Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Объектно-ориентированное программирование

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

на тему

СОЗДАНИЕ API ДЛЯ РАЗРАБОТКИ ДВУМЕРНЫХ ИГР ДЛЯ ОПЕРАЦИОННЫХ СИСТЕМ WINDOWS И LINUX

БГУИР КП 1-40 04 01 01 009

Студент Тимофеев К.А.

Руководитель Рогов М.Г.

Минск 2023

СОДЕРЖАНИЕ

Введение 6
1 Постановка задачи 7
1.1 Анализ рынка и пользователей 7
1.2 Анализ существующих аналогов 7
1.3 Функциональные требования 8
2 Теоретическое обоснование разработки 9
2.1 Технологии программирования 9
2.2 Паттерны программирования 10
3 Архитектура разработанной системы 11
3.1 Структура и архитектура серверной части 11
3.2 Структура и архитектура тестовой игры…. 15
3.3 Структура классов 17
4 Функциональные возможности программы 18
4.1 Описание функций программного средства 18
4.2 Функционал тестовой игры 19
Заключение 22
Список использованных источников 23
Приложение А (обязательное) Исходный код 24

ОПРЕДЕЛЕНИЯ И СОКРАЩЕНИЯ

Объектно-ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Паттерн проектирования – повторяемая архитектурная конструкция в сфере проектирования программного обеспечения, предлагающая решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Архитектура программного средства – это структурная организация и компоненты программного средства, их взаимосвязи и взаимодействие, которые определяют его общую форму и функциональность. Она описывает различные аспекты программного средства, такие как его компоненты, модули, слои, интерфейсы, взаимодействие между ними, принципы организации кода и данные.

Архитектура разработанной системы – это описание структуры, организации и компонентов системы, которая была разработана на основе определенной архитектурной концепции или дизайна. Она представляет собой реализацию архитектурных решений, учитывающих функциональные и нефункциональные требования системы.

Simple DirectMedia Layer (или коротко SDL) — свободная кроссплатформенная мультимедийная библиотека, реализующая единый программный интерфейс к графической подсистеме, звуковым устройствам и средствам ввода для широкого спектра платформ. SDL API доступны для языков программирования C, C++, C#, Java и многих других.

Use-case диаграмма – диаграмма, отражающая отношения между группами пользователей и являющаяся составной частью модели прецедентов, позволяющей описать систему на концептуальном уровне.

UML-диаграмма – это схема, нарисованная с применением символов UML. Она может содержать множество элементов и соединений между ними.

ВВЕДЕНИЕ

Игровая индустрия активно развивается, поэтому актуален вопрос в разработке специализированных API для разработки игр – игровых движков.

Игровой движок – это центральный программный компонент компьютерных и видео игр или других интерактивных приложений с графикой, обрабатываемой в реальном времени. Он обеспечивает основные технологии, упрощает разработку и часто даёт игре возможность запускаться на нескольких платформах, таких как игровые консоли и настольные операционные системы, например, Linux, Mac OS X и Microsoft Windows. Основную функциональность обычно обеспечивает игровой движок, включающий движок рендеринга («визуализатор») для 2D или 3D графики, систему скриптов, управление памятью и графические сцены.

Игровые движки помогают упростить разработку игр по многим направлениям. Они позволяют неоднократно использовать код, разделить разработку игры по направлениям (разработка графической части, программирование скриптов, разработка и модифицирование самого движка).

2D игры всегда были популярны. Во времена начала игровой индустрии все игры были двумерными, но в конце 90-ых с развитием 3D-графики они на время потеряли популярность. Однако с развитием игровой индустрии и появлением свободно распространяемых игровых движков они вновь обрели популярность, перейдя из категории крупнобюджетных проектов в сегмент игр от независимых разработчиков с малым бюджетом. Еще один скачок популярности 2D игр пришелся на начало 10-ых, когда технологии позволили запускать игры в браузерах и в телефонах.

1 ПОСТАНОВКА ЗАДАЧИ

1.1 Анализ рынка и пользователей

Основную долю рынка свободно распространяемых игровых движков занимают Unity и Unreal Engine. Они имеют большое количество встроенного функционала, активно поддерживаются разработчиками и регулярно получают обновления, добавляющие передовые технологии в области графики. Они оба имеют бесплатные версии с немного ограниченным функционалом, что существенно облегчает обучение новых специалистов за счет широкого сообщества. Так же они поддерживают разработку 2D игр, хоть и в основе всех технологий лежит работа с трехмерной графикой.

Серьезную долю занимают движки Source и CryEngine. Они общедоступны, но менее популярны и используются в основном для разработки модификаций к уже вышедшим на этих движках играм. Оба не поддерживают разработку 2D игр.

Набирает популярность многоцелевой игровой движок Godot. К его достоинствам можно отнести открытый исходный код, возможность разработки 2D игр, активную поддержку и быстро развивающееся сообщество.

Так же существуют узконаправленные движки для разработки конкретного жанра игр – RPG Maker(jRpg) , Ren'Py (визуальные новеллы), jMonkeyEngine (платформеры) и многие другие.

1.2 Анализ существующих аналогов

Самым популярным движком в разработке 2D игр считается Unity. В качестве языка для написания скриптов в нем используется C#. Для работы с Unity необходимо установить официальный редактор. Система скриптов вместе с возможностями языка C# позволяют легко настраивать поведение игровых объектов и настраивать связи между объектами. В специальном режиме есть возможность запуска игры без отключения большинства возможностей редактора, что дает возможность корректировать поведение объектов для лучшего игрового опыта. По умолчанию представлена широкая библиотека компонентов для работы с физикой и светом. Редактор позволяет проводить диагностику используемой памяти, легко добавлять и сохранять игровые объекты. Однако Unity является в первую очередь 3D-движком, поэтому ресурсы компьютера используются не самым эффективным образом, некоторые встроенные модули не адаптированы по работу в 2D. Разрабатываемый в данном проекте движок будет приспособлен для работы только с 2D играми, что позволит увеличить производительность и упростить разработку.

