# **Обзор проекта**

Разработанная в рамках дипломного проекта кроссплатформенная видеоигра представляет собой 2D платформер - приключение, в котором игрок должен взять на себя роль доставщика почты, и на своем пути из начальной точки в конечную раздать посылки, а также познакомиться с сюжетом игры. Виртуальный мир видеоигры (место, где происходит действие) разбит на 3 основных зоны, каждая зона представлена сценой в среде Unity. Сцена содержит фоновые декорации, внутренности помещений, акторов-персонажей (представляющих собой игровой объект, состоящий из физической модели, визуальной части и компонентов базового класса Unity Monobehavior), а также множество предметов (также являющиеся игровыми объектами, обладающими физической моделью, визуальной частью, программными компонентами), которые будут использоваться игроком.

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

(Картинка - параллакс)

Эффект параллакса - это визуальный эффект, используемый в 2D-видеоиграх, в которых декорации заднего плана представляются игроку объектами, имеющими меньшую скорость, по сравнению с декорациями переднего плана. Этот эффект используется для создания ощущения глубины в игровой среде, придает реалистичности игровому опыту пользователя. Эффект параллакса достигается за счет использования слоев изображений, которые перемещаются с разной скоростью относительно камеры и прорисовываются на разной глубине. Например, слой гор на заднем плане относительно камеры перемещается медленнее, чем слой деревьев на переднем плане.

(Скрин из движка со слоями)

Кроме того, эффект параллакса используется для создания ощущения движения при статичной сцене, поскольку объекты на заднем плане движутся с разной скоростью при движении камеры. Этот эффект может быть использован для создания ощущения динамичного движения, что делает игровую среду более живой.

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 - управляемый игроком с помощью кнопок интерфейса и жестов ( на мобильных платформах) актор, игровой персонаж. Игровой персонаж обладает плоской физической моделью, что позволяет ему участвовать в физической симуляции мира игры.

Основные функции 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 мелкие объекты в руки, возможность взаимодействия с различными механизмами и приспособлениями (рычаги, вентили, крюки для игровых объектов);
* -Взаимодействие с источниками освещения (возможность отбрасывать тень);

Персонаж игры может передвигаться по игровому миру как самостоятельно, так и с помощью парового колесного механизма. Паровая машина состоит из множества модулей, позволяющих ей функционировать. Игрок должен поддерживать техническое состояние модулей для их работы;

Основные функции паровой машины:

* -Возможность движения 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 и персонажей игры;
* -Обеспечение работы модулей;
* -Поломка модулей и возможность их починки;

На пути игрока через игровой мир созданы преграды - головоломки, которые должен решить игрок, чтобы получить вознаграждение или открыть путь дальше.

Дополнительные модули для паровой машины, а также головоломки и сюжетные моменты вводятся в игру согласно кривой 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 игры, а также предоставляя награды и стимулы для поддержания интереса игроков. Кроме того, разработчики игр 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, разработчики игр могут гарантировать, что их игроки останутся вовлеченными и заинтересованными на протяжении всей игры.

В игре представлено множество механизмов и головоломок, с которыми взаимодействует игрок с помощью физического воздействия главного персонажа или взаимодействия с элементами интерфейса. Рассмотрим пример взаимодействия игрока и одной из головоломок игры.

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

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 предмета 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

# **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 для одновременного использования игровых ассетов и кода, а также позволяет использовать один и тот же интерфейс приложения для разных платформ. Настройка приложения для различных платформ производится автоматически или в ручном режиме. Для проектов на Unity также доступны библиотеки и плагины, разработанные сторонними компаниями и пользователями, которые позволяют ускорить и упростить процесс разработки. Все это делает Unity отличным выбором для разработки приложений на разных платформах.

Благодаря представленным возможностям кроссплатформенной разработки, разработчику приходится тратить меньше денежных ресурсов на разработку и поддержку версий для различных устройств.

