2.1 Архитектурная схема

Игровой движок построен по модульному принципу, где каждый модуль отвечает за отдельную функциональность. Основные модули движка:

- **Менеджер ресурсов:** Управляет загрузкой, хранением и распределением игровых ресурсов, таких как текстуры, модели, звуки.
- **Подсистема рендеринга:** Отвечает за отображение графики, взаимодействие с графическим API (OpenGL), обработку шейдеров и рендеринг объектов на экран.
- **Система физики:** Обрабатывает столкновения, симуляцию движения объектов, гравитацию и другие физические эффекты.
- **Менеджер сцены:** Сохраняет и управляет игровыми объектами, сценами и их иерархией.
- **Система ввода:** Перехватывает и обрабатывает сигналы от клавиатуры, мыши и других устройств.
- **Аудиосистема:** Обеспечивает воспроизведение звуков и музыкальных треков.
- **Скриптовая подсистема:** Предоставляет возможность динамического взаимодействия с игровым миром через скрипты.
- **Сетевой модуль:** Реализует сетевую синхронизацию и взаимодействие (например, в многопользовательских играх).

Список менеджеров

1. **Менеджер ресурсов:** - Управляет загрузкой, хранением и распределением игровых ресурсов, таких как текстуры, модели, звуки.
2. **Менеджер сцены:** - Сохраняет и управляет игровыми объектами, сценами и их иерархией.
3. Менеджер Инпутов: - Управляет вводом выводом переферии , отвечает за подключение систем
4. Менеджер физики: - Управляет отработкой физических событий
5.

описаны

- Менеджер ресурсов
- Менеджер сцены
- Скриптовая подсистема
- Система ввода
- Система физики
-

надо описать

- Аудиосистема
- Сетевой модуль
- Модуль делегатов
- Интерфейс взаимодействия

2.2 Краткое описание модулей

**2.2.1 Менеджер ресурсов

Описание

Менеджер ресурсов отвечает за эффективное управление игровыми ресурсами, разделяя их на статичные и динамичные данные, оптимизируя доступ и использование памяти. Он также предоставляет интерфейсы для управления объектами и их состояниями.

---

1. Основные задачи:

1. **Загрузка и хранение ресурсов:**
   - На этапе инициализации ресурсы из списка предзагружаются в холодную память (`cold_memory_pool`).
   - Выделенные ресурсы предоставляются через указатели, с увеличением счётчика ссылок.
2. **Разделение ресурсов:**
   - **Холодная память:** Статичные ресурсы, которые не изменяются во время выполнения (например, текстуры, геометрия).
   - **Горячая память:** Динамичные данные, которые временно изменяются (например, координаты объектов, состояния).
3. **Обновление данных:**
   - Изменённые значения временно дублируются в горячую память.
   - Периодически проверяется состояние ресурсов, чтобы вернуть неизменяемые значения из горячей памяти в холодную.
4. **Динамическое управление объектами:**
   - Используется структура `std::map` для хранения объектов и их параметров. Каждый ключ (объект) может иметь несколько связанных значений (параметры).
5. **Удаление объектов:**
   - Ресурсы могут быть помечены флагом удаления.
   - Удаление происходит в следующем игровом цикле, освобождая память.

---

2. Архитектура:

1. **Cold Memory Pool (Холодная память):**
   - Хранилище статичных ресурсов.
   - Использует `std::unordered_map` для быстрого доступа.
2. **Hot Memory Pool (Горячая память):**
   - Хранилище изменяемых данных.
   - Использует `std::unordered_map` для доступа.
3. **Управление объектами:**

- Основные объекты хранятся в `std::vector`.
- Два интерфейса доступа:
  - `std::set` (сортировка по флагу удаления).
  - `std::map` (поиск по имени).

---

3. Интерфейсы:

1. **API работы с ресурсами:**
   - `LoadResource(name, parameters, memory_type)` — Загрузка ресурса в указанную память (холодную или горячую).
   - `GetResource(name)` — Получение указателя на ресурс.
   - `UnloadResource(name)` — Удаление ресурса.
2. **API управления объектами:**
   - `AddObject(name)` — Добавление нового объекта.
   - `GetObjectByName(name)` — Получение объекта по имени.
   - `MarkForDeletion(name)` — Пометка объекта на удаление.
   - `CleanupDeletedObjects()` — Очистка всех объектов, помеченных на удаление.

