Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра электронных вычислительных машин
Дисциплина: Программирование на языках высокого уровня

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

Игра «Tetra Peace»

БГУИР КП 1–40 02 01 202 ПЗ

Студент: гр. 250502,
Бекетова М. А.

Руководитель: ассистент каф. ЭВМ
Богдан Е. В.

Минск 2023

Учреждение образования
«Белорусский Учреждение образования
«Белорусский государственный университет информатики
и радиоэлектроники»

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ
Заведующий кафедрой

_____
(подпись)
_____2023 г.

ЗАДАНИЕ
по курсовому проектированию

Студенту *Бекетовой Марии Александровне*

1. Тема проекта *Игра «Tetra Peace»*

2. Срок сдачи студентом законченного проекта *15 декабря 2023 г.*

3. Исходные данные к проекту *Язык программирования – C++, мультимедийная библиотека SFML*

4. Содержание расчетно-пояснительной записки (перечень вопросов, которые подлежат разработке)

1. Лист задания.
2. Введение.
3. Обзор литературы.
 3.1. Обзор методов и алгоритмов решения поставленной задачи.
4. Функциональное проектирование.
 4.1. Структура входных и выходных данных.
 4.2. Разработка диаграммы классов.
 4.3. Описание классов.
5. Разработка программных модулей.
 5.1. Разработка схем алгоритмов (два важных метода).
 5.2. Разработка алгоритмов (описание алгоритмов по шагам, для двух методов).
6. Результаты работы.
7. Заключение

8. Литература
9. Приложения

5. Перечень графического материала (с точным обозначением обязательных чертежей и графиков)

1. Диаграмма классов.

2. Схема алгоритма метода run() класса Game

3. Схема алгоритма метода run() класса GameOver

6. Консультант по проекту (с обозначением разделов проекта) Е. В. Богдан

7. Дата выдачи задания *15.09.2023г*

8. Календарный график работы над проектом на весь период проектирования (с обозначением сроков выполнения и трудоемкости отдельных этапов):

1. Выбор задания. Разработка содержания пояснительной записки. Перечень графического материала к 06.10 – 15 %;

разделы 2, 3 к 27.10 – 10 %;

разделы 4 к 06.11 –20 %;

разделы 5 к 20.11 – 35 %;

разделы 6,7,8,9 к 27.11 – 10 %;

оформление пояснительной записки и графического материала к 01.12.23 – 10 %

Защита курсового проекта с 21.12 по 28.12.23г.

РУКОВОДИТЕЛЬ *Богдан Е. В.*

(подпись)

Задание принял к исполнению *Бекетова М. А.*

(дата и подпись студента)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Игра «Тетрис» – это одна из самых популярных и известных аркадных игр, созданная и разработанная Алексеем Пажитновым в 1984 году.

Данная игра обрела популярность благодаря следующим факторам:

1. Простота и доступность. «Тетрис» прост в понимании и игровой механике, что делает его доступным для широкой аудитории, включая как опытных игроков, так и новичков. Правила игры легко освоить, и играть в неё можно уже при первом запуске.

2. Увлекательность и вызов. Падающие в игре фигуры требуют быстрого принятия решений и точного позиционирования, чтобы максимально заполнить горизонтальные ряды. С каждой заполненной линией сложность игры увеличивается, что поддерживает интерес и мотивацию игрока.

3. Стратегический аспект. Игра также требует стратегического мышления. Игроку необходимо планировать и прогнозировать, как наилучшим образом разместить фигуры на игровом поле, чтобы избежать его полного заполнения, и создать возможность для удаления нескольких линий одновременно (удаление 4 линий одновременно называется «тетрисом» и приносит игроку больше игровых очков).

4. Короткие игровые сессии. Данная игра идеально подходит для коротких игровых сессий. Один игровой цикл может занять всего несколько минут, что делает «Тетрис» удобным для игры в перерывах или в свободное время.

5. Конкуренция и рекорды. В «Тетрисе» зачастую используется система рекордов, которая стимулирует игроков соревноваться между собой и стремиться установить новые рекорды. Это создаёт дополнительный элемент привлекательности и мотивации для игры.

Несмотря на кажущуюся простоту правил, процесс разработки игры «Тетрис» требует определённых навыков, таких как обработка событий, работа с графикой и пользовательским интерфейсом, реализация игровой логики, использование алгоритмов и структур данных.

Таким образом, создание аналога данной игры отлично подходит для демонстрации навыков программирования на высокоуровневом объектно-ориентированном языке программирования C++.

Для расширения возможностей языка будет использована библиотека SFML (Simple and Fast Multimedia Library) – это кроссплатформенная библиотека, которая предоставляет удобный интерфейс для работы с графикой, звуком, управлением и другими аспектами игрового процесса.

В итоге, совместное использование данного стека технологий обеспечивает возможность создания производительной, переносимой и функциональной игры.

1 ПОСТАНОВКА ЗАДАЧИ

Разрабатываемая игра «Tetra Peace» должна предоставлять игрокам увлекательный игровой процесс, чёткую и отзывчивую механику управления, а также должна обладать интуитивным пользовательским интерфейсом с соответствующими игровыми функциональностями.

Необходимо разработать игровой движок, обеспечивающий основные механики и правила игры "Тетрис", создать графический интерфейс с использованием библиотеки SFML для отображения окна приложения, игрового поля, фигур, интерактивных элементов и фоновых объектов, реализовать систему управления, позволяющую игроку взаимодействовать с игровым процессом. Также следует реализовать точную обработку пользовательского ввода, обработку игрового состояния (стартовое меню, непосредственно игровой цикл, окончание игры, пауза и др.), добавить звуковые эффекты и анимацию движения фигур.

Дополнительно нужно рассмотреть добавление таких деталей как счёт, увеличение уровня сложности на основе количества убранных линий, сохранение конечного счёта в файл при завершении игры используя введённое пользователем имя, изменение таблицы лидеров основываясь на данных, сохранённых в файле.

Для реализации программы используется объектно-ориентированный язык программирования C++, мультимедийная библиотека SFML, среда разработки Visual Studio 2022.

2 ОБЗОР ЛИТЕРАТУРЫ

2.1 Анализ существующих аналогов

Тема курсового проекта была выбрана в первую очередь для получения знаний в области разработки десктопных игр с использованием ООП. В процессе создания игры будут затронуты главные принципы ООП, создание графического оформления игры с помощью онлайн-сервиса для разработки интерфейсов Figma [1], использован источник [2] при работе с библиотекой SFML.

2.1.1 Игра Tetris Twist

Tetris Twist – иное видение классической игры. В течение 100 уровней игрок путешествует по различным мегаполисам. Игра вышла в 2016 году и с тех пор считается одной из лучших вариаций незабвенного оригинала.

В процессе игры пользователь может видеть свой текущий счёт, число убранных линий и уровень (с количеством звёзд, набранных за его прохождение). Как и в классической версии, в Tetris Twist отображаются предыдущая и следующая фигуры. Возможен выбор дополнительных модификаций игры. Имеется возможность играть в полноэкранном режиме, приостановить игру или перейти к карте уровней.

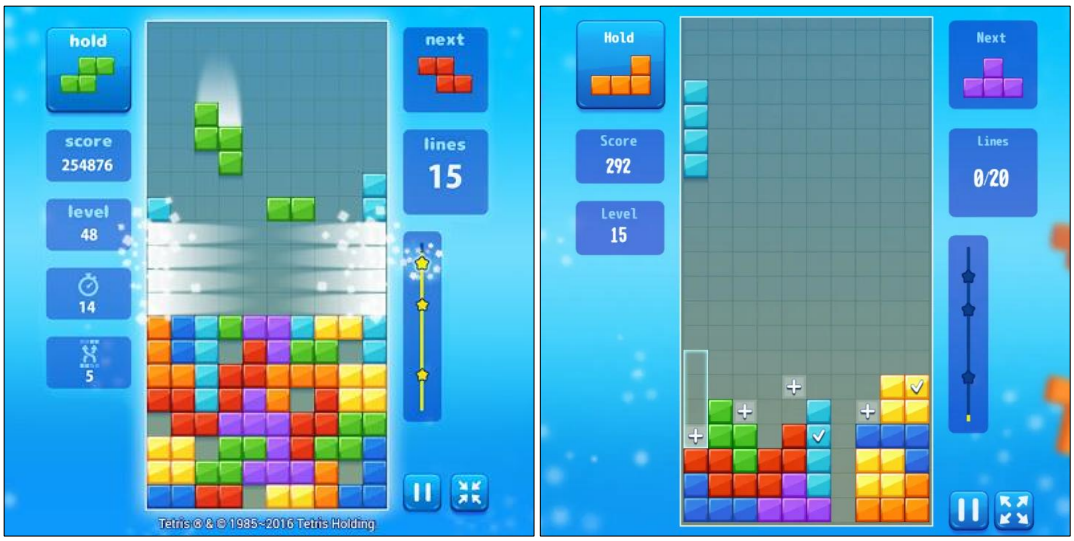

Рисунок 2.1 – Скриншоты Tetris Twist

2.1.2 Игра Tetra Blocks

Tetra Blocks – эта игра, которая также является свежей интерпретацией классики 1984 года от компании Nocodesoft. Особенностью данной версии

является возможность самостоятельного выбора оформления фона игры и текстуры блоков.

