ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет об программном проекте на тему:

3D игровой движок на C++

Выполнили:

студент группы БПМИ215
Баландин Кирилл Андреевич _____ (подпись) _____ (дата)

студент группы БПМИ215
Балаян Степан Хосровович _____ (подпись) _____ (дата)

студент группы БПМИ215
Рысин Денис Павлович _____ (подпись) _____ (дата)

студент группы БПМИ215
Нгуен Фук Туе _____ (подпись) _____ (дата)

Принял руководитель проекта:

Сосновский Григорий Михайлович
Приглашенный преподаватель
Факультета компьютерных наук НИУ ВШЭ

_____ (подпись) _____ (дата)

Москва 2023

Содержание

Аннотация

В данной работе описывается создание игрового 3D движка на C++. Цель проекта - создать простой, но гибкий кроссплатформенный игровой движок для инди-игр с поддержкой физики, звука и скриптинга.

Ключевые слова

Рендеринг, OpenGL, OpenAL, Физический движок, Скриптинг, Анимации, PBR, Постпроцессинг

1 Введение

В современном мире потребности в инструментах для производства интерактивного медиа контента становится всё больше и больше, из-за чего появилась необходимость в специальных программах, содержащих обширные наборы различных инструментов для создания виртуальных миров. Такие программы называются игровыми движками и используется в различных сферах от создания визуализаций до графики в кино, но преимущественно для создания видеоигр. Игровой движок делится на две основные части: библиотека и редактор. Библиотека рассчитана на использование её функционала в независимых проектах, например создании игры с нуля в обход низкоуровневых API операционных систем и драйверов. В функционал библиотеки включают самые главные компоненты, отвечающие за работу итогового проекта такие как: рендерер, физический движок, менеджер скриптов, логика приложения и т.д. Редактор со стороны библиотеки является независимым проектом и его основной целью является процесс разработки продукта, из-за чего в него добавляют множество различных инструментов для упрощения разработки. Целью данной курсовой работы является написание библиотеки и редактора для создания видеоигр.

1.1 Разделение задач по участникам

Рысин Денис Павлович:

- PBR – physical based rendering
- Реализовать поддержку анимации и её импорта из хотя бы одного формата 3D моделей
- Реализовать механизм скриптинга на C++ с возможностью перезагружать скрипты во время работы программы.
- Реализовать Entity Component System

Баландин Кирилл Андреевич:

- Физическая симуляция мира
- Просчет коллизий у физических тел
- Разрешение коллизий

Балаян Степан Хосровович:

- Звуковой движок

- Compute Shaders
- Deferred Shading

Нгуен Фук Туе:

- Портирование движка на разные ОС
- Постпроцессинг, эффект bloom

2 Физический движок

2.1 Введение

Физический движок - немаловажная часть любой современной игры. Он помогает создать некоторый виртуальный мир, в котором мы описываем законы взаимодействия, которые чаще всего приближены к реальности.

2.2 Анализ

Среди физических движков с открытым исходным кодом можно выделить несколько.

Первый, ReactPhysics3D - исходный код написан на C++. Достаточно прост в базовом использовании, но сложен в изучении и интеграции с уже готовой базой.

Второй, PhysX - нет открытого исходного кода, предоставляет API для C++. Поддерживает все операционные системы, но не все графические процессоры. Сложен в использовании и изучении. Идеально подходит для требовательных игр в AAA-индустрии.

Третий, qu3e - исходный код написан на C++, нет зависимостей. Прост в использовании, нет возможности сериализации состояния мира. Был архивирован в ноябре 2022 года, больше не поддерживается

В результате сравнения этих движков выбор сделан в пользу написания самописного решения. Требования к физическому движку минимальные, так как игровой движок позиционируется для создания легких инди-игр.

2.3 Фазы симуляции

Симуляция физики происходит в несколько фаз, давайте подробнее разберем каждую из них.

Первая фаза является опциональной и в англоязычных источниках называется broadphase. В эту фазу происходит проверка, могут ли два тела пересечься. В эту фазу допустимы ложно положительные проверки.

Вторая фаза - проверка на пересечение. Во время этой фазы проверяются все пары объектов точным алгоритмом. Ложно положительные проверки недопустимы. В этой фазе так же считается вся информация о коллизии, а именно: точки контакта, вектора нормалей для каждой из точек, глубина пересечения.

Третья фаза - разрешение коллизий. Во время этой фазы физический движок пытается урегулировать все нарушения физики. Именно в этой фазе меняется положение тел в пространстве, их скорости(линейная и угловая).

2.4 Обозначения

Вначале, давайте введем некоторые обозначения

Твердое тело - объект, перемещение которого мы симулируем. Все части этого объекта не меняют свое положение относительно друг друга. Иными словами, это тело не деформируется. У такого тела есть несколько основных характеристик: масса, момент инерции, положение и ориентация в пространстве, линейная и угловые скорости.

Коллайдер(Хитбокс) - геометрическая оболочка тела, описывающая физические границы тела.

Коллизия - пересечение двух коллайдеров. Касание коллайдеров тоже считается коллизией.

Пятно контакта - множество точек, в которых происходит пересечение. Дальше будет употребляться термин контакт.

2.5 Перемещение тел в пространстве

У любого физического тела есть основные характеристики - положение тела в пространстве и его вращение. Обе эти величины можно задать вектором в трехмерном евклидовом пространстве.

Линейная скорость, обозначаемая $v(t)$ - мера изменения положения тела в пространстве. Изменение положения - интеграл $\int v(t)dt$. Но мы будем использовать приближенное вычисление, используя метод Эйлера: $x(t + \Delta t) = x(t) + v(t) \cdot \Delta t$

Скорость вращения(угловая скорость), обозначаемая $\omega(t)$ - мера изменения вращения тела в пространстве. Аналогично, мы будем использовать метод Эйлера и вращение тела можно считать как $o(t + \Delta t) = o(t) + \omega(t) \cdot \Delta t$

Когда мы применяем силу к телу, меняется его линейная скорость и, в зависимости от направления и точки применения, его скорость вращения.

Любое применение силы приводит к появлению момента силы, обозначаемого $\tau(t)$. Если момент силы не равен 0, то изменяется скорость вращения. Момент силы можно посчитать как векторное произведение силы и вектора r, исходящего из центра масс тела до точки применения силы: $\tau(t) = r(t) \times F(t)$

Момент инерции I - это угловой эквивалент массы, который определяет, насколько трудно повернуть объект вдоль определенной оси. При выборе другой оси вращения момент инерции может быть другим. Для кубов, момент инерции одинаков для всех осей вращения.

Линейный импульс $P(t)$ - мера, описывающая, насколько трудно изменить линейную скорость тела. Вычисляется по формуле: $P(t) = m \cdot v(t)$.

Угловой импульс $L(t)$ - мера, описывающая, насколько трудно изменить угловую скорость. Вычисляется по формуле $L(t) = I \cdot \omega(t)$

Научимся считать изменение скоростей после применения силы:

$$\Delta P(t) = F(t) \cdot \Delta t \tag{1}$$

$$\Delta v(t) = \Delta P(t) \cdot m^{-1} = F(t) \cdot m^{-1} \cdot \Delta t \tag{2}$$

$$\Delta L(t) = \tau(t) \cdot \Delta t \tag{3}$$

$$\Delta \omega(t) = \Delta L(t) \cdot I^{-1} = \tau(t) \cdot I^{-1} \cdot \Delta t \tag{4}$$

2.6 Проверка на пересечение

В основе проверки на пересечение был выбран алгоритм, основанный на теореме о разделяющей оси. Давайте остановимся на ней поподробее.

Теорема. Две выпуклые геометрии не пересекаются, тогда и только тогда, когда между ними существует плоскость, которая их разделяет. Ось, ортогональная разделяющей гиперплоскости, называется разделяющей осью, а проекции фигур на нее не пересекаются.

