ФГАОУ ВО «Национальный исследовательский технологический университет «МИСИС»

КУРСОВАЯ РАБОТА

по дисциплине ООП

Разработка игры "Arkanoid"на C++ с использованием библиотеки SFML

Выполнил:
Шарифов Хабиб Аминжонович
Группа: БПМ-24-4

2025

Содержание

Введение

Арканоид — это культовая аркадная игра, в которой игрок управляет платформой, отбивая мяч и разрушая ряды блоков. В рамках данного проекта реализована версия Arkanoid на языке программирования C++ с использованием библиотеки SFML (Simple and Fast Multimedia Library). Такой выбор технологий позволяет создать кроссплатформенное приложение с графическим интерфейсом, поддержкой звуковых эффектов и реалистичной физикой столкновений.

Ключевые особенности реализации:

- **Модульность** — код организован в независимые компоненты
- **ООП** — инкапсуляция, наследование, полиморфизм
- **Паттерны проектирования** — Singleton, Factory, State, Observer
- **SFML** — графика, звук, обработка ввода

Цель проекта

Целью проекта является разработка функциональной, расширяемой и удобной для пользователя игры "Arkanoid"с применением ООП и освоением инструментов библиотеки SFML для создания мультимедийных приложений.

Задачи проекта

Для достижения поставленной цели были сформулированы следующие задачи:

1. **Реализовать основную игровую механику:**
   Движение платформы, отскок мяча, разрушение блоков различного типа.
2. **Организовать систему подсчёта очков и жизней:**
   Вести учёт прогресса игрока и отображать его на экране.
3. **Добавить звуковые эффекты и визуальную обратную связь:**
   Реализовать воспроизведение звуков при столкновениях, разрушении блоков, сборе бонусов и других событиях.
4. **Обеспечить реалистичную и отзывчивую физику:**
   Внедрить продвинутую систему коллизий с отдельными солверами для разных типов столкновений (мяч-блок, мяч-платформа, мяч-стена).
5. **Создать менеджеры ресурсов:**
   Организовать удобную загрузку, хранение и кэширование текстур, шрифтов, звуков и других ресурсов.
6. **Реализовать систему уровней:**
   Поддерживать загрузку и парсинг уровней из JSON-файлов, а также продвижение по уровням с возрастающей сложностью.

7. **Добавить систему бонусов:**
   Реализовать выпадение и применение различных бонусов при разрушении блоков, с соответствующими анимациями и изменением логики/физики игры.

8. **Организовать различные состояния игры:**
   Главное меню, меню выбора уровня, настройки и т.д, а также реализовать плавное переключение между этими состояниями.

Архитектура проекта: модульность и ООП

Чёткая модульность и ответственность

Весь проект разбит на независимые модули, каждый из которых отвечает только за свою область:

Модуль	Роль и ответственность
`core/`	Инициализация движка (Главный игровой цикл). Переключение состояний
`entities/`	Игровые объекты (мяч, платформа, блоки, бонусы). Инкапсуляция поведения и свойств
`game_states/`	Игровые состояния (меню, игра, пауза и т.д.). Изоляция логики режимов и упрощенное переключение между сценами
`managers/`	Менеджеры ресурсов, уровней, бонусов, звука
`systems/`	Физика и обработка коллизий. Каждый solver за свой тип столкновения. Отрисовка графики игрового процесса

Таблица 1: Модульная структура проекта

ООП: абстракции, наследование, полиморфизм

Базовые классы:

- **Entity** — общий интерфейс для всех игровых объектов (мяч, платформа, блоки, бонусы)
- **BaseBlock** — абстракция для всех видов блоков
- **State** — интерфейс для игровых состояний
- **PowerUpEffect** — интерфейс для эффектов бонусов

Наследование:

Все игровые объекты наследуют от Entity, блоки — от BaseBlock. Это позволяет хранить их в одном контейнере и обрабатывать через общий интерфейс.

Полиморфизм:

Виртуальные методы обеспечивают корректное поведение для каждого типа объекта. Например, обработка коллизий не зависит от конкретного типа блока — всё работает через базовый интерфейс.

Инкапсуляция:

Вся логика и данные скрыты внутри классов, внешний код работает только через публичные методы.