Информация выводимая пользователю: нынешний счёт, максимально набранный ранее счёт, суммарный счёт за всё время игры, уровень, количество убранных линий, время, прошедшее с начала игры. Отображаются следующая и предыдущая фигуры.

Есть кнопки выхода в меню настроек, паузы, режима изменения темы, информации по управлению игрой.

Находясь на стартовом окне можно открыть окно со статистикой, список похожих игр и, самое главное, можно начать новую игру.

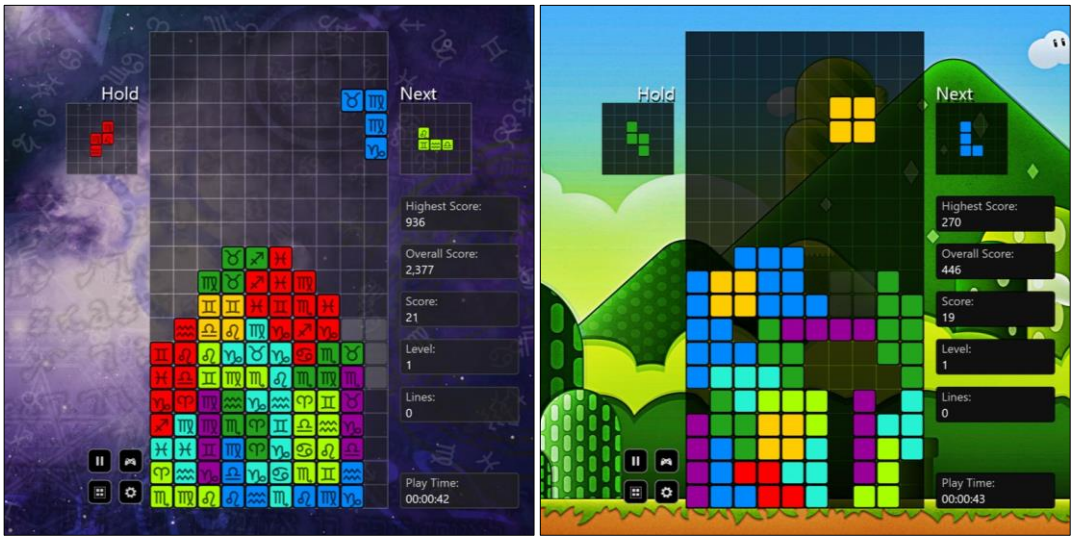

Рисунок 2.2 – Скриншоты Tetra Blocks

2.2 Требования к работе программы

После рассмотрения аналогов данного курсового проекта, можно определить более точные требования к будущему «Тетрису» :

- наличие стартового окна с возможностью начала новой игры, выхода из игры, просмотра таблицы лидеров;
- наличие основного окна игры, которое используется для отображения всей игровой логики и для взаимодействия с пользователем;
- в процессе игры игроку должны выводиться данные о текущем счёте и количестве убранных линий, необходимо наличие окна следующей в очереди фигуры;
- наличие окна конца игры с выводом на экран набранных очков;
- наличие окна с отображением таблицы лидеров.

Дополнительно в игре «Tetra Peace» должна быть разработана функция отображения тени падающей фигуры, что упростит позиционирование фигур при использовании игры для снижения уровня стресса.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе описывается функционирование и структура разрабатываемой игры.

Диаграмма классов представлена в Приложении А.

3.1 Входные и выходные данные

Таблица 3.1 - Файл scores.txt (информация о результатах прошлых игр):

Никнейм	Игровой счет
byaketava	8642

Файл tiles.png хранит изображение клеток фигур.

Файл menuBackground.png хранит фоновое изображение окна главного меню.

Файл gameoverBackground.png хранит фоновое изображение окна конца игры.

Файл leaderboardBackground.png хранит фоновое изображение окна таблицы лидеров.

Файл background.png хранит фоновое изображение окна процесса игры.

Файл gamePaused.png хранит изображение окна паузы.

Файл exitNormal.png хранит изображение нормальной кнопки выхода.

Файл exitPressed.png хранит изображение нажатой кнопки выхода.

Файл leadersNormal.png хранит изображение нормальной кнопки перехода к таблице лидеров.

Файл leadersPressed.png хранит изображение нажатой кнопки перехода к таблице лидеров.

Файл menuNormal.png хранит изображение нормальной кнопки перехода к главному меню.

Файл menuPressed.png хранит изображение нажатой кнопки перехода к главному меню.

Файл pauseNormal.png хранит изображение нормальной кнопки паузы.

Файл pausePressed.png хранит изображение нажатой кнопки паузы.

Файл playNormal.png хранит изображение нормальной кнопки старта игры.

Файл playPressed.png хранит изображение нажатой кнопки старта игры.

Файл shadowNormal.png хранит изображение нормальной кнопки отображения тени фигуры.

Файл shadowPressed.png хранит изображение нажатой кнопки отображения тени фигуры.

Файл volumeon.png хранит изображение нормальной кнопки звука.

Файл volumeoff.png хранит изображение нажатой кнопки звука.

Файл icon.png хранит изображение иконки игры.

Файл Exo-ExtraBold.ttf хранит шрифт, используемый для вывода текста в игре.

Файл background.ogg хранит основное музыкальное сопровождение игры.

Файл button.ogg хранит звук нажатия кнопки в игре.

Файл cleared.ogg хранит звук удаления заполненного ряда.

3.2 Описание классов

3.2.1 Класс Board

Этот класс представляет собой реализацию игрового поля.

Поля:

- `int boardWidth` – ширина поля;
- `int boardHeight`– высота поля;
- `sf::Texture tileset` – текстура клеток;
- `sf::VertexArray vertices` – для представления каждой клетки поля четырьмя вершинами.

Методы:

- `Board(int boardWidth, int boardHeight)` – конструктор класса;
- `Board()`– деструктор класса;
- `void updateTexture(Cell piecePos, Cell shapeCell, int color, int tileSize)` – обновляет текстуру клетки поля;
- `void updateAllTextures(int tileSize)` – обновляет текстуры всех клеток поля;
- `bool collidesWith(int x, int y, Cell* shape)` – проверяет столкновения с другими фигурами/границами поля. Возвращает `false`, если столкновения не обнаружены. Возвращает `true`, если любая из клеток фигуры столкнулась с чем-либо;
- `bool add(Piece* piece)` – добавляет фигуру на поле и обновляет текстуры поля. Возвращает `false`, если игра окончена (фигура была добавлена в запрещенную зону по Y). Возвращает `true`, если фигура была добавлена успешно;
- `int updateBoard()` – удаляет заполненные линии. Возвращает число удалённых за раз линий;

- `void pushRowDown(int row)` – опускает все линии выше удаляемой вниз, обновляет текстуры поля;
- `void draw(sf::RenderTarget& target, sf::RenderStates states) const override` – реализация интерфейса отрисовки SFML.

3.2.2 Класс Button

Этот класс представляет функциональность одиночной кнопки, расположенной в окне.

Поля:

- `sf::Sprite normal` – объект нормальной кнопки;
- `sf::Sprite pressed` – объект нажатой кнопки;
- `sf::Sprite* currentState` – указатель на объект текущего состояния кнопки;
- `bool isPressed` – флаг, определяющий в нажатом ли состоянии находится кнопка, изначально равен `false`;
- `bool isNormal` – флаг, определяющий в нажатом ли состоянии находится кнопка, изначально равен `true`.

Методы:

- `Button(sf::Vector2f position, const sf::Texture& normal, const sf::Texture& pressed)` – конструктор класса. Устанавливает текстуры для нормального и нажатого состояния кнопки;
- `bool Button::mouseOnButton(sf::Vector2f mousePos)` – проверяет наведена ли мышь на кнопку. Возвращает `true`, если мышь наведена на кнопку, иначе возвращает `false`;
- `bool Button::updateButton(sf::Vector2f mousePos)` – изменяет состояние кнопки при нажатии на неё. Возвращает `true`, если произошло нажатие на кнопку, иначе возвращает `false`;
- `void draw(sf::RenderTarget& target, sf::RenderStates states) const override` – реализация интерфейса отрисовки SFML.

3.2.3 Класс Cell

Этот класс представляет одиночную клетку игрового поля.

Поля:

- `int x` – координата x клетки;
- `int y` – координата y клетки.

Методы:

- `Cell()` – конструктор класса;
- `Cell(int x, int y)` – конструктор класса, инициализирующий поля класса определённым значением;
- `int getX() const` – получает координату x;
- `int getY() const` – получает координату y;

- `void setX(int x)` – устанавливает координату x;
- `void setY(int y)` – устанавливает координату y;
- `void setPos(int x, int y)` – устанавливает обе координаты.