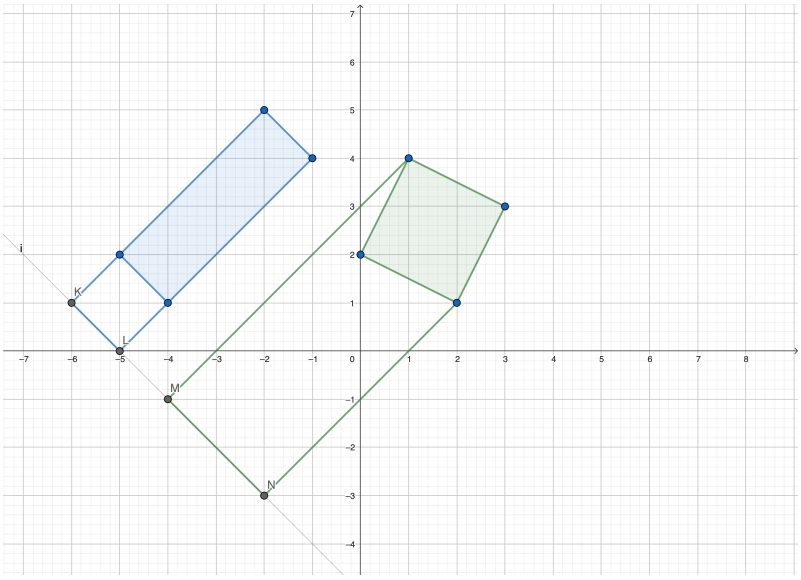

Рис. 2.1: Пример оси, ортогональной разделяющей плоскости.

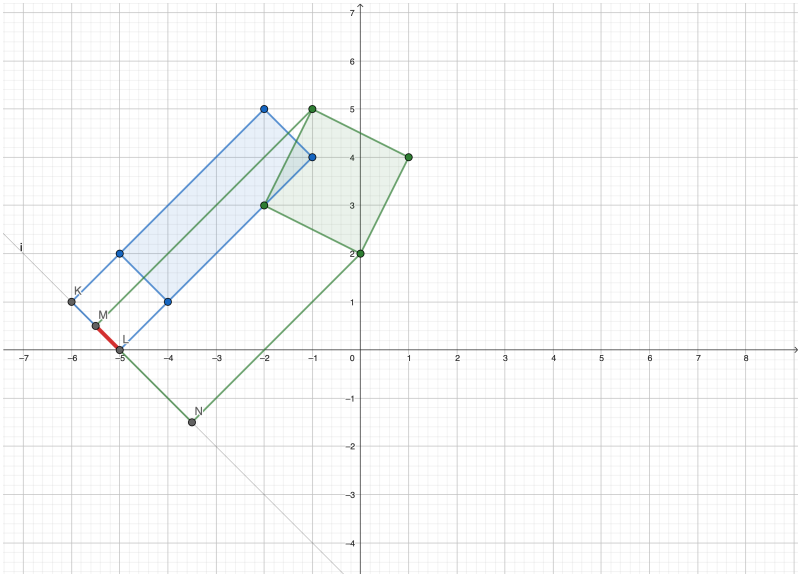

Рис. 2.2: Пример работы теоремы при наличии пересечения.

Для выпуклых многогранников эта ось будет либо нормалью какой-то грани, либо векторным произведением ребер из разных многогранников. При этом нормаль коллизии будет лежать вдоль оси, где пересечение проекций минимально.

Следующим шагом в проверке на пересечение будет определение точек контакта. Для этого необходимо пересечь коллайдеры с плоскостью, проходящей через точку с минимальной(максимальной) проекцией на разделяющую ось и ортогональной вектору нормали коллизии.

Пересечение выпуклого многогранника и плоскости реализовано следующим образом. Для каждой вершины считаем ориентированное расстояние до плоскости. Если оно равно 0, то эта точка находится на пятне контакта. Затем для всех ребер проверяем произведение расстояний от вершин до плоскости, подсчитанных на прошлом этапе. Если оно **строго** отрицательное, то значит, что точка контакта находится где-то на этом ребре, найдем ее. Таким образом мы получим точки контакта.

2.7 Разрешение коллизий

Фаза разрешения коллизий использует концепцию ограничений. Твердое тело в трех измерениях имеет 6 степеней свободы: 3 позиционных и 3 угловых. Ограничение уменьшает степени свободы тела. Например, ограничение, которое удерживает объект в пространстве в его центре масс, уменьшает степени свободы объекта на 3: все позиционные степени свободы удаляются, и, таким образом, объект теперь может вращаться только с 3 степенями свободы.

Любое взаимодействие с телом реализовано за счет ограничений. С точки зрения математики, ограничение это некоторое уравнение вида $C = 0$, где C - это некоторая линейная комбинация характеристик тела(положение, вращение).

Есть разные способы решения ограничений. Самый популярный метод называется Sequential Impulse и был предложен автором библиотеки Box2D. Его суть в том, что мы будем допускать нарушения ограничений в некоторые моменты времени. Изменение положения тела или его вращения с целью мгновенного решения ограничений может привести к дрожанию тела, и физика не будет выглядеть реалистично. Поэтому предлагается, дифференцировать выражение $C = 0$, чтобы получить соответствующее ограничение скорости C', а затем решить для C'.

Ограничение скорости C' между двумя твердыми телами моделируется как линейная комбинация линейных и угловых скоростей обоих твердых тел плюс смещение:

$$C' : JV + b = 0 \tag{5}$$

Где V это вектор скоростей, содержащий скорости(как линейные, так и угловые) обоих тел.

$$V = \begin{bmatrix} \overrightarrow{V_A} \\ \overrightarrow{\omega_A} \\ \overrightarrow{V_B} \\ \overrightarrow{\omega_B} \end{bmatrix} \tag{6}$$

V_A и V_B обозначают линейные скорости тел A и B соответственно; ω_A и ω_B обозначают угловые скорости тел A и B соответственно

J - вектор, содержащий коэффициенты линейной комбинации скоростей. В англоязычной литературе его называют якобианом, но ничего общего с матрицей Якоби он не имеет.

$$J = \begin{bmatrix} \overrightarrow{J_{V_A}}^T & \overrightarrow{J_{\omega_A}}^T & \overrightarrow{J_{V_B}}^T & \overrightarrow{J_{\omega_B}}^T \end{bmatrix} \tag{7}$$

Где $\overrightarrow{J_{V_A}}, \overrightarrow{J_{\omega_A}}, \overrightarrow{J_{V_B}}, \overrightarrow{J_{\omega_B}}$ - векторы, представляющие коэффициенты линейной комбинации.

Смещение b рассмотрим позже.

Чтобы уметь удовлетворять ограничением, заданным уравнением 5, нам необходимо как-то менять скорости тел.

$$J(V + \Delta V) + b = 0,$$

Рассмотрим

$$M = \begin{bmatrix} M_A & 0 & 0 & 0 \\ 0 & I_A & 0 & 0 \\ 0 & 0 & M_B & 0 \\ 0 & 0 & 0 & I_B \end{bmatrix}$$

$$M_A = \begin{bmatrix} m_A & 0 & 0 \\ 0 & m_A & 0 \\ 0 & 0 & m_A \end{bmatrix}, M_B = \begin{bmatrix} m_B & 0 & 0 \\ 0 & m_B & 0 \\ 0 & 0 & m_B \end{bmatrix}$$

.

I_A, I_B - вектора, описывающие моменты инерции тел вдоль осей, параллельных базисным векторам.

Исходя из (2) и (4), получаем

$$\Delta V \sim M^{-1} \tag{8}$$

Мы можем думать о процессе определения ΔV, в виде: $V = J^T \cdot M^{-1} \cdot \lambda$.

$$J(V + M^{-1}J^T\lambda) + b = 0\lambda = \frac{-(JV + b)}{JM^{-1}J^T} \tag{9}$$

Если у нас есть несколько ограничений, то мы итеративно повторяем этот процесс для всех ограничений и система придет к глобальному решению. Отсюда и пошло название метода Sequential Impulse

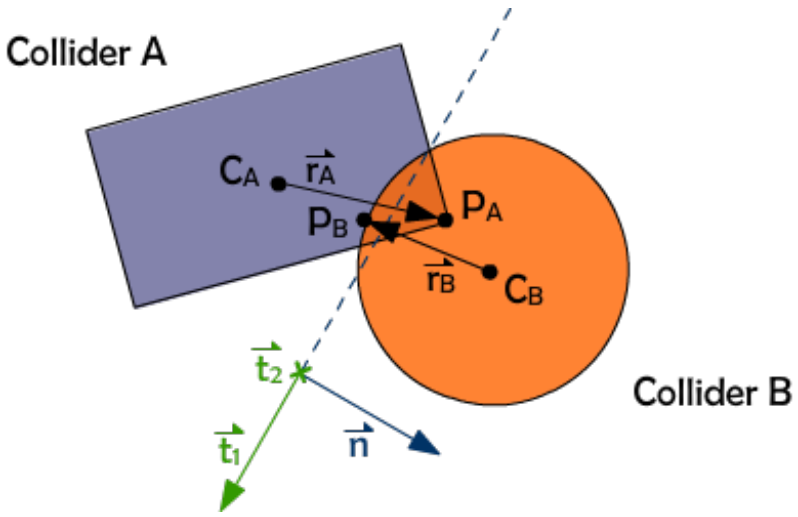

Рис. 2.3: Пример пересечения и обозначения

После того как мы научились разрешать ограничения, необходимо научиться их строить. Рассмотрим коллизию на примере 2.3. C_A, C_B - центры масс тел A и B соответственно. r_A, r_B - вектора от центра масс до наиболее глубокой точки пересечения. $\overrightarrow{n}$ - нормаль коллизии. $\overrightarrow{t_1}, \overrightarrow{t_2}$ - касательные к пересечению, в примере t_2 направлен от нас.