Преимущества архитектуры

Ключевые преимущества:

- **Масштабируемость:** Добавить новый тип бонуса или состояния — просто реализовать новый класс
- **Переиспользуемость:** Менеджеры ресурсов, уровней, звука легко использовать в других проектах
- **Гибкость:** Легко расширять игру новыми механиками, не ломая существующий код

Ключевые классы и их назначения

Класс/Модуль	**Назначение**
`GameEngine`	Главный цикл игры, инициализация, управление состояниями, точка входа
`State`, `StateMachine`	Паттерн State: экраны (игра, меню, пауза, победа, поражение), Машина переходов между состояниями
`GameState`	Главное игровое состояние. Координирует весь игровой процесс: физику, коллизии, управление уровнями, бонусы, счет и жизни.
`LevelManager`	Загрузка, парсинг и генерация уровней из JSON
`Entity`, `MovableEntity`	Базовые классы для всех игровых сущностей
`Ball`, `Paddle`	Основные игровые объекты: мяч и платформа
`Block`, `Rock`	Разрушаемые и неразрушаемые блоки
`PowerUp`, `PowerUpManager`, `PowerUpFactory`	Система бонусов: создание, спавн, применение
`PowerUpEffect`	Эффекты бонусов как отдельные классы
`PhysicsSystem`	Физика игры: движение, коллизии
`ICollisionObserver`	Паттерн Observer: интерфейс наблюдателя
`BlockCollisionSolver`	Солвер: столкновения мяча с блоками
`PaddleCollisionSolver`	Солвер: столкновения мяча с платформой
`WallCollisionSolver`	Солвер: столкновения мяча со стенами
`AssetManager`	Менеджер ресурсов (Singleton): текстуры, шрифты, звуки, изображения
`SoundManager`	Менеджер звука и музыки (Singleton): эффекты, громкость
`RenderSystem`	Вся отрисовка игрового процесса: игровые объекты, UI, оверлеи
`MainMenuState`, `LevelSelectState`, `OptionsState`	Состояния меню: главное, выбор уровня, настройки

Таблица 2: Основные классы системы

Паттерны проектирования, используемые в проекте

State Pattern (Состояние)

Где используется

Управление состояниями игры: `GameState`, `MainMenuState`, `GameOverState`, `OptionsState` и др.
Реализовано через базовый класс `State` и менеджер состояний `StateMachine`.

Реализация

- **State** — абстрактный базовый класс для всех игровых состояний
- **Конкретные состояния** — наследуют `State`, реализуют свою логику (игра, меню, настройки конец игры и т.д.)
- **StateMachine** — управляет текущим состоянием, хранит стек состояний, делегирует обновление и обработку событий

Переключение состояний

Переходы через методы `StateMachine`:

- `pushState()` — добавить новое состояние поверх текущего
- `changeState()` — заменить текущее состояние
- `popState()` — вернуться к предыдущему состоянию

При переходах вызываются методы жизненного цикла состояний: `enter()`, `exit()`, `pause()`, `resume()`.
В основном цикле игры все действия делегируются текущему состоянию через `StateMachine`.

Зачем нужен

- Изоляция логики каждого режима
- Лёгкое добавление новых состояний
- Централизованное и прозрачное управление переходами
- Гибкость и расширяемость архитектуры

Observer Pattern (Наблюдатель)

Где используется

Система физики регистрирует наблюдателей и уведомляет их о коллизиях между игровыми объектами.
Примеры наблюдателей: `GameState` (реализует интерфейс `ICollisionObserver`).