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

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 отлично подходит для реализации плагинов для разработки внутри проекта, которые можно с лёгкостью интегрировать для отображения в интерфейсе движка. Для реализации собственных инструментов можно взаимодействовать с макетом редактора UnityEditor, консолью и инструментами отладки и редактирования сцены Gizmos напрямую, или воспользоваться сторонним дополнением для создания инструментов и настройки пользовательских интерфейсов созданных инструментов. Подобные системы ускоряют процесс разработки и положительно влияют на качество продукта, благодаря повышению удобства взаимодействия разработчика с движком, а также повышению 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 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 мобильных устройств с различными характеристиками 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.

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

# **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 предпета, itemMagnet - компонент, придающий предмету свойство магнита, itemScrap - компонент, позволяющий разбирать предмет ЕЩЕ КАКИЕ\_НИБУДЬ). 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 представляет данные или информацию игры, представление представляет 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.

(Картика 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.

# **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 на сцене.

В иерархии сцены содержатся дочерние и родительские объекты - это важная концепция в движке Unity Engine. Дочерние объекты - это объекты, которые создаются внутри других объектов, также известных как родительские объекты. 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, по сути, 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. Пользовательские компоненты также могут использоваться 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.

# **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.

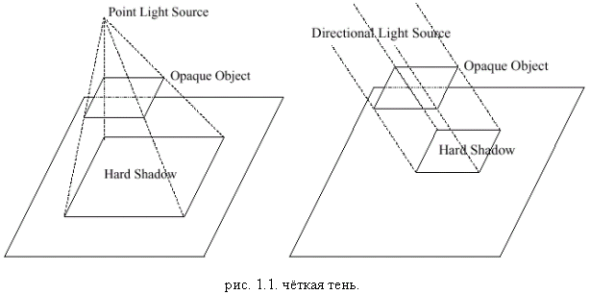
**СВЕТОТЕНЬ**

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.

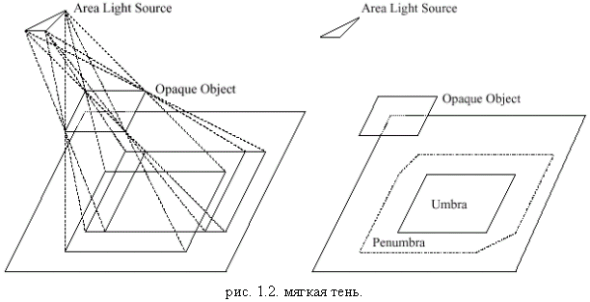
В компьютерной графике в реальном времени объекты часто отображаются без теней, что создает впечатление, будто объект не привязан к окружающей среде. Тень передает большое количество информации - по сути, она представляет объект с другой точки зрения и закрепляет его в сцене.

Тень - один из эффектов освещения на глобальном уровне. Для построения тени учитывается не столько ориентация минимального полигона модели, отбрасывающей тень, сколько взаимное расположение всех полигонов в сцене. Современные графические укорители работают на уровне минимальных полигонов – треугольников и отдельных векторов, поэтому тени не поддерживаются ими напрямую и разработка рендера теней становится программной задачей, которую необходимо воплотить в видеоигре. В наиболее распространённых алгоритмах отрисовки тени практически все стадии построения тени реализуются именно через графический ускоритель.

Тени бывают чёткие (hard shadows) и мягкие (soft shadows). Чёткие тени получаются, когда объект, отбрасывающий тень подвергается воздействию точечного источника света или источника направленного параллельного света. В этом случае, согласно геометрической оптике Френеля, тень получается, как показано на рисунке



В случае наличия протяжённого источника света, от объекта получается не одна тень, а серия теней, которые накладываются друг на друга и на участках многократного наложения образуют более затемнённые области, степень затемнения зависит от количества наложенных теней. Эти участки образуют области которые и составляют основную тень (*umbra*) и область полутени (*penumbra*). Процесс многократного проецирования тени на геометрию сцены слишком ресурсоемкий для использования в современных видеоиграх. На данный момент графические ускорители не способны визуализировать тени, построенные на таком алгоритме.



