для двух и более аппаратных платформ. В рамках данной дипломной работы рассматривается процесс разработки видеоигры для персональных компьютеров под управлением операционной системы Windows 10+ и для мобильных устройств под управлением ОС Android версии 5.0+.

Индустрия видеоигр, включающая разработку, издание и продвижение, – одна из самых быстро развивающихся отраслей компьютерных технологий и одновременно глобального сектора развлечений. По данным аналитиков TelecomDaily’s рост рынка видеоигр за 2022 год составил 15 процентов, а рост рынка мобильных приложений и видеоигр составил 25 процентов.

В настоящее время на российском рынке видеоигр наблюдается отток иностранных компаний, связанный с внешнеполитическими решениями правительства России. Такие иностранные компании как CD Projekt RED, Bloober Team и другие отказались от продаж в РФ, компании Steam и Nintendo блокируют покупки за российскую валюту. Вследствие образования рыночного вакуума актуализировалась отечественная разработка видеоигр.

# **Сравнение проекта с аналогами**

Необходимо рассмотреть аналогичные проекты для расширения понимания темы и увеличить набор данных для анализа. Изучение аналогичных игр позволит узнать, какие принципы и механики уже были использованы и как они были реализованы. Это поможет развить существующие идеи, а также приобрести новые. Кроме того, изучение аналогов поможет определить, какие функции и возможности могут быть добавлены в проект, чтобы улучшить его концепцию и привлекательность для пользователей. В целом, данный этап в разработке нового проекта позволяет повысить качество его реализации.

Для сравнения были выбраны Side-scrolling 2D игры в жанре приключения:

FAR: Lone Sails - это оригинальная игра в жанре приключенческой 2D-головоломки. Она отличается от большинства других игр этого жанра механиками геймплея, основанными на физическом взаимодействии с предметами, а также прекрасной визуальной стилизацией. FAR: Lone Sails имеет проработанную механику управления своим транспортом для исследования уникального мира.

Среди аналогов проекта можно выделить головоломку Limbo, которая также является 2D-приключением с уникальным визуальным стилем. Также можно отметить Inside, которая имеет сходства с FAR: Lone Sails в плане геймплея и атмосферы. Кроме того, игра не содержит насилия и позволяет наслаждаться красотой пустынных ландшафтов.

Анализируя данный сегмент игровой индустрии можно заметить сравнительно небольшие системные требования игр. Представленные проекты не требуют мощной видеокарты (до 2 Гб видеопамяти) и процессора (с частотой до 2.4 ГГц). Игры имеют хорошую оптимизацию и версии для множества операционных систем и устройств (PC, мобильные платформы, консоли). Среди прочих достоинств можно выделить:

* Увлекательный геймплей: механики сочетают экшен и головоломки, исследование мира.
* Стильная графика: в играх используются многоуровневые декорации и другие объекты, созданные профессиональными дизайнерами и художниками.
* Расслабляющая атмосфера: игра создает уникальную атмосферу спокойствия и умиротворения, что делает ее идеальной для отдыха.

Также, у представленных проектов есть свои недостатки:

1. Ограниченный геймплей: FAR: Lone Sails может быстро надоесть игрокам, так как геймплей основан на повторении одних и тех же действий.

2. Отсутствие сюжетной линии: ни одна из представленных игр не имеет сильной сюжетной линии, что может разочаровать тех, кто ищет глубокий и захватывающий сюжет.

3. Настроение: игры обладают давящей, мрачной атмосферой, что может отпугнуть некоторых игроков.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Платформы** | **Минимальные системные требования (PC)** | **Представлеение** | **Наличие сюжета** | **Стилизация** | **Красочный мир** | **Настроение** | **Исследование мира** | **Транспорт** | **Головоломки** |
| **Inside** | * Windows * macOS * PlayStation 4 * Xbox One * Nintendo Switch * iOS | * ОС: Windows 7/8/10 (64-bit) * Процессор: Intel Core 2 Quad Q6600 @ 2.4 GHz, AMD FX 8120 @ 3.1 GHz. * Оперативная память: 4 GB ОЗУ * Видеокарта: NVIDIA GT 630 / 650m, AMD Radeon HD6570. * DirectX: Версии 9.0c. * Место на диске: 3 GB. * Звуковая карта: 100% DirectX 9.0c | 2.5D | Нет | Есть | Нет | Тревожное | Нет | Нет | Есть |
| **Limbo** | * Xbox 360 (XBLA) * Xbox One * PlayStation 3 (PSN) * Windows, (Steam) * Mac OS X (Steam и Mac App Store) * Linux * Cloud (OnLive) PlayStation Vita * iOS * Android * Nintendo Switch | * ОC: Windows XP, Vista, 7 * Процессор: 2 GHz * Оперативная память: 512MB * Диск 150MB * Видеокарта: NVIDIA GT 630 / 650m, AMD Radeon HD6570. * DirectX 9.0c | 2D | Нет | Есть | Нет | Тревожное | Нет | Нет | Есть |
| **FAR: Lone Sails** | * Windows * macOS * Xbox One * PlayStation 4 * Nintendo Switch * iOS * Android | * ОС: Windows 7+ (64-bit OS required) * Процессор: Intel Core 2 Quad Q6600 @ 2.4 GHz, AMD FX 8120 @ 3.1 GHz. * Оперативная память: 4 GB ОЗУ * Видеокарта: NVIDIA GTX 460 / 650m, AMD Radeon HD6570. * DirectX: Версии 11. * Место на диске: 3 GB. * Дополнительно: Не поддерживаются Intel HD Graphics видеокарты | 2.5D | Нет | Есть | Нет | Печальное | Есть | Есть | Есть |

# **Обоснование выбора платформы**

В качестве среды разработки приложения была выбрана кроссплатформенная среда Unity, созданная американской компанией Unity Technologies. Такой выбор обоснован техническими возможностями Unity, гибкостью движка и подходом к кроссплатформенной разработке.

Unity позволяет создавать приложения, работающие на более чем 25 актуальных платформах, среди которых ПК под управлением ОС Windows, macOS, Linux, мобильные устройства под управлением ОС Android, IOS, игровые консоли PlayStation 4 и PlayStation 5, XBox One, интернет-приложения WebGL.

Для создания приложения для нескольких платформ в одном проекте Unity предоставляет инструменты одновременного использования игровых библиотек, плагинов (ассетов) и кода, а также позволяет использовать один и тот же интерфейс приложения для разных платформ. Настройка приложения производится автоматически или в ручном режиме. Для проектов на Unity также доступны ассеты, разработанные сторонними компаниями и пользователями, что позволяет ускорить и упростить процесс разработки.

Благодаря представленным возможностям кроссплатформы, разработчику приходится тратить меньше денежных ресурсов на развитие и поддержку версий для различных устройств.

Unity имеет несколько различных тарифных планов, благодаря чему любая компания или разработчик может подобрать себе наиболее оптимальный вариант.

Unity Personal - это бесплатный тарифный план, предназначенный для компаний и разработчиков с денежным оборотом менее 100 тыс. долларов США в год. Он предоставляет доступ ко всем инструментам Unity и позволяет создавать приложения для более чем 25 платформ. При запуске созданное приложение выводит на экран устройства вотермарку с логотипом среды Unity, которую возможно скрыть при переходе на план Unity Plus, в котором также доступны дополнительные инструменты аналитики приложений. При использовании тарифного плана Unity Plus потолок прибыли разработчика составляет 200 тыс. долларов США в год.

Unity Pro - платный тарифный план, который предоставляет доступ к техподдержке продуктов Unity Technologies и расширенным инструментам и возможностям, таким как поддержка физического движка Havok, инструменты для работы с VR Unity Mars, профессиональные аналитические инструменты, доступ к исходному коду движка.

Unity Enterprise - тарифный план для больших игровых компаний, предлагающий специальные отраслевые решения и расширенную LTS.