---

4. Логика работы:

1. На этапе загрузки:
   - Все ресурсы из списка загружаются в холодную память.
   - Для изменяемых данных создаётся копия в горячей памяти при их изменении.
2. На этапе выполнения:
   - Изменения в ресурсах обновляют значения в горячей памяти.
   - Периодически проверяется состояние ресурсов:
     - Если значение больше не изменяется, оно переносится обратно в холодную память.
3. На этапе очистки:
   - Объекты, помеченные флагом удаления, удаляются из всех структур данных.

---

5. Структура данных:

1. **Холодная память:**

```
std::unordered_map<std::string, std::shared_ptr<Resource>>;
```

2. **Горячая память:**

```
std::unordered_map<std::string, std::shared_ptr<Resource>>;
```

3. **Объекты:**

```
std::vector<std::shared_ptr<Object>>;
std::set<std::shared_ptr<Object>, DeletionComparator>;
std::map<std::string, std::shared_ptr<Object>>;
```

2.2.3 Система физики

Описание

Система физики основана на библиотеке `Bullet Physics` и отвечает за симуляцию физических взаимодействий, включая столкновения, гравитацию и динамику объектов. Каждый игровой класс имеет лениво инициализированный базовый класс физики, который создаётся при необходимости. В процессе разработки проекта возможно добавление новых параметров в базовый класс физики. Физические классы взаимодействуют только через API классов, в которых они инкапсулированы, что минимизирует зависимость между модулями.

Список менеджеров

1. **Менеджер ресурсов:** Управляет загрузкой, хранением и распределением игровых ресурсов, таких как текстуры, модели, звуки.
2. **Менеджер сцены:** Сохраняет и управляет игровыми объектами, сценами и их иерархией.
3. **Менеджер инпутов:** Управляет вводом-выводом периферии, отвечает за подключение систем.
4. **Менеджер физики:** Управляет отработкой физических событий и синхронизирует обработку событий с основным потоком выполнения.

---

1. Основные задачи:

1. **Симуляция физических взаимодействий:**
   - Обработка столкновений между объектами.
   - Реализация сил, таких как гравитация, трение и упругость.
2. **Управление физическими объектами:**
   - Создание и удаление физических объектов в сцене.
   - Настройка свойств объектов, таких как масса, форма и скорость.
3. **Интеграция с другими модулями:**
   - Связывание с Менеджером ресурсов для управления геометрией объектов.
   - Обеспечение взаимодействия с Менеджером сцены для визуализации.
4. **Lazy Initialization (Ленивая инициализация):**
   - Базовый класс физики создаётся только при первом обращении.
   - Дополнительные параметры могут быть добавлены в процессе разработки проекта.

5. **Изолированное взаимодействие:**
    - Классы физики обращаются друг к другу только через API, предоставляемые игровыми классами.
6. **Синхронизация с основным потоком:**
    - Физический менеджер обрабатывает события в синхронизации с основным потоком выполнения, чтобы исключить расхождения данных.

---

2. Архитектура:

1. **Основные компоненты:**
    - `btDiscreteDynamicsWorld`: Основной мир симуляции, содержащий все физические объекты.
    - `btCollisionShape`: Определяет форму объектов для обработки столкновений.
    - `btRigidBody`: Представляет физический объект с параметрами массы, инерции и скорости.
2. **Управление объектами:**
    - Все физические объекты хранятся в `std::map` для быстрого доступа по идентификатору.
    - Система использует ленивую инициализацию для создания объектов физики.

---

3. Интерфейсы:

1. **API для работы с физическими объектами:**
    - `AddPhysicsObject(name, shape, mass)` — Добавление нового физического объекта.
    - `RemovePhysicsObject(name)` — Удаление физического объекта.
    - `ApplyForce(name, force)` — Применение силы к объекту.
    - `GetObjectState(name)` — Получение текущего состояния объекта (позиция, скорость).
2. **API для симуляции:**
    - `StepSimulation(delta_time)` — Выполнение шага симуляции с заданным шагом времени.
    - `SyncWithScene()` — Синхронизация состояния объектов с Менеджером сцены.