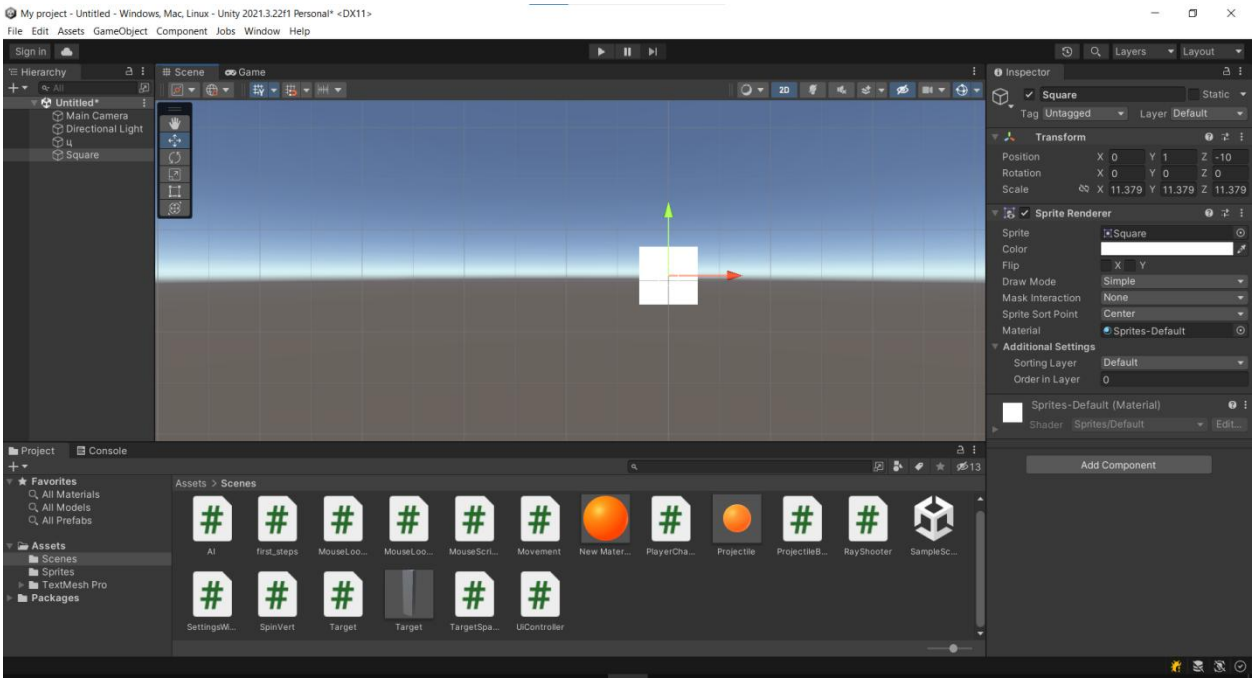


Рисунок 1.1 –Интерфейс редактора Unity

1.3 Функциональные требования

Движок должен позволять создавать игровые сцены как в ручную через код, так и при выгрузке из файла. Должны быть реализованы система скриптов, позволяющая менять поведение игровых объектов во время выполнения, система контроля используемых ресурсов, система родитель-потомок для связей между игровыми объектами. Необходима возможность настройки реагирования программы на нажатие клавиш. Техническое изложение предоставляемого функционала можно увидеть на Use-case диаграмме (см. рисунок 1.2):

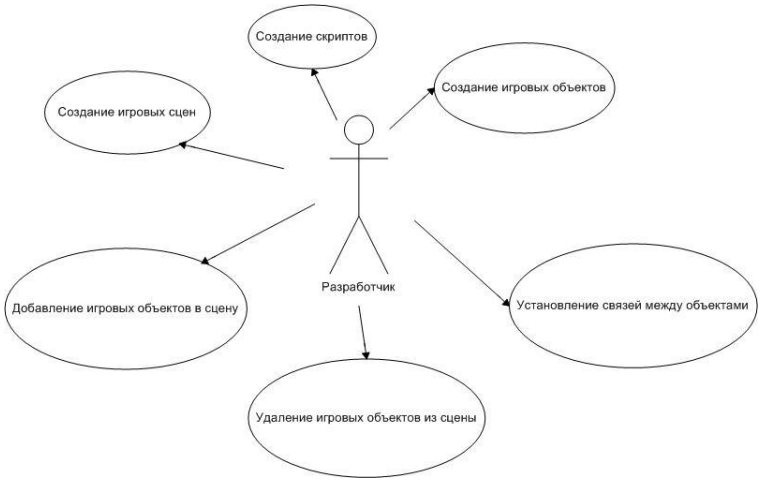


Рисунок 1.2 – Use-case диаграмма приложения

2 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ

2.1 Технологии программирования

Для решения поставленных задач был выбран язык программирования C++. Данный выбор позволит добиться требуемых результатов в производительности. Так же это дает возможность использовать огромное количество библиотек, написанных для языка программирования C.

Для вывода изображения на экран была использована кроссплатформенная библиотека SDL. Данная библиотека позволяет не только выводить изображение на экран, но и считывать нажатия клавиш, выводить звуковые файлы и видео.

Будут использованы различные технологии программирования, основанные на принципах ООП и SOLID.

Классы используют инкапсуляцию для скрытия внутренней реализации и данных от внешнего доступа. Они определяют приватные поля и методы для обеспечения контролируемого доступа к данным и выполнения операций.

Использование принципа абстракции позволило скрыть детали реализации объектов и предоставить только необходимый интерфейс для взаимодействия с ними. Внутренние механизмы работы объектов остаются скрытыми и недоступными для внешнего кода. Это позволяет изменять или улучшать реализацию объектов без влияния на клиентский код, который взаимодействует только с их абстракцией.

Помимо этого, в проекте был соблюден принцип единственной ответственности (Single Responsibility Principle, SRP), поскольку каждый класс имеет четко определенную задачу и отвечает только за свою функциональность.

Соблюдается принцип открытости/закрытости (Open/Closed Principle, OCP), который позволяет добавлять новую функциональность без изменения существующего кода.

Соблюдается принцип подстановки Барбары Лисков (Liskov Substitution Principle, LSP). Этот принцип устанавливает, что объекты в программе могут быть заменены их наследниками без изменения свойств программы. Другими словами, объекты наследников могут использоваться вместо объектов базового класса без изменения поведения программы.

Таким образом, применение принципов инкапсуляции, полиморфизма, наследования и абстракции в сочетании с принципами SOLID в проекте позволяет создать гибкую и расширяемую архитектуру, где каждый класс имеет четко определенные обязанности и может быть легко модифицирован или заменен при необходимости.

2.2 Паттерны программирования

В основе программного продукта были использован паттерн Strategy.

Strategy – это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

Вы решили написать приложение-навигатор для путешественников. Оно должно показывать красивую и удобную карту, позволяющую с лёгкостью ориентироваться в незнакомом городе. Одной из самых востребованных функций являлся поиск и прокладывание маршрутов. Пребывая в неизвестном ему городе, пользователь должен иметь возможность указать начальную точку и пункт назначения, а навигатор — проложит оптимальный путь. Первая версия вашего навигатора могла прокладывать маршрут лишь по дорогам, поэтому отлично подходила для путешествий на автомобиле. Но, очевидно, не все ездят в отпуск на машине. Поэтому следующим шагом вы добавили в навигатор прокладывание пеших маршрутов. Через некоторое время выяснилось, что некоторые люди предпочитают ездить по городу на общественном транспорте. Поэтому вы добавили и такую опцию прокладывания пути. Но и это ещё не всё. В ближайшей перспективе вы хотели бы добавить прокладывание маршрутов по велодорожкам. А в отдалённом будущем — интересные маршруты посещения достопримечательностей. Любое изменение алгоритмов поиска, будь то исправление багов или добавление нового алгоритма, затрагивало основной класс. Это повышало риск сделать ошибку, случайно задев остальной работающий код. Кроме того, осложнялась командная работа с другими программистами, которых вы наняли после успешного релиза навигатора. Ваши изменения нередко затрагивали один и тот же код, создавая конфликты, которые требовали дополнительного времени на их разрешение. Паттерн Стратегия предлагает определить семейство схожих алгоритмов, которые часто изменяются или расширяются, и вынести их в собственные классы, называемые стратегиями. Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию. Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.