Unity подходит для реализации плагинов, используемых при разработке приложений внутри проекта, которые можно с лёгкостью интегрировать для отображения в интерфейсе движка. Для реализации собственных инструментов можно взаимодействовать с макетом редактора UnityEditor, консолью и Gizmos (инструментами отладки и редактирования сцены) напрямую, или воспользоваться сторонним дополнением для создания инструментов и настройки их пользовательских интерфейсов. Подобные системы ускоряют процесс разработки и положительно влияют на качество продукта, благодаря повышению удобства взаимодействия разработчика с движком и автоматизации разработки.

Unity предоставляет полную поддержку разработки игр и приложений в 3D и 2D. Движок располагает мощными инструментами для создания игровой механики, симуляции 2D и 3D физики, трехмерной графики (работа с моделями, освещением, шейдерами, системами частиц, текстурами и материалами), двумерной графики (работа со спрайтами, оптимизация 2D –графики, 2D-освещение), костной векторной, покадровой анимации, работы с объемным звуком. Unity может одновременно работать с 2D и 3D объектами в одном проекте.

Другим преимуществом Unity является мощная система разработки интерфейса пользователя, которая позволяет создавать интерфейсы для видеоигр и приложений, используя якорные привязки (Anchors) и правила трансформации GUI-объектов, благодаря чему интерфейс созданного приложения адаптирован практически к любому соотношению сторон и разрешению экрана целевого устройства, что особенно важно при разработке для мобильных устройств, т.к в данный момент в эксплуатации потребителей находится множество моделей с различными параметрами дисплея.

Движок Unity является основным продуктом Unity Technologies и поддерживает современные, актуальные технологии, такие как:

* Поддержка разработки игр с виртуальной и дополненной реальностью.
* Система трассировки лучей в реальном времени (RTX), которая моделирует аналогичные природным свойства лучей света, их взаимодействие с физическими объектами и материалами.
* Horizon-based Ambient occlusion (HBAO) - на данный момент наряду с HDAO наиболее совершенная технология затенения и рендеринга, используемая для расчета того, насколько каждая точка сцены подвержена воздействию окружающего освещения.

В результате рассмотрения технологий, возможных подходов к разработке и технических решений, воплощенных в среде Unity, движок был выбран в качестве среды разработки.

# **Обзор проекта**

Разработанная в рамках дипломного проекта кроссплатформенная видеоигра “CogitaMechanica” представляет собой 2D платформер – приключение, в котором игрок должен взять на себя роль доставщика почты – андроида Сай. Игрок должен пройти через множество локаций, раздать посылки, решить головоломки, а также познакомиться с сюжетом игры. Мир видеоигры разбит на 3 основных зоны, каждая из которых представлена сценой (отдельным 3D пространством) в среде Unity. Сцена содержит фоновые декорации, интерьеры помещений, акторов (персонажей, представляющих собой игровой объект, состоящий из физической модели, визуала и программных компонентов), а также множество предметов, являющихся игровыми объектами, которые будут использоваться игроком.

Виртуальный мир видеоигры представлен в виде многослойного плоского пространства с эффектом параллакса.

Так как путь через игровой мир будет занимать множество внутриигровых суток, в дипломном проекте реализована визуализация смены дня и ночи, а также имитация освещения и светотени. Игровой мир имеет 4 состояния (полдень, полночь, рассвет, закат). Состояния плавно переключаются между собой, рендер теней реагирует на положение солнца. При каждом состоянии становятся активны соответствующие элементы окружения (например, в дневное время суток на небе можно увидеть Солнце).

Главный герой игры управляется игроком с помощью клавиатуры (на ПК), а также кнопок интерфейса и жестов (на мобильных платформах). Игровой персонаж обладает плоской физической моделью, что позволяет ему участвовать в физической симуляции мира игры.

Основные возможности игрока:

* передвижение по твёрдой, неровной поверхности, прыжки в различных направлениях, реакция на контакт с землёй;
* физически корректное взаимодействие с движущимися объектами;
* взаимодействие с игровыми объектами (возможность брать выделенные объекты в руки, возможность взаимодействия с различными механизмами и приспособлениями (рычаги, вентили, крюки для игровых объектов);
* взаимодействие с источниками света.

Персонаж игры может передвигаться по игровому миру как самостоятельно, так и с помощью транспорта. Паровая машина состоит из множества модулей, позволяющих ей реализовывать различные механические функции. Игрок должен поддерживать техническое состояние модулей для их работы.

Основные возможности транспорта:

* возможность путешествовать по игровому миру под управлением игрока;
* физически корректное взаимодействие с поверхностью земли;
* расход топлива и других компонентов при движении;
* возможность пополнения расходных материалов;
* возможность перевозки и сохранения игровых объектов, найденных игроком;
* обеспечение работы модулей;
* поломка модулей и возможность их починки;

Дополнительные модули для паровой машины, а также головоломки и сюжетные моменты вводятся в игру согласно кривой интереса. Решая головоломки, игрок получает вознаграждение или открывает путь дальше. С механизмами и головоломками игрок взаимодействует с помощью физического воздействия главного персонажа или элементов интерфейса. Рассмотрим пример.

(Тут пример какой-нибудь головоломки и её блок-схема решения)

В проекте реализовано большое количество (~ 100) игровых предметов, которые игрок может использовать в прохождении и коллекционировать.

Основные функции предметов:

* возможность “брать их в руки”;
* физическое взаимодействие предметов с окружением;
* возможное использование предметов игроком в качестве расходных материалов для модулей машины (топливо, жидкость для парового котла);
* возможное использование предметов игроком для помощи в прохождении игры (например, фонарик, развеивающий темноту ночью);
* возможное использование предметов в качестве необходимого элемента для решения головоломки (например, шестерёнки в механизме);
* развлекательные функции (например, радио, музыкальный инструмент).

Таким образом, можно выделить основные составляющие игры:

(Картинка как связаны все компоненты, общая)

* Окружение

1. интерактивные механизмы и предметы
2. фоновые объекты
3. имитация 3D освещения
4. машина и модули

* Акторы

1. Игрок
2. Сюжетные персонажи

# **Особенности использования движка Unity**

При использовании Unity в разработке видеоигр важно знать основные концепции и особенности работы в среде.

Разработанная в Unity видеоигра состоит из множества связанных между собой сцен. Сцена - это пространство для заполнения контентом, использующееся разработчиком в качестве игрового мира. Взаимодействие со сценами происходит с помощью инструмента UnityEngine SceneManagement, позволяющего работать со сменой сцен, загрузкой и выгрузкой их ресурсов в память.

Контент, добавленный разработчиком на сцену, помещается в список объектов на сцене. Unity поддерживает концепцию ООП в формировании иерархии объектов. В иерархии сцены содержатся дочерние и родительские объекты - это важная концепция в движке Unity Engine. Дочерние объекты могут наследовать свойства родительского объекта, такие как положение, поворот и масштаб. Дочерние объекты также можно перемещать, вращать и масштабировать относительно родительского объекта. Это позволяет создавать комплексные игровые объекты, а также упрощает управление ими.

GameObjects - игровые объекты - это основные элементы любого приложения на движке Unity Engine. GameObjects - это контейнеры, которые могут содержать такие компоненты, как компоненты MonoBehavior, аудио, графические компоненты, игровую физику. Они также могут содержать другие GameObjects, которые могут быть вложены друг в друга для создания сложных иерархий объектов. Объекты GameObjects можно перемещать, вращать и масштабировать в игровом мире, а также изменять из пользовательского интерфейса редактора их публичные свойства.

Движок Unity Engine предоставляет ряд стандартных компонентов, обеспечивающих базовые возможности объектов. Рассмотрим некоторые из них.