Наша цель - довести глубину пересечения до 0, поэтому мы хотим, чтобы $\overrightarrow{P_A P_B} \cdot \overrightarrow{n} \geq 0$

$$P_A = C_A + \overrightarrow{r_A} \tag{10}$$

$$P_B = C_B + \overrightarrow{r_B} \tag{11}$$

$$C \ : \ (P_B - P_A) \cdot \overrightarrow{n} \geq 0 \tag{12}$$

$$C \ : \ (C_B + \overrightarrow{r_B} - C_A - \overrightarrow{r_A}) \cdot \overrightarrow{n} \geq 0 \tag{13}$$

Продифференцировав это выражение по времени, получаем

$$C' \ : \ (-\overrightarrow{V_A} - \overrightarrow{\omega_A} \times \overrightarrow{r_A} + \overrightarrow{V_B} + \overrightarrow{\omega_B} \times \overrightarrow{r_B}) \cdot \overrightarrow{n} \geq 0 \tag{14}$$

Воспользовавшись свойством $\overrightarrow{A} \times \overrightarrow{B} \cdot \overrightarrow{C} = \overrightarrow{C} \times \overrightarrow{A} \cdot \overrightarrow{B}$, мы получим

$$\dot{C} \ : \ JV + b \geq 0, \tag{15}$$

$$J = \begin{bmatrix} -\overrightarrow{n}^T & (-\overrightarrow{r_A} \times \overrightarrow{n})^T & \overrightarrow{n}^T & (\overrightarrow{r_B} \times \overrightarrow{n})^T \end{bmatrix}, \tag{16}$$

$$V = \begin{bmatrix} \overrightarrow{V_A} \\ \overrightarrow{\omega_A} \\ \overrightarrow{V_B} \\ \overrightarrow{\omega_B} \end{bmatrix} \tag{17}$$

Изменение скорости тела повлияет на результат последующего разрешения того же ограничения. В какой-то момент мы можем столкнуться с изменением скорости, которое на самом деле сближает пары коллайдеров, вместо того чтобы раздвигать их. По этой причине нам нужно ограничить изменение скорости с помощью λ.

Мы хотим убедиться, что на протяжении всех итераций для коллизии в пределах одного временного шага выполняется следующее неравенство:

$$\Sigma\lambda_i \geq 0 \tag{18}$$

Для этого на каждой итерации ограничим λ_i снизу 0.

Разрешив это ограничение, мы заметим, что умеем останавливать тела от дальнейшего проникновения друг в друга, но не удовлетворять ограничению. Поэтому воспользуемся методом стабилизации Баумгарте. Его идея заключается в добавлении глубины проникновения тел в смещение в уравнении (5).

$$b = -\frac{\beta}{\Delta t} d \tag{19}$$

где d - глубина проникновения, β - некоторый коэффициент. Эксперементальным путем было выяснено, что лучше выбирать в отрезке $[0.2, 0.3]$

Мы разобрали только нормальную компоненту столкновения. Перейдем к тангенциальным, которые так же называют компонентами трения(friction).

Силы трения пропорциональны нормальным силам, поэтому они зависят от результата разрешения нормальной компоненты коллизии. Поэтому, тангенциальная часть ограничений разрешается после нормальной части.

Нормальная часть ограничений контакта пытается обнулить проекцию относительной скорости двух точек контакта на нормаль коллизии. Тангенциальная часть выполняет схожую роль: она пытается обнулить проекцию относительной скорости двух наиболее глубоких точек контакта на плоскость контакта (образованную двумя касательными t_1, t_2).

Касательная и нормальная компоненты ограничений в основном выполняют одно и то же действие, за исключением того, что направления, на которые проецируется относи-

тельная скорость тел, различны. Поэтому "якобианы"для двух касательных к контакту выглядят очень похожими на "якобиан"для нормальной помпоненты. Давайте переобозначим "якобиан"для контактной нормали $J_{\overrightarrow{n}}$ и посмотрим на него рядом с "якобианами"для двух касательных контакта, обозначенных $J_{\overrightarrow{t_1}}$ и $J_{\overrightarrow{t_2}}$.

$$J_{\overrightarrow{n}} = \left[\begin{array}{cccc} -\overrightarrow{n}^T & (-\overrightarrow{r_A} \times \overrightarrow{n})^T & \overrightarrow{n}^T & (\overrightarrow{r_B} \times \overrightarrow{n})^T \end{array} \right] \tag{20}$$

$$J_{\overrightarrow{t_1}} = \left[\begin{array}{cccc} -\overrightarrow{t_1}^T & (-\overrightarrow{r_A} \times \overrightarrow{t_1})^T & \overrightarrow{t_1}^T & (\overrightarrow{r_B} \times \overrightarrow{t_1})^T \end{array} \right] \tag{21}$$

$$J_{\overrightarrow{t_2}} = \left[\begin{array}{cccc} -\overrightarrow{t_2}^T & (-\overrightarrow{r_A} \times \overrightarrow{t_2})^T & \overrightarrow{t_2}^T & (\overrightarrow{r_B} \times \overrightarrow{t_2})^T \end{array} \right] \tag{22}$$

Как говорилось ранее, силы трения пропорциональны нормальным силам и их норма ограничена интервалом $[-C_F\lambda_{\overrightarrow{n}}; C_F\lambda_{\overrightarrow{n}}]$, где C_F - коэффициент трения

2.8 Реализация

Вся работа с матрицами и векторами реализована с помощью библиотеки glm.

У каждого объекта при его подгрузке из файла считается его коллайдер oriented bouncing box(OBB) и центр масс. Происходит это в классе ConvexCollider.

Входной точкой движка является класс Solver, который инициализируется при открытии сцены. Вся логика находится в методе OnUpdate. Изменение позиции и вращения тел происходит посредством взаимодействия с EntityComponentSystem(ECS). Для физических характеристик тела существует отдельная компонента PhysicsComponent, в ней содержится линейная скорость, скорость вращения, обратная масса и обратный момент инерции. Решено было хранить обратную массу и момент инерции, так как во многих местах используется операция деления, а так же удобство создания статичным объектов, например, поверхности. Чтобы сделать объект статичным, достаточно установить обратную массу равную 0. Тогда при любом решении ограничений, скорость этого объекта будет оставаться равной 0.

В начале цикла обновления к телам применяется гравитация сцены и у коллайдеров обновляется матрица трансформации.

Из-за отсутствия необходимости в симуляции большого числа тел, было решено отказаться от broadphase в прямом понимании этой фазы, так что после применения гравитации, сразу идет фаза проверки на пересечение.

Для каждой пары тел запускается алгоритм, описанный выше. В случае коллизии создается объект структуры Collision, инициализируется 3 объекта Jacobian. Один для нормальной компоненты, два других для тангенциальной.

После разрешения коллизий, в методе OnUpdate обновляется положение и вращение тела.

3 Звуковой движок

3.1 Введение

Аудиодвижок — это фундаментальная составляющая любого современного игрового движка, значение коротой часто преуменьшается. Он вносит огромный вклад в общую атмосферу и реализм игры, обеспечивая поддержку для аудиофайлов и звуков, которые воспроизводятся в ответ на игровые события.

Звуковое сопровождение игры включает в себя разнообразные элементы, такие как музыкальный фон, звуки окружающей среды, звуковые эффекты от взаимодействия с объектами и звуковое сопровождение персонажей. Например, шаги персонажа, голосовые фразы, звуки оружия и многие другие элементы подчеркивают действия и создают атмосферу.

Современные аудиодвижки могут обрабатывать большое количество звуковых источников одновременно и применять к ним различные эффекты в реальном времени. Они также обеспечивают трехмерное аудио, создают звуковое поле, позволяющее слушателю точно определить, откуда идет звук. Это может быть критически важно для игр, где ориентация в пространстве играет ключевую роль, например, в шутерах от первого лица.