3.2.4 Класс Game

Этот класс представляет отдельный игровой цикл, который. Объединяет всю игровую логику. Является отдельным состоянием игры согласно `enum gamePart`.

Поля:

- `int boardWidth` – ширина поля;
- `int boardHeight` – высота поля;
- `PieceCreating* pieceCreating` – указатель на объект класса `PieceCreating` для создания фигур;
- `Board* gameboard` – указатель на объект класса `Board`;
- `bool gameOver` – флаг конца игры, изначально равен `false`;
- `int score` – рейтинг;
- `int level` – уровень, изначально равен 1;
- `int totalRows` – количество убранных линий;
- `bool volume` – флаг звука (on/off), изначально равен `true`;
- `bool shadowFlag` – флаг определяющий отображать или нет тень фигуры, изначально равен `true`;
- `sf::Text scoreText` – объект текста для отображения счёта;
- `sf::Text linesText` – объект текста для отображения количества убранных линий;
- `sf::Font textFont` – объект, хранящий используемый для отображения текста шрифт;
- `int* gamePartPtr` – указатель на выполняемую часть игры;
- `int* scorePtr` – указатель на основную переменную рейтинга;
- `int* levelPtr` – указатель на основную переменную уровня;
- `int* linesPtr` – указатель на основную переменную количества линий;
- `sf::Image icon` – иконка игры;
- `sf::RenderWindow* window` – указатель на окно SFML;
- `sf::Sprite pauseWindow` – спрайт окна паузы;
- `sf::Texture pauseWindowTexture` – текстура окна паузы;
- `Button* pauseButton` – указатель на объект кнопки паузы;
- `Button* volumeButton` – указатель на объект кнопки звука;
- `Button* shadowButton` указатель на объект кнопки тени.

Методы:

- `bool moveLeft (Piece* piece, Piece* shadowPiece)` – отвечает за движение фигуры влево, `true` – успешно, `false` – нет;

- `bool moveRight(Piece* piece, Piece* shadowPiece)` – отвечает за движение фигуры вправо, `true` – успешно, `false` – нет;
- `bool rotate(Piece* piece, Piece* shadowPiece)` – отвечает за поворот фигуры, `true` – успешно, `false` – нет;
- `bool fallDown(Piece* piece)` – отвечает за опускание фигуры вниз на 1 клетку, `true` – успешно, `false` – нет;
- `void setShadowPosition(Piece* currentPiece, Piece* shadowPiece)` – устанавливает позицию тени;
- `void updateScore(int clearedRows)` – измененяет рейтинг относительно количества удалённых строк;
- `void updateLevel()` – измененяет рейтинг относительно количества удалённых строк;
- `Game(sf::RenderWindow* window, int boardWidth, int boardHeight, bool volume, int* gamePartPtr, int* scorePtr, int* levelPtr, int*linesPtr, bool shadowFlag)` – конструктор класса;
- `~Game()` – деструктор класса;
- `void run()` – основной цикл игры;
- `void setScore(int score)` – выводит текущий рейтинг на экран;
- `void setLines(int lines)` – выводит текущее количество убранных линий на экран.

3.2.5 Класс GameOver

Этот класс представляет окно конца игры. Является отдельным состоянием игры согласно `enum gamePart`.

Поля:

- `sf::RenderWindow* window` – указатель на окно SFML;
- `sf::Font textFont` – объект, хранящий используемый для отображения текста шрифт;
- `sf::Text nickText` – объект текста для отображения никнейма, вводимого игроком;
- `sf::Text scoreText` – объект текста для отображения счёта;
- `sf::Text linesText` – объект текста для отображения количества убранных линий;
- `Button* menuButton` – указатель на объект кнопки перехода в основное меню;
- `Button* playButton` – указатель на объект кнопки старта игры;
- `Button* leaderboardButton` – указатель на объект кнопки тени;
- `int* gamePartPtr` – указатель на выполняемую часть игры;
- `int* scorePtr` – указатель на основную переменную рейтинга;
- `int* linesPtr` – указатель на основную переменную количества линий;

- `std::string filename` – переменная, содержащая название файла;
- `std::string nick` – переменная, хранящая никнейм, вводимый игроком.

Методы:

- `GameOver(sf::RenderWindow* window, std::string filename, int* gameStatePtr, int* scorePtr, int* linesPtr)` – конструктор класса;
- `~GameOver()` – деструктор класса;
- `void run()` – главный цикл конца игры (вывод результатов игры, ввод пользователем никнейма для их сохранения);
- `void addScore()` – ввод нового рейтинга в файл.

3.2.6 Класс GameSound

Этот класс отвечает за взаимодействие со звуковыми эффектами.

Поля:

- `static const int n` – константа, хранящая количество звуков в игре;
- `std::array<sf::Sound,n> gSounds` – массив объектов звуковых эффектов;
- `static GameSound* sInstance` – отвечает за наличие лишь одного менеджера звуковых эффектов во всей игре.

Методы:

- `GameSound()` – конструктор класса;
- `static void play(int index)` – функция включения звука;
- `static void stop(int index)` – функция выключения звука.

3.2.7 Класс LeaderBoard

Этот класс представляет окно таблицы лидеров. Является отдельным состоянием игры согласно `enum gamePart`.

Поля:

- `sf::RenderWindow* window` – указатель на окно SFML;
- `Button* menuButton` – указатель на объект кнопки перехода в основное меню;
- `std::vector<SingleScore*> scores` – вектор, хранящий указатели на единичные записи рейтинга;
- `int* gamePartPtr` – указатель на выполняемую часть игры;
- `int* scorePtr` – указатель на основную переменную рейтинга.

Методы:

- `LeaderBoard(sf::RenderWindow* window, std::string filename, int* gamePartPtr, int* scorePtr)` – конструктор класса;

- `~LeaderBoard()` – деструктор класса;
- `void run()` – главный цикл таблицы лидеров (вывод десяти самых высоких записей рейтинга);
- `void sortScores()` – сортировка рейтингов.

3.2.8 Класс Menu

Этот класс представляет главное меню игры. Является отдельным состоянием игры согласно `enum gamePart`.

Поля:

- `sf::RenderWindow* window` – указатель на окно SFML;
- `Button* startButton` – указатель на объект кнопки старта игры;
- `Button* exitButton` – указатель на объект кнопки выхода из игры;
- `Button* leaderboardButton` – указатель на объект кнопки перехода к таблице лидеров;
- `int* gamePartPtr` – указатель на выполняемую часть игры.

Методы:

- `Menu(sf::RenderWindow* window, int* gamePartPtr)` – конструктор класса;
- `~Menu()` – деструктор класса;
- `void run()` – главный цикл меню игры.

3.2.9 Класс Resources

Этот класс отвечает за загрузку ресурсов, таких как текстуры, звуки, шрифты.

Поля:

- `std::map< std::string, sf::Texture> textures` – карта, содержащая все текстуры, загруженные в игру;
- `std::map <std::string, sf::Font> fonts` – карта, содержащая все шрифты, загруженные в игру;
- `std::map <std::string, sf::SoundBuffer> sounds` – карта, содержащая все звуки, загруженные в игру;
- `static Resources* sInstance` – отвечает за наличие лишь одного менеджера ресурсов во всей игре.

Методы:

- `static sf::Texture& getTexture(std::string const& filename)` – добавляет текстуру в карту текстур;
- `static sf::SoundBuffer& getSound(std::string const& filename)` – добавляет звук в карту звуков;
- `static sf::Font& getFont(std::string const& filename)` – добавляет шрифт в карту шрифтов.

3.2.10 Класс SingleScore

Этот класс представляет единичную запись рейтинга.

Поля:

- `sf::Font font` – объект, хранящий используемый для отображения текста шрифт;
- `sf::Text nickText` – объект текста для отображения никнейма;
- `sf::Text scoreText` – объект текста для отображения счёта;
- `std::string nick` – переменная, хранящая никнейм;
- `int score` – переменная, хранящая счёт.

Методы:

- `SingleScore(const std::string& nick, int score)` – конструктор класса;
- `long getScore() const` – возвращает значение рейтинга;
- `void update(int x, int y)` – устанавливает позицию рейтинга на экране;
- `void draw(sf::RenderTarget& target, sf::RenderStates states) const override` – реализация интерфейса отрисовки SFML.

3.2.11 Класс NextPieceBoard

Этот класс представляет поле следующей фигуры.

Поля:

- `Piece* piece` – указатель на объект фигуры в окне следующей фигуры;
- `int tileSize` – переменная, хранящая размер единичной плитки;
- `sf::Text scoreText` – объект текста для отображения счёта;
- `std::string nick` – переменная, хранящая никнейм;
- `int score` – переменная, хранящая счёт.

Методы:

- `NextPieceBoard(Piece* piece, int tileSize)` – конструктор класса;
- `void setPiece(Piece* piece)` – устанавливает позицию следующей фигуры в поле предпросмотра фигуры;
- `void draw(sf::RenderTarget& target, sf::RenderStates states) const override` – реализация интерфейса отрисовки SFML.