3 АРХИТЕКТУРА РАЗРАБОТАННОЙ СИСТЕМЫ

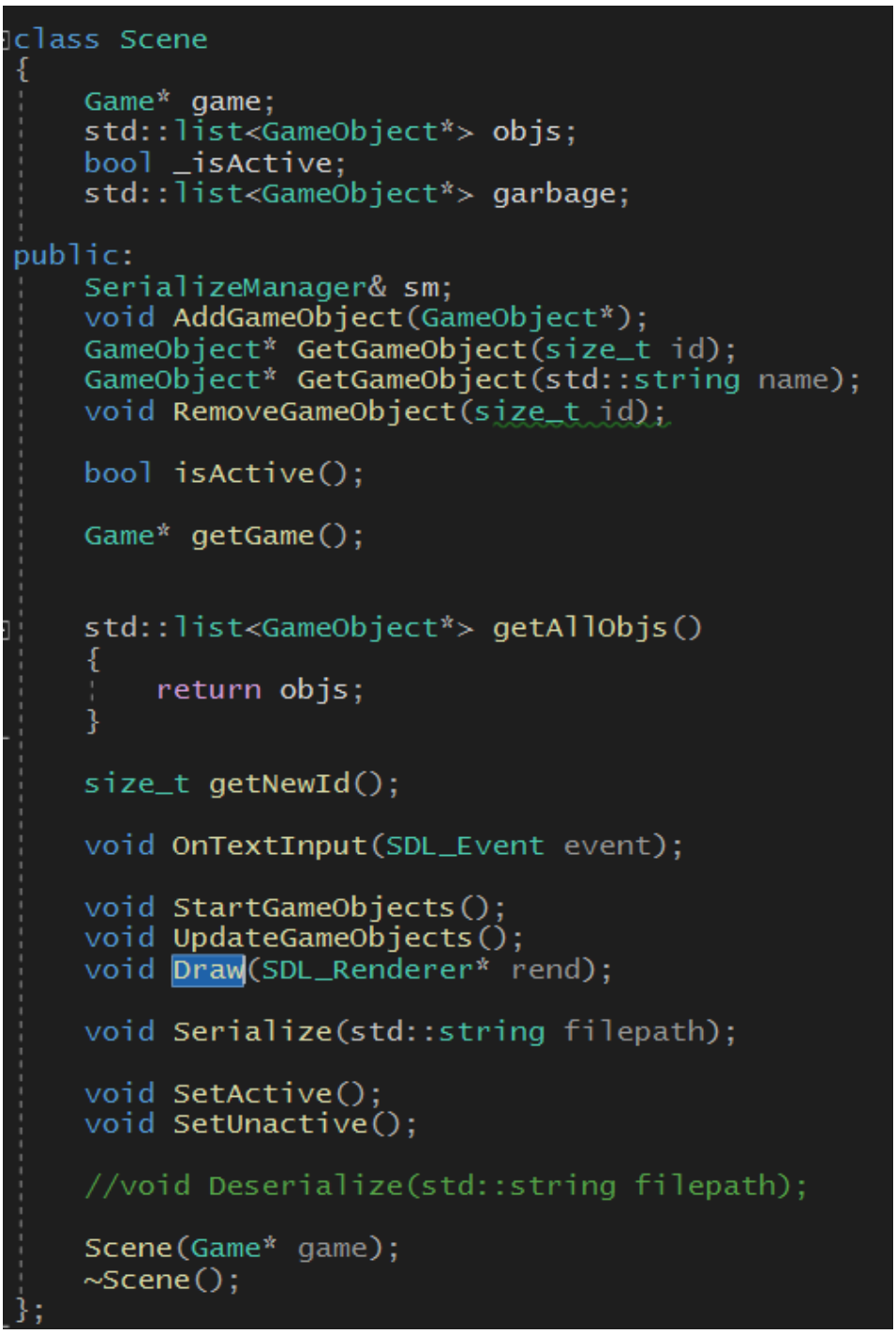
3.1 Структура и архитектура

Главным объектом является объект класса Game. Он хранит в себе текущую игровую сцену, ресурсы необходимые для отрисовки и объект специального класса ResourceManager, необходимого для регистрации загруженных ресурсов. Двумя главными методами класса Game являются Run() и setCurrentScene(). Run() запускает основной цикл игры, а setCurrentScene() устанавливает игровую сцену.

```
class Game
{
    Scene* currentScene;
    bool toBeClosed;
    std::string title;
    Vector size;
    SDL_Window* win;
    SDL_Renderer* renderer;
    ResourceManager rm;
    Scene* nextScene;


public:
    SerializeManager serializeManager;
    size_t getId();
    void Run();
    void SetCurrentScene(Scene* scene)
    {
        currentScene = scene;
    }
    //void SetTitle(std::string title);

    void Close();

    SDL_Renderer* getRenderer();

    Game(Vector);

    void ChangeScene(Scene* newScene);
    ResourceManager& resourceManager() { return rm; }



    ~Game();
};
```

Рисунок 3.1 – Заголовочный файл класса Game

Объекты класса Scene представляют собой игровую сцену и является в сути коллекцией объектов класса GameObject. Объекты класса Scene отвечают за их отрисовку и удаление из памяти. Имеются методы доступа к конкретным объектам через уникальный идентификатор и метод доступа к копии коллекции игровых объектов. В качестве коллекции для хранения ссылок на игровые объекты был выбран класс std::list из стандартной библиотеки шаблонов языка C++. Выбор обусловлен регулярными вставками, удалениями и отсутствием необходимости быстрого доступа по индексу.



```
class Scene
{
    Game* game;
    std::list<GameObject*> objs;
    bool _isActive;
    std::list<GameObject*> garbage;

public:
    SerializeManager& sm;
    void AddGameObject(GameObject*);
    GameObject* GetGameObject(size_t id);
    GameObject* GetGameObject(std::string name);
    void RemoveGameObject(size_t id);

    bool isActive();

    Game* getGame();


    std::list<GameObject*> getAllObjs()
    {
        return objs;
    }

    size_t getNewId();

    void OnTextInput(SDL_Event event);

    void StartGameObjects();
    void UpdateGameObjects();
    void Draw(SDL_Renderer* rend);

    void Serialize(std::string filepath);

    void SetActive();
    void SetUnactive();

    //void Deserialize(std::string filepath);

    Scene(Game* game);
    ~Scene();
};
```

Рисунок 3.2 – Заголовочный файл класса Scene

Объекты класса GameObject хранят в себе коллекцию скриптов, объект специального класса Texture для хранения текстур, маркеры активности и видимости объекта, коллекцию потомков, ссылку на родительский игровой объект, ссылку на текущую сцену и объект структуры Transform, хранящей данные о положении игрового объекта в сцене.

```
class GameObject
{
    friend class Scene;
private:
    static size_t idCount;
    std::unordered_map<size_t, Component*, TypeHash> components;
    size_t id;
    std::list<GameObject*> childs;
    GameObject* parent;
    Scene* scene;
    bool _isVisible;
    bool _isActive;
    Texture texture;
    std::string name;
public:
    Transform transform;
    template <class T>
    void AddComponent(T* componentReference);
    template <typename T>
    requires std::derived_from<T, Component>
    Component* GetComponent() const;

    template<class T>
    bool hasComponent() const;
    template<class T>
    void RemoveComponent();

    void Update();
    void Start();

    Scene* getScene();

    GameObject* GetChild(size_t id) const;
    GameObject* GetChild(std::string name) const;

    void AddChild(GameObject*);
    void RemoveChild(GameObject*);

    size_t GetId() const;

    GameObject(std::string name = "");

    GameObject* getParent();
    void setParent(GameObject*);

    bool isVisible();
    void SetVisible();
    void SetInvisible();

    bool isActive();
    void SetActive();
    void SetUnactive();

    Texture GetTexture();

    void SetTexture(Texture text);

    std::string GetName();

    void Serialize(std::fstream&);
    void Deserialize(std::fstream&);

    void OnCollision(GameObject*);
    void Die();
    void OnTextInput(SDL_Event event);

    ~GameObject();
};
```

Рисунок 3.3 – Заголовочный файл класса GameObject

Скрипты представляют собой классы, унаследованные от абстрактного класса Component. В этом классе имеются два виртуальных метода: Start() и Update(), которые необходимо переопределить в классе скрипта. Метод Start() вызывается при появлении компонента в игровом объекте, а метод Update() вызывается игровым объектом в каждом кадре прорисовки.