---

4. Логика работы:

1. **Инициализация:**
    - Создаётся экземпляр `btDiscreteDynamicsWorld` при первом обращении к базовому классу физики.
    - Настраиваются параметры гравитации и среды.
2. **Добавление объектов:**
    - Создаётся `btCollisionShape` и `btRigidBody` для нового объекта.
    - Объект добавляется в `btDiscreteDynamicsWorld` и в локальное хранилище.
3. **Симуляция:**
    - Метод `StepSimulation` вызывается каждый кадр с передачей времени шага.
    - Обновляются состояния всех объектов.
    - События синхронизируются с основным потоком выполнения.
4. **Удаление объектов:**

- Объект удаляется из мира симуляции и локального хранилища.

---

5. Структура данных:

1. **Физические объекты:**

```
std::map<std::string, std::shared_ptr<btRigidBody>> physics_objects;
```

2. **Форма объектов:**

```
std::map<std::string, std::shared_ptr<btCollisionShape>> collision_shapes;
```

3. **Мир симуляции:**

```
std::unique_ptr<btDiscreteDynamicsWorld> dynamics_world;
```

2.2.4 Менеджер сцены

- **Функция:**
  Менеджер сцены управляет игровыми объектами, их состояниями и взаимодействием с подсистемами рендеринга, физики, звука и других модулей. Он является центральной точкой обработки игровых данных и отвечает за согласованность работы системы.

---

1. Основные задачи

1. **Управление объектами:**
   - Хранение объектов текущей сцены и их состояний.
   - Обеспечение доступа к объектам через API.
2. **Оптимизация обновлений:**
   - Использование флага `is_update` для обновления только изменяющихся объектов.
   - Снижение нагрузки на систему за счёт пропуска неизменяемых данных.
3. **Синхронизация с другими модулями:**
   - Передача объектов в подсистемы рендеринга, физики и звука.
   - Поддержание консистентности данных через двойную буферизацию и временные метки.
4. **Очистка объектов:**
   - Управление временем жизни объектов и удаление тех, что помечены флагом `is_deleted`.

5. **Масштабируемость:**
    - Поддержка добавления и удаления объектов во время выполнения.
    - Поддержка нескольких сцен.

---

2. Архитектура

1. **Структура данных:**
    - **Список объектов:** Основное хранилище объектов (`std::vector`).
    - **Индексы для быстрого доступа:**
        - `std::map` для поиска объектов по имени.
        - `std::set` для сортировки объектов по состояниям, например, по флагу `is_deleted`.
2. **Поддержка двойной буферизации:**
    - Два буфера для обновления (`write_buffer`) и доступа (`read_buffer`) к данным.
3. **Интеграция временных меток:**
    - Каждое событие и изменение помечается временной меткой текущего игрового кадра.
4. **Флаги состояния:**
    - `is_visible`: Объект видим в текущем кадре.
    - `is_update`: Объект нуждается в обновлении.
    - `is_deleted`: Объект помечен на удаление.

---

3. Методы менеджера сцены

Добавление объекта

Добавляет объект в сцену и обновляет индексы.

```
void SceneManager::AddObject(std::shared_ptr<Object> L_object) {
std::lock_guard<std::mutex> lock(scene_mutex);
scene_objects.push_back(L_object);
object_map[L_object->name] = L_object;
}
```

Удаление объекта

Помечает объект на удаление.

```
void SceneManager::MarkForDeletion(const std::string& L_name) {
auto obj = GetObjectByName(L_name);
```

```
    if (obj) {
    obj->is_deleted = true;
    }
 }
```

Обновление объектов

Обрабатываются только объекты с `is_update = true`.

```
void SceneManager::UpdateObjects() {
std::lock_guard<std::mutex> lock(scene_mutex);
for (auto& object : scene_objects) {
    if (object->is_update && !object->is_deleted) {
                object->Update();
        }
    }
}
```

Получение видимых объектов

Возвращает только видимые и активные объекты.

```
std::vector<std::shared_ptr<Object>> SceneManager::GetVisibleObjects() {
std::lock_guard<std::mutex> lock(scene_mutex);
std::vector<std::shared_ptr<Object>> visible_objects; for (const auto& object :
scene_objects) { if (object->is_visible && !object->is_deleted) {
visible_objects.push_back(object); } } return visible_objects; }
```