3.2.12 Класс Piece

Этот класс представляет единичную фигуру. Является базовым для восьми классов наследников, представляющих различные варианты игровой фигуры.

Поля:

- `Cell piecePosition` – позиция фигуры на поле;
- `int rotation` – текущий поворот фигуры. У каждой фигуры есть 4 варианта вращения, каждый вариант описан в массиве Cell shapes[4][4];
- `int color` – цвет текущей фигуры (число определяет смещение в tileset), 0/9 – прозрачный блок, 1-7 - цветные блоки, 8 – блок тени;
- `int currentPiece` – текущая фигура (её представление в перечислении `pieceEnum`);
- `Cell shape[4]` – текущая форма описывается 4 клетками, каждый поворот определяетя одним из четырёх вариантов массива Cell shapes[4][4];
- `Cell shapes[4][4]` – основной массив, хранящий положение всех клеток фигуры при разных вариантах вращений;
- `sf::Texture tiles` – объект текстуры блоков.
- `sf::Sprite tileSprite[4]` – массив 4 спрайтов для представления одной фигуры.

Методы:

- `Piece(int rotation, int currentPiece, Cell* shapes, int spawnX = 4)` – конструктор класса, где `spawnX` позиция появления фигуры;
- `void setCurrentShape()` – устанавливает текущую форму фигуры (положение каждого из 4 блоков) в зависимости от поворота;
- `void draw(sf::RenderTarget& target, sf::RenderStates states) const override` – реализация интерфейса отрисовки SFML.
- `void setShapes(Cell* newShapes)` – копирует все формы фигуры в массив;
- `Cell* const getShapes() const` – возвращает массив всех поворотов;
- `int getRotation() const` – возвращает текущий поворот фигуры (0-3);
- `void setRotation(int rotation)` – устанавливает желаемый поворот фигуры;
- `int getCurrentColor() const` – возвращает текущий цвет фигуры;
- `int getCurrentShapeInt() const` – возвращает текущий тип фигуры (согласно перечислению `pieceEnum`);
- `const Cell getPiecePosition() const` – возвращает текущую позицию фигуры на поле;
- `void setPiecePosition(int x, int y, bool fixedToBoard)` – устанавливает позицию фигуры (позицию 4 спрайтов) на желаемую. Перегрузка для поля следующей фигуры;
- `void setPiecePosition(int x, int y)` – устанавливает позицию фигуры (позицию 4 спрайтов) на желаемую. Перегрузка для основного поля;

- `void setPiecePosition(Cell x)` – устанавливает позицию фигуры (позицию 4 спрайтов) на желаемую. Перегрузка для тени;
- `void rotate()` – поворот фигуры (изменение поля `rotation`);
- `Cell* getCurrentShape()` – возвращает текущую форму фигуры;
- `Cell* getRotationShape()` – возвращает форму фигуры после поворота.

3.2.13 Классы IPiece, JPiece, LPiece, OPiece, SPiece, TPiece, ZPiece и ShadowPiece

Данные классы являются наследниками класса `Piece`. Они используется для создания фигур разных форм и тени данных фигур.

Поле:

- `static Cell cells[4][4]` – массив, определяющий положение спрайтов при всех вариантах поворота фигуры. В классе `ShadowPiece` данное поле отсутствует, так как фигура тени использует данные, полученные от основной фигуры;

В каждом классе определяется только конструктор класса, инициализирующий поля базового класса

3.2.14 Класс PieceCreating

Этот класс реализует процесс создания случайных фигур для появления на поле.

Поля:

- `std::vector <int> pieces` – вектор, хранящий представления фигур согласно перечислению `pieceEnum`, для последующего их вывода на игровое поле;
- `int spawnX` – координата x появления фигуры на пол.

Методы:

- `void fillVector()` – заполняет вектор всеми возможными фигурами, а затем перемешивает их;
- `PieceCreating(int spawnX)` – конструктор класса;
- `Piece* getPiece()` – возвращает случайную фигуру из вектора, а затем удаляет её оттуда. Если вектор при этом полностью освобождается, то вызывается метод `fillVector()`;
- `Piece* getShadowPiece(Piece* currentPiece)` – возвращает объект текущей фигуры тени.

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

В данном разделе рассмотрены описания алгоритмов, используемых в программе.

4.1 Алгоритм метода run() класса Game

Шаг 1. Инициализация объекта игрового поля используя метод `initialization()` класса `Board`;

Шаг 2. Инициализация фоновой текстуры;

Шаг 3. Инициализация объектов класса `Piece`, представляющих собой начальные игровые фигуры (`currentPiece`, `nextPiece`, `shadowPiece`);

Шаг 4. Инициализация окна следующей фигуры (`nextBoard`);

Шаг 5. Инициализация флагов ускоренного падения, мгновенного падения, звука;

Шаг 6. Инициализация часов для обработки пользовательского ввода и контроля игрового процесса;

Шаг 7. Запуск музыкального сопровождения;

Шаг 8. Проверка состояния окна (открыто ли оно);

Шаг 9. Если окно закрыто, переход к шагу 18;

Шаг 10. Если окно открыто, производится проверка состояния флага текущей части, в случае равенства его значению `MENU` или `GAMEOVER` осуществляется переход к соответствующим алгоритмам классов `Menu` и `GameOver`;

Шаг 11. Пока очередь событий не пуста, выполняется обработка событий очереди;

Шаг 12. Обработка нажатия кнопок внутри игры (позволяют изменить состояние флага `shadowFlag`, значение флага `volume`, значение флага `pausedFlag`);

Шаг 13. Обработка нажатия кнопок клавиатуры (аналогичное шагу выше изменение флагов, поворот фигуры, выход в основное меню, изменение флагов `fastFallFlag` и `instantFall`);

Шаг 14. При пустой очереди событий производится проверка флага `pausedFlag`;

Шаг 15. Если игра не приостановлена, выполняется обработка нажатия кнопок клавиатуры (движение фигуры влево и вправо, изменение флагов ускоренного `fastFallFlag` и мгновенного `instantFall` падений);

Шаг 16. Если установлен `pausedFlag` или выполнен шаг 15 происходит отрисовка всех игровых объектов (фон, фигуры, кнопки, окно паузы);

Шаг 17. Совершается переход к шагу 8;

Шаг 18. Завершается алгоритм, передача управления вызывающей функции.

Схема алгоритма представлена в Приложении Б.

4.2 Алгоритм метода run() класса GameOver

Шаг 1. Инициализация фоновой текстуры;

Шаг 2. Проверка состояния окна (открыто ли оно);

Шаг 3. Если окно закрыто, переход к шагу 14;

Шаг 4. Если окно открыто, производится проверка состояния флага текущей части, в случае не равенства его значению `GAMEOVER` осуществляется занесение единичной записи рейтинга в файл и переход к соответствующим состоянию флага алгоритмам других классов;

Шаг 5. Пока очередь событий не пуста, выполняется обработка ввода пользователем текста;

Шаг 6. Обработка нажатия кнопок внутри игры (позволяют изменить состояние флага `shadowFlag`, значение флага `volume`, значение флага `pausedFlag`);

Шаг 7. Если была нажата кнопка backspace клавиатуры удаляется последний символ из строки `nick`;

Шаг 8. Если символ отличен от пробела и находится в пределах A-z, а также длина строки `nick` не более 15 символов, к переменной `nick` дописывается введённый символ;

Шаг 9. Инициализация строки объекта текста `nickText` и установка его позиции на экране;

Шаг 10. Если установлен `pausedFlag` или выполнен шаг 15 происходит отрисовка всех игровых объектов (фон, фигуры, кнопки, окно паузы);

Шаг 11. Обработка нажатия кнопок внутри игры (позволяют перейти к главному меню, начать новую игру и перейти к просмотру таблицы лидеров);

Шаг 12. Отрисовка всех игровых объектов (фон, текст для отображения счёта, количества убранных линий, введённого пользователем ника, кнопки меню, старта, таблицы лидеров);

Шаг 13. Совершается переход к шагу 2;

Шаг 14. Завершается алгоритм, передача управления вызывающей функции.

Схема алгоритма представлена в Приложении В.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

5.1 Системные требования

Данная игра разработана в Visual Studio 2022 на 64–разрядной операционной системе Windows 11. Процессор AMD Ryzen 5 5500U, размер оперативной памяти 16 Гб. Для нормальной работы данной игры требуется не менее 120 Мб свободной оперативной памяти.

5.2 Игровой процесс

Для запуска игры необходимо открыть исполняемый файл TETRA PEACE.exe (Рисунок 5.1).