* Компонент Transform используется для перемещения, вращения и масштабирования объектов в игровом мире.
* Компонент Rigidbody используется для создания взаимодействий на основе физики, таких как гравитация и столкновения.
* Вариации компонента Collider используется для создания физической формы объекта, использующейся в столкновениях, а компонент Joints - для создания физических зависимостей между объектами.
* Компоненты Mesh Renderer и Sprite Renderer используются в Unity для рендеринга 3D и 2D объектов соответственно. Компонент Mesh Renderer используется для рендеринга 3D-объектов в игровом мире, таких как модели и рельеф. Компонент Sprite Renderer используется для рендеринга 2D-объектов, таких как спрайты и элементы пользовательского интерфейса.
* Компонент Audio Source используется для воспроизведения звука в игровом мире, а компонент Audio Listener - для обнаружения звука.

Кроме того, Unity предоставляет ряд других компонентов, которые можно использовать для создания сложных игровых механик.

Пользовательские компоненты в Unity - это компоненты, которые создаются разработчиками для расширения функциональности движка Unity Engine. Пользовательские компоненты можно использовать для создания новых игровых механик, таких как движение персонажа, взаимодействие с объектами, взаимодействие на основе физики и для создания элементов интерфейса. Кроме того, такие компоненты можно использовать для создания шейдеров и эффектов, а также для интеграции внешних библиотек и API в Unity.

Класс MonoBehaviour - это базовый класс, от которого происходят все компоненты Unity. Он обеспечивает базовую функциональность скриптов, такую как возможность обновления, запуска и остановки скрипта, а также доступ к различным API Unity. Он также отвечает за отправку сообщений другим скриптам и предоставляет доступ к компонентам объекта GameObject.

Стандартные методы MonoBehaviour в Unity включают Awake(), Start(), Update(), FixedUpdate(), LateUpdate(), OnGUI(), OnDisable(), OnEnable() и OnDestroy().

* Метод Awake() вызывается во время загрузки экземпляра компонента, до вызова любых других методов.
* Метод Start() вызывается перед первым обновлением кадра и используется для инициализации переменных или состояния игры.
* Метод Update() вызывается каждый кадр и используется для игровой логики, например, создания движения объектов.
* Метод FixedUpdate() вызывается каждый кадр с фиксированной частотой кадров и используется для расчетов физики.
* Метод LateUpdate() вызывается после вызова всех остальных методов Update() и используется для постобработки.
* Метод OnGUI() вызывается для рендеринга и обработки событий GUI.
* Метод OnDisable() вызывается, когда компонент на привязанном объекте отключен.
* Метод OnEnable() вызывается, когда компонент включен.
* Метод OnDestroy() - деструктор объекта.

# **Архитектура приложения**

Архитектура приложения - это общая структура приложения и то, как его компоненты взаимодействуют друг с другом. Важно, чтобы архитектура приложения была разработана таким образом, чтобы обеспечить масштабируемость и легкую отладку. Кроме того, при разработке архитектуры необходимо учитывать конкретные аппаратные и программные компоненты целевых платформ, а также специфические потребности пользователя. Учитывая специфику видеоигр, при создании приложений на движке Unity обычно используют архитектуру на базе КОП - Компонентно-ориентированного программирования.

КОП - это тип программирования, при котором следует фокусироваться на разбиении сложных задач на небольшие, управляемые компоненты. Это позволяет разработчикам создавать сложные приложения, комбинируя несколько компонентов различными способами. КОП упрощает разработчикам создание и отладку отдельных модулей, а также позволяет упростить повторное использование компонентов в различных модулях. Кроме того, КОП также позволяет сократить время разработки и упростить интеграцию с другими системами.

На практике данный подход был реализован в том числе в создании игровых предметов. Представленный на рисунке игровой предмет является объектом, содержащим компоненты itemWeight, itemMagnet, itemScrap и.т.д

(Картинка – предмет и его компоненты).

Каждый компонент объекта отвечает за определенное свойство (itemWeight - компонент, отвечающий за вес предмета, itemMagnet - компонент, придающий предмету свойство магнита, itemScrap - компонент, позволяющий разбирать предмет ЕЩЕ). Таким образом, был создан ограниченный набор компонентов, благодаря уникальным сочетаниям и настройкам которых, реализовано множество различающихся игровых предметов. На практике благодаря КОП переиспользование кода возрастает в разы.

(Картинка - предметы и их компоненты, переиспользование компонентов)

Архитектура Model-View-Controller (MVC) является популярным шаблоном проектирования, используемым при разработке программного обеспечения, и может быть использована также при разработке игр. В этом шаблоне модель - это данные или информация игры, представление - графический интерфейс, а контроллер отвечает за обработку событий, пользовательский ввод и обновление представления и модели. Такая архитектура обеспечивает большую гибкость и масштабируемость при разработке игр. Это также отделяет логику от уровня представления, облегчая разработчикам модификацию или расширение игры без необходимости вносить изменения во всю систему. Кроме того, использование MVC позволяет создать более безопасную систему, поскольку модель может быть защищена от вредоносного пользовательского ввода.

(Картинка схема MVC модели)

В проекте используется изменённая архитектура MVC - MVVMC, содержащая модель, представление, прослойку View-Model и контроллер. Данная архитектура реализована во всех модулях паровой машины, используется в различных элементах игры (головоломках, устройствах на локации).

(Картика схема MVVMC)

Рассмотрим реализацию архитектуры MVVMC на примере созданной системы подачи топлива в паровую машину.

(Картинка взаимодействие компонентов и модулей).

Организация компонентов представлена на рисунке. В данном случае пользователь с помощью кнопки с задержкой активирует топку, сжигающую попавшие внутрь предметы, после чего пополняется уровень топлива.

(Блок-схема взаимодействия)

Внутриигровая UI - кнопка с компонентом HBar (View) передаёт сигнал о срабатывании компоненту module Base (VM), вызывает установленный метод. VM определяет контроллер module Furnance, вызываемый компонентом кнопки, контроллер обрабатывает событие и выводит показатель топлива на пользовательский интерфейс (View).

Таким образом, в дипломном проекте была реализована гибридная архитектура приложения, что позволило добиться масштабируемости модулей с помощью различных шаблонов, применяя их там, где это было удобно и необходимо.

# **Освещение и затенение**

Основываясь на нарративных требованиях, а также для улучшения внешнего вида видеоигры была создана система смены времени суток, связанные с ней компоненты теней и прочие вспомогательные компоненты.

Созданная система освещения состоит из следующих компонентов:

* Контроллер смены времени суток
* Цветовые фильтры
* Контроллер тени
* Источники света
* Контроллер освещения

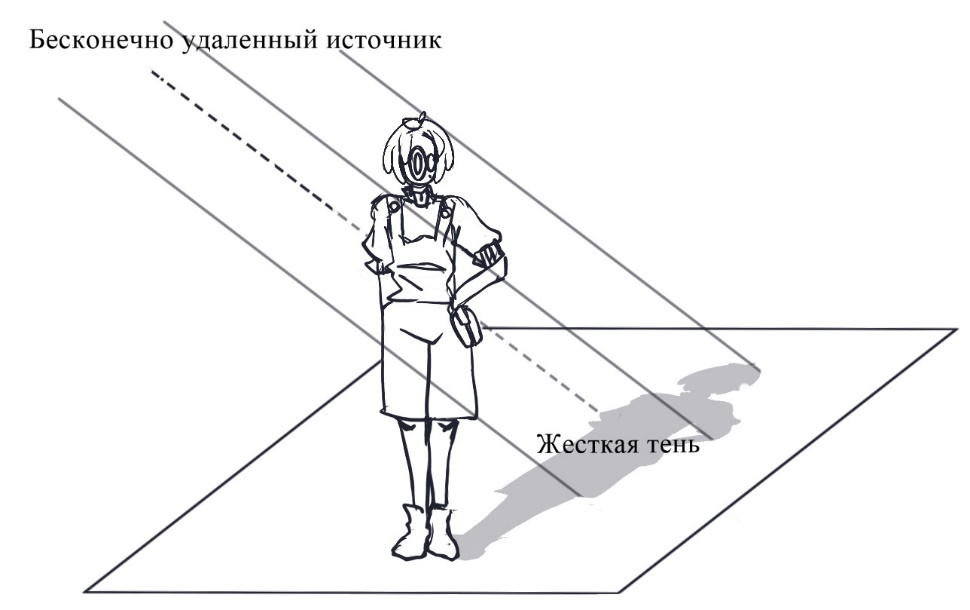
Цветовые фильтры, позволяют создать эффект утреннего, вечернего, ночного естественного освещения, путем наложения цветового слоя на объекты сцены. Слои фильтра имеют различный оттенок и прозрачность. Наряду с фильтрами применены эффекты цветовой коррекции. Для каждого времени суток создан пресет, позволяющий создать привлекательный вид окружения.