Основной сложностью при разработке аудиодвижка является эффективное использование системных ресурсов. Звуковые файлы могут быть весьма объемными, и без должного управления они могут значительно увеличить время загрузки игры и уровня потребления памяти. Аудиодвижок, следовательно, должен быть способен управлять звуковыми ресурсами, загружая их при необходимости и выгружая, когда они больше не нужны, чтобы оптимизировать производительность.

Кроме того, для создания действительно погружающего аудиовпечатления аудиодвижок должен уметь эмулировать звуковую физику. Он должен учитывать такие аспекты, затухание, эхо, эффект Доплера и другие.

В заключении, аудиодвижок играет важную роль в создании реалистичных и впечатляющих игровых опытов. Он работает в тесном взаимодействии с другими компонентами игрового движка, чтобы создать убедительный и живой игровой мир. Поскольку звук является ключевым элементом в создании атмосферы и ощущения присутствия, качественный аудиодвижок может существенно улучшить общее впечатление от игры.

3.2 Обозначения

Listener (слушатель): Это объект, который представляет точку прослушивания в игровой среде. Свойства слушателя, такие как положение и ориентация, влияют на то, как звуки воспроизводятся.

Source (источник): Это объект, который воспроизводит звук. Источники имеют свойства, такие как положение, скорость и уровень громкости, которые влияют на то, как слушатель воспринимает звук.

Buffer (буфер): Это место, где хранятся аудиоданные до того, как они будут воспроизведены. Буферы заполняются данными и затем связываются с источниками для воспроизведения.

Streaming audio (потоковое аудио): Это техника, при которой аудиофайлы читаются и воспроизводятся по мере получения данных, а не после полного загрузки файла. Это особенно полезно для больших аудиофайлов или звуков в реальном времени. Пример использования потокового аудио − воспроизведение эмбиентов (фоновой музыки). Буферы поддерживают потоковое аудио.

Attenuation (затухание): Это процесс уменьшения громкости звука с увеличением расстояния между источником звука и слушателем. Это естественное явление, которое происходит из-за рассеивания и поглощения звука в окружающей среде.

Doppler effect (эффект Доплера): Это изменение частоты звука, воспринимаемого наблюдателем, когда источник звука или наблюдатель движутся относительно друг друга. Когда источник звука приближается к наблюдателю, частота звука, воспринимаемая наблюдателем, увеличивается, что приводит к эффекту повышения тона звука. Когда источник звука отдаляется от наблюдателя, частота звука уменьшается, вызывая эффект понижения тона звука.

Sound Positioning (позиционирование звуков): Это процесс определения и управления положением звуковых источников в пространстве относительно слушателя. Это позволяет создавать эффект присутствия и пространственного восприятия звуков.

Surround Sound (объемный звук): это технология и методика воспроизведения звука, которая позволяет создать ощущение звучания из разных направлений вокруг слушателя. Вместо простого воспроизведения звука через два передних динамика (стерео), окружающий звук добавляет дополнительные акустические каналы, располагаемые вокруг слушателя, для создания более реалистичного и пространственного звукового опыта.

3D audio (трехмерное аудио): Это технология и методика создания пространственного

восприятия звука, которая позволяет услышать звук так, как будто он исходит из определенных точек в трехмерном пространстве вокруг слушателя. 3D-аудио имитирует пространственные характеристики и эффекты, которые мы воспринимаем в реальном мире, чтобы создать эффект присутствия и погружения в звуковую среду. Включает в себя затухания, эффект Доплера, позиционирование, объёмный звук и многие другие.

3.3 Обзор аналогов

Существует множество фреймворков для создания звуковых движков, каждый из которых имеет свои особенности и преимущества. В этом обзоре мы сосредоточимся на трех ключевых игроках в этой области: OpenAL, Wwise и FMOD.

OpenAL, Wwise и FMOD − это три из самых известных и широко используемых аудиодвижка на рынке. Они были выбраны для обзора из-за своей репутации, функциональности и широкой поддержки платформ. При выборе аудиодвижка важно учитывать не только его функциональность, но и другие критерии, такие как поддерживаемые платформы, простота использования, производительность, гибкость и расширяемость, а также сегмент рынка, на который он ориентирован.

1 Функциональность

- OpenAL: Поддерживает 3D-аудио, эффекты Доплера и затухания, потоковое аудио, множество звуковых буферов и источников.
- Wwise: Обладает широким набором функций, включая поддержку 3D-аудио, MIDI, интерактивного музыкального оформления, настраиваемых звуковых эффектов и обработки звука.
- FMOD: Очень функционален, поддерживает 3D-аудио, реальное преобразование звука, создание пользовательских эффектов, а также имеет встроенную поддержку многих аудиоформатов.

2 Поддерживаемые платформы

- OpenAL: Кроссплатформенный, поддерживает Windows, macOS, Linux и мобильные платформы.
- Wwise: Поддерживает широкий спектр платформ, включая Windows, macOS, Linux, iOS, Android, и большинство консолей.
- FMOD: Поддерживает большое количество платформ, включая Windows, macOS, Linux, iOS, Android, и все основные игровые консоли.

3 Простота использования

- OpenAL: OpenAL может быть немного сложнее для новичков из-за его низкоуровневого характера.

- Wwise: Wwise предлагает удобный графический интерфейс пользователя и обширную документацию, что облегчает работу с ним.
- FMOD: FMOD известен своим простым и интуитивно понятным интерфейсом, который облегчает начало работы.

4 Производительность

- OpenAL: Обеспечивает достойную производительность, но, как правило, менее эффективен по сравнению с FMOD и Wwise.
- Wwise: Имеет мощные оптимизационные инструменты, что позволяет добиться высокой производительности.
- FMOD: Имеет очень хорошую производительность и оптимизирован для работы на различных платформах.

5 Расширяемость и гибкость

- OpenAL: OpenAL является открытым и гибким, позволяя создавать сложные звуковые эффекты, но требует больше работы на стороне программиста.
- Wwise: Wwise предлагает большую гибкость и расширяемость благодаря своей модульной структуре и поддержке плагинов.
- FMOD: FMOD также очень гибкий и поддерживает пользовательские плагины и DSP эффекты.

6 Сегмент рынка

- OpenAL: Часто используется в открытых проектах и инди-играх, где важна поддержка кроссплатформенности и доступность кода.
- Wwise: Широко используется в AAA-играх и профессиональных проектах, где требуются продвинутые аудиофункции.
- FMOD: Популярен как в инди-сообществе, так и среди AAA-разработчиков благодаря своей простоте использования и гибкости.

В результате сравнения этих трёх аудиодвижков выбор был сделан в пользу OpenAL. Это решение было принято на основании нескольких ключевых критериев.

Во-первых, единообразие – графический движок основан на OpenGL, и использование OpenAL, который является частью той же семьи технологий, обеспечивает высокую степень совместимости и гладкости интеграции между графическим и звуковым движками.

Во-вторых, простота использования и реализации. Наш игровой движок предназначен для создания легковесных инди-игр и, следовательно, не требует сложных и комплексных аудиоредакторов. OpenAL предлагает прямолинейный и интуитивно понятный интерфейс для работы со звуком, что упрощает процесс интеграции и использования в рамках нашего движка.

Наконец, прицел на инди-рынок, а не AAA-индустрию. OpenAL, несмотря на свою функциональность и производительность, ориентирован в основном на инди-разработчиков и открытые проекты, что делает его идеальным выбором для нашего проекта.

В заключение, учитывая эти факторы, OpenAL представляется наиболее подходящим звуковым движком для нашего проекта. Это гибкое, мощное и надежное решение, которое поможет нам создать убедительный и реалистичный звуковой ландшафт для наших игр. В перспективах дальнейшей разработки возможна интеграция FMOD.

3.4 Реализация

Перейдём к описании реализации аудиодвижка. Но перед этим нужно упомянуть, что из-за особенностей обработки ошибок в OpenAL, требуется написать обёртку для вызова функций. Для этого, следуя паттерну Фасад, создан класс AudioEngineFacade, который будет оборачивать все сложные детали работы с OpenAL и предоставлять простые методы для выполнения общих задач и обработки ошибок.

Процесс реализации состоял из следующих этапов:

1 Инициализация

Этап инициализации является важнейшим этапом в жизненном цикле любого движка. В случае аудиодвижка на базе OpenAL, инициализация включает в себя создание устройства и контекста OpenAL, которые являются основой для всей дальнейшей работы с звуками.