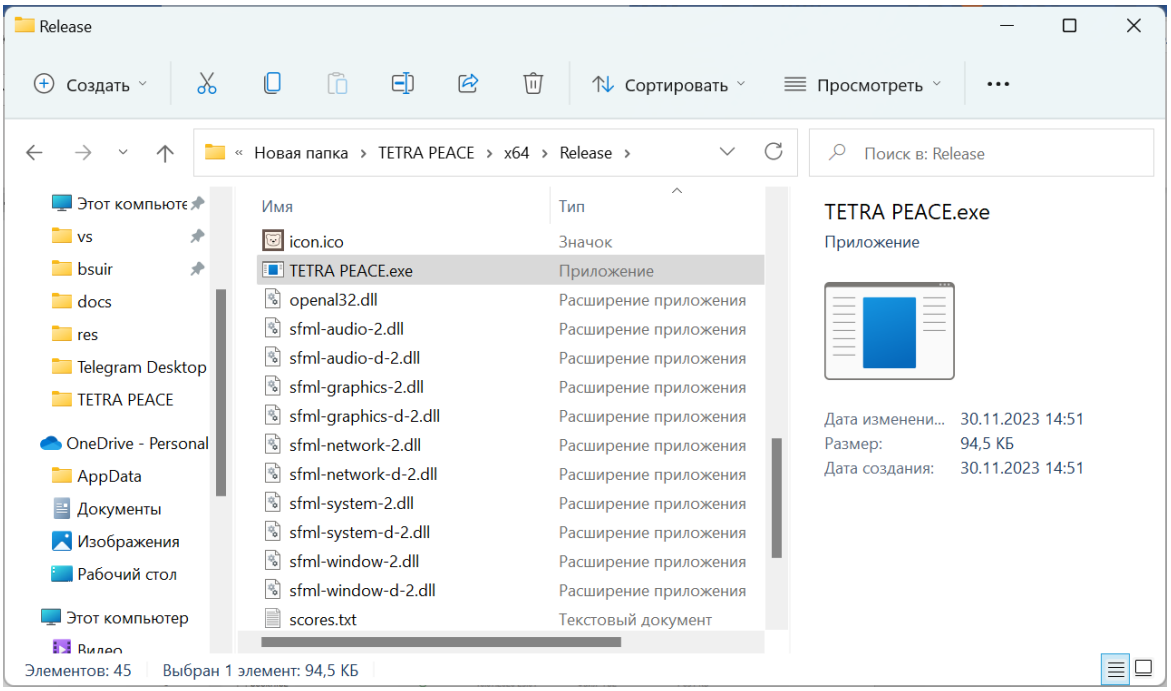

Рисунок 5.1 – Запуск игры.

После этого откроется главное меню игры с возможностью старта игры, просмотра таблицы лидеров и выхода (Рисунок 5.2).

Рисунок 5.2 – Главное меню игры.

При нажатии на кнопку старта игры откроется основное игровое окно, с которым будет взаимодействовать игрок (изменения режима игры, изменение громкости, управление фигурами) (Рисунок 5.3 и Рисунок 5.4).

На рисунках отображена одна и таже игровая сессия с активным или нет режимом тени падающей фигуры.

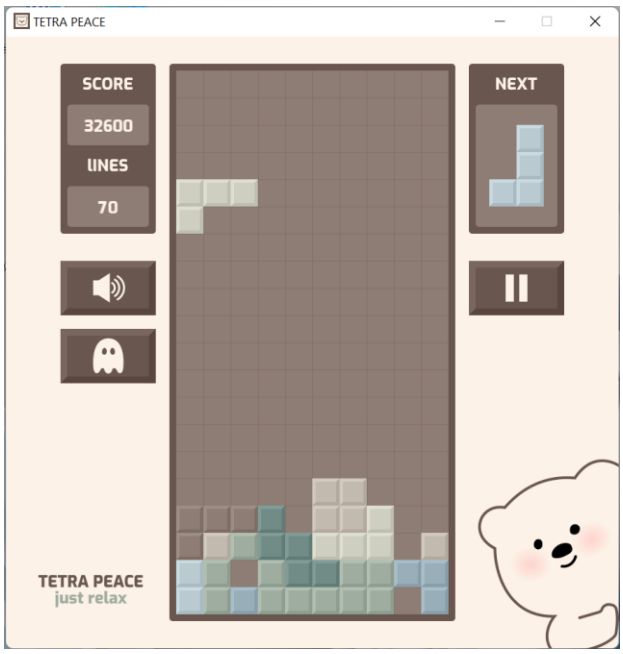

Рисунок 5.3 – Основное игровое окно (с тенью фигуры).

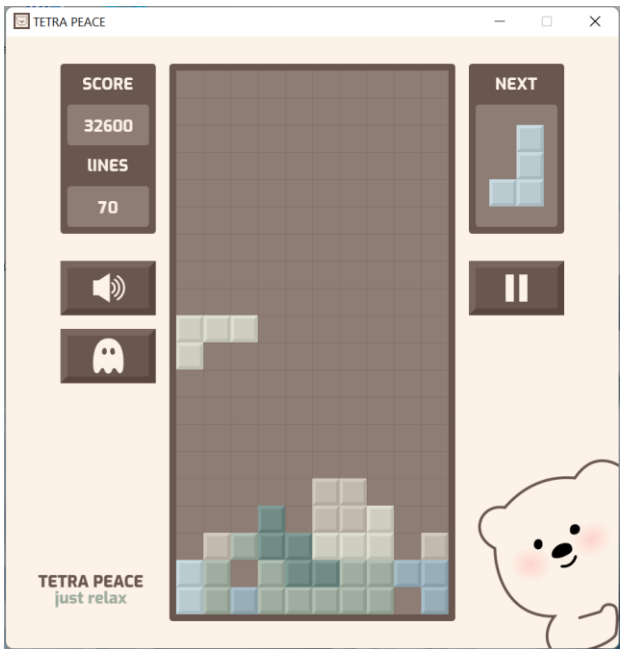

Рисунок 5.4 – Основное игровое окно (без тени фигуры).

Также в «Tetra Peace» возможно приостановить игру. Тогда пользователь увидит следующее окно (Рисунок 5.5):

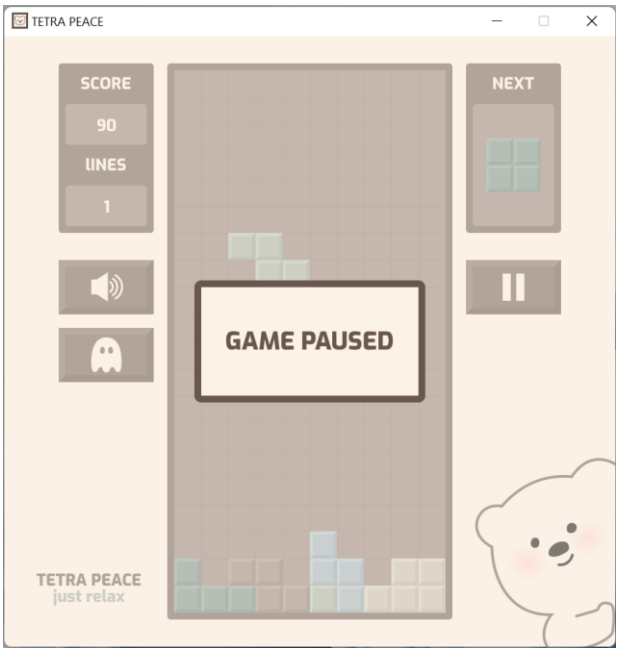

Рисунок 5.5 – Окно паузы.

После окончания игры выполняется переход к окну конца игры. Пользователю предлагается ввести никнейм для сохранения результата в таблице лидеров (Рисунок 5.6 и Рисунок 5.7).

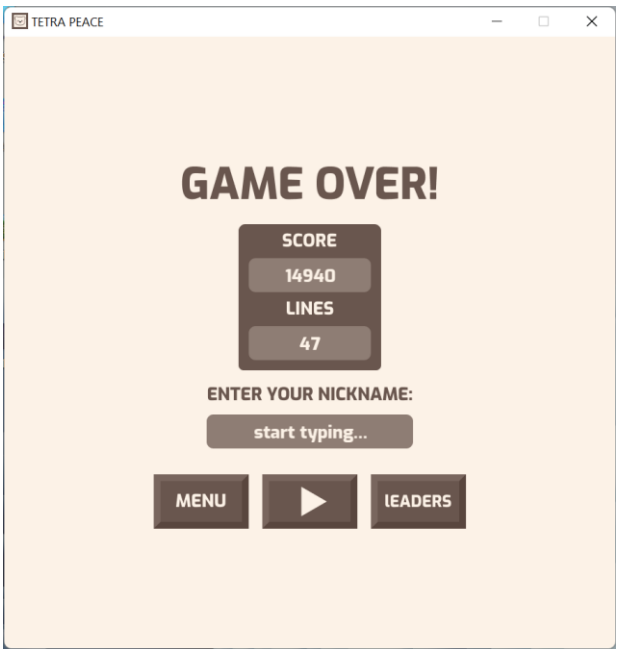

Рисунок 5.6 – Окно конца игры.