Очистка объектов

Удаляет объекты, помеченные флагом `is_deleted`.

```
void SceneManager::CleanupDeletedObjects() {
    std::lock_guard<std::mutex> lock(scene_mutex);
    scene_objects.erase(
        std::remove_if(scene_objects.begin(), scene_objects.end(),
                       [](const std::shared_ptr<Object>& obj) { return obj-
>is_deleted; }),
        scene_objects.end());
}
```

Управление флагом обновления

Позволяет включать или отключать обновление для конкретного объекта.

```
void SceneManager::SetObjectUpdate(const std::string& L_name, bool L_is_update) {
    auto obj = GetObjectByName(L_name);
    if (obj) {
        obj->is_update = L_is_update;
    }
}
```

---

4. Интеграция с GameLoop

1. **Однопоточная версия:**
   - Все методы (`UpdateObjects`, `GetVisibleObjects`, `CleanupDeletedObjects`) вызываются последовательно в одном потоке.

```
void GameLoop::Run() {
    while (is_running) {
        scene_manager.UpdateObjects();
        auto visible_objects = scene_manager.GetVisibleObjects();
        render_manager.Render(visible_objects);
        scene_manager.CleanupDeletedObjects();
    }
}
```

2. **Многопоточная версия:**
   - Каждый менеджер работает в отдельном потоке.
   - Барьер синхронизации обеспечивает согласованность выполнения.

2.2.5 Система ввода

- **Функция:**
  Обеспечивает связь между игроком и игровым миром через обработку событий ввода с использованием библиотеки SFML.

---

1. Основные задачи:

1. **Обработка устройств ввода:**
   - Клавиатура (нажатие, удержание, отпускание).
   - Мышь (движение, клики, прокрутка колеса).
   - Геймпад (подключение, отслеживание осей и кнопок).

2. **Поддержка событий:**

- События на уровне окон (закрытие окна, изменение размера).
- Ввод текстовых данных (например, для чата или интерфейсов).

3. **Динамическая привязка действий:**
   - Возможность сопоставления пользовательских действий с определёнными комбинациями ввода.
   - Реализация системы горячих клавиш (hotkeys).
4. **Интеграция с игровой логикой:**
   - Передача событий в менеджер сцены или другие модули.

---

2. Архитектура:

1. **Событийный цикл:**
   - Используется метод `pollEvent` для получения событий SFML.
   - Обработка событий в игровом цикле `GameLoop`.
2. **Система привязок (bindings):**
   - Структура данных для хранения соответствия действий (например, "стрельба", "перезарядка") комбинациям клавиш/кнопок.
   - Возможность переназначения действий через интерфейс.
3. **Механизм обратных вызовов (callback):**
   - События ввода вызывают заранее определённые функции-обработчики.

---

3. Интерфейсы API:

- Регистрация обработчиков для событий.
- Получение текущего состояния клавиши, мыши, или других устройств ввода.
- Обработка событий и их преобразование в действия.

---

4. Логика работы:

1. **Обработка событий SFML:**
   События обрабатываются в каждом игровом цикле с помощью метода `pollEvent`.
2. **Вызов действий:**
   После обработки события вызываются соответствующие функции-обработчики, зарегистрированные в системе привязок.
3. **Передача событий в другие модули:**
   Например, движение камеры или взаимодействие с объектами передаётся в менеджер сцены.

2.2.6 Аудиосистема

- **Функция:** Воспроизведение и управление звуковыми эффектами и музыкой.
- **Поддержка форматов:**
  - WAV, MP3.
- **Особенности:**
  - Пространственный звук (3D-звук).
  - Микширование нескольких треков.

2.2.7 Скриптовая подсистема

- **Функция:** Позволяет разработчикам и пользователям создавать логику игрового мира с помощью встроенного функционала.
- **Подходы к реализации:**
  - **C++ API:** Предоставляет интерфейсы для прямого программирования на C++.
  - **Нодовая система:** Удобный графический инструмент для создания игровой логики, схожий с Blender3D.
    - Узлы представляют собой функциональные блоки (например, "Событие", "Движение объекта", "Проигрывание анимации").
    - Логика строится через соединение узлов (потоки данных или событий).
- **Особенности:**
  - Возможность переключения между нодовым интерфейсом и кодированием на C++.
  - Расширяемость: пользователи могут добавлять свои узлы или методы.
  - Визуализация состояния во время выполнения (в нодовой системе).