- Создание устройства: устройство в OpenAL соответствует аудиоустройству на компьютере. Создание устройства OpenAL — это процесс получения ссылки на аудиоустройство компьютера, которое будет использоваться для воспроизведения звука.
- Создание контекста: контекст в OpenAL представляет собой среду, в которой происходит работа со звуками. Контекст связан с устройством и позволяет управлять параметрами звука на более высоком уровне. OpenAL предполагает, что в любой момент времени будет существовать только один активный контекст. В связи с этим, паттерн Синглтон (одиночка) может быть полезен для управления этим контекстом. Класс AudioContextSingleton ответственен за создание и управление единственным экземпляром контекста OpenAL.
- Создании фабрики: объектов звука и буферов с большим количеством параметров в проекте может быть очень много. Поэтому для удобства создания таких объектов используется паттерн Фабрика и классы SoundFactory и BufferFactory.

2 Работа с источниками звука

- Создание источника: сначала создается объект источника звука. Этот объект содержит все необходимые параметры, такие как позиция источника, громкость и панорамирование звука, а также ссылку на буфер, содержащий аудиоданные (в буфере открывается нужный звуковой файл). Буфер создаётся с помощью BufferFactory.

- Добавление на сцену: после создания, источник звука может быть добавлен на сцену. Звук будет воспроизводиться из точки на сцене, где размещен источник.
- Управление источниками: после добавления на сцену, источник звука может быть перемещен, его громкость может быть изменена, может быть изменён звуковой файл и другие его параметры.

3 Развитие работы со звуками

Далее была внедрена поддержка потокового чтения звуковых файлов. Теперь буферы делятся на два типа:

- Буферы с "честным чтением": эти буферы используются для маленьких звуковых эффектов, которые можно полностью загрузить в память. Они быстро и просто воспроизводятся, но не подходят для больших аудиофайлов.
- Буферы с потоковым чтением: эти буферы используются для больших аудиофайлов, которые не могут быть полностью загружены в память. Они динамически загружают небольшие части аудиофайла по мере необходимости.

В зависимости от длительности и размера звукового файла определяется тип буфера. Однако пользователь также может вручную выбрать тип буфера.

4 Управление звуками

Теперь необходимо не только позиционировать и выставлять базовые параметры, но и добавить возможность им на лету обрабатывать события: остановить проигрывание, стать громче/тише, добавить эффекты.

- Для удобства работы с событиями, создаётся класс SoundEventManager. Он хранит ссылки на все источники и буферы, а также предоставлять функции для их управления. Он получает команды от других частей системы и обрабатывает их в соответствии с их приоритетом и временем поступления. SoundEventManager использует паттерн Синглтон, чтобы обеспечить единственный экземпляр для управления всеми звуковыми событиями.
- Для работы с событиями требуется ввести обработку состояний. Для каждого объекта введена система состояний: воспроизводит, остановлен (и должен быть освобождён), приостановлен.
- Реализация самих событий: это либо смена состояний, либо изменение параметров источника звука (без смены состояния).

5 Управление ресурсами и оптимизация

После имплементации общего функционала требуется обратить внимание на расход ресурсов. Управление ресурсами — важная часть оптимизации движка. Она состоит из управление памятью, источниками звука и потоками данных.

- Менеджер ресурсов, Синглтон, класс ResourceManager. Этот класс будет отвечать за отслеживание и управление всеми источниками звука и буферами.
- Создание пула источников звука.

  Простое создание и удаление источников звука является ресурсоемким процессом и может существенно влиять на производительность. Причины, почему постоянное создание и удаление источников не является оптимальным подходом:

  Во-первых, источники звука являются ограниченным ресурсом. Большинство аудиосистем ограничивают максимальное количество активных источников звука, которые могут быть использованы одновременно. Постоянное создание новых источников может привести к исчерпанию этих ресурсов.

  Во-вторых, создание и удаление источников звука требует времени и ресурсов. При большом количестве звуков это может негативно сказаться на производительности.

  В-третьих, остоянное создание и удаление объектов может привести к утечкам ресурсов, если они не удаляются должным образом.

  Таким образом, вместо создания нового источника каждый раз, когда требуется проиграть звук, движок может просто взять уже существующий источник из пула, обновить его параметры и начать воспроизведение. Когда звук завершает воспроизведение, источник возвращается в пул для будущего использования. Изначально пул пустует, в него добавляются источники до достижения верхней границы вместимости. Пул очищается при окончании работы программы.
- Логирование: добавление всех элементов аудио движка к общей системе логирования всего проекта.

4 Compute Shaders

4.1 Мотивация

Compute Shaders представляют собой относительно новую функцию, которая была добавлена в спецификацию OpenGL начиная с версии 4.3. Это особый тип шейдеров, который используется для выполнения вычислительно-интенсивных задач, которые не связаны напрямую с рендерингом графики.

До появления Compute Shaders, GPU обычно использовались только для графических операций. Тем не менее, они имеют значительную параллельную вычислительную мощность, которую можно использовать для других видов обработки данных, включая физические расчеты, обработку изображений и многое другое.

Compute Shaders позволяют программистам использовать эту вычислительную мощность более эффективно. Они предоставляют более прямой доступ к ресурсам GPU и памяти, что позволяет реализовывать более сложные алгоритмы, чем это возможно с использованием традиционных шейдеров.

Однако при разработке возникла следующая проблема: портировать на MacOS и Linux удалось до версии OpenGL 4.1, а compute shaders были добавлены в 4.3. Поэтому поддержку данной функциональности было решено вырезать.

5 Entity Component System

5.1 Мотивация

В играх практически всегда разработчику приходится иметь дело с большим количеством игровых объектов. Все они имеют разнообразную функциональность, но порой удобно выделять набор функциональностей в блоки и переиспользовать. Так же часто важно уметь быстро производить одинаковые манипуляции над одинаковыми блоками данных у различных объектов. В этот момент встаёт выбор между ООП и Data Oriented Design - методология написания кода ставящая на первое место данные и их расположение в памяти.

5.2 Сравнение

Рассмотрим пример. Допустим есть три логических объекта в игровой логике: Health, Hitbox и Mesh и важно уметь создавать объекты с произвольным подмножеством этой логики. При реализвации с ООП единственным выбором будет создать три отдельных класса для этих трёх элементов, но так же надо будет создавать по одному классу на каждую комбинацию этих компонентов и использовать множественное наследование. Какие есть минусы и плюсы у этого подхода:

- Наследование зачастую влечёт за собой оверхед по производительности защёт использования виртуальных функций. Так же при неообходимости создать объект с множеством компонентов прийдётся использовать множественное наследование, чего порой нету в язык, а так же ухудшает и без того негативный оверхед по производительности. И это ещё рассматривается пример с тремя компонентами, а в реальных задачах их число может привышать сотню, что взрывает экспоненциально количество возможных подмножеств этих компонентов, а следовательно и классов.

- В играх часто нужно обрабатывать множество компонентов одного типа в независимости от их владельца. Для решения этой проблемы в ООП скорее всего надо будет хранить все объекты в виде указателей на какой-то базовый класс, что влечёт за собой надобность в downcast, а следовательно ухудшение производительности, а также компоненты будут разбросаны в памяти как попало, что повлечёт множественные пробития кэша и так же потерю производительности. Так же для каждой такой системы надо будет хранить целый массив указатель, то есть указатель на один объект будет дублироватся в каждой системе что его использует, то есть опять же потеря производительности.

- Так же часто важно иметь возможность у уже созданного объекта забирать компонент или наоборот добавлять новый, не разрушая этот объект и пересобирая его заного. Такую проблему ООП в принципе решить не может, и единственным методом будет делать отдельные функции меняющие поведение уже созданного объекта.

- Вопреки всем вышеупомянутым недостаткам у ООП есть одно большое преимущество, а именно независимость объектов. Любой созданный класс может существовать сам по себе, не опираясь на иные существующие системы или контейнеры, что делает его предпочтительным вариантом при написании библиотек.

Таким образом со временем в играх стали использовать Design Oriented Patern, а именно Entity Component System.

5.3 Описание

Как можно понять из названия Entity Component System основывается на 3 основных объектах.