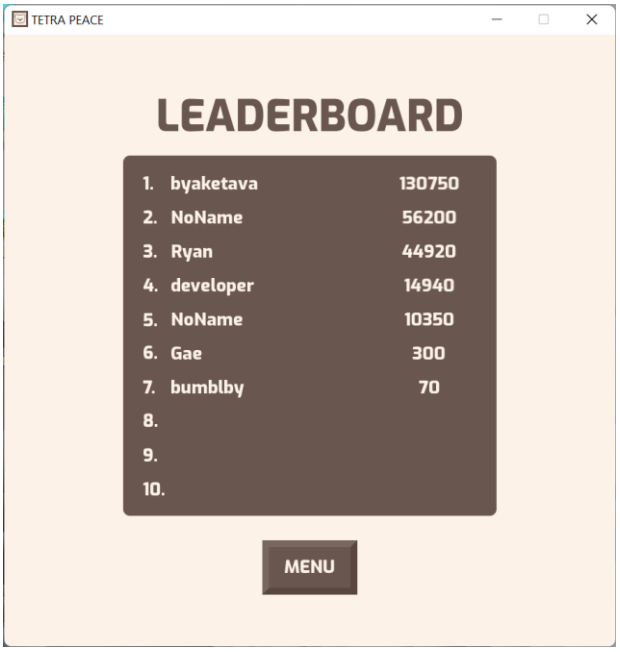

Рисунок 5.7 – Окно таблицы лидеров.

ЗАКЛЮЧЕНИЕ

Данный курсовой проект представляет собой разработанную игру «Tetra Peace», которая обладает полным игровым функционалом и интуитивно понятным пользовательским интерфейсом. В ходе его создания были успешно достигнуты поставленные цели, и весь запланированный функционал был реализован.

На глубоком уровне были изучены возможности для создания игр, которые предлагает мультимедийная библиотека SFML. Однако основное внимание было уделено языку программирования C++. В процессе разработки игры были усвоены основы объектно-ориентированного программирования (ООП), что позволило эффективно использовать его возможности на различных этапах написания кода.

Работа над проектом была разбита на этапы: анализ аналогов и предпочтений пользователей, постановка требований, проектирование, конструирование, разработка программного продукта и его тестирование. Благодаря последовательности выполнения каждого этапа был достигнут желаемый результат, а именно – аналог классической игры «Тетрис».

В будущем планируется улучшение текущего функционала. Планируется добавить поддержку 2 игроков, дополнительные блоки (взрывной блок и блок-стена), поддержку различного цветового оформления, а также улучшить звуковое сопровождение игры.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Figma: The Collaborative Interface Design Tool [Электронный ресурс]. -Электронные данные. -Режим доступа: https://www.figma.com/ - Дата доступа: 27.11.2023
2. Documentation of SFML 2.6.1 [Электронный ресурс]. Электронные данные. -Режим доступа: https://www.sfml-dev.org/documentation/2.6.1/ - Дата доступа: 27.11.2023
3. Настройка библиотеки SFML в Visual Studio [Электронный ресурс]. Электронные данные. -Режим доступа: https://habr.com/ru/articles/703500/ - Дата доступа: 27.11.2023
4. The C++ Programming Language, – Bjarne Stroustrup, 1985
5. The C Programming Language. 2nd Edition, –Dennis Ritchie, Brian Kernighan, 1978

ПРИЛОЖЕНИЕ А
(обязательное)

Диаграмма классов

ПРИЛОЖЕНИЕ Б

(обязательное)

Схема алгоритма метода run() класса Game

ПРИЛОЖЕНИЕ В
(обязательное)

Схема алгоритма метода run() класса GameOver

ПРИЛОЖЕНИЕ Г
(обязательное)

Листинг кода

Файл Board.cpp:

```
#include "Board.h"
void Board::initialization(int tileSize) {
  // позиция доски
  this->setPosition(BOARD_OFFSET_X, BOARD_OFFSET_Y);
  // изменение массива вершин
  vertices.setPrimitiveType(sf::Quads);
  vertices.resize(this->boardWidth * this->boardHeight * 4);
  // инициализация текстур
  updateAllTextures(tileSize); }
void Board::draw(sf::RenderTarget& target, sf::RenderStates states)
const        {
  states.transform *= getTransform();
  states.texture = &tileset;
  target.draw(vertices, states); }
Board::Board(int boardWidth, int boardHeight)  {
  this->boardWidth = boardWidth;
  this->boardHeight = boardHeight;
  this->board = new int* [this->boardWidth];
  for (int i = 0; i < this->boardWidth; i++)
    this->board[i] = new int[boardHeight];
  for (int i = 0; i < this->boardWidth; i++)  {
    for (int j = 0; j < this->boardHeight; j++)  {
      if (i == 0 || i == this->boardWidth - 1 || j == this-
>boardHeight - 1)  {
        board[i][j] = WALL; }
      else  {
        board[i][j] = NONE; }}}}
bool Board::collidesWith(int x, int y, Cell* shape) {
  // каждые 4 клетки фигуры
  for (int i = 0; i < 4; i++) {
    // если какая-либо клетка не пуста вернуть истину (столкновение
обнаружено)
    if (this->board[x + shape[i].getX()][y + shape[i].getY()] !=
NONE)
      return true; }
  // если столкновения не обнаружено вернуть ложь
  return false; }
bool Board::add(Piece* piece) {
  // каждая клетка фигуры заполняется числом цвета фигуры
  for (int i = 0; i < 4; i++)  {
    // если клетка в зоне появления фигур (~4 блока в высоту,
центральные по иксу), вернуть false (gameover)
    if ((piece->getPiecePosition().getY() + piece-
>getCurrentShape()[i].getY() < 4) &&
```

```
      (piece->getPiecePosition().getX() + piece-
>getCurrentShape()[i].getX() >4 && piece->getPiecePosition().getX()
+ piece->getCurrentShape()[i].getX() < 7))
      return false;
    // иначе заполнить клетку цветом фигуры (число)
    board[piece->getPiecePosition().getX() + piece-
>getCurrentShape()[i].getX()]
       [piece->getPiecePosition().getY() + piece-
>getCurrentShape()[i].getY()] = piece->getCurrentColor();
    // обновить текстуру клетки
    updateTexture(piece->getPiecePosition(), piece-
>getCurrentShape()[i], piece->getCurrentColor(), 40); }
  return true; }
void Board::updateTexture(Cell piecePos, Cell shapeCell, int color,
int tileSize) {
  // обновить текстуру одиночной клетки
  sf::Vertex* quad = &vertices[(piecePos.getX() + shapeCell.getX()
+ (piecePos.getY() + shapeCell.getY()) * this->boardWidth) * 4];
  quad[0].texCoords = sf::Vector2f(color * tileSize, 0);
  quad[1].texCoords = sf::Vector2f((color + 1) * tileSize, 0);
  quad[2].texCoords = sf::Vector2f((color + 1) * tileSize,
tileSize);
  quad[3].texCoords = sf::Vector2f(color * tileSize, tileSize); }
int Board::updateBoard()  {
  bool FullRow = true;
  int lineCounter = 0;
  // поиск с нижней строки (не учитывая границу) двигаясь вверх
  for (int y = this->boardHeight-2; y > 0; y--)  {
    FullRow = true;
    for (int x = 1; x < this->boardWidth - 1; x++)  {
      // если хотя бы одна клетка пуста - линия не заполнена
      if (board[x][y] == NONE)
        FullRow = false; }
    if (FullRow) {
      pushRowDown(y);
      lineCounter++;
      // возвращаем поиск в то же место, на случай если следующая
строка тоже заполнена
      y++; }}
  return lineCounter; }
void Board::pushRowDown(int row)  {
  for (int y = row; y > 0; y--)  {
    for (int x = 1; x < this->boardWidth - 1; x++)  {
      board[x][y] = board[x][y - 1];
      updateAllTextures(40); { { {
void Board::updateAllTextures(int tileSize)  {
  // не отрисовываются стены
  for (int i = 1; i < this->boardWidth - 1; i++)  {
    for (int j = 0; j < this->boardHeight-1; j++)  {
      // число в клетке - её цвет
      int tileNumber = board[i][j];
      // указатель на текущий квадрат
```

```
      sf::Vertex* quad = &vertices[(i + j * this->boardWidth) * 4];
      // определение его 4 вершин
      quad[0].position = sf::Vector2f(i * tileSize, j * tileSize);
      quad[1].position = sf::Vector2f((i + 1) * tileSize, j *
tileSize);
      quad[2].position = sf::Vector2f((i + 1) * tileSize, (j + 1) *
tileSize);
      quad[3].position = sf::Vector2f(i * tileSize, (j + 1) *
tileSize);
      // определение его 4х координат текстуры
      quad[0].texCoords = sf::Vector2f(tileNumber * tileSize, 0);
      quad[1].texCoords = sf::Vector2f((tileNumber + 1) * tileSize,
0);
      quad[2].texCoords = sf::Vector2f((tileNumber + 1) * tileSize,
tileSize);
      quad[3].texCoords = sf::Vector2f(tileNumber * tileSize,
tileSize);     }}}
Board::~Board()  {
  for (int i = 0; i < this->boardWidth; i++)
    delete[] this->board[i];
  delete[] this->board; }
```