В игровом объекте хранится коллекция ссылок на объекты классов, унаследованных от класса Component. Однако есть необходимость в доступе к конкретному копоненту игрового объекта со всеми добавленными полями и методами, а не только к тем, чьё наличие гарантировано классом Component. Поэтому выбор для коллекции ссылок на компоненты пал на std::unordered_map и стандартной библиотеки. Данный класс является реализацией хеш-таблицы, которая идеально подходит под поставленную задачу хранения элементов разных типов. Благодаря встроенной функции языка C++ typeid мы можем вычислить хеш некоторого типа данных. Таким

образом при добавлении скрипта объекту мы используем шаблонную функцию AddComponent<T>(T*), где T – тип, унаследованный от класса Component. В качестве хеша берется хеш типа T, благодаря чему имеется возможность хранить в одной коллекции объекты разных типов, не теряя возможности обращаться ко всем их полям и методам.

```
template <typename T>
requires std::derived_from<T, Component>
Component* GameObject::GetComponent() const
{
    Component* result = components.at(typeid(T).hash_code());
    return result;
}

template<class T>
void GameObject::AddComponent(T* component)
{
    if (component->GetGameObject() != this && component->GetGameObject() != nullptr) throw std::exception("Wrong game object");
    component->gameObject = this;
    components.insert({ typeid(T).hash_code(), component });
    if (scene != nullptr)
    {
        int a = 0;
        ++a;
        if (scene != nullptr)
        {
            if (scene->isActive()) component->Start();
        }
    }
}

template <class T>
void GameObject::RemoveComponent()
{
    delete components[typeid(T).hash_code()];
    components.erase(typeid(T).hash_code());
}

template<class T>
bool GameObject::hasComponent() const
{
    try
    {
        components.at(typeid(T).hash_code());
    }
    catch (std::out_of_range)
    {
        return false;
    }

    return true;
}
```

Рисунок 3.4 – Реализация работы с компонентами

Для сохранения объектов и сцен в файлах в соответствующие классы был добавлен метод Serialize(std::fstream). Аналогично с выгрузкой из файлов при помощи метода Deserialize(std::fstream). Однако при сохранении игровых объектов стоит вопрос о сохранении объектов скриптов в памяти. Если загрузку в память можно обеспечить добавлением в родительский класс Component виртуального метода для сериализации, то десериализовать объект не зная базовый тип из-за особенностей языка мы не сможем. Для этого был создан специальный класс SerializeManager. Он отвечает за сериализацию базовых типов данных и имеет в себе специальную таблицу функций, отвечающих за сериализацию и десериализацию объектов и скриптов. Данная таблица представляет собой объект std::unordered_map. Хранение пары функций организовано по тому же принципу что и хранение компонентов (через хеш типа). Поэтому при необходимости сериализации объектов разработчику необходимо вручную зарегистрировать все используемые в сцене компоненты вызовом метода register_component<T>().

Так же уделено внимание сериализации ссылок на другие объекты или компоненты. Напрямую сохранять объекты не имеет смысла, поэтому проблема была решена по-другому. При сериализации в файл сохраняется идентификатор объекта, на который делается ссылка. При десериализации в специальную коллекцию добавляются пара значений идентификаторов, представляющие пару держателя ссылки и объекта, на который ссылаются. Возможности языка C++ позволяют обращаться к участкам памяти

программы, даже если они не являются доступными для текущего объекта. Поэтому мы можем хранить держателя ссылки в виде указателя на указатель на объект или компонент, что позволит потом их заполнить актуальными данными. Так как идентификаторы при десериализации будут изменены, пара значений старого и нового идентификаторов заносится в другую таблицу для дальнейшего использования. В конце десериализации, в держателей ссылок заносятся актуальные ссылки на компоненты и игровые объекты.

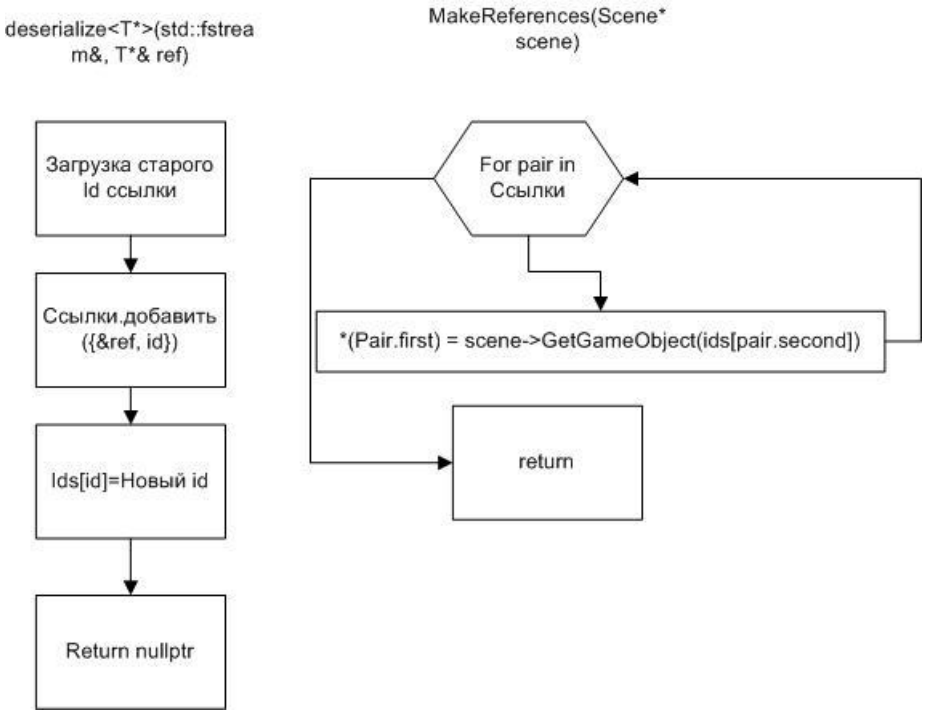


Рисунок 3.5 – Схема алгоритма десериализации ссылок

3.2 Структура и архитектура тестовой игры

В качестве демонстрационного проекта была разработана игра, являющаяся аналогом «Arcanoid».

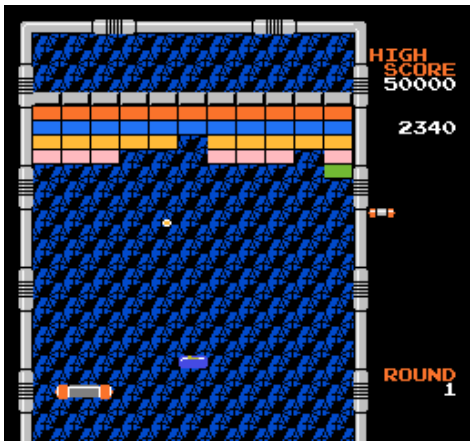


Рисунок 3.6 – Игра «Arcanoid»

В разработанной игре присутствуют следующие типы игровых объектов: разрушаемые блоки, платформа игрока, отскакивающий от поверхностей шар, границы, окна выводы информации о рекорде и объект, координирующий действия остальных

Для хранения данных о рекордах и пользователях была использована свободно распространяемая база данных SQLite3. Для просмотра базы данных во время разработки была использована программа SQLiteStudio.