Реализация

- **ICollisionObserver** — абстрактный интерфейс наблюдателя с методом обработки коллизии `onCollision(CollisionType type, Entity* obj1, Entity* obj2)`
- **PhysicsSystem** — хранит список наблюдателей (`std::vector<ICollisionObserver*> observers`)
- При обнаружении коллизии с помощью метода `notifyObservers()` вызывает у всех наблюдателей метод `onCollision(...)` с деталями события
- **GameState** — подписывается на события физики и реализует свою логику реакции на коллизии (начисление очков, проигрывание звуков с помощью `SoundManager`, спавн бонусов с помощью `PowerUpManager`)

Зачем нужен

- **Отделяет обработку последствий столкновений от самой физики**
- **Ослабление связей (Loose Coupling):** `PhysicsSystem` не знает, что делать при столкновении — она просто сообщает о факте коллизии всем подписчикам
- **Гибкая расширяемость:** Можно добавить новых наблюдателей без изменения существующего кода
- **Упрощение архитектуры:** Физика отвечает только за вычисления, все игровые последствия обрабатываются отдельно

Factory Pattern (через function map)

Где используется

Создание бонусов (`PowerUp`) в `PowerUpManager`. Вся генерация бонусов централизована через класс `PowerUpFactory`.

Реализация

- **PowerUpFactory** — отдельный класс, отвечающий за создание бонусов по их типу (`PowerUpType`)
- Внутри фабрики используется неупорядоченный ассоциативный контейнер между типом бонуса и функцией его создания:

```
1 std::unordered_map<PowerUpType, Creator> creators
```

где `Creator` — это `std::function<std::unique_ptr<PowerUp>(float, float)>`

- Для каждого типа бонуса в конструкторе фабрики регистрируется функция-"создатель":

```
creators[PowerUpType::ShrinkPaddle] = [](float x, float y) {
    return std::make_unique<ShrinkPaddlePowerUp>(x, y);
};
```

- `PowerUpManager` работает только с интерфейсом фабрики, не зная о конкретных классах бонусов

Зачем нужен такой подход

- **Простое добавление новых бонусов:** Достаточно реализовать новый класс бонуса и зарегистрировать его в фабрике
- **Централизует логику создания объектов:** Все бонусы создаются в одном месте
- **Ослабление связей:** `PowerUpManager` не зависит от конкретных классов бонусов
- **Принцип OCP (Open/Closed Principle):** Для расширения не нужно менять существующий код

Singleton Pattern (Одиночка)

Где используется

- **AssetManager** — централизованное управление ресурсами (текстуры, шрифты, изображения)
- **SoundManager** — управление звуками и музыкой во всей игре

Реализация

Статический метод `getInstance()`:

```
AssetManager& AssetManager::getInstance() {
    static AssetManager instance; //

    return instance;
}
```

Защита от копирования:

```
AssetManager() = default;
AssetManager(const AssetManager&) = delete;
AssetManager& operator=(const AssetManager&) = delete;
```

Зачем нужен

- **Централизация управления:**
  - Все ресурсы и звуки контролируются в едином месте
  - Единая точка доступа ко всем медиа-ресурсам игры
- **Экономия памяти и защита от повторных загрузок:**
  - Каждый файл (текстура, звук, шрифт) загружается в память единожды через `AssetManager`
  - При повторных обращениях возвращается уже загруженный ресурс из кэша
  - Исключает дублирование ресурсов в памяти
- **Удобство доступа:**
  - Доступ к методам через `getInstance()` в любой точке кода
  - Не требуется передавать указатели или ссылки между объектами
- **Безопасность:**
  - Запрет копирования исключает случайные дубликаты экземпляров
  - Гарантия существования только одного экземпляра каждого менеджера
- **Оптимизация:**
  - `AssetManager` кэширует ресурсы, ускоряя последующие запросы
  - `SoundManager` использует пул звуков, снижая нагрузку на CPU
  - Ленивая инициализация — создание только при первом обращении
- **Расширяемость:**
  - Легко добавить новые ресурсы или типы звуков без изменения архитектуры
  - Централизованное место для модификации логики загрузки ресурсов

Особенности AssetManager

- **Универсальная загрузка ресурсов:** Шаблонный метод `getResource` централизует загрузку всех типов ресурсов
- **Автоматическое кэширование:** Ресурсы хранятся в `std::unordered_map` и загружаются единожды
- **Оптимизированная предзагрузка:** Методы `preloadTextures()`, `preloadFonts()`, `preloadSounds()`
- **Централизованная очистка:** `clear()` освобождает все ресурсы одной командой