Файл Board.h:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "Enums.h"
#include "Cell.h"
#include "Piece.h"
#define BOARD_OFFSET_X 210
#define BOARD_OFFSET_Y 50
class Board : public sf::Drawable, sf::Transformable  {
private:
  int boardWidth;   // ширина поля
  int boardHeight;  // высота поля
  int** board;    // массив поля
  // текстура клеток
  sf::Texture tileset = Resources::getTexture("res/tiles.png");
  // каждая клетка доски будет представлена 4 вершинами
  sf::VertexArray vertices;
  // обновляет текстуру клетки доски
  void updateTexture(Cell piecePos, Cell shapeCell, int color, int
tileSize);
  // обновляет все клетки
  void updateAllTextures(int tileSize);
public:
  // конструктор класса
  Board(int boardWidth, int boardHeight);
  // деструктор класса
  ~Board();
  // создает массив вершин, представляющий доску.
  // добавляет текстуры спользуя метод UpdateAllTextures()
  void initialization(int tileSize);
```

```
  // проверка столкновений с другими фигурами/границами доски
  // возвращает false, если столкновений не обнаружено
  // возвращает true, если любая из клеток фигуры столкнулась с
чем-либо
  bool collidesWith(int x, int y, Cell* shape);
  // добавляет фигуру на доску и обновляет текстуру доски
  // возвращает false, если игра окончена (фигура была добавлена в
запрещенную зону по Y)
  // возвращает true, если фигура была добавлена успешно
  bool add(Piece* piece);
  // удаляет заполненные линии, возвращает число удалённых за раз
линий
  int updateBoard();
  // опускает все линии выше удаляемой вниз, обновляет текстуры
  void pushRowDown(int row);
protected:
  // отрисовывает поле
  void draw(sf::RenderTarget& target, sf::RenderStates states)
const override;
};
```

Файл Button.cpp:

```
#include "Button.h"
Button::Button(sf::Vector2f position, const sf::Texture& normal,
const sf::Texture& pressed) {
  this->normal.setTexture(normal);
  this->pressed.setTexture(pressed);
  this->normal.setPosition(position);
  this->pressed.setPosition(position);
  currentState = &this->normal;  }
void Button::draw(sf::RenderTarget& target, sf::RenderStates
states) const  {
  target.draw(*currentState); }
bool Button::mouseOnButton(sf::Vector2f mousePos) {
  return currentState->getGlobalBounds().contains(mousePos);  }
bool Button::updateButton(sf::Vector2f mousePos)  {
  if (mouseOnButton(mousePos)) {
    if (sf::Mouse::isButtonPressed(sf::Mouse::Button::Left)) {
      isPressed = true;
      isNormal = false;
      currentState = &this->pressed;}
    else if (isPressed)  {
      isPressed = false;
      isNormal = true;
      currentState = &this->normal;
      return true; }}
  else if (isPressed){
    isPressed = false;
    isNormal = true;
    currentState = &this->normal; }
  return false; }
```

Файл Button.h:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "Resources.h"
class Button : public sf::Drawable {
private:
  sf::Sprite normal;          // объект нормальной кнопки
  sf::Sprite pressed;         // объект нажатой кнопки
  sf::Sprite* currentState;       // указатель на текущий вид
кнопки
  bool isPressed = false;
  bool isNormal = true;
protected:
  void draw(sf::RenderTarget& target, sf::RenderStates states)
const override;
public:
  Button(sf::Vector2f position, const sf::Texture& normal, const
sf::Texture& pressed);
  bool updateButton(sf::Vector2f mousePos);
  bool mouseOnButton(sf::Vector2f mousePos);
};
```

Файл Cell.cpp:

```
#include "Cell.h"
#include <iostream>
Cell::Cell(int x, int y)  : x(x), y(y) {}
Cell::Cell()        : x(0), y(0) {}
int Cell::getX() const {
  return x; }
int Cell::getY() const {
  return y; }
void Cell::setX(int x) {
  Cell::x = x; }
void Cell::setY(int y) {
  Cell::y = y; }
void Cell::setPos(int x, int y) {
  Cell::x = x;
  Cell::y = y; }
```

Файл Cell.h:

```
#pragma once
class Cell {
private:
  int x;
  int y;
public:
  Cell();
  Cell(int x, int y);
  int getX() const;
  int getY() const;
  void setX(int x);
  void setY(int y);
```

```
  void setPos(int x, int y);
};
```

Файл Enums.h:

```
#pragma once
enum gamePart {
  MENU, GAME, GAMEOVER, LEADERBOARD, EXIT
};
enum pieceEnum {
  NONE = 0, IP, JP, LP, OP, SP, TP, ZP, SHADOW, WALL
};
```