2.2.8 Сетевой модуль

- **Функция:** Обеспечивает сетевую синхронизацию и взаимодействие между клиентами и сервером.
- **Поддержка:**
  - Основы TCP/UDP.
  - Позднее добавление синхронизации объектов в реальном времени.

2.2.9 Модуль делегатов

Описание :

Модуль делегатов предоставляет механизм сигналов и ответных сигналов для взаимодействия между объектами. Делегаты позволяют отправлять сигналы от одного объекта к другому (слушателю делегата) с возможностью получения ответного сигнала, передачи значений и указателей на функции. Обеспечивается удобный интерфейс для подключения и отключения слушателей.

---

1. Основные задачи:

1. **Отправка сигналов:**
   - Отправка сигнала от объекта-инициатора к одному или нескольким слушателям.
2. **Получение ответных сигналов:**
   - Возможность слушателей отправлять ответные сигналы обратно инициатору.
3. **Передача данных:**

- Поддержка передачи значений и указателей на функции в рамках сигнала.

4. **Управление слушателями:**
   - Удобный интерфейс для добавления и удаления слушателей.
5. **Инкапсуляция взаимодействий:**
   - Все взаимодействия через делегаты строго изолированы через их API.

---

2. Архитектура:

1. **Основные компоненты:**
   - `Delegate`: Базовый класс, обеспечивающий отправку и обработку сигналов.
   - `Listener`: Объект-слушатель, подключённый к делегату.
   - `Signal`: Объект, представляющий сигнал с параметрами.
2. **Связь между компонентами:**
   - `Delegate` управляет списком `Listener`.
   - Каждый `Listener` подписывается на определённые события.
   - `Signal` передаётся от делегата к слушателям.

---

3. Интерфейсы:

1. **API для работы с делегатами:**
   - `AddListener(listener)` — Добавление слушателя.
   - `RemoveListener(listener)` — Удаление слушателя.
   - `Broadcast(signal)` — Отправка сигнала всем подключённым слушателям.
2. **API для слушателей:**
   - `OnSignalReceived(signal)` — Обработка полученного сигнала.

---

4. Логика работы:

1. **Инициализация:**
   - Создаётся экземпляр `Delegate`.
   - Подключаются необходимые `Listener` через метод `AddListener`.
2. **Отправка сигнала:**
   - Объект-инициатор вызывает метод `Broadcast`, передавая `Signal`.
   - Делегат передаёт сигнал всем слушателям.
3. **Обработка сигнала:**
   - Каждый слушатель обрабатывает полученный сигнал в методе `OnSignalReceived`.
4. **Отключение слушателя:**
   - Слушатель удаляется из делегата через метод `RemoveListener`.

---

5. Структура данных:

1. **Делегаты:**

```
class Delegate {
private:
    std::vector<std::shared_ptr<Listener>> listeners; // Список слушателей

public:
    void AddListener(const std::shared_ptr<Listener>& listener);
    void RemoveListener(const std::shared_ptr<Listener>& listener);
    void Broadcast(const Signal& signal);
};
```

2. **Слушатели:**

```
class Listener {
public:
    virtual void OnSignalReceived(const Signal& signal) = 0; // Обработка сигнала
};
```

3. **Сигналы:**

```
class Signal {
public:
    std::string name; // Название сигнала
    std::any data;    // Передаваемые данные

    Signal(const std::string& name, const std::any& data) : name(name), data(data)
{}
};
```

2.2.10 Интерфейс взаимодействия

- **Функция:** Обеспечивает унифицированный способ предоставления свойств или методов объектов для внешнего взаимодействия.
- **Особенности:**
    - Контракт взаимодействия: классы, реализующие интерфейс, предоставляют заранее определенные свойства/методы.
    - Возможность изменения значений в реальном времени.
    - Поддержка расширяемости для добавления новых свойств.