- Entity - он собственно и является единственным универсальным представлением для любого объекта, и никак не зависит от своих компонентов(в примере выше он заменяет все различные классы создаваемые для каждого подмножества компонентов)

- Component - логический кусок данных и функций который можно добавлять к любому Entity в любое время(из примера выше компоненты это Health, Hitbox и Mesh)

- System - объект производящий какие-то операциями над компонентами указанных типов зачастую в не зависимости от их владельцев(наприм если есть AnimationComponent отвечающий за анимирование mesh, его надо каждый кадр обновлять вне зависимости от владельца, как раз для этого можно создать отдельную систему)

Entity в сущности является просто уникальным номером(handle) и не хранит никакие компоненты на прямую. Сам по себе он существовать не может поэтому создаётся отдельный объект контейнер отвечающий за хранение компонентов и создание новых Entity. Внутри этого контейнера так же есть 3 важные части.

- EntityManager - объект отвечающий за генерацию новых и уникальных handle и поддерживает информацию о ныне созданных Entity

- ComponentManager - объект который содержит все компоненты всех Entity и информацию о их владельцах.

- SystemManager - объект обеспечивающий возможность добавления различных систем для работы над конмпонентами и предоставления к ним доступа написанным пользователем системам.

Самое главное в таком дизайне это то как компоненты расположены в памяти, а именно каждый тип компонента лежит в непрерывном массиве с компонентами только этого типа. То есть Вместо того чтобы объединять владельца и его данные как это обычно происходит в ООП, ECS наоборот хранит данные в независимости от владельца. Какие же в итоге это даёт плюсы и минусы?

- Главный плюс это последовательность данных в памяти. В ООП каждый класс лежит в памяти как скажет его пользователь, а следовательно его компоненты тоже и при необходимости создания систем произойдёт сильная потеря производительности из-за пробития кэша. Так как в ECS компоненты лежат в компактных и непрерывных массивах пробитий кэша станет намного меньше.

- Хоть и по эффективности работы с памятью такой вариант лучше, он теряет одну степень независимости. В большинстве реализаций ECS компоненты могут быть перенесены в памяти без ведома пользователя, чего не может быть в ООП. В связи с этим пользователю надо писать компоненты работающие с внезапными переносами в памяти.

- Ещё одним важным плюсом является гибкость. ECS концептуально решает проблему добавления и удаления компонентов без пересборки объекта, так как компоненты не зависят друг от друга, а самое важное от их владельца.

- Так же засчёт того что компоненты лежат не в Entity, а в Storage, получение доступа к компоненту добавляет некоторые overhead, но во-первых он не значительный, а во-вторых сопоставим с виртуальными функциями в ООП.

- Минусом же является необходимость в хранении Storage при использовании Entity, что может быть не удачным вариантом для создания библиотеки, но так как движок это по большей части фреймворк, storage идеально вписывается в его концепцию и хранится в Scene, который так или иначе всегда будет в игре.

5.4 Реализация

Практически люая реализация ECS имеет 3 важных аспекта: генерация handle, тактика создания и удаления компонентов, поиск всех Entity с указанным подмножеством компонентов.

- Handle - Генерация и алгоритм напрямую зависит от времени жизни Entity и его интерфейса. Так как Entity порой существуют только как номер в Storage и используется в системах просто уникальный номер и подсчёт количества созданных копий не сработает. Так же в не зависимости от количество копий Entity порой надо его принудительно уничтожить. Эти условия дают условия на время жизни Entity - от создания первой инстанции в Storage до вызова Destroy. Исходя из этого handle - это уникальный номер и поколение. Для уникальных номеров можно просто создать очередь, при создании Entity доставать номер, при уничтожении класть его обратно, если же номера кончились добавлять в очередь номер равный её размеру. Поколение же нацелено на решение проблемы с уже созданными Entity. Если было две копии одного Entity A и B и на A вызвали Destroy, а через некоторое время создали новый Entity C которому достался тот же номер что был у A. В итоге Entity B будет точной копией C и сможет напрямую его менять. Для этого вводится поколение каждого номера, увеличивающееся на 1 каждый раз как Entity уничтожают. Тогда при попытке B изменить какие-то компоненты C можно будет проверить поколение номера B в Storage и увидеть не совпадение.

- Существуют разные тактики удаления и создания компонетов, так как по сути своей это проблема аллокатора, но так как цель у ECS не абстрактная как у аллокатора, можно просто создать динамический массив с компонентами, добавлять компонент в его конец, а при удалении копмонента откуда-то из середины, просто класть на его место посдений компонент в массиве. Таким образом минимизируется количество ипсользуемой памяти, достигается минимум пробития кэша из-за отсутствия пустых ячеек и все операции происходят за амортизированные O(1)

- Для поиска Entity по выбранному набору компонентов внутри ComponentManager для каждого Entity хранится битовая маска содержащая информацию о его компонентах. Таким образом поиск сводится к битовым операциям и работает быстро. В качестве результата такого поиска можно было просто возвращать массив Entity, но это было бы не эффективно по памяти. Поэтому в качестве такого объекта возвращается StorageView, который содержит в себе только указатель на storage и битовую маску множества ком-

понетов. Когда же надо пройтись по результату, можно просто проитерироватся по этому объекту создавая одного Entity за раз, что в разы эффективнее по памяти чем возвращать массив Entity.

6 Скриптинг

6.1 Мотивация

Для написания игр часто не обязательно писать всю программу с нулям от точки входа, если есть хороший фреймворк. Главное отличие фреймворка от библиотеки это большая инфраструктура в которую пользователь фреймворка встраивает свой код, а вызывает его фреймворк. В контексте движка, пользователю не нужно писать с нуля все системы и взаиможейтсвие в ними. В большинстве случаев вполне достаточно написать несколько небольших кусков кода которые будут взаиможействовать со сценой, entity и компонентами. Для этого собственно и нужны скрипты.

6.2 Требования

Главным требованием является возможность изменять скрипт во время работы редактора, для кратного ускорения процесса разработки, а так же достаточный для понимания состояния сцены интерфейс.

6.3 Реализация

Первой проблемой реализации становится возможность изменять код во время работы программы. C++ - компилируемый язык, то есть для изменения кода программы надо пересобрать и перезапустить. Однако существуют динамические библиотеки, которые можно с помощью системных вызовов подгружать во время работы программы и находить в них указатели на выбранные функции. С помощью этого механизма и реализован скриптинг и его перезагрузка во время работы программы. Как собственно это происходит. Пользователь в отдельном файле пишет свой класс наследующий Scipt. У него есть методы OnCreate/OnDestroy - вызываеются при первом создании объекта скрипта, OnRuntimeStart/OnRuntimeStart - вызывается при начале и конце симуляции сцены в редакторе(кнопки play и stop) и OnUpdate который вызвается каждый кадр обновления сцены. Затем Compiler при поступлении команды build каждый файл в отдельности компилирует в объектный файл, затем линкует все объектные файлы в одну библиотеку. Compiler не тащит за собой в движок огромную зависимость в виде компилятоар C++, а лишь вызывает доступный в системе компилятор и передаёт ему нужные аргументы. Затем ScriptManager создаёт класс SharedObject, отвечающий за непосредственную загрузку динамисческой библиотека полученной от Compiler и поик в ней нужных функций. Затем управление переходит

к ScriptEngine являющимся самым главным среди всех этих компонентов. Он манипулирует тремя типами объектов: ScriptClass, SharedClass и SciptComponent. ScriptClass содержит в себе информацию о конретном скрипте написанном пользователем, а конкретно его название и указатели на полученные из SharedObejct функции для создания и деструкции объектов скриптов. SciptComponent - является handle для обращения к объекту скрипта хранящемуся в ScriptEngine. В данной ситуации handle имеет весомое преимущество над прямым указателем на скрипт по следующей причине: когда поступает команда пересобрать какой-то скрипт надо перезагрузить динамическую библиотека, а следовательно разобрать существующие скрипты и собрать новые, что будет означать необходимость изменять укзатели у произвольного количества пользователей скрипта. С помощью handle модно изменять только указатель внутри ScriptEngine, а пользователь ScriptComponent не будет знать о каких либо манипуляциях с его скрипитом. Ещё одим не мало важной функциональностью скриптов является сохранение состояния между перезагрузками библиотек. Для это есть отдельный макрос FIELD который добавляет поле класса в отслеживаемое множество и при перезагрузки библиотеки ScriptManager сохраняет поля из этого множества и после создания новых скриптов восстанавливает им значения соответсвующих полей.

7 Animations

7.1 Мотивация

Одним из важнейших систем игрового движка, дающих значимай вклад в визальную состовляющую итогового продукта являются анимации моделей. В большинстве популярных движках есть даже множество встроенных инструментов для создания, редакции и проигрывания анимаций. Так как поддерживать инструменты для анимаций весьма трудозатратная задача, в движке будет поддруживатся загрузка анимаций из других программ и их последующее проигрывание.

