Объектно-ориентированное программирование (ООП)

Описание:

ООП используется как основная парадигма построения движка. Она позволяет представить подсистемы (сцены, сущности, менеджеры) в виде изолированных объектов с чёткой иерархией и поведением.

Применение в движке:

Абстракция иерархии: Scene, Entity, UIButton, AssetManager.

Расширение через наследование: например, конкретные реализации Scene.

Инкапсуляция поведения и данных в объектах.

Полиморфизм — для обработки различных сцен или сущностей через общие интерфейсы.

Компонентный подход (Entity-Component Architecture, ECS-light)

Описание:

Компонентная модель позволяет отделить данные от логики поведения и повторно использовать их в разных контекстах. Это особенно удобно при создании сложных игровых объектов.

Применение в движке:

Добавление к сущностям логики через подключаемые компоненты (HealthComponent, MovementComponent, RenderComponent).

Позволяет собирать объекты "по частям", а не через глубокую иерархию.

Компоненты хранят состояние и могут обновляться независимо.

Событийно-ориентированная модель

Описание:

Модель, при которой действия инициируются событиями (нажатие клавиши, столкновение, клик по UI). Это улучшает разделение обязанностей и уменьшает связанность кода.

Применение в движке:

Система событий (EventManager) для связи между сценой, UI и игровыми объектами.

Поддержка пользовательских событий (PLAYER_HIT, BUTTON_CLICKED) с подпиской и реакцией.

Упрощение коммуникации между модулями.

Модульность и внедрение зависимостей (Dependency Injection)

Описание:

Позволяет сделать архитектуру гибкой и расширяемой за счёт явной передачи зависимостей между модулями, а не жёсткого связывания компонентов напрямую.

Применение в движке:

Подсистемы (ввода, аудио, сцены) можно заменить через DI.

Инициализация движка с пользовательским конфигом/контейнером модулей.

Упрощает тестирование и конфигурируемость.

Функциональные элементы (вспомогательно)

Описание:

Хотя Python — не функциональный язык, элементы ФП (чистые функции, map, filter, lambda) повышают читаемость и удобство некоторых операций.

Применение в движке:

Функции обработки массивов сущностей (filter_alive_entities, map_to_rects).

Генерация временных данных и вспомогательных структур.

Обработка конфигураций, данных, шаблонов.

Вывод

Комбинирование нескольких парадигм позволяет построить архитектуру, которая будет:

гибкой (за счёт компонент и событий),

расширяемой (через внедрение зависимостей и модули),

наследуемой и инкапсулированной (через ООП),

тестируемой и **поддерживаемой** на разных этапах развития проекта.

Такая модель особенно актуальна для соло-разработки, где важно заранее заложить прочную основу для роста и развития движка без необходимости его полной переделки.

Распределение парадигм по модулям проекта

Модуль	Используемые парадигмы	Обоснование применения
`core/game.py`	ООП, событийная модель	Объект `Game` инкапсулирует главный цикл и события
`core/scene.py`	ООП, шаблон `State`	`Scene` — абстрактный базовый класс для логики игры
`core/scene_manager.py`	ООП, событийная модель, DI	Управление сценами, динамическая замена
`input/input_manager.py`	ООП, событийная модель	Централизованная обработка пользовательского ввода
`entities/entity.py`	Компонентная модель, ООП	Базовая сущность с возможностью добавления компонентов
`entities/components/`	Компонентная модель	Поведение реализуется через подключаемые модули
`ui/`	ООП, событийная модель	Виджеты как объекты, реагирующие на события
`audio/audio_manager.py`	ООП	Инкапсулированное управление звуками и музыкой
`resources/asset_manager.py`	ООП, DI, функциональные элементы (`map`, кеширование)	Кеширование и централизованная загрузка ресурсов

Модуль	Используемые парадигмы	Обоснование применения
`events/event_manager.py`	Событийная модель, ООП	Подписка, публикация и обработка пользовательских событий
`utils/helpers.py`	Функциональные элементы	Утилиты для фильтрации, генерации и форматирования
`config/`	DI, функциональный подход (чтение и хранение параметров)	Параметры и настройки в виде словарей/объектов