(Картинка – время суток, наполнение слоя и пресет, его конфигурации)

# **Визуализация теней объектов**

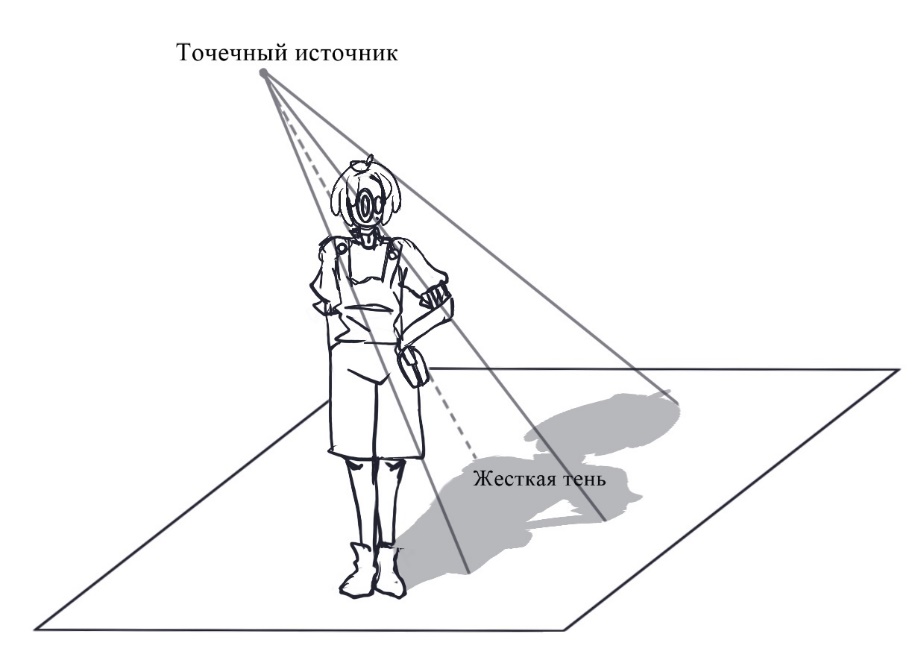
В компьютерной графике важным аспектом при создании сцены является работа над светом. Свет и тень добавляют изображению глубины, контраста и реализма. Также, свет является мощным художественным инструментом, который помогает автору передать заложенные в сцену настроение и атмосферу.

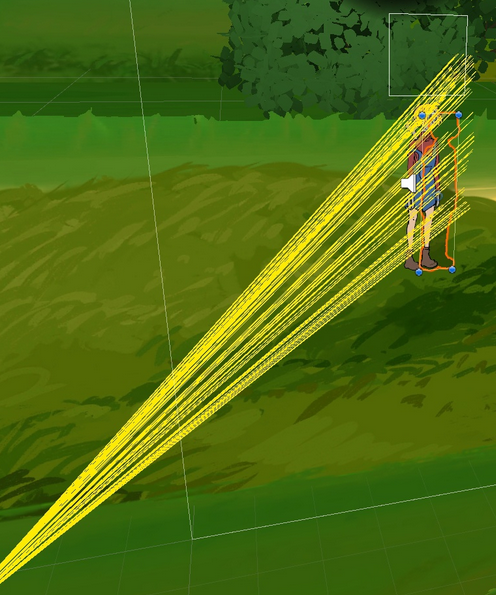
В задаче моделирования освещения в пространстве с источником света и облучаемым объектом обычно рассматривается два основных случая:

Первый – сцена с бесконечно удаленным источником. Данный источник света можно представить в виде плоскости, развернутой поверхностью в направлении излучения. Из каждой точки плоскости исходят лучи света, перпендикулярные самой плоскости и параллельные между собой. Такие источники применяются, когда для сцены нужно создать имитацию солнечного света, освещающего все объекты пространства под одним и тем же углом.



Второй случай – сцена с точечным источником света. В качестве источника выступает гипотетическая точка, размерами которой можно пренебречь. В этом сценарии излучение происходит во всех направлениях от данной точки.





Существуют частные случаи точечного источника света:

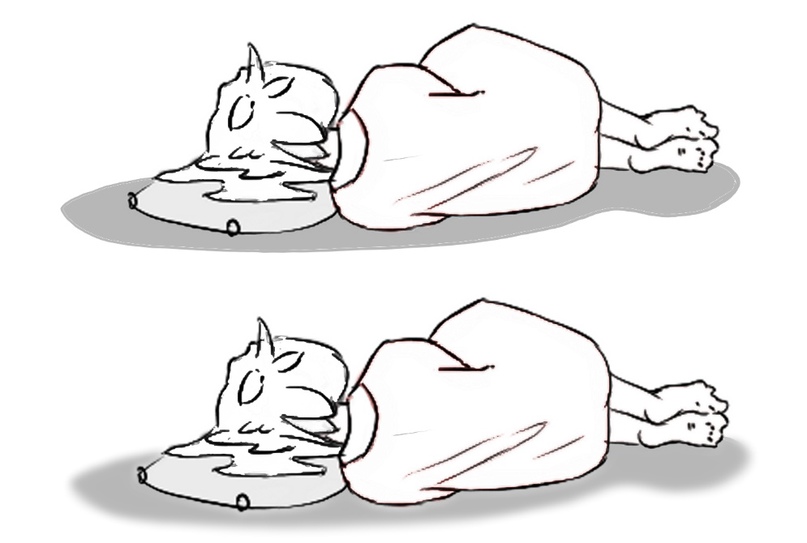
– Прожектор – направленный точечный источник, освещающий определенную область. Такой тип источника применяется в 3D графике для имитации света прожекторов, уличных фонарей и т.д.

– Протяженный источник, рассредоточенный по площади и излучающий свет из серии точек. Все реальные источники света (лампы, звезды и т.д.) можно представить в виде протяженного источника.

В 3D графике различают два типа теней, получающихся от источников света – жесткие и мягкие. Жесткие тени имеют четкие, определенные грани, в то время как мягкие тени “расплываются” ближе к краю.

Жесткие тени получаются в результате облучения точечного или бесконечного удаленного источника – лучи света либо падают на освещаемые поверхности, либо сталкиваются с препятствующим объектом и оставляют темные области, создавая четкую тень.

Мягкие тени – результат воздействия на объекты протяженного источника света. Так как протяженный источник состоит из множества точечных, объект отбрасывает не одну, а множество теней. В некоторых областях тени накладываются друг на друга и создают большее затенение. Наиболее затененные участки образуют основную тень, менее затененные участки – полутень.



В настоящее время в 3D графике реального времени невозможно реализовать алгоритм наложения теней в силу отсутствия достаточно мощного графического процессора. Зачастую, разработчики применяют эффекты размытия для имитации мягкой тени.

Современные графические ускорители оптимизированы для выполнения сложных операций с полигонами (т.е. плоскостей, ограниченных векторами), таких как преобразование, затенение и текстурирование. Графический конвейер включает в себя такие этапы как обработка вершин полигонов, затенение геометрии, растеризация и затенение пикселей. Рендеринг теней происходит за счет анализа и проецирования полигонов, из которых состоит поверхность объекта, на окружающее пространство. Разработка инструмента для визуализации теней - программная задача, которую необходимо воплотить в движке игры.