7.2 Описание

Для понимания работы анимаций необходимо иметь представление как работает отрисовка моделей на экран. Зачастую на GPU отправляются два массива на одну единицу отрисовки mesh: вершинный буфер и буфер индесов. Вершинный буфер содержит множество вершин, сама вершина же может содержать в себе множество аттрибутов, таких как позиция, текстурная коордита, нормаль поверхности и тд. Буфер индексов же содержит в себе последовательность троек индесов вершин, образующие треугольники, которые собственно и отрисовываются на экран. Так же на GPU почти всегда передаюся uniforms - поля шейдера которые можно менять напрямую без буферов. Они зачастую используются для данных которые надо обновлять каждый кадр, например View(матрица трансформации содержащая информацию об ориентации камеры), Projection(матрица проекции камеры, содержит FOV и тд) и Transform(матрица содержащая информацию об ориентации mesh в пространстве). Следовательно чтобы анимировать модель надо либо как-то менять вершинный буфер, либо передавать ещё какие-то данные на через uniform GPU. Очевиднейшим решением кажется просто каждый кадр обновлять позиции вершин в вершинном буфере, но это было бы крайне не эффективно. Иным вариантом было бы для каждой веришны передавать матрицу трансформации через uniform, что то же было бы долго и потребляло бы много памяти. Средним вариантом было бы разделить модель на множество mesh и применять к ним отдельно матрицы трансформации. Это было бы довольно эффективно, но в месте вращения mesh появлялись бы пропуски и тратилось бы намного больше долгих вызовов отрисовки. В итоге пришли к модели скелетной анимации.

7.3 Реализация

В ней к 3D модели добавлялся скелет с костями. Кости выстраиваюстся в иерархию с фиксированным корнем. Каждая кость будет иметь свою матрицу трансформации из координатного пространства модели в локальное для кости координатное пространство. Когда же надо отрисовать модель внешняя трансформация применяется к корню этой иерархии, а затем ппроисходит спуск по каждой ветки иерархии с последующим умножением матриц трансформаций для каждой кости, что даёт итоговое положение статичной позы модели. Для каждой же вершины вводятся 2 новых аттрибута: номера костей и их веса. В виду работы OpenGL аттрибуты могут быть только фиксированными типами и не могут быть массивами. Поэтому в качестве номеров костей берётся 2 четырёхмерных векторов из натуральных чисел, а для весов берётся 2 четырёхмерных вектора чисел с плавающей точкой. Так же в шейдере вводится сравнительно небольшой массив unfiorm типа четырёхмерных матриц в которых будут хранится матрицы трансформации для каждой кости. Теперь надо понять как это всё связанно. Номера костей и веса у каждой вершины обозначают трансформации каких костей влияют на конкретную вершину и с каким весом. Тогда имея эти значения можно к каждой вершине применить итоговое средневзвешенную матрицу трансформации и получить нужную анимированную позицию. Матрицы же эти будем брать из вышеупомянутого массива uniform. Сама же анимация хранится отдельно в информации для каждой кости на CPU в виде ключевых кадров. Ключевые кадры есть 3 типов: позиция, вращение и маштабирование. Если позиция и маштабирование можно задать просто трёхмерным вектором, то с вращением всё немного сложнее. Важным свойством данных кадра является удобство интерполяции, а если представлять вращение с помощью трёхмерного вектора, а именно эйлеровых углов, то надо будет отдельно обрабатывать несколько угловых случеаев(например интерполяция угла 350 к нулю будет длится 350 градусов, а не 10). Эту проблему решают кватернионы и их сверическая интерполяция, из-за чего они часто используется в игровой индустрии. Кадр соответсвенно содержит одно из этих полей и временную отметку этого кадра. Когда настаёт время анимировать модель, удобным интерфейсом будет передавать точку времени, а результатом будет итоговые матрицы трансформации для этой точки времени. По полученной точки времени для каждой кости и каждого типа ключегого кадра находятся два кадра ближайшие к этой точке во времени, который был раньше этой точки и позде соответвенно. Затем берутся значения этих двух кадров и интерполируются с учётом расстояния от двух кадров до точки. Затем перемножаются матрицы трансформации 3-х типов кадров полученные с помощью интерполяции и они являются локальной

трансформацией заданной кости в конретную точку времени. Затем происходит проход от корня иерархии костей и трансформация каждой кости умножается на трансформации всех костей выше её по иерархии, что даёт финальную матрицу трансформации для конретной кости. Эти матрицы как раз и кладутся в массив uniform для последующего использования в шейдере. Таким образом достигается большая производительность и небольшие затраты памяти относительно иных вариантов.

8 PBR

8.1 Мотивация

Довольно долгое время доминирующей моделью рендеринга была модель Phong. Она основывалась на примитивных свойствах света и поверхности. В ней было 3 компоненты света diffuse(преломлённая), specular(отражённая) и ambient(цвет без освещения). Материал задавали 3 текстуры, ambient, diffuse и specular которые содержали информацию о ток как каждая точка объекта отражает каждую компоненту света. Итоговый же цвет для каждой компоненты считался через пару углов и пару цветов (материала и источника света). Модель была популярной потому что давала много контроля пользователю, можно было менять цвет отдельных компонент света как у материала так и у источника. Главным её недостатком была не физичность, а именно то, что при разном освещении одни и те же модели могли выглядеть совершенно по разному. Спустя некоторое появилась новая модель PBR.

8.2 Теория

PBR - physical based rendering, модель которая основана на физических свойствах света. На самом деле она основана на одном законе: энергия приходящая на поверхность материала не меньше энергии ушедшей(отражённой) и поглощённой(преломлённой) материалом. Вся модель строится на уравнение отражения:

$$L_o(p, w_o) = \int_\Omega f_r(p, w_i, w_o) L(p, w_i)(nw_i)\delta w_i \tag{23}$$

где p_o - точка для которой пишется уравнение, $L_o(p, w_o)$ - суммарная интенсивность исходящая из точки в напрпавлении вектора $f_r(p, w_i, w_o)$ - BRDF(двунаправленная функция распределения отражения), $L(p, w_i)$ - интенсивность света приходящего на точку p с направления w_i и n - вектор нормали. Важно сразу заметить, что уже на этом этапе делаются многие допущения. В исходной модели из физики w_o является не вектором, а конусным углом вокруг какого-то вектора, интенсивность цвета считается из длины волны света, а не просто цвета RGB, но для применения в реальном времени прибегают к следующим условностям. BRDF определяется следующим образом:

$$f_r(p, w_i, w_o) = k_d f_{lambert} + k_s * f_{cook-torrance} \tag{24}$$

где k_d - доля преломлённого света, k_s - доля отражённого света, $f_{lambert}$ - функция определяющая цвет преломлённого света и $f_{cook-torrance}$ - функция определяющая цвет отражённого света. Сами k_s и k_d на деле не числа, а цвета и чтобы удовлетворять закону сохранения энергии $k_d = 1 - k_s$ и k_s - считается через уравнение Френеля. Функция $f_{lambert}$ определяется как $\frac{c}{\pi}$, где c - цвет материала в точке p. Функция $f_{cook-torrance}$ определяется как $\frac{NDF*G}{4(nw_i)(nw_o)}$, где в знаменателе коэфициент нормализации, а в числителе две функции: NDF - определяет долю поверхности сонаправленную с её нормалью основываясь на коэффициенте шероховатости и G - определяет коэффициенте затенения поверхности засчёт щероховатости.

8.3 Реализация

Для нахождения цвета каждого пикселя, берётся его позиция, позиция камеры и источников света, их свойства и свойства материалов и подставляются в уравнение отражения. Так же в большинстве моделей используются только 3 типа источников света, точечный(излучает во всех направлениях из точки), точечный-направленный(излучает в некоторых направлениях из точки) и прямой(излучает в одном направлении в любой точке, например солнце). Все эти источники попадают на выбранную точку только под одним углом, а следовательно интегралл можно свести к сумме. В основе материала будут лежать два главных свойства, степень металла и шероховатость. Уже ясно где используется шероховатость, а степень металла учитывается при подсчёте k_d. По законам физики материал может быть либо металлом или нет, а следовательно k_d может быть либо 0, либо $1 - k_s$. Но на практике красивые результаты дают промежуточные значения металла, из-за чего k_d просто домножают на $1 - k_m$, где $k_m = 1$ - материал полностью металический, а $k_m = 0$, материал не металический. Так же в материале есть ещё две важные составляющие: карта нормалей и карта самозатенения. Первая даёт возможность отрисовывая один прямоугольник на GPU задавать каждой его точке отдельное направление нормали, а карта самозатенения даёт возможность искуственно затенять части модели труднодостижимые для света.