Все существующие алгоритмы построения теней позволяют строить чёткие тени, но некоторые из них позволяют размыть полученную тень, создавая псевдо-мягкую тень, имитацию тени и полутени.

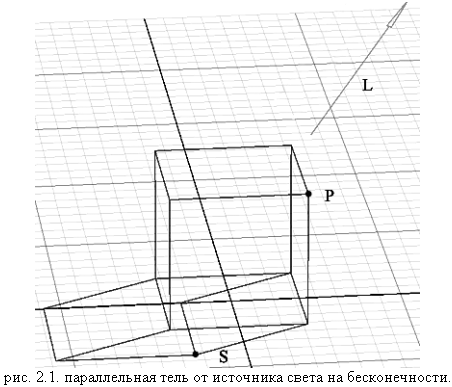
Для создания физически верной тени в игре был применен так называемый метод преобразования “на землю”.

Фактически, это первый алгоритм построения тени, который был применён в играх. (Turok II: Seeds of Evil (1998), Shogo: Mobile Armor Division (1998), Quake (1996), Half-Life (1998)). Он отличается простотой реализации и хорошим качеством получаемой тени. Этот алгоритм был впервые описан Джимом Блинном. В своей статье он описал уравнения для проектирования полигона "на землю", т.е. на плоскость z=0, в направлении от источника света. Он рассмотрел два случая:

1. Источник на бесконечности (параллельный направленный свет)
2. Локальный источник (точечный источник недалеко от объекта)

Этот метод использует геометрическое взаимоотношение источника света и полигона, т.е. подобные треугольники, для вычисления проекции каждого полигона модели "на землю". "Теневые полигоны" должны быть рассчитаны для каждого источника света, т.е. если объект освещается N источниками света, то необходимо рассчитать N его "теневых проекций".

#### Источник на бесконечности



В случае бесконечно удалённого источника света мы предполагаем, что лучи света, приходящие к объекту, полностью параллельны. Это позволит нам решить уравнение проекции только раз и применять полученное решение ко всем вершинам объекта.

**Общая постановка задачи:**

Имея точку источника света https://www.ixbt.com/video/theor/shadows/f1.gifи вершину объекта https://www.ixbt.com/video/theor/shadows/f2.gif, мы хотим получить проекцию вершины объекта на плоскость z=0, т.е. точку тени https://www.ixbt.com/video/theor/shadows/f3.gif

Из подобных треугольников получаем: https://www.ixbt.com/video/theor/shadows/f4.gif

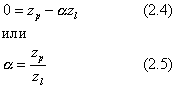
решая это уравнение относительно https://www.ixbt.com/video/theor/shadows/4-1.gif, получаем:

https://www.ixbt.com/video/theor/shadows/f5.gif

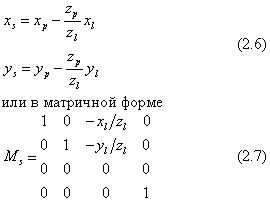
если принять, что L это вектор из точки P к источнику света, то точку S можно выразить как

https://www.ixbt.com/video/theor/shadows/f6.gif

т.к. мы производим проекцию на плоскость z=0, то уравнение (2.3) можно переписать в следующем виде:



решая (2.3) относительно https://www.ixbt.com/video/theor/shadows/f8.gif, получаем



Теперь имея координаты точки P в мировом координатном пространстве, можно получить её проекцию на плоскость z=0 просто путём умножения на матрицу https://www.ixbt.com/video/theor/shadows/f10.gif

https://www.ixbt.com/video/theor/shadows/f11.gif

#### Локальный источник

Уравнение (2.6) для бесконечно удалённого источника света может быть обобщено для случая, когда источник света находится на конечном расстоянии от объекта. В этом случае нам понадобятся дополнительные вычисления на каждую вершину, т.к. каждая вершина имеет, в общем случае, своё собственное направление на источник света. Тем не менее, в этом случае мы тоже можем перенести большую часть вычислений в матрицу https://www.ixbt.com/video/theor/shadows/f10.gif

Если L это точка расположения источника света, то (2.3) принимает вид:

и снова нам необходимо произвести проекцию на плоскость z=0, т.ч.