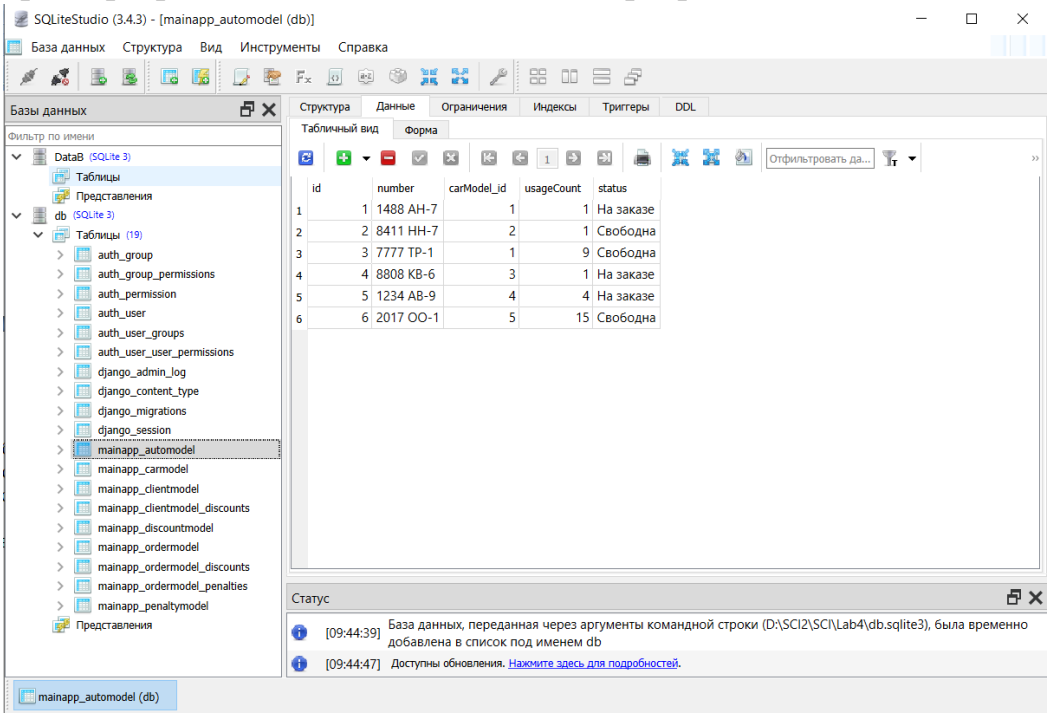


Рисунок 3.7 – Интерфейс SQLiteStudio

3.3 Структура классов

При разработке программного решения для игрового движка для разработки 2D игр была применена архитектура, основанная на объектно-ориентированной парадигме программирования. Данная парадигма позволяет создать систему, где реальные сущности и их свойства могут быть точно описаны в виде объектов.

Детальнее с построением зависимостей между классами можно ознакомиться на UML-схеме (см. рисунок 3.8).

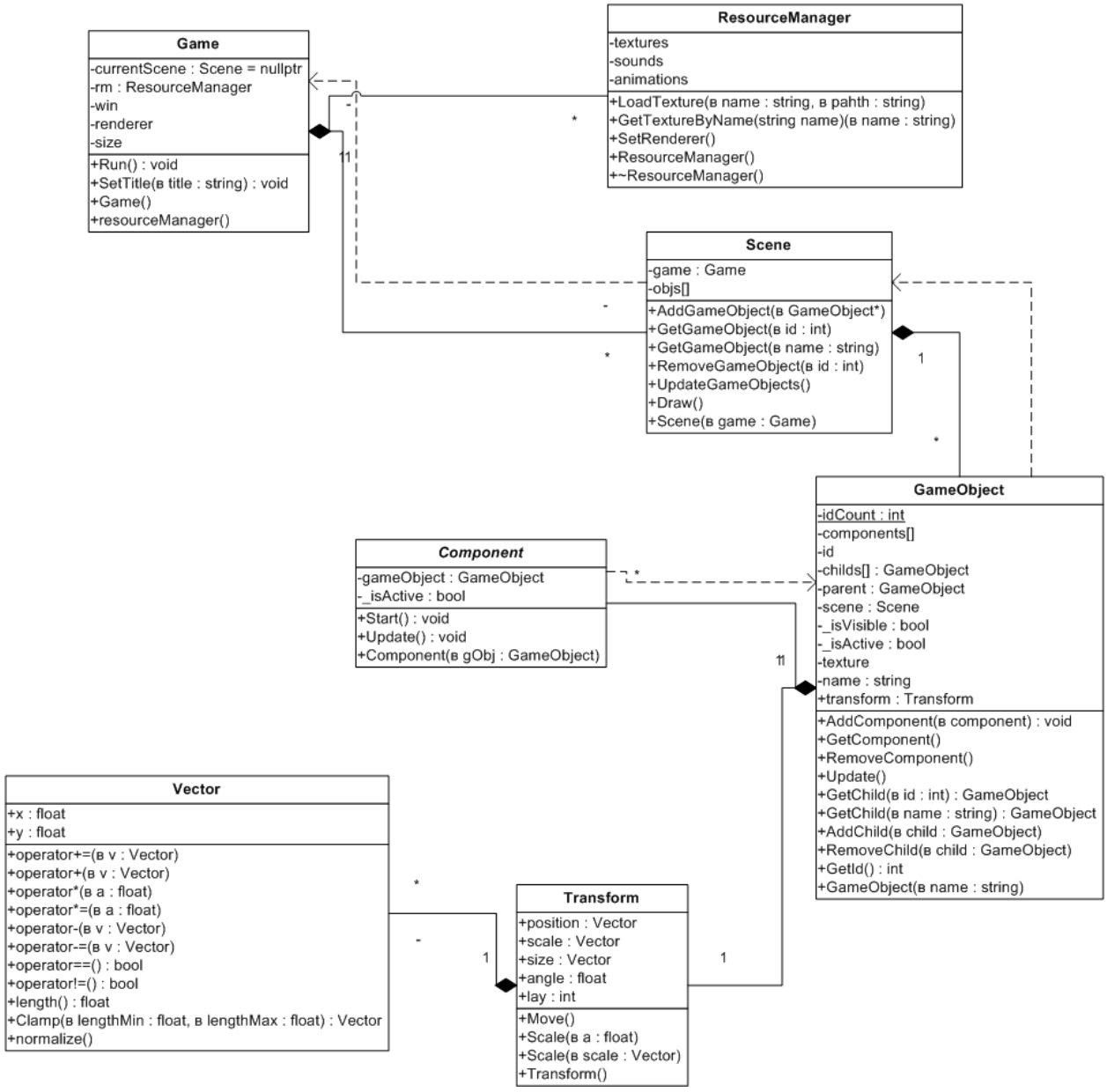


Рисунок 3.8 – UML-диаграмма классов

4 ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ

4.1 Описание функций программного средства

Основной функционал программного средства включает в себя возможность создания игровых объектов, добавление их в игровые сцены, возможность создания игровых скриптов для изменения поведения объектов в сцене, возможность добавлять и удалять игровые компоненты у объектов во время игры. Функционал по реагированию на нажатие клавиш клавиатуры и мыши, по выводу изображения на экран и выводу звука обеспечен возможностями используемой библиотеки SDL.

Для старта необходимо создать объект класса Game. В конструктор передается двумерный вектор, представляющий размеры окна.

```
const size_t WIDTH = 800;
const size_t HEIGHT = 600;
Game game = Game({ WIDTH, HEIGHT });
```

Рисунок 4.1 – Создание объекта класса Game

Для работы с игровыми объектами необходимо создать игровую сцену и разместить в ней игровые объекты. Объектам можно добавлять компоненты для определения их поведения в сцене.

```
Scene* scene = new Scene(&game);
GameObject* go = new GameObject();
go->AddComponent(new GameDirector(pair.first, pair.second));
scene->AddGameObject(go);
```

Рисунок 4.2 – Создание игрового объекта и добавление его в сцену

Для создания компонента необходимо создать класс, унаследованный от класса Component и переопределить в нем методы Start() и Update().

```
class GameDirector :public Component
{
    TextOutput* recOut;
    TextOutput* usrOut;
    BallComponent* ball;
    int level;
    int points;

public:
    Player rec;
    Player current;
    Scene* MakeScene() { return nullptr; };
    void Start();
    void Update();

    GameDirector(Player current, Player rec, int points = 0, int speedLevel = 0);
};
```

Рисунок 4.3 – Объявление класса GameDirector, являющегося компонентом

Для взаимодействия между игровыми объектами в классе GameObject есть методы доступа к конкретному компоненту и методы проверки наличия этого компонента в объекте.

```
obj->hasComponent<BoxComponent>()
```

Рисунок 4.4 – Метод проверки наличия компонента у объекта

```
BlockComponent* bl = static_cast<BlockComponent*>(obj->GetComponent<BlockComponent>());
```

Рисунок 4.5 – Метод доступа к компоненту

Менеджер внешних ресурсов позволяет оптимизировать загрузку внешних ресурсов.

```
Texture gb = game.resourceManager().LoadTexture("greenblock", "D:/Kursach/greenblock.png", { {0,0}, {50,20} });
Texture yb = game.resourceManager().LoadTexture("yellowblock", "D:/Kursach/yellowblock.png", { {0,0}, {50,20} });
Texture rb = game.resourceManager().LoadTexture("redblock", "D:/Kursach/redblock.png", { {0,0}, {50,20} });
Texture bg = game.resourceManager().LoadTexture("background", "D:/Kursach/background.png", { {0,0}, {600, 600} });

Texture pt = game.resourceManager().LoadTexture("player", "D:/Kursach/playerblock.png", { {0,0},{100,30} });

Texture bord = game.resourceManager().LoadTexture("border", "D:/Kursach/border.png", { {0,0}, {600, 50} });

Texture bt = game.resourceManager().LoadTexture("ball", "D:/Kursach/ball.png", { {0,0},{20,20} }, 255, 255, 255);
```

Рисунок 4.6 – Загрузка внешних ресурсов через менеджер ресурсов

4.2 Функционал тестовой игры

Тестовая игра представляет собой вариацию игры «Arcanoid».