Особенности SoundManager

- **Пул звуков для оптимизации:** SoundManager использует пул из `MAX_CONCURRENT_SOUNDS` проигрывателей для эффективного управления звуками. Каждый проигрыватель может воспроизводить только один звук за раз — новый звук прерывает предыдущий на том же проигрывателе. Система автоматически переиспользует свободные проигрыватели, а при перегрузке (когда все заняты) заменяет самый старый звук (прерывая его). Это позволяет одновременно воспроизводить до `MAX_CONCURRENT_SOUNDS` звуковых эффектов. Такой подход экономит ресурсы, предотвращает утечки памяти и обеспечивает стабильную работу даже при активном звуковом сопровождении.
- **Динамическое переиспользование:** Метод `getAvailableSound()` находит первый незанятый звук или переиспользует старый, предотвращая утечки памяти.
- **Гибкое управление аудио:** Независимые настройки громкости для звуков (`soundVolume`) и музыки (`musicVolume`). Глобальный mute через `setMuted(true)` мгновенно отключает все звуки и музыку, включая остановку активных эффектов.

Система физики и коллизий

Основные компоненты:

Компонент	**Роль**
`PhysicsSystem`	Центральный координатор физики. Обновляет позиции объектов, проверяет коллизии и уведомляет наблюдателей
`ICollisionObserver`	Интерфейс наблюдателя коллизий (паттерн Observer)
`BlockCollisionSolver`	Точный расчёт столкновений мяча с блоками (нормали, коррекция позиции)
`PaddleCollisionSolver`	Отражение мяча от платформы с учётом точки удара и её скорости
`WallCollisionSolver`	Обработка отскоков от стен и границ экрана

Таблица 3: Компоненты системы физики

PhysicsSystem — ядро физики

Функционал:

- Координирует работу всех солверов
- Уведомляет подписчиков (например, `GameState`) о коллизиях через `ICollisionObserver`

Ключевые методы:

- `void update(Ball& ball, Paddle& paddle, Blocks& blocks, PowerUps& powerups, float deltaTime)`
- `void notifyObservers(CollisionType type, Entity* obj1, Entity* obj2)`

Солверы коллизий:

BlockCollisionSolver

Особенности:

- Использует Sweep-тест (алгоритм, предсказывающий столкновение на основе траектории движения) для точного определения момента столкновения, что предотвращает "проскок"мяча через блок на высокой скорости.
- Рассчитывает нормаль отскока в зависимости от стороны удара (верх/низ/бок)
- Корректирует позицию мяча, чтобы избежать "залипания"в блоке

PaddleCollisionSolver

Особенности:

- Угол отскока зависит от точки удара: чем ближе к краю платформы, тем сильнее отклонение
- Учитывает скорость платформы (`applyPaddleInfluence`)
- Коррекция позиции: исправляет позицию мяча после столкновения
- Прогрессия сложности: ускоряет мяч после удара на фиксированное значение (`SPEED_INCREAS` превышая верхний лимит скорости)

WallCollisionSolver

Особенности:

- Граничные условия: обрабатывает столкновения мяча со стенами (левой, правой и верхней границами мира)
- Нижняя граница не проверяется, так как её пересечение означает потерю мяча
- Рассчитывает нормаль отскока и отражает мяч через `ball.reflect(normal)`

Система уровней

LevelManager: загрузка и управление уровнями

Класс `LevelManager` реализует:

- Загрузку уровней из JSON-файлов
- Парсинг данных уровня (расположение блоков, их типы, параметры макета)
- Генерацию игровых объектов (блоков) на основе загруженных данных
- Проверку завершения уровня (все разрушаемые блоки уничтожены)

Ключевые особенности:

Формат уровней в JSON

Уровни описываются в структурированном виде, где `layout` представляет матрицу блоков:

- **1** — зелёный блок
- **2** — жёлтый блок
- **3** — красный блок
- **9** — камень (неразрушаемый)

Пример структуры уровня:

```
{
    "number": 1,
    "name": "Classic Start",
    "description": "Basic Level",
    "centerHorizontally": true,
    "centerVertically": false,
    "marginTop": 120,
    "marginSides": 50,
    "spacing": 5,
    "layout": [
        [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
        [ 2, 2, 2, 9, 2, 2, 2, 9, 2, 2, 2 ],
        [ 3, 3, 3, 3, 9, 3, 9, 3, 3, 3, 3 ],
        [ 1, 1, 9, 1, 1, 1, 1, 1, 9, 1, 1 ]
    ]
}
```