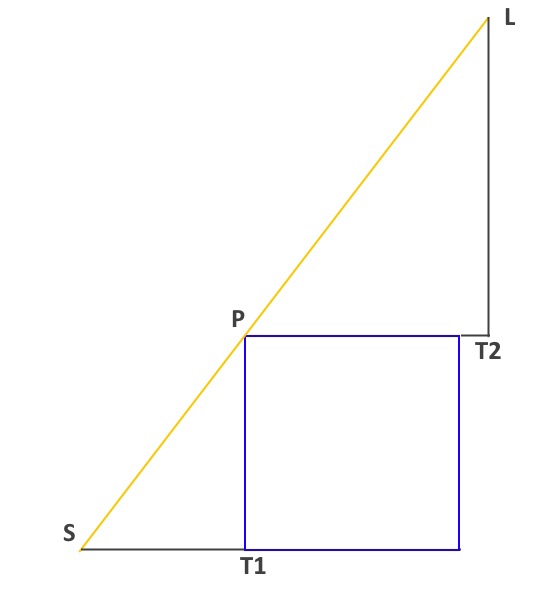
Рассмотрим метод преобразования полигона “на землю” Джима Блинна, который позволит детально разобрать процесс получения тени объекта от источника света в 3D графике.

С помощью данного метода полигоны модели проецируются на плоскость сцены, в результате чего получается жесткая тень. Метод использует свойство подобных треугольников для вычисления позиции и геометрии каждого полигона тени. Тень должна быть рассчитана для каждого источника света индивидуально, т.е. если объект освещается N источниками света, то необходимо применить метод для каждого источника и получить N теневых проекций.

Разберем преобразование координат вершин объекта на примере проекции на плоскость z =0 (плоскость “земли”). В случае направленного параллельного света:

Имея точку источника света L и вершину объекта P, необходимо получить проекцию вершины объекта на плоскость z=0, т.е. точку тени S

Используя свойство подобных треугольников можно составить зависимость расстояния *SP* от теневой точки *S* до вершины *P* объекта и расстояния от источника света *L* до вершины объекта:



Треугольник *SPT1* подобен треугольнику *PLT2*, т.е. коэффициент подобия *α* равен

, то есть,

*SP = PL α*

Если принять, что *l* это вектор из точки *P* к источнику света, то точку *S* можно выразить как:

*S = P – α l*

Применив эту формулу для всех вершин реального полигона мы получим вершины “теневого” полигона.

В случае точечного источника понадобятся дополнительные вычисления на каждую вершину модели, т.к. каждая вершина имеет, в общем случае, своё собственное направление на источник света. Если L это точка расположения источника света, то уравнение принимает вид:

*S = P + α (P - L)*

Для построения проекции на объекты сложной формы (композиции плоскостей, сферы), был применен рейкастинг:

Для точечного источника света *L* в направлении через ключевые *N* вершин объекта направляются лучи *R1-N*

*Ri(d) = L + d(Pi-L)*

При столкновении луча с плоскостью поверхности, точка столкновения *Si*становится вершиной будущей тени.

В проекте, в качестве объекта, отбрасывающего тень, используется спрайт, изображение, не имеющее геометрии (не представленное в виде взаимосвязанных полигонов). Поэтому, первой задачей на пути к рендеру тени является создание 3D сетки из полигонов по форме спрайта. Таким образом, детализация полученной тени зависит от количества выделенных для проекции вершин *N* объекта, отбрасывающего тень. По полученным точкам *S1-N*возможно выполнить триангуляцию, для последующего формирования 3D-сетки и визуализации тени.

В результате, инструмент, основанный на приведенных алгоритмах позволил создать источники света, на которые бы реагировали объекты окружения, отбрасывая тень.

# **Управление тенью**

В игре было реализовано два типа источников света – источник параллельного света и точечный источник. Точечные источники света применены в фонарях, фарах, свечах, а источник параллельного света используется для имитации естественного освещения.

Для уменьшения вычислительной нагрузки объект может отбрасывать только одну тень. Целевой источник света выбирается исходя из дистанции до источника.

Также, в зависимости от времени суток меняется положение источника параллельного света, имитируя положение солнца относительно игрока в утреннее, дневное и вечернее время.

Рисунок с положениями и тенью

# **Визуализация света**

Так как затенение в темное время суток реализовано помощью фильтров с определенной непрозрачностью, источники света, такие как фонари, фары, свечи, и т.д., должны образовывать просветы в слоях фильтров. Для создания мягких краев областей просвета была применена техника теневых масок и проективного наложения изображения.

Объект, который создает световое пятно рисуется с точки зрения дополнительной виртуальной камеры чёрным цветом в белую текстуру. Размер текстуры зависит от того, насколько высокое разрешение маски мы хотим видеть в игре. В реализованном проекте текстура имеет разрешение X что позволяет добиться высокой четкости изображения.

В проекте все объекты, излучающие свет имеют невидимую для основной камеры, но видимую для дополнительной темную размытую текстуру. Цветовые фильтры используют текстуру, в которую и проецируется изображение с дополнительной камеры. Темные участки этой текстуры интерпретируются как прозрачные.

# **Пользовательский интерфейс**

Система пользовательского интерфейса Unity состоит из нескольких ключевых компонентов, включая Canvas, который является корневым компонентом. В этом компоненте определяется режим рендеринга и виртуальная камера, используемая для всех элементов Canvas. Стандартные элементы GUI включают в себя текст, изображения, ползунки и кнопки. Эти компоненты объединяются с помощью макета, такого как Grid или горизонтальная/вертикальная Layout Group. Наконец, пользовательский интерфейс оформляется с использованием эффектов анимации и переходов. С помощью стандартных компонентов можно создавать иммерсивные и интерактивные пользовательские интерфейсы.

Подходы к разработке пользовательского интерфейса в Unity

1. Проектирование пользовательского интерфейса: Основное внимание при разработке пользовательского интерфейса уделяется интуитивности и эффективности взаимодействий. Чтобы достичь этого, пользовательский интерфейс должен быть разработан с учетом основных сценариев ваимодействия с видеоигрой, потребностей пользователя. Элементы интерфейса должны быть структурированы по смыслу и расположены согласно иерархии объектов UI. Следует учитывать частоту, удобство взаимодействия с элементами.

Макет и визуальный дизайн - это расположение элементов на экране, использование цвета и типографики, а также общая эстетика GUI. Эти элементы задают настроение и стиль GUI.

Важные аспекты дизайна, на которые следует обратить внимание:

1. Элементы управления

Элементы управления пользовательского интерфейса должны быть максимально комфортны для пользователя.

2. Навигация

Навигация разрабатывается таким образом, чтобы направлять пользователя в необходимый раздел. (Интуитивно понятная навигация).

3. Доступность

Пользовательский интерфейс должен быть разработан таким образом, чтобы быть доступным для максимально широкого круга пользователей, независимо от их навыков работы с ПО.

4. Адаптивность

Пользовательский интерфейс должен быть разработан с расчетом на работу на различных устройствах, с разными параметрами экрана.

5. Отзывчивость

Элементы интерфейса должны выполнять закрепленные за ними функции с минимальной задержкой, управление должно быть комфортным, каждое действие пользователя должно иметь обратную связь.

6. Привлекательность

Интерфейс должен быть разработан таким образом, чтобы быть приятным для пользователя, с привлекательными визуальными элементами и анимацией.

7. Тестирование

Пользовательский интерфейс следует тестировать и дорабатывать на протяжении всего процесса разработки, чтобы убедиться в соответствии потребностям и ожиданиям пользователя.

# **Реализация в среде Unity**

Для создания видеоигры потребовалось создать игровой мир и объекты для его наполнения, реализовать необходимые компоненты для управления объектами и их взаимодействия, компоненты для создания привлекательного внешнего вида игры.

# **Реализация теней**

Создание сетки 3D тени происходит в несколько этапов:

**1) Выделение точек контура.**

На этом этапе следует определить точки, которые лежат на границе изображения и фона (в данном случае – альфа-слоя). Для выбора контурных точек на PNG изображении с прозрачным фоном можно использовать алгоритм маршрута границы:

1) Выбрать начальную точку (За начальную точку применяется самый верхний непрозрачный пиксель).