Если использовать гомогенизацию после преобразования, то (2.11) можно записать в виде матрицы

Опять, имея координаты точки P в мировом координатном пространстве, можно записать:

после чего провести гомогенизацию точки для получения проекции точки P на плоскость z=0.

### Построение теневой маски и проективное наложение

Этот метод является логическим продолжение предыдущего, но, в отличие от него, обладает целым рядом преимуществ.

Объект, который отбрасывает тень (shadow caster) тем или иным способом рисуется с точки зрения источника света чёрным цветом в белую текстуру. Размер текстуры зависит от того, насколько мелкие элементы объекта мы хотим видеть в тени (например, в игре Drakan: Order of the flame "теневая" текстура была размером примерно 32х32 пиксела).

Представьте себе плоскость, перпендикулярную направлению на источник света и расположенную сразу за объектом по лучу света — это и есть текстура, в которую производится отрисовка объекта. Для того, чтобы получить проекцию объекта на эту текстуру, применим элементы стандартного конвейера трансформации. Предположим, что текстура, в которую нам необходимо нарисовать объект, это экран. Тогда последовательность преобразований становится просто очевидной (стандартный конвейер преобразования): пространство объекта => мировое пространство => пространство камеры (у нас это источник света) => проективное пространство => нормализованное проективное пространство => экранное пространство.

Для наших целей необходимо модифицировать матрицы переходов в пространство камеры, проективное пространство и в экранное пространство.

**Пространство камеры**

Практически, это обыкновенная матрица камеры, но только расположена эта виртуальная камера в точке нахождения источника света и направлена на shadow caster. В случае бесконечно удалённого источника света виртуальная камера может располагаться в любой точке на прямой между shadow caster и источником света (обычно её выбирают близко к shadow caster).

**Проективное пространство**

Эта матрица существенно зависит от того, является ли источник света локальным или бесконечно удалённым. Для бесконечно удалённого источника света это матрица ортографического преобразования:

где width, height и depth это ширина, высота и глубина проективного объёма. В этом случае (ортографическое преобразование) не требуется гомогенизация.

Для локального источника света необходима матрица перспективного преобразования:

где — угол зрения, aspect — отношение ширины к высоте экрана (при отрисовке в квадратную текстуру aspect=1), far и near — дальняя и ближняя стенки проективного объёма соответственно. После перспективного преобразования необходима гомогенизация, как и в случае локального источника при преобразовании "на землю".

Для качественной картинки рекомендуется выбирать проективный объём таким образом, чтобы он максимально плотно охватывал shadow caster и не оставлял зазоров.

**Экранное пространство**

Здесь всё просто — модель из нормализованного проективного пространства должна лечь в квадратную текстуру с минимальными зазорами.

После того, как shadow caster нарисован в текстуру, необходимо эту текстуру наложить на затенённые объекты (shadow receiver). Для этого можно воспользоваться технологией [проективного наложения текстур](https://www.ixbt.com/video/projectivetexture.shtml), которая легко реализуется на современном железе. Для избежания повторения теневой текстуры необходимо выставить clamp или border color для режима "обёртки" текстуры.

В результате мы получаем тень от одного объекта (shadow caster) на другом (shadow receiver). Что ещё хочется сделать с этой тенью:

1. Размыть её (получить псевдо-мягкую тень). Для этого необходимо размыть теневую текстуру. Сделать это можно несколькими способами:
   1. "Вручную" путём наложения какого-нибудь blur-фильтра. Такой подход удобен при программной отрисовке самой теневой текстуры.
   2. Использовать железо. Можно отрисовать текстуру саму в себя со смещением, либо сгенерировать mip-level.
2. Уводить тень в прозрачность на основе расстояния до источника света. К сожалению, реализовать это без vertex shader при помощи железа невозможно. Единственное, что можно здесь сделать для улучшения вида тени, так это сделать её не угольно чёрной, а равномерно полупрозрачной.
3. Минимизировать затраты на отрисовку тени. Это достигается при использовании какой-либо технологии LOD — просто выбирается оптимально-минимальный уровень у shadow caster и вся теневая текстура рисуется именно с этим уровнем.