Параметры уровня:

- `number` — номер уровня
- `name` — название уровня
- `description` — описание уровня
- `centerHorizontally` — центрирование блоков по горизонтали
- `centerVertically` — центрирование блоков по вертикали
- `marginTop` — отступ сверху
- `marginSides` — отступы по бокам
- `spacing` — расстояние между блоками
- `layout` — двумерный массив, определяющий расположение блоков

Валидация и резервный уровень

Уровни загружаются при старте игры или при переходе между ними. Если загрузка не удалась, создаётся уровень по умолчанию.

Интеграция с GameState

Класс `GameState` использует `LevelManager` для:

- **Инициализации уровня**
- **Проверки завершения уровня:** Метод `isLevelComplete()` проверяет, остались ли разрушаемые блоки
- **Перехода между уровнями:** через `LevelManager::nextLevel()`

Состояния игры

В игре реализована система состояний (State Pattern), которая позволяет управлять различными экранами и режимами игры. Каждое состояние инкапсулирует свою логику, что делает код модульным и легко расширяемым.

Примечание

Все состояния из главного меню наследуются от `MenuStateBase` — базового класса меню.

Состояние	Описание
`MainMenuState`	Главное меню игры. Позволяет начать игру, выбрать уровень, перейти в настройки или выйти из игры. Интегрируется с `SoundManager` для воспроизведения звуков нажатия кнопок
`LevelSelectState`	Меню выбора уровня. Отображает сетку уровней, загруженных через `LevelManager`, с возможностью выбора
`OptionsState`	Меню настроек. Позволяет регулировать громкость музыки и звуковых эффектов через `SoundManager`
`GameState`	Основное игровое состояние. Управляет всем игровым процессом: движением мяча и платформы, системой коллизий через `PhysicsSystem`, логикой уровней (`LevelManager`), системой бонусов (`PowerUpManager`), подсчетом очков и жизней. Обрабатывает игровые события (пауза, запуск мяча, движение)
`GameOverState`	Состояние завершения игры. Отображает итоговый счёт и уровень. Предлагает вернуться в главное меню

Таблица 4: Состояния игры

Заключение

Достигнутые результаты

В результате выполнения курсовой работы была успешно разработана игра "Arkanoid"на языке C++ с использованием библиотеки SFML. Проект демонстрирует практическое применение принципов объектно-ориентированного программирования в создании мультимедийных приложений.

Все поставленные задачи выполнены:

- Реализована игровая механика с физикой столкновений
- Организована система подсчёта очков и жизней
- Добавлены звуковые эффекты для всех игровых событий
- Создана система уровней с загрузкой из JSON-файлов
- Реализована система бонусов с различными эффектами
- Организованы состояния игры с плавными переходами между меню

Освоенные технологии

SFML:

Освоена работа с графикой, звуком и обработкой пользовательского ввода для создания интерактивных приложений.

ООП:

Применены основные принципы объектно-ориентированного программирования: инкапсуляция, наследование и полиморфизм.

Паттерны проектирования:

Изучены и реализованы четыре ключевых паттерна:

- **State** — для управления состояниями игры
- **Observer** — для обработки событий коллизий
- **Factory** — для создания игровых объектов
- **Singleton** — для менеджеров ресурсов

Архитектурные преимущества

Созданная архитектура обладает важными качествами:

- **Модульность** — код разделён на независимые компоненты
- **Расширяемость** — легко добавлять новые игровые элементы
- **Читаемость** — понятная структура и логика программы

Практические навыки

Получен ценный опыт в области:

- Проектирования игровых приложений
- Работы с мультимедийными библиотеками
- Создания модульной архитектуры программ
- Применения паттернов проектирования на практике

Выводы

В ходе выполнения курсовой работы я получил ценный практический опыт разработки игрового приложения с применением объектно-ориентированного подхода и паттернов проектирования. Работа над проектом "Arkanoid"позволила мне на практике применить теоретические знания ООП и понять, как они работают в реальных условиях.

Проект успешно выполнен!