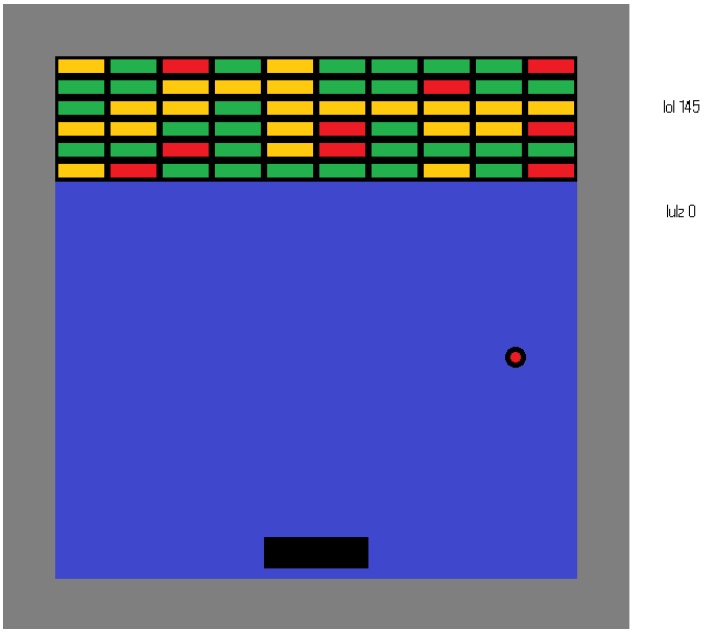


Рисунок 4.7 – Тестовая игра

В игре присутствуют блоки трех видов: зеленые, желтые и красные. Для их уничтожения по ним необходимее нанести 1, 2 и 3 удара соответственно. При нанесении по блоку удара его цвет меняется на цвет,

соответствующий оставшемуся для его уничтожения количеству ударов шара.

Игрок управляет черной платформой. Уничтожаемые блоки и платформа игрока имеют компонент BoxComponent, необходимый для обнаружения столкновений.

```
player->AddComponent(new BoxComponent());
```

Рисунок 4.8 – Добавление игроку компонента BoxComponent

За обнаружение столкновений отвечает компонент BallComponent, отвечающий так же за поведение шара. При активации компонента он запоминает все игровые объекты, имеющие компонент BoxComponent. При вызове метода Update() проверяется наличие столкновения с каким-либо объектом.

```
for (int i = 0; i < blocks.size(); ++i)
{
    auto obj = blocks[i];
    Transform o_tr = obj->transform;
    Rect r = { {o_tr.position.x - o_tr.size.x / 2, o_tr.position.y - o_tr.size.y / 2},
        {o_tr.size.x, o_tr.size.y} };
    if(checkCollisionToBlock(r))
    {
        std::cout << "Collision check";
        obj->OnCollision(gameObject);
        if (!obj->isActive() && !obj->isVisible())
        {
            BlockComponent* bl = static_cast<BlockComponent*>(obj->GetComponent<BlockComponent>());
            points += 2*bl->lvl - 1;
            blocks.erase(blocks.begin() + i);
            std::cout << obj->GetName() << std::endl;
            --i;
            ++kills;
        }
    }

}
```

Рисунок 4.9 – Проверка наличия столкновений

Так же этот компонент отвечает за движение шара по игровой сцене.

```
gameObject->transform.position += dir * speed;
```

Рисунок 4.10 – Изменение положения шара в сцене

При столкновении шар вызывает у компонентов объекта метод OnCollision(GameObject*). У разрушаемых блоков этот метод вызовется у компонента BlockComponent. Метод изменит направление движения шара, уменьшит количество оставшихся жизней и изменит его текстуру.

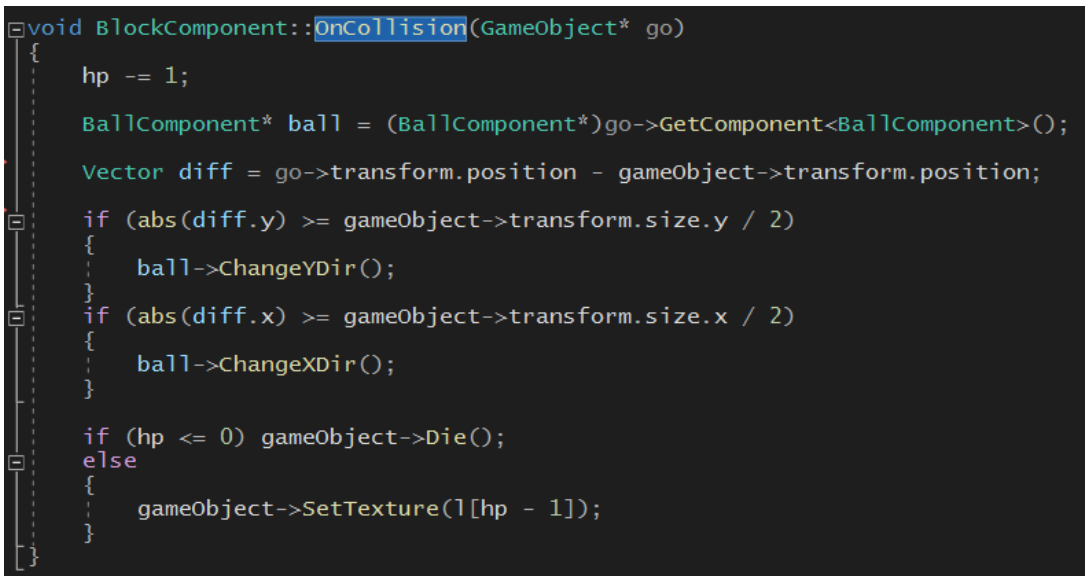


Рисунок 4.11 – Метод OnCollision в компоненте BlockComponent

В тестовой игре реализована возможность регистрации пользователей и входа в учетную запись. Этот функционал реализован в консоли. Данные пользователей хранятся в базе данных. В таблице хранятся логин пользователя, хеш его пароля и его рекорд.

Рисунок 4.12 – Запрос в базу данных на создание таблицы пользователей

В базе данных ищутся текущий пользователь и пользователь с наибольшим количеством очков. Они передаются особому компоненту, который отвечает за отображение текста в игре. У текущего пользователя отображается текущее количество очков, а у пользователя с рекордом отображается его ник и максимальное количество набранных очков.

ЗАКЛЮЧЕНИЕ

В данной курсовой работе был разработан игровой движок для создания 2D игр на языке программирования C++ с использованием кроссплатформенной библиотеки на языке C SDL. При разработке проекта были использованы принципы ООП (инкапсуляция, наследование, полиморфизм, абстракция), а также принципы SOLID и паттерны программирования.

У данного проекта есть дальнейшие перспективы модификации и развития. Например, добавление системы физики с различными формами объектов, упрощение вывода текстовой информации на экран, создание классов для взаимодействия с библиотекой SDL и много другое.

