Урок 3: От простого к правильному — как мы улучшили архитектуру игры

Цель урока:
☞ Понять, зачем нужно разделять ответственность в коде
☞ Увидеть, как мы постепенно улучшали структуру игры
☞ Научиться использовать класс `AssetLoader` для управления ресурсами

Было: Всё в одном файле и в одном классе

На первом этапе у нас был простой код:

```
class Player:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.width = 50
        self.height = 50
        self.rect = pygame.Rect(x, y, self.width, self.height)
        self.sprite = pygame.Surface((50, 50))
        pygame.draw.rect(self.sprite, BLUE, (0, 0, 50, 50))  # Рисуем внутри __init__
```

Проблема: Класс Player делал слишком много — и двигался, и рисовал себя, и "знал", как выглядит.

Шаг 1: Вынесли загрузку спрайтов в отдельную функцию

Мы улучшили код, создав функцию:

```
def load_player_sprite():
    try:
        return pygame.image.load("assets/player.png")
    except:
        # создаём заглушку
```

И использовали её в Player.**init**().

Плюс: Код стал чище — логика загрузки вынесена

Минус: Функция — глобальная, не масштабируется Если появятся враги, нужно будет писать load_enemy_sprite() и т.д.

Шаг 2: Перенесли загрузку внутрь класса Player

```
class Player:
    def __init__(self, x, y):
        self.sprites = self.load_sprites()  # ← вызов метода

    def load_sprites(self):
        # загрузка спрайтов idle, walk, jump
```

Плюс: Всё, что нужно игроку — внутри него

Минус: Теперь Player отвечает и за поведение, и за внешний вид Это нарушает принцип единственной ответственности

Теперь: Создали отдельный класс AssetLoader

Новый подход:

```
class AssetLoader:
    def __init__(self):
        self.sprites = {}
        self.load_all()

    def load_image(self, path, w, h):
        # загружает или создаёт заглушку
        return pygame.transform.scale(image, (w, h))

    def load_all(self):
        self.sprites["player_idle"] = self.load_image("assets/player_idle.png", 50, 50)
        self.sprites["player_walk"] = self.load_image("assets/player_walk.png", 50, 50)
        self.sprites["player_jump"] = self.load_image("assets/player_jump.png", 50, 50)

    def get(self, name):
        return self.sprites.get(name)
```

Теперь Player получает загрузчик в конструкторе:

```
class Player:
    def __init__(self, x, y, assets):  # ← получает загрузчик
        self.assets = assets
        # ...
        self.current_sprite = self.assets.get("player_idle")
```

Почему это важно?

1. Разделение ответственностей (Single Responsibility) Player — отвечает за движение, прыжок, анимации AssetLoader — отвечает за загрузку и хранение ресурсов

2. Масштабируемость python enemy = Enemy(x=500, y=400, assets=asset_loader) # тот же загрузчик! coin = Coin(x=300, y=300, assets=asset_loader)
3. Поддерживаемость Если нужно поменять формат изображений — правим только AssetLoader Если файл не найден — заглушка работает везде 🛠 Как использовать AssetLoader в будущем? Можно расширить его:

python class AssetLoader: def load_sound(self, path): try: return pygame.mixer.Sound(path) except: return None

```
def load_font(self, path, size):
    try:
        return pygame.font.Font(path, size)
    except:
        return pygame.font.SysFont("arial", size)
```

Теперь можно:

python self.jump_sound = assets.load_sound("sounds/jump.wav") self.font = assets.load_font("fonts/game.ttf", 24) Вывод Мы прошли путь от простого к профессиональному подходу:

Прямоугольник → Спрайт в классе → Отдельный загрузчик ресурсов Теперь код:

Чистый Модульный Готов к расширению Что дальше? Добавить класс Platform с коллизиями Сделать менеджер уровней Внедрить систему анимаций Совет: Попробуй создать класс Enemy, который тоже использует asset_loader. Убедись, что всё работает без дублирования кода!

Готов к следующему уроку? Скажи: «Дальше!» — и мы добавим платформы 🎮

Этот урок:

- Показывает **эволюцию мышления** программиста
- Объясняет **архитектурные принципы** простым языком
- Готов к использованию в обучении, курсах, самообразовании