2) Выбрать следующую точку контура. Контурная точка выбирается по числу непрозрачных “соседей”. Точка, имеющая наименьшее число является пограничной.

2) Пункт 2 повторяется, пока не будет снова выбрана начальная точка.

нужна блок-схема

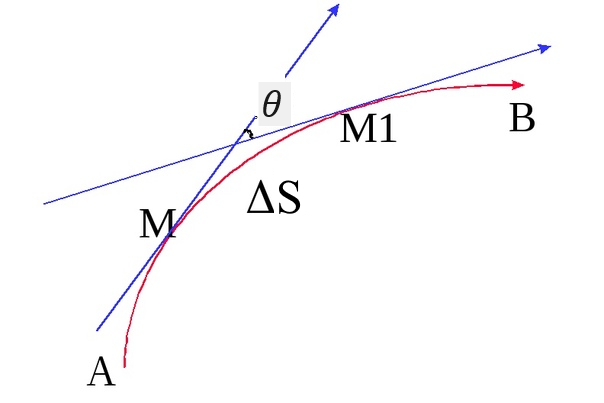
В PNG изображении могут быть прозрачные отверстия, а также изображение может быть фрагментировано, разбито на несколько частей. В таких случаях алгоритму необходимо совершить несколько обходов для определения всех имеющихся контуров.

**2) Выделение узловых точек.**

Примененный метод заключается в том, чтобы определить точки контура, в которых происходит смена направления или кривизны линии. Кривизна в точке M на контуре определяется как отношение:

, где

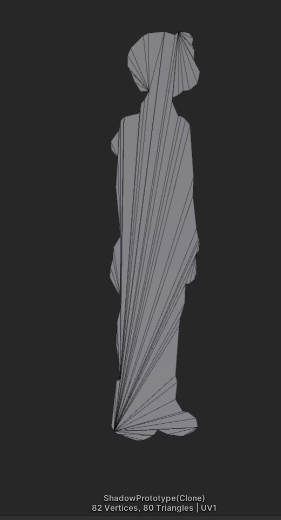
*k* – кривизна, – угол между касательной к узловой точке *M*i-1 и *M*, S – длина дуги от *M*i-1 до *M*

**

Точка становится узловой, когда кривизна точки контура превышает некоторый заданный пороговый уровень. Этот пороговый уровень выбирается экспериментально.

**3) Триангуляция**

Триангуляция – процесс разбиения многоугольника (фигуры, образованной узловыми точками) на треугольники с непересекающимися гранями. Эти треугольники необходимы для создания 3D сетки тени.



Существует множество различных алгоритмов триангуляции. В проекте был применен алгоритм “отрезания ушей” или алгоритм Ван Гога. Он основывается на том, что любой выпуклый многоугольник может быть разбит на треугольники без пересечения ребер.

Шаги алгоритма:

1. Выбрать одну из вершин многоугольника.

2. Найти две соседние вершины, образующие с текущей вершиной выпуклый угол.

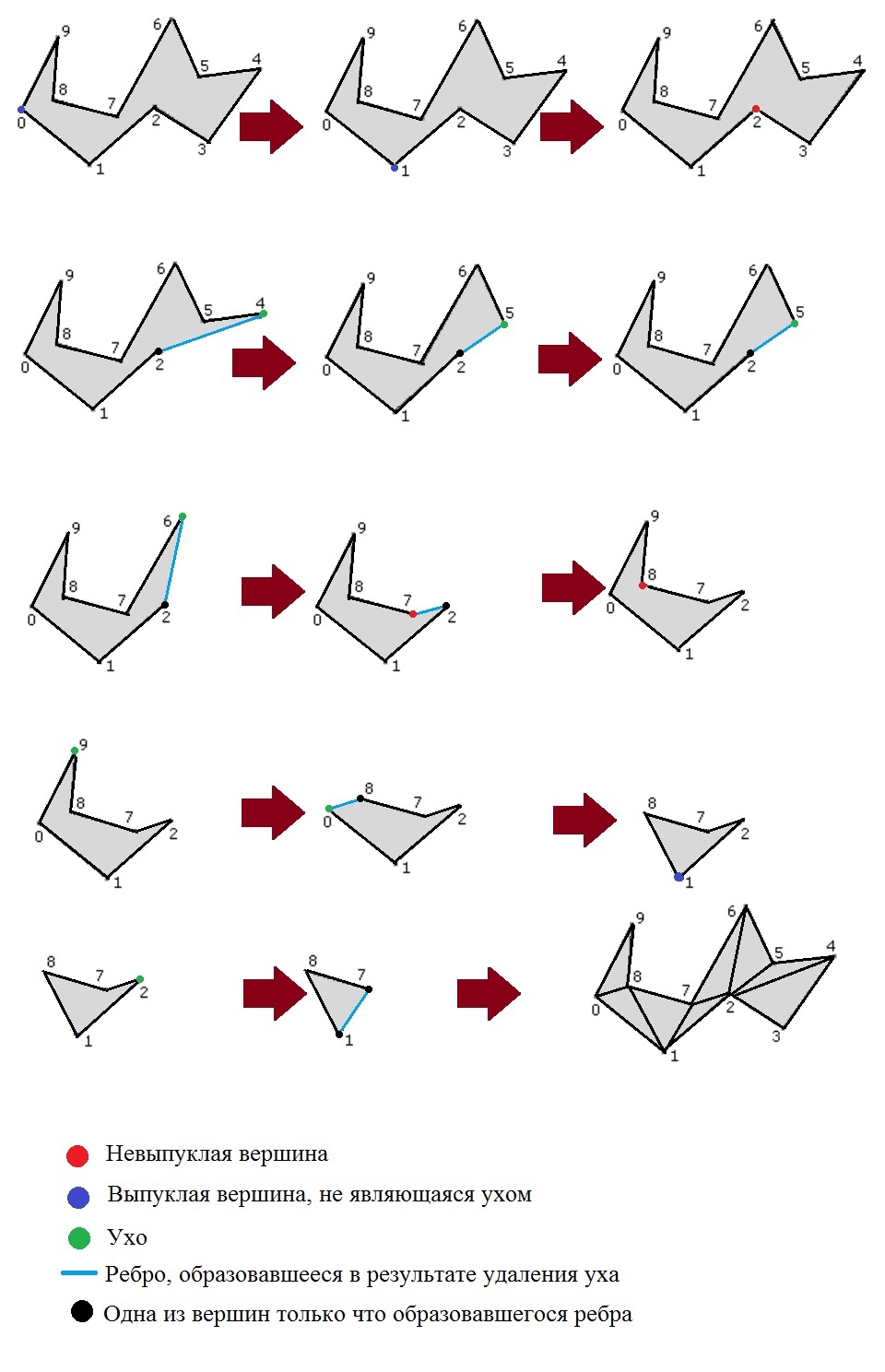
3. Проверить, лежит ли внутри этого угла некоторая другая вершина многоугольника. “Ушная" проверка вершины осуществляется алгоритмом принадлежности точки *n*-угольнику.

4. Если такая вершина найдена, то текущая вершина не является “ухом”, и ее можно пропустить. Если же такой вершины нет, то текущая вершина является “ухом”.

5. Соединить вершину уха с соседними вершинами, образующими угол. Это образует треугольник, который заносится в список.

6. Удалить вершину уха из многоугольника.

7. Повторять шаги 1-6, пока не останется только треугольник.



**4) Генерация 3D-сетки**

Наконец, после произведения триангуляции полученные полигоны можно объединить в сетку средствами движка. На 3D модель со сгенерированной сеткой накладывается материал тени.

Генерация 3D-сетки тени спрайта производится один раз и обновляется в течении игры вместе с изменением геометрии тени спрайта.

