Обоснование архитектурных решений

Многоуровневая модульная архитектура

Причина выбора:

Позволяет чётко разделить ответственность между слоями (логика, API, низкоуровневые сервисы).

Упрощает навигацию по проекту, поддержку и внедрение изменений.

Обеспечивает изоляцию между пользовательским кодом игры и внутренними подсистемами.

Результат:

Проект легко масштабируется, каждая подсистема (аудио, UI, сцены, сущности) может развиваться независимо.

Использование ООП как базовой парадигмы

Причина выбора:

Python предоставляет удобные средства для реализации ООП.

Иерархии, инкапсуляция и полиморфизм позволяют строить логично организованный код.

Упрощает реализацию базовых и абстрактных классов, таких как Scene, Entity, UIButton.

Результат:

Объекты могут наследоваться, переопределяться и расширяться, не затрагивая остальной код.

Компонентный подход в системе сущностей

Причина выбора:

Позволяет гибко настраивать поведение объектов без глубокой иерархии.

Реализует композицию вместо наследования.

Обеспечивает повторное использование и независимую разработку логики.

Результат:

Поведение объекта можно изменить «на лету» путём добавления или удаления компонентов.

Внедрение архитектурных паттернов

Причина выбора:

Каждый паттерн решает конкретную задачу: управление состоянием (State), расширяемость (Component), уведомление (Observer), контроль доступа (Singleton).

Избавляют от дублирования логики и повышают читаемость архитектуры.

Результат:

Система легко расширяется, поддерживает масштабирование и написание повторно используемых модулей.

Централизованное управление ресурсами и событиями

Причина выбора:

Ресурсы (изображения, звуки) и события (ввод, коллизии) являются глобальными для всей игры.

Централизованные менеджеры позволяют избежать повторной загрузки и упростить логику.

Результат:

Высокая производительность, минимальные издержки на IO и понятная точка доступа к общим данным.

Использование pygame как нижнего уровня

Причина выбора:

pygame предоставляет достаточную абстракцию для рендеринга, аудио и ввода.

Не требует дополнительных зависимостей, хорошо документирован, стабилен.

Легко интегрируется с Python-экосистемой.

Результат:

Снижен порог входа, можно сосредоточиться на логике движка, а не на низкоуровневой реализации.

Поддержка DI и расширяемости

Причина выбора:

Важна гибкость — например, возможность заменить AudioManager или InputManager на свои реализации.

DI и регистрация сервисов упрощают тестирование и внедрение новых модулей без модификации ядра.