Итоговый продукт удовлетворяет не только техническим, но и функциональным требованиям. Благодаря соблюдению принципов ООП и SOLID, в приложение можно легко добавлять новые функциональные модули и расширять его возможности без необходимости переписывать или усложнять код. Новые модули могут быть реализованы путем разработки новых классов, наследующихся от базовых, что позволит использовать уже существующие методы и функции программного средства.

Таким образом, данная курсовая работа демонстрирует возможности и преимущества использования ООП-подходов при разработке программного средства для программирования 2D игр.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Lazy Foo’s Productions – Beginning Game Programming v2.0 [Электронный ресурс]. – Режим доступа: https://lazyfoo.net/tutorials/SDL/index.php

[2] Простой алгоритм определения пересечения двух отрезков [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/267037/

[3] Стандарт C++20: обзор новых возможностей C++. Часть 3 «Концепты» [Электронный ресурс]. – Режим доступа: https://habr.com/ru/companies/yandex_praktikum/articles/556816/

[4] Стратегия [Электронный ресурс]. – Режим доступа: https://refactoring.guru/ru/design-patterns/strategy.

ПРИЛОЖЕНИЕ А
(обязательное)
Исходный код

Файл Game.h

```
#pragma once
#include <SDL.h>
#include <string>
#include "Vector.h"
#include "Texture.h"
#include "ResourceManager.h"
#include "Scene.h"
#include "SerializeManager.h"
#include <SDL_ttf.h>
//#include "GameObjectTemps.cpp"

//class Scene;
class Game
{
	Scene* currentScene;
	bool toBeClosed;
	std::string title;
	Vector size;
	SDL_Window* win;
	SDL_Renderer* renderer;
	ResourceManager rm;
	Scene* nextScene;


public:
	SerializeManager serializeManager;
	size_t getId();
	void Run();
	void SetCurrentScene(Scene* scene)
	{
		currentScene = scene;
	}
	//void SetTitle(std::string title);

	void Close();

	SDL_Renderer* getRenderer();

	Game(Vector);

	void ChangeScene(Scene* newScene);
	ResourceManager& resourceManager() { return rm; }



	~Game();
};
```

Файл Game.cpp

```
#include "Game.h"
#include "Scene.h"

size_t Game::getId()
{
	static size_t count = 0;
```

```
	return count++;
}

void Game::Run()
{
	currentScene->SetActive();
	currentScene->StartGameObjects();
	while (true)
	{
		currentScene->UpdateGameObjects();
		SDL_Event ev;
		while (SDL_PollEvent(&ev))
		{
			if (ev.type == SDL_EventType::SDL_QUIT)
			{
				std::cout << "to quit" << std::endl;
				toBeClosed = true;
				break;
			}
			if (ev.type == SDL_EventType::SDL_TEXTINPUT || ev.type ==
SDL_EventType::SDL_TEXTEDITING)
			{
				currentScene->OnTextInput(ev);
			}
		}

		if (toBeClosed) break;

		currentScene->Draw(renderer);
		if (nextScene)
		{
			delete currentScene;
			currentScene = nextScene;
			nextScene = nullptr;
			currentScene->StartGameObjects();
		}
	}

	SDL_Quit();
}

void Game::Close()
{
	toBeClosed = true;
}

SDL_Renderer* Game::getRenderer()
{
	return renderer;
}

Game::Game(Vector _size) : size(_size), serializeManager(SerializeManager(&rm)),
nextScene(nullptr)
{
	SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO);
	win = SDL_CreateWindow(title.c_str(), SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED,
		size.x, size.y, SDL_WindowFlags::SDL_WINDOW_SHOWN);

	renderer = SDL_CreateRenderer(win, -1,
SDL_RendererFlags::SDL_RENDERER_ACCELERATED|
SDL_RendererFlags::SDL_RENDERER_PRESENTVSYNC);

	SDL_SetRenderDrawColor(renderer, 255, 255, 255, 0);
	rm.SetRenderer(renderer);
```

```
	TTF_Init();
}

void Game::ChangeScene(Scene* newScene)
{
	nextScene = newScene;
}

Game::~Game()
{
	delete currentScene;
	TTF_Quit();
	IMG_Quit();
	SDL_Quit();

}
```

Файл Scene.h

```
#pragma once
#include "GameObject.h"
#include <list>
#include "SerializeManager.h"


class Game;

class Scene
{
	Game* game;
	std::list<GameObject*> objs;
	bool _isActive;
	std::list<GameObject*> garbage;

public:
	SerializeManager& sm;
	void AddGameObject(GameObject*);
	GameObject* GetGameObject(size_t id);
	GameObject* GetGameObject(std::string name);
	void RemoveGameObject(size_t id);

	bool isActive();

	Game* getGame();


	std::list<GameObject*> getAllObjs()
	{
		return objs;
	}

	size_t getNewId();

	void OnTextInput(SDL_Event event);

	void StartGameObjects();
	void UpdateGameObjects();
	void Draw(SDL_Renderer* rend);

	void Serialize(std::string filepath);

	void SetActive();
	void SetUnactive();
```

```
	//void Deserialize(std::string filepath);

	Scene(Game* game);
	~Scene();
};
```

Файл Scene.cpp

```
#include "Scene.h"
#include "Game.h"
#include <iostream>


void Scene::AddGameObject(GameObject* obj)
{
	obj->scene = this;
	if (_isActive) obj->Start();
	auto it = objs.begin();
	for (; it != objs.end(); ++it)
	{
		if ((*it)->transform.lay <= obj->transform.lay) break;
	}
	objs.insert(it, obj);
}
GameObject* Scene::GetGameObject(size_t id)
{
	for (auto it : objs)
	{
		if (it->GetId() == id) return it;
	}
	throw std::exception(("There is no gameobject with id " + std::to_string(id)
+ " in the scene").c_str());
}
GameObject* Scene::GetGameObject(std::string name)
{
	for (auto it : objs)
	{
		if (it->GetName() == name) return it;
	}
	throw std::exception(("There is no gameobject with name " + name + " in the
scene").c_str());
}
void Scene::RemoveGameObject(size_t id)
{
	auto it = objs.begin();
	for (; it != objs.end(); ++it)
		if ((*it)->GetId() == id) break;
	garbage.push_back(*it);
	objs.erase(it);
}
void Scene::UpdateGameObjects()
{
	_isActive = true;
	for (auto obj : objs)
	{
		if (obj->isActive()) obj->Update();
	}

	for (auto obj : garbage)
	{
		delete obj;
	}

	garbage.clear();
}
```

```
void Scene::StartGameObjects()
{
	_isActive = true;
	for (auto obj : objs)
	{
		if (obj->isActive()) obj->Start();
	}
}
void Scene::Draw(SDL_Renderer* rend)
{
	SDL_RenderClear(rend);
	for (auto obj : objs)
	{
		if (obj->isVisible() == false) continue;

		const Transform& tr = obj->transform;
		SDL_Rect rect;
		float sizeX = tr.scale.x * tr.size.x;
		float sizeY = tr.scale.y * tr.size.y;
		Vector glob = tr.getGlobalPosition();
		rect.x = glob.x - sizeX / 2;
		rect.y = glob.y - sizeY / 2;
		rect.w = sizeX;
		rect.h = sizeY;
		//std::cout << obj->name << " x:" << rect.x << " y:" << rect.y <<
std::endl;

		//std::cout
			//<< "x: " << tr.position.x << " y: " << tr.position.y << "\n"
		//	<< "Center x: " << center.x << " y: " << center.y << "\n";
		auto text = obj->GetTexture();
		SDL_Rect rec = text.GetRect().toSdlRect();

		SDL_RenderCopyEx(rend, text.GetTexture(), nullptr, &rect, -
tr.getGlobalAngle(), nullptr, SDL_FLIP_NONE);
	}

	SDL_RenderPresent(rend);
}


void Scene::Serialize(std::string filepath)
{
	std::fstream stream = std::fstream(filepath, std::ios_base::binary |
std::ios_base::in);
	for (GameObject* go : objs)
	{
		go->Serialize(stream);
	}
}

size_t Scene::getNewId() { return game->getId(); }

Game* Scene::getGame()
{
	return game;
}

bool Scene::isActive()
{
	return _isActive;
}

Scene::Scene(Game* game) : game(game), sm(game->serializeManager), _isActive(false)
{}
```

```
Scene::~Scene()
{
	for (auto obj : objs)
	{
		delete obj;
	}

	for (auto obj : garbage)
	{
		delete obj;
	}
}

void Scene::OnTextInput(SDL_Event event)
{
	for (auto obj : objs)
	{
		obj->OnTextInput(event);
	}
}

void Scene::SetActive()
{
	_isActive = true;
}

void Scene::SetUnactive()
{
	_isActive = false;
}
```