# **Физика объектов**

Rigidbody и Collider - это два важных компонента, использующиеся в игровом движке Unity для создания реалистичной физики объектов в игре. Rigidbody - это компонент, позволяющий объекту перемещаться под воздействием силы и взаимодействовать с другими объектами в игровом пространстве, Collider определяет физическую форму и размер объекта, а также реагирует на столкновения. Когда объект в игре имеет как Rigidbody, так и Collider, он обладает физической моделью. Unity использует физический движок PhysX, который позволяет создавать реалистичные физические эффекты в игре. PhysX использует формулы классической механики для определения движения объектов, учитывает гравитацию и другие факторы, влияющие на движение.

Простые физические модели, построенные с помощью Collider и Rigidbody используются для динамических объектов окружения, в контроллере игрока.

Главный герой игры должен сталкиваться с твердыми поверхностями слева и справа от себя, передвигаться по плоскости “земли”, определять, касается ли его “голова” препятствия (т.е может ли он совершить прыжок). Картинка с физ моделью игрока

Для реализации этих функций с помощью контроллера, необходима физическая модель. Circle collider (1) – круговая коллизия, лучше всего подходит для объекта, движущегося по поверхности, т.к позволяет лучше обрабатывать столкновения с другими объектами благодаря закругленным краям. Кроме того, такая коллизия использует меньше ресурсов, чем более детализированный Mesh collider. Circle collider (1) не позволяет игроку провалиться под плоскость “земли”, позволяет сталкиваться со стенами. Компонент Rigidbody обрабатывает касания, что позволяет определить, касается игрок статического или движущегося объекта.

Overlapped circle – виртуальная круговая зона, необходима для определения коллизий, препятствующих прыжку.

Wheel joint и Hinge joint - это два типа соединительных элементов, которые используются в Unity для создания физической связи двух объектов. Wheel joint используется для соединения вращающегося объекта с осью и передачи силы вращения на другой объект. Wheel joint имеет несколько настраиваемых параметров, такие как масса объекта, радиус колеса и сила торможения.

Hinge joint используется для соединения объектов, которые должны вращаться вокруг фиксированной оси, например, двери на шарнирах. В Hinge Joint настраиваются такие параметры как масса объекта, угол поворота и ограничения на поворот.

С помощью данных компонентов была реализована физическая симуляция веревки (цепи), колес и подвески транспорта, свободно закрепленных объектов.

Картинка с физ моделью веревки

Первое и последнее звено подвешеной фрагментированной веревки или цепи с помощью HingeJoint связано со статической точкой, не подверженной влиянию гравитации, каждое звено связано со следующим и предыдущим (если они существуют).

Картинка с физ моделью машины

Машина имеет три палубы, имеющие коллизию и триггер для взаимодействия с контроллером игрока, а также Wheel Joint (1) для заднего колеса, Wheel Joint (2) со смещенной точкой подвеса (*C*) для имитации раскачки транспорта.

Картинка с физ моделью крюка

Крюк подвешен с помощью Hinge Joint к статической точке, центру вращения (*C*) и ограничен в колебаниях (±*α*).

# **Параллакс**

Параллакс - эффект глубины в 2D-играх, позволяющий с помощью нескольких плоских подвижных слоев имитировать объем объекта или сцены при движении камеры. Параллакс достигается за счет использования слоев изображений, которые перемещаются с разной скоростью относительно движения камеры. Например, слой *Layer1* на заднем плане относительно камеры перемещается медленнее, чем *Layer2* на переднем плане, создавая 3D эффект в игре. Кроме того, эффект параллакса может использоваться для создания ощущения движения при статичной сцене.

(слои)

Для вычисления параллакса можно использовать следующие формулы:

1. Для нахождения новой позиции фона:

*newPosition = (cameraPosition \* parallaxFactor) - (cameraSize \* (parallaxFactor - 1))*

где:

- *newPosition* - новая позиция фона

- *cameraPosition* - текущая позиция камеры

- *parallaxFactor* - коэффициент параллакса (обычно от 0 до 1, где 0 - фон не движется, 1 - фон движется с той же скоростью, что и камера)

- *cameraSize* - размер камеры

2. Для нахождения смещения фона:

*offset = (newPosition - oldPosition) \* parallaxFactor*

где:

- *offset* - смещение фона

- *newPosition* - новая позиция фона

- *oldPosition* - старая позиция фона

- *parallaxFactor* - коэффициент параллакса

Каждый слой имеет множитель *parallaxFactor*, отвечающий за смещение фона относительно камеры.

БЛОК-СХЕМА АЛГОРИТМА

# **Создание контроллера игрока**

Контроллер игрока должен выполнять следующие основные задачи:

* получение управляющих сигналов с тачскрина, клавиатуры и реакция на них;
* возможность взаимодействия с предметами;
* возможность взаимодействия с механизмами;
* взаимодействие с движущимися предметами.

Полученные сигналы управления обрабатываются следующим образом:

Блок-схема обработки нажатий

Главный герой имеет анимации движения с предметом в руках и без него, прыжка, смены направления движения, использования предметов. Для героя составлена карта переходов анимации:

Карта анимаций

Статичные объекты, использующиеся для передвижения по ним (пол дома, земля) помечены тегом Ground, динамичесие – тегом Moving.

Картинка с силами

При касании динамической поверхности, игроку сообщается сила Rigidbody-компонента поверхности, что заставляет персонажа двигаться вместе с платформой.

# **Создание модуля**

Все модули транспорта независимы между собой, что упрощает апгрейд машины и разработку дополнительных модулей.

КОНТЕНТ

# **Связь компонентов внутри проекта**

КОНТЕНТ

# **Создание интерфейса**

КОНТЕНТ

# **Кривая интереса**

Кривая интереса - это концепция, используемая в дизайне игр, которая объясняет, как интерес игрока к игровому процессу увеличивается и уменьшается со временем, а также диаграмма, отображающая интерес игрока. Эта концепция основана на том, что игроки часто становятся более увлеченными процессом по мере прохождения игры, но в определённый момент достигают точки, где их интерес начинает ослабевать. Чтобы удержать игроков в этот момент, разработчики игр должны создавать контент, который будет увлекательным и интересным на протяжении всей игры. Этого можно добиться, вводя новые механики, персонажей и элементы сюжета в соответствующие моменты, а также предоставляя награды и стимулы для поддержания интереса игроков. Кроме того, разработчики должны знать, сколько времени игроки проводят на каждом уровне или участке игры, и соответствующим образом регулировать сложность и содержание. Для этого в игру вводятся специальные метрики для отслеживания действий игрока в виртуальном мире. Понимая и управляя кривой интереса, можно гарантировать, что игроки останутся вовлеченными и заинтересованными на протяжении всего прохождения игры.

Картинка кривой интереса

На картинке представлена кривая интереса успешного развлекательного продукта. Используя данную кривую была составлена собственная для помощи в распределении контента по карте первой зоны игры.

Новая кривая интереса

Точка A – начало пользовательского опыта, в этой точке у пользователя имеется начальный интерес, сформированный грамотной рекламной компанией, оформлением страницы игры, меню и т.д. Точка B – “Крючок”, заставляющий игрока окунуться в игру.

Sequence B

Отрезок AB в проекте игрок проводит за просмотром вступительного ролика, раскрывающего атмосферу, тематику, настроение, представляющего вселенную игры и главного героя, решает первую головоломку, осваивает транспорт и получает для него первый необязательный модуль, встречает первого второстепенного персонажа – Феми.

Sequence C

На протяжении игры интерес игрока будет спадать, поэтому, в точках C и E на карту добавлен контент – в точке C – загадка с цветовой последовательностью, которая позволяет получить модуль транспорта. Почтовый модуль игрок наконец сможет перевозить и раздавать посылки. Также, у игрока возникают сложности – после этой точки модули машины станут регулярно выходить из строя.

Sequence E

В точке E игрок познакомится с новым персонажем – принцессой Лилли, а также в случае успешного решения загадки с запасами еды сможет принять ее в свой экипаж.