Для создания теней был применён изменённый алгоритм проецирования тени на плоскость. Фактически, это первый алгоритм построения тени, который был применён в играх. Алгоритм отличается простотой реализации и настраиваемым разрешением получающейся тени. Этот алгоритм был впервые описан Джимом Блинном. В статье Me and my (fake) shadow (Jan 1988, vol 8, issue 1)

How to cast shadows on the ground plane using perspective transformations

Применим уравнения для проектирования полигона 3D-модели на плоскость земли, т.е. на плоскость z=0, в направлении от источника света.

Существует два случая:

Источник света на бесконечности (параллельный направленный свет)

Локальный источник (точечный источник недалеко от объекта)

Этот метод использует геометрическое взаимоотношение источника света и полигона, т.е. подобные треугольники, для вычисления проекции каждого полигона модели "на землю". "Теневые полигоны" должны быть рассчитаны для каждого источника света, т.е. если объект освещается N источниками света, то необходимо рассчитать N его "теневых проекций".

Источник на бесконечности

В случае бесконечно удалённого источника света мы предполагаем, что лучи света, приходящие к объекту, полностью параллельны. Это позволит нам решить уравнение проекции только раз и применять полученное решение ко всем вершинам объекта.

... здесь рассчитывать....

Адаптируя решение под задачи проекта и работу с 2D графикой,

Параллакс - эффект глубины в 2D-играх, позволяющий с помощью нескольких плоских подвижных слоев имитировать объем объекта при движении камеры.

В реализованном проекте все слои фона движутся вслед за камерой. Формула, задающая положение слоя в каждом кадре :

ФОРМУЛА

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

ПРО ПОКАЗАТЕЛИ МНОЖИТЕЛЯ РАССКАЗАТЬ

БЛОК СХЕМА АЛГОРИТМА

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 формляется 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 взаимодействия с видеоигрой, потребностей пользователя. Элементы интерфейса должны быть структурированы по смыслу и расположены согласно иерархии объектов UI. Следует учитывать частоту, удобство взаимодействия с элементами UI

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).

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

5. Адаптация: Пользовательский интерфейс должен быть разработан с рассчетом на работу на различных устройствах, с разными параметрами экрана.

Отзывчивость: Элементы интерфейса должны выполнять закрепленные за ними функции с минимальной задержкой.

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

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

Здесь про макет ui игры из чего он состоит и тп

Оптимизация в видеоиграх необходима для создания приятного игрового процесса. Она включает в себя процессы по обеспечению плавности выводящегося на экран изображения (работа на увеличение частоты кадров), уменьшение вычислительной нагрузки на аппаратуру, а также адаптация программного кода для конкретных устройств.

Существует несколько методов оптимизации сцены и программного кода видеоигр:

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.

2. Использование низкополигональных модели: Низкополигональные модели - это модели, которые менее детализированы, чем их высокополигональные аналоги. Использование низкополигональных моделей позволяет значительно сократить объем отрисовываемой геометрии.

3. Использовать Level Of Detail: Уровень детализации (LOD) - это технология, используемая для уменьшения детализации объектов по мере их отдаления от виртуальной камеры в игровом мире. Это может значительно повысить производительность, так как требуется рендеринг меньшего числа геометрии сцены.

4. Использование окклюзии: Occlusion culling - это технология, используемая для уменьшения количества объектов, которые рисуются на экране. Объекты, которые не видны виртуальной камере, не отрисовываются.

8. 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.