9 Постпроцессинг, эффект bloom

9.1 Мотивация

Наиболее трудоемкой частью в реализации эффекта стало его теоретическое изучение. После изучения оставалось реализовать его в виде кода и внедрение в игровой движок. В данной части большое внимание будет посвящено самому эффекту и его математическое

объяснение.

Иногда яркие источники света и ярко освещенные области трудно передать зрителю из-за ограниченного диапазона интенсивности монитора. Один из способов выделения ярких источников света и создания иллюзии сильного освещения - использование эффекта постобработки под названием Bloom. Bloom добавляет эффект свечения на ярко освещенные участки сцены.

Bloom может значительно улучшить визуальное восприятие яркости объектов на сцене. При правильной настройке он может усилить освещение и создать разнообразные эффекты, что является важным элементом визуальной составляющей игр.

Рис. 9.1: эффект bloom

9.2 Реализация

Для реализации эффекта Bloom мы сначала рендерим освещенную сцену и сохраняем изображение в цветовом буфере HDR, а также извлекаем яркие участки сцены. Затем мы размываем извлеченное яркостное изображение и добавляем результат на исходное HDR-изображение сцены.

Итак, первый шаг: извлечение двух изображений из сцены (изображение сцены и изображение только с яркими участками). Для этого мы в основном будем использовать предлагаемые возможности OpenGL. Укажем номер цветовой привязки при привязке текстуры, используя $GL\_COLOR\_ATTACHMENT1$, чтобы иметь два цветовых буфера. При рендеринге будет использоваться соответствующий цветовой буфер. Это позволяет извлечь яркие области из фрагмента.

Второй шаг: блюр изображения с помощью гауссово размытия. Размытие Гаусса основано на гауссовой кривой, которая обычно описывается как кривая нормального распределения. Поскольку гауссова кривая имеет большую площадь вблизи ее центра, использование ее значений в качестве весов для размытия изображения дает более естественные результаты, поскольку образцы, находящиеся рядом, имеют более высокий приоритет.

Для реализации фильтра Гаусса нам нужен двумерный ящик весов, который мы можем получить из уравнения двумерной гауссовой кривой. У уравнения Гаусса есть хорошее свойство, которое позволяет нам разделить двумерное уравнение на два одномерных уравнений: одно, которое описывает горизонтальные веса, и другое, которое описывает вертикальные веса. Сначала мы делаем горизонтальное размытие с горизонтальными весами на текстуре сцены, а затем на полученной текстуре делаем вертикальное размытие.

Для реализации двухпроходного гауссова размытия нам нужно размыть изображение как минимум два раза, используя пинг-понговые фреймбуферы. Для размытия мы создаем два основных фреймбуфера, каждый из которых содержит только текстуру цветового буфера. Затем мы заполняем один из "пинг-понговых"фреймбуферов текстурой яркости, а затем размываем изображение заданное количество раз. При каждой итерации мы привязываем один из двух фреймбуферов в зависимости от того, хотим ли мы размыть изображение по горизонтали или по вертикали, и привязываем цветовой буфер другого фреймбуфера в качестве текстуры для размытия.

Финальный шаг: обьединение сцен. В финальном фрагментном шейдере мы аддитивно смешиваем обе текстуры.

10 Deffered Shading

10.1 Мотивация

Deferred Shading - это метод рендеринга, который сначала собирает информацию о геометрии сцены и затем откладывает вычисление освещения до более позднего этапа. Этот подход противоположен прямому (или forward) рендерингу, где освещение вычисляется в момент прохода каждого полигона.

Стадии Deferred Shading включают следующие шаги:

1 Geometry Pass: На этом этапе сцена рендерится, и информация о геометрии и материалах (например, позиция, нормаль, цвет диффузии, спекулярность и т.д.) сохраняется в буфере кадра, известном как G-Buffer.

2 Lighting Pass: После завершения Geometry Pass, Lighting Pass вычисляет освещение сцены. Для каждого источника света на сцене вычисляется вклад в освещение каждого пикселя, используя информацию, сохраненную в G-Buffer.

Одним из основных преимуществ Deferred Shading является то, что он позволяет обрабатывать большое количество источников света с меньшим влиянием на производительность, чем в методах прямого рендеринга. Это потому, что освещение вычисляется только для пикселей, которые действительно видны в конечном изображении, а не для каждого полигона сцены.

Однако, Deferred Shading также имеет свои недостатки. Он не хорошо работает с полупрозрачными материалами или при выполнении антиалиасинга. Этот подход также требует больше памяти для хранения G-Buffer и может иметь проблемы с производительностью на устройствах с ограниченной пропускной способностью памяти.

10.2 Реализация

1 G-Buffer:

G-Buffer – это набор текстур, каждая из которых хранит определенный тип информации, такой как положение пикселя, его нормаль, альбедо (базовый цвет), спекулярность, и т.д. Для создания G-Buffer создаётся несколько текстур с соответствующими форматами, а затем они объединяются в один Framebuffer Object (FBO) для использования при рендеринге.

2 Geometry Pass:

На этом этапе рендерится каждый объект сцены, записывая информацию о его геометрии и материалах в G-Buffer. Для этого написан шейдер, который извлекает необходимую информацию из каждого объекта и записывает её в соответствующие текстуры G-Buffer.

3 Lighting Pass:

На этом этапе вычисляется освещение сцены, используя информацию, хранящуюся в G-Buffer. Для каждого источника света на сцене вычисляется его влияние на каждый пиксель изображения, снова используя шейдер. Этот шейдер берет в качестве входных данных текстуры G-Buffer и параметры источников света, а затем вычисляет цвет каждого пикселя на основе этих данных.

4 Shading и Post-Processing:

После этого применяются дополнительные эффекты освещения и постобработки к изображению.

11 Портирование на разные ОС

В процессе реализации портирования игрового движка на macOS было выявлено множественное количество проблем на этапах компиляции и функциональности движка.

Первой проблемой стала компиляция скриптов внутри самого движка на macOS. Было добавлены аргументы компилятора, которые указывают на то, что необходимо использовать версию macOS 12.5 при компиляции. В коде также были добавлены функции, которые добавляют аргументы компилятора для директив препроцессора, пути к файлам и библиотекам. Эти функции необходимы при настройке компилятора. Были добавлены дефайны, которые позволили компилятору настроиться на нужные параметры для успешной компиляции проекта на macOS.

Следующей проблемой стала работа с файловыми системами на macOS, в частности открывание диалогового окна для открытия/сохранения сцен. Была найдена и использована кроссплатформенная библиотека nativefiledialog-extended, а так же реализовано открытие диалогового окна для платформы macOS. Написанные методы открывают возможность составить описание файлов и фильтров выбора файла, их открытие и сохранение.

Так же были проведены работы с переписыванием класса SharedObject, отвечающий

за загрузку динамической библиотеки и получение доступа к её функциональности. Была использована встроенная в C библиотека dlfcn.h и знания с курса АКОС.

Иной проблемой стали форматы файлов шейдеров и конфигурации. Для того, чтобы эти файлы могли быть прочитаны и обработаны на macOS я изменил их синтаксис таким образом, чтобы они могли быть прочитаны и компилированы.

Так же был проведен рефакторинг кода, для более удобной работы с движком, были настроены файлы cmake и организована файловая иерархия для портирования на macOS.

Список литературы

[1] Ming-Lun Chou, Lecture on constraint-based physics for the Game Physics Club at DigiPen Institute of Technology(дата обр. 24.05.2023)

[2] Erin Catto, Sequential Impulses — GDC 2006(дата обр. 24.05.2023)

[3] Johnny Huynh, Separating Axis Theorem for Oriented Bounding Boxes(дата обр. 24.05.2023)

[4] Спецификация OpenGL(дата обр. 24.05.2023)

[5] Спецификация OpenAL(дата обр. 24.05.2023)

[6] Спецификация GLFW(дата обр. 24.05.2023)

[7] Спецификация ASSIMP(дата обр. 24.05.2023)

[8] LearnOpenGL(дата обр. 24.05.2023)

[9] Кроссплатформенная библиотека для открытия файлового диалога(дата обр. 24.05.2023)