Sequence G

В точке G зона подходит к концу, игрока необходимо замотивировать к продолжению игры. Ему предстоит решить сложную головоломку, чтобы открыть мост в новую зону, которая отображается в декорациях совершенно новой и незнакомой игроку. Там же игрок встречает Феми, которая предупреждает об опасности путешествия дальше, в Регни Нитори – земли блестящей руды. Точка H – конец первой зоны.

# **Оптимизация приложения**

Оптимизация в приложениях необходима для плавной работы и создания приятного пользовательского опыта. В разработке видеоигр оптимизация является одним из важнейших этапов, поскольку возможные задержки в выводе изображения на экран в результате недостаточных действий по оптимизации приложения неприемлемы.

Оптимизация видеоигры включает в себя процессы по обеспечению плавности и стабильности работы приложения, а также работы, направленные на увеличение частоты кадров приложения, уменьшение вычислительной нагрузки на ОЗУ, процессор и графический ускоритель, а также адаптация программного кода с учетом особенностей конкретных целевых устройств.

В среде Unity существует несколько методов оптимизации приложения:

1. Сокращение количества DrawCalls:

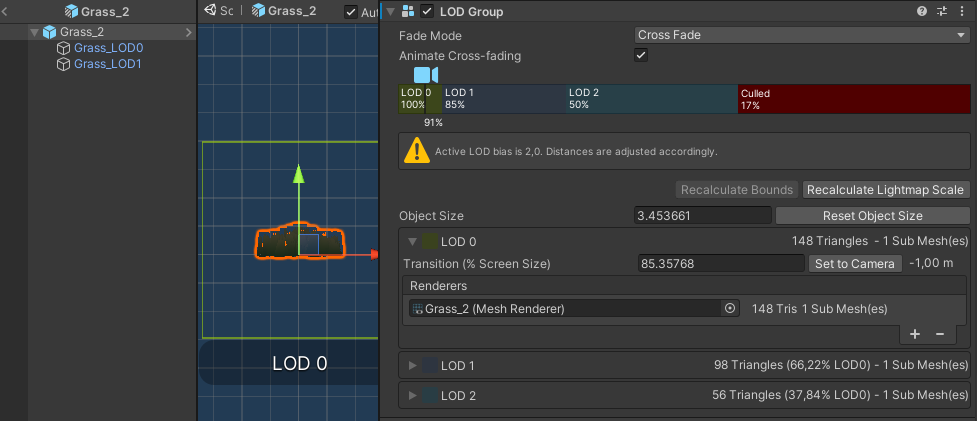
DrawCall - ресурсоемкая операция и может стать основным узким местом в игре. Draw Calls, вызовы отрисовки, выполняются графическим процессором устройства для отображения объектов на экране. Сокращение количества вызовов отрисовки за счет объединения объектов и использования текстур вместо отдельных объектов может значительно снизить нагрузку на GPU.

1.1 Для сокращения количества вызовов отрисовки может применяться технология импостеринга (impostor).

Импостеринг - это способ оптимизации отображения объектов, которые находятся далеко от виртуальной камеры и которые можно заменить на более простые модели, называемые импостерами. Импостеры представляют собой двухмерные изображения, накладываемые на простейшие геометрические фигуры, такие как плоские многоугольники. Эти геометрические фигуры, в свою очередь, могут быть масштабированы и повернуты таким образом, чтобы имитировать формы объектов, которые они заменяют. Импостеринг позволяет значительно снизить нагрузку на процессор и графический ускоритель, так как вместо отображения детальных моделей, состоящих из множества полигонов, которые потребляют много вычислительных ресурсов для рендеринга, мы отображаем более простые модели, имеющие до нескольких десятков полигонов. Для реализации импостеринга определяется, какие объекты на сцене нуждаются в отображении импостеров, а какие должны отображаться в полном объеме. Это может быть осуществлено с помощью вычисления расстояния между объектами и камерой, и заменой реальных моделей на импостеры, когда расстояние достигает определенного порога. Эти вычисления могут быть применены в дальнейшем в технологии LOD. Импостеринг не подходит для игровых персонажей или сложных архитектурных элементов, т.к. недостаток деталей может привести к ухудшению визуального опыта игрока.

1.2 Технология импостеринга часто применяется вместе с Level Of Detail (Уровень детализации) (LOD). Это технология, используемая для уменьшения детализации объектов по мере их отдаления от виртуальной камеры в игровом мире. На определенном уровне детализации будут использоваться низкополигональные модели - модели, которые менее детализированы, чем их высокополигональные аналоги. Они также могут быть использованы отдельно от LOD.

Технология LOD позволяет повысить производительность приложения, поскольку требуется рендеринг меньшего числа геометрии сцены.



1.3. При оптимизации сцены может быть применена окклюзия: Occlusion culling - это технология, используемая для уменьшения количества объектов, которые рисуются на экране. Технология решает проблему лишних затрат на отрисовку объектов, которые не будут видны на финальном кадре. В Unity применяется два вида окклюзии:

1. Static Occlusion Culling (статическая окклюзия) - используется для оптимизации сцен, содержащих статические объекты. Это означает, что объекты на сцене, которые не двигаются или изменяются, могут быть помечены как статические, после чего будут использоваться для расчета того, какие части сцены могут быть скрыты от камеры и не будут отрисованы во время игры. Unity использует алгоритм BSP (Binary Space Partitioning) для вычисления статической окклюзии.
2. Dynamic Occlusion Culling (динамическая окклюзия) - используется для оптимизации сцен, содержащих динамические объекты. Объекты на сцене, которые могут двигаться или изменяться, будут использоваться для расчета того, какие части сцены могут быть скрыты от камеры в определенный момент времени.

1.4. Использование GPU Instancing (Создание экземпляров объектов с помощью графического процессора) - с помощью создания экземпляров объектов можно уменьшить количество вызовов отрисовки, за счет многократного использования одного и того же набора геометрии.

2. Сокращение объема файлов: сжатие файлов проекта (текстур, аудио- и видео-файлов) уменьшает время динамической подгрузки объектов и объем исполняемого файла и ресурсов игры.

3. Оптимизация физических компонентов: Физические взаимодействия (использование коллизий, компонента rigidbody, constraint и.т.д) требуют большого объема вычислений, поэтому сокращение количества объектов, участвующих в физической симуляции значительно влияет на производительность приложений.

4. Использование Sprite Packing - это процесс объединения нескольких спрайтов в один большой атлас спрайтов, с целью оптимизации загрузки ресурсов и производительности в игре. В Unity это возможно благодаря присутствию в среде инструмента для автоматической и ручной упаковки спрайтов. При сборке игры, движок автоматически объединяет все используемые в игре спрайты в один или более атласов спрайтов, уменьшая количество вызовов рендеринга и повышая производительность игры. Разработчик может создать собственный атлас спрайтов, размещая набор связанных спрайтов на одном изображении и разметив атлас с помощью инструментов Unity. Применение технологии упаковки спрайтов особенно полезно для сборок под мобильные устройства, где производительность крайне ограничена и требуется оптимизация ресурсов.

3. Оптимизация программного кода:

Существует несколько методов оптимизации программного кода с учетом особенностей среды Unity:

3.1 Сокращение количества команд, вызываемых каждый кадр (содержащихся в методе Update()), в особенности сокращение количества циклов существенно повысит производительность приложения. Также важно избегать вызовов метода GetComponent() для получения определенного компонента объектов внутри Update(). Вместо этого рекомендуется получить необходимые компоненты в результате одиночного события или в методе Start().

3.2 Использование Object pooling: Объединение объектов в пул - это техника, используемая для уменьшения объема памяти, используемой игрой. Вместо того чтобы создавать новые объекты, можно повторно использовать уже существующие. Это позволяет сократить объем используемой памяти и повысить производительность.

3.3 Использование Coroutine для функций задержки: В Unity. Функция WaitForSeconds() может быть использована для приостановки выполнения Сoroutine на определенное количество секунд. Это намного эффективнее, чем использование цикла внутри Update(), и не блокирует основной поток выполнения программы.