Файл GameObject.h

```
#pragma once
#include <unordered_map>
#include <unordered_set>
#include "Component.h"
#include <string>
#include <typeinfo>
#include "Transform.h"
#include "Texture.h"
#include "TypeHash.h"
#include <SDL.h>

//template <class T>
//concept ComponentType = std::derived_from<T, Component>;
//
//template <class T, typename ...Args>
//concept EmplaceType = ComponentType<T> && std::constructible_from<T, GameObject*,
Args...>;



class Scene;

class GameObject
{
	friend class Scene;
private:
	static size_t idCount;
	std::unordered_map<size_t, Component*, TypeHash> components;
	size_t id;
```

```
        std::list<GameObject*> childs;
        GameObject* parent;
        Scene* scene;
        bool _isVisible;
        bool _isActive;
        Texture texture;
        std::string name;
public:
        Transform transform;
        template <class T>
        void AddComponent(T* componentReference);
        template <typename T>
        requires std::derived_from<T, Component>
        Component* GetComponent() const;

        template<class T>
        bool hasComponent() const;
        template<class T>
        void RemoveComponent();

        void Update();
        void Start();

        Scene* getScene();

        GameObject* GetChild(size_t id) const;
        GameObject* GetChild(std::string name) const;

        void AddChild(GameObject*);
        void RemoveChild(GameObject*);

        size_t GetId() const;

        GameObject(std::string name = "");

        GameObject* getParent();
        void setParent(GameObject*);

        bool isVisible();
        void SetVisible();
        void SetInvisible();

        bool isActive();
        void SetActive();
        void SetUnactive();

        Texture GetTexture();

        void SetTexture(Texture text);

        std::string GetName();

        void Serialize(std::fstream&);
        void Deserialize(std::fstream&);

        void OnCollision(GameObject*);
        void Die();
        void OnTextInput(SDL_Event event);

        ~GameObject();
};
```

Файл GameObject.cpp

```
#include "GameObject.h"
```

```cpp
#include "Component.h"
#include "Scene.h"
#include "SerializeManager.h"
#include "SerializeManagerTemps.cpp"

size_t GameObject::idCount = 0;


GameObject* GameObject::getParent()
{
	return parent;
}

GameObject::GameObject(std::string name):transform(this, {100,100}), scene(nullptr)
{
	id = idCount++;
	_isActive = true;
	_isVisible = true;
	if (name == "") name = "GameObject" + std::to_string(id);
	this->name = name;

}


void GameObject::setParent(GameObject* newParent)
{
	parent = newParent;

	newParent->childs.push_back(this);

}

void GameObject::OnTextInput(SDL_Event event)
{
	for (auto comp : components)
	{
		comp.second->OnTextInput(event);
	}
}

GameObject::~GameObject()
{
	for (auto comp : components)
	{
		delete comp.second;
	}
}

void GameObject::AddChild(GameObject* go)
{
	childs.push_back(go);
	go->parent = this;
}

void GameObject::RemoveChild(GameObject* go)
{
	auto i = childs.begin();
	for ( ;i != childs.end(); ++i)
		if (*i == go) break;
	childs.erase(i);
}

void GameObject::Update()
{
	for (auto comp : components)
```

```
	{
		if(comp.second->isActive()) comp.second->Update();
	}
}

void GameObject::Start()
{
	for (auto comp : components)
	{
		comp.second->Start();
	}
}

Scene* GameObject::getScene()
{
	return scene;
}

GameObject* GameObject::GetChild(size_t id) const
{
	for (auto it : childs)
	{
		if (it->id == id) return it;
	}
	std::string message = "There is no child with id " + std::to_string(id) + "
of object " + name;
	throw std::exception(message.c_str());
}

GameObject* GameObject::GetChild(std::string name) const
{
	for (auto it : childs)
	{
		if (it->name == name) return it;
	}
	std::string message = "There is no child with name " + name + " of object " +
this->name;
	throw std::exception(message.c_str());
}
//GameObject::GameObject(std::initializer_list<Component*> components, std::string
name = "")
//{
//}
size_t GameObject::GetId() const { return id; }

bool GameObject::isVisible() { return _isVisible; }

void GameObject::SetVisible()
{
	_isVisible = true;
}

void GameObject::SetInvisible()
{
	_isVisible = false;
}

bool GameObject::isActive() { return _isActive; }

void GameObject::SetActive()
{
	_isActive = true;
}

void GameObject::SetUnactive()
```

```
{
	_isActive = false;
}

Texture GameObject::GetTexture()
{
	return texture;
}

void GameObject::SetTexture(Texture text)
{
	texture = text;
}

std::string GameObject::GetName()
{
	return name;
}

void GameObject::Serialize(std::fstream& stream)
{
	scene->sm.serialize(stream, texture);

	scene->sm.serialize(stream, parent);

	scene->sm.serialize(stream, childs);

	scene->sm.serialize(stream, (int)components.size());

	for (auto pair : components)
	{
		Component* comp = pair.second;

		scene->sm.serialize(stream, comp);
	}

}

void GameObject::Deserialize(std::fstream& stream)
{
	texture = scene->sm.deserialize<Texture>(stream);
	parent = scene->sm.deserialize<GameObject*>(stream, parent);

	childs = scene->sm.deserialize<std::list<GameObject*>>(stream);

	int count = scene->sm.deserialize<int>(stream);
	for (int i = 0; i < count; ++i)
	{
		Component* comp = scene->sm.deserialize<Component>(stream);
		components.insert({ typeid(*comp).hash_code(), comp});
	}
}

void GameObject::OnCollision(GameObject* coll)
{
	for (auto comp : components)
	{
		comp.second->OnCollision(coll);
	}
}

void GameObject::Die()
{
	_isActive = false;
	_isVisible = false;
```

```
	scene->RemoveGameObject(id);
}
```

Файл GameObject.cpp

```
#include "GameObject.h"
#include "Scene.h"


template <typename T>
requires std::derived_from<T, Component>
Component* GameObject::GetComponent() const
{
	Component* result = components.at(typeid(T).hash_code());
	return result;
}

template<class T>
void GameObject::AddComponent(T* component)
{
	if (component->GetGameObject() != this && component->GetGameObject() !=
nullptr) throw std::exception("Wrong game object");
	component->gameObject = this;
	components.insert({ typeid(T).hash_code(), component });
	if (scene != nullptr)
	{
		int a = 0;
		++a;
		if (scene != nullptr)
		{
			if (scene->isActive()) component->Start();
		}
	}
}

template <class T>
void GameObject::RemoveComponent()
{
	delete components[typeid(T).hash_code()];
	components.erase(typeid(T).hash_code());
}

template<class T>
bool GameObject::hasComponent() const
{
	try
	{
		components.at(typeid(T).hash_code());
	}
	catch (std::out_of_range)
	{
		return false;
	}

	return true;
}
```