Файл Game.cpp:

```
#include"Game.h"
Game::Game  (sf::RenderWindow* window, int boardWidth, int
boardHeight,
      bool volume, int* gamePartPtr, int* scorePtr,
      int* levelPtr, int* linesPtr, bool shadowFlag)  {
  // ширина + 2 (боковые границы)
  this->boardWidth  = boardWidth + 2;
  // высота + 1 (нижняя граница)
  this->boardHeight   = boardHeight + 1;
  // инициализация объектов классов поля и создания фигур
  this->gameBoard   = new Board(this->boardWidth, this-
>boardHeight);
  this->pieceCreating = new PieceCreating(this->boardWidth / 2 -
2);
  // определение всех указателей
  this->window    = window;
  this->volume    = volume;
  this->gamePartPtr   = gamePartPtr;
  this->scorePtr    = scorePtr;
  this->levelPtr    = levelPtr;
  this->linesPtr    = linesPtr;
  this->shadowFlag  = shadowFlag;
  // отображение рейтинга
  scoreText.setFont(textFont);
  scoreText.setCharacterSize(24);
  scoreText.setFillColor(sf::Color(252, 242, 231, 255));
  setScore(score);
  // отображение количества убранных линий
  linesText.setFont(textFont);
  linesText.setCharacterSize(24);
  linesText.setFillColor(sf::Color(252, 242, 231, 255));
  setLines(totalRows);
  // определение окна паузы
  pauseWindowTexture = Resources::getTexture("res/gamePaused.png");
  pauseWindow.setTexture(pauseWindowTexture);
  // кнопки
  pauseButton = new Button(sf::Vector2f(680, 330),
Resources::getTexture("res/pauseNormal.png"),
Resources::getTexture("res/pausePressed.png"));
```

```
  volumeButton = new Button(sf::Vector2f(80, 330),
Resources::getTexture("res/volumeon.png"),
Resources::getTexture("res/volumeoff.png"));
  shadowButton = new Button(sf::Vector2f(80, 430),
Resources::getTexture("res/shadowNormal.png"),
Resources::getTexture("res/shadowPressed.png")); }
void Game::run() {
  // инициализация поля
  gameBoard->initialization(40);
  // фон игры
  sf::Sprite background;
  sf::Texture backgroundTexture =
Resources::getTexture("res/background.png");
  background.setTexture(backgroundTexture);
  // получение текущей фигуры
  Piece* currentPiece = pieceCreating->getPiece();
  // получение следующей фигуры
  Piece* nextPiece = pieceCreating->getPiece();
  // получение тени фигуры
  Piece* shadowPiece = pieceCreating->getShadowPiece(currentPiece);
  setShadowPosition(currentPiece, shadowPiece);
  // окно следующей фигуры
  NextPieceBoard nextBoard(nextPiece, 40);
  if (volume)
    GameSound::play(0);
  else
    GameSound::stop(0);
  // флаг быстрого падения
  bool fastFallFlag = false;
  // флаг мгновенного падения
  bool instantFall = false;
  // флаг паузы
  bool pausedFlag = false;
  // установка часов для подсчёта времени
  sf::Clock frameClock;
  sf::Clock keyClock;
  sf::Time frameTime = frameClock.getElapsedTime();
  sf::Time keyTime = keyClock.getElapsedTime();
  // основной игровой цикл
  while (window->isOpen()) {
    // позиция мыши
    sf::Vector2f mousePos = window-
>mapPixelToCoords(sf::Mouse::getPosition(*window));
    // вход в основное меню
    if (*gamePartPtr == MENU) {
      GameSound::stop(0);
      break; {
    // если истинен флаг конца игры
    if (gameOver) {
      GameSound::stop(0);
      *gamePartPtr = GAMEOVER;
      *levelPtr = level;
```

```
      *scorePtr = score;
      *linesPtr = totalRows;
      delete currentPiece;
      delete nextPiece;
      delete shadowPiece;
      break; }
    sf::Event event;
    // обработка событий в очереди (закрытие окна и клавиши
поворота фигуры/выхода/паузы)
    while (window->pollEvent(event)) {
      if (event.type == sf::Event::Closed)
        window->close();
      // если игра не на паузе
      if (!pausedFlag) {
        // нажата кнопка паузы, то поставить игру на паузу
        if (pauseButton->updateButton(mousePos)) {
          GameSound::play(1);
          GameSound::stop(0);
          pausedFlag = !pausedFlag; {
        // нажата кнопка звука, то включить/выключить звук
        else if (volumeButton->updateButton(mousePos)) {
          GameSound::play(1);
          volume = !volume;
          if (volume)
            GameSound::play(0);
          else
            GameSound::stop(0); }
        // нажата кнопка тени, то включить/выключить тень
        else if (shadowButton->updateButton(mousePos)) {
          GameSound::play(1);
          shadowFlag = !shadowFlag; }}
      // если игра на паузе, то продолжить играть можно нажав
ввод/пробел (в условии дальше ещё P)
      if (pausedFlag &&
(sf::Keyboard::isKeyPressed(sf::Keyboard::Space) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::Enter))) {
        pausedFlag = !pausedFlag;
        if (volume)
          GameSound::play(0);
        else
          GameSound::stop(0); }
      if (event.type == sf::Event::KeyPressed) {
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::W)) {
          if (!pausedFlag)
            rotate(currentPiece, shadowPiece); }
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape)){
          *gamePartPtr = MENU; }
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::P)) {
          pausedFlag = !pausedFlag;
          if (pausedFlag)
            GameSound::stop(0);
```

```
          else if (volume)
            GameSound::play(0); }
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::G)) {
          shadowFlag = !shadowFlag; {
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::V)) {
          volume = !volume;
          if (volume)
            GameSound::play(0);
          else
            GameSound::stop(0); }}}
    // обработка событий непрерывного ввода клавиш (для более
плавной игры независимо от очереди событий)
    keyTime = keyClock.getElapsedTime();
    if (!pausedFlag) {
      if ((sf::Keyboard::isKeyPressed(sf::Keyboard::Down) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::S))
        && keyTime.asSeconds() > 0.04) {
        fastFallFlag = true;
        keyClock.restart(); }
      else if ((sf::Keyboard::isKeyPressed(sf::Keyboard::Space) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::F))
        && keyTime.asSeconds() > 0.2) {
        instantFall = true;
        keyClock.restart(); }
      else if ((sf::Keyboard::isKeyPressed(sf::Keyboard::Left) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::A))
        && keyTime.asSeconds() > 0.1) {
        moveLeft(currentPiece, shadowPiece);
        keyClock.restart(); }
      else if ((sf::Keyboard::isKeyPressed(sf::Keyboard::Right) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::D))
        && keyTime.asSeconds() > 0.1) {
        moveRight(currentPiece, shadowPiece);
        keyClock.restart(); } }
    // отрисовка
    window->clear();
    window->draw(background);
    window->draw(nextBoard);
    window->draw(*gameBoard);
    window->draw(scoreText);
    window->draw(linesText);
    window->draw(*pauseButton);
    window->draw(*volumeButton);
    window->draw(*shadowButton);
    if (shadowFlag)
      window->draw(*shadowPiece);
    window->draw(*currentPiece);
    if (pausedFlag)
      window->draw(pauseWindow);
    window->display();
    // падение фигур
    frameTime = frameClock.getElapsedTime();
```

```
    if (!pausedFlag && (fastFallFlag || (frameTime.asSeconds() > 1
- (level - 1) * 0.025))) {
      frameClock.restart();
      // если падение фигуры на 1 вниз невозможно
      if (!fallDown(currentPiece)) {
        // add возвращает true если фигура добавлена успешно
        gameOver = !gameBoard->add(currentPiece);
        delete currentPiece;
        delete shadowPiece;
        // следующая фигура становится текущей
        currentPiece = nextPiece;
        currentPiece->setPiecePosition(this->boardWidth / 2 - 1,
0);
        // новая следующая фигура
        nextPiece = pieceCreating->getPiece();
        nextBoard.setPiece(nextPiece);
        // новая фигура тени
        shadowPiece = pieceCreating->getShadowPiece(currentPiece);
        // удаление заполненных линий
        // получаем число очищенных строк с помощью метода
updateBoard()
        int clearedRows = gameBoard->updateBoard();
        if (clearedRows)
          GameSound::play(2);
        // обновление позиции тени фигуры
        setShadowPosition(currentPiece, shadowPiece);
        // обновление рейтинга и уровня
        this->totalRows += clearedRows;
        updateScore(clearedRows);
        updateLevel(); }
      fastFallFlag = false;
      // обновление отображения рейтинга и количества убранных
линий на экране
      setScore(score);
      setLines(totalRows); }
    if (!pausedFlag && instantFall) {
      // увеличение рейтинга при мгновенном падении
      this->score += 10;
      currentPiece->setPiecePosition(shadowPiece-
>getPiecePosition().getX(), shadowPiece-
>getPiecePosition().getY());
      // add возвращает true если фигура добавлена успешно
      gameOver = !gameBoard->add(currentPiece);
      delete currentPiece;
      delete shadowPiece;
      // следующая фигура становится текущей
      currentPiece = nextPiece;
      currentPiece->setPiecePosition(this->boardWidth / 2 - 1, 0);
      // новая следующая фигура
      nextPiece = pieceCreating->getPiece();
      nextBoard.setPiece(nextPiece);
      // новая фигура тени
```

```
      shadowPiece = pieceCreating->getShadowPiece(currentPiece);
      // удаление заполненных линий
      // получаем число очищенных строк с помощью метода
updateBoard()
      int clearedRows = gameBoard->updateBoard();
       if (clearedRows)
         GameSound::play(2);
      // обновление позиции тени фигуры
      setShadowPosition(currentPiece, shadowPiece);
      // обновление рейтинга и уровня
      this->totalRows += clearedRows;
      updateScore(clearedRows);
      updateLevel();
      instantFall = false;
      // обновление отображения рейтинга и количества убранных
линий на экране
      setScore(score);
      setLines(totalRows); }}}
bool Game::moveLeft(Piece* piece, Piece* shadowPiece)  {
  // если не будет столкновения при движении влево
  if (!gameBoard->collidesWith(piece->getPiecePosition().getX() -1,
                piece->getPiecePosition().getY(),
                piece->getCurrentShape()))  {
    // установить новую позицию фигуры на поле
    piece->setPiecePosition(piece->getPiecePosition().getX() - 1,
               piece->getPiecePosition().getY());
    // установить новую позицию тени на поле
    setShadowPosition(piece, shadowPiece);
    return true; }
  return false; }
bool Game::moveRight(Piece* piece, Piece* shadowPiece) {
  // если не будет столкновения при движении вправо
  if (!gameBoard->collidesWith(piece->getPiecePosition().getX() +1,
                piece->getPiecePosition().getY(),
                piece->getCurrentShape()))  {
    // установить новую позицию фигуры на поле
    piece->setPiecePosition(piece->getPiecePosition().getX() + 1,
              piece->getPiecePosition().getY());
    // установить новую позицию тени на поле
    setShadowPosition(piece, shadowPiece);
    return true; }
  return false; }
bool Game::rotate(Piece* piece, Piece* shadowPiece) {
  // если не будет столкновения после поворота
  if (!gameBoard->collidesWith(piece->getPiecePosition().getX(),
                piece->getPiecePosition().getY(),
                piece->getRotationShape()))  {
    // повернуть фигуру
    piece->rotate();
    // повернуть тень
    setShadowPosition(piece, shadowPiece);
    return true; }
```

```
  return false; }
//
bool Game::fallDown(Piece* piece)  {
  // если после падения не будет столкновения
  if (!gameBoard->collidesWith(piece->getPiecePosition().getX(),
                piece->getPiecePosition().getY() + 1,
                piece->getCurrentShape()))  {
    // опустить фигуру
    piece->setPiecePosition(piece->getPiecePosition().getX(),
piece->getPiecePosition().getY() + 1);
    return true; }
  return false; }
void Game::setShadowPosition(Piece* currentPiece, Piece*
shadowPiece) {
  // установить поворот тени таким же как у фигуры
  shadowPiece->setRotation(currentPiece->getRotation());
  // установить позицию тени такой же как у фигуры
  shadowPiece->setPiecePosition(currentPiece->getPiecePosition());
  // найти нижайшее возможное положение тени
  // (движение вниз до обнаружения столкновения)
  while (!gameBoard->collidesWith(shadowPiece-
>getPiecePosition().getX(),
                  shadowPiece->getPiecePosition().getY() + 1,
                  shadowPiece->getCurrentShape()))  {
    shadowPiece->setPiecePosition(shadowPiece-
>getPiecePosition().getX(),
                    shadowPiece->getPiecePosition().getY() + 1); }}
void Game::updateScore(int clearedRows)  {
  // увеличение рейтинга на основании количества удалённых линий
  switch (clearedRows)  {
  case 1:
    this->score += 50 * this->level;
    break;
  case 2:
    this->score += 150 * this->level;
    break;
  case 3:
    this->score += 350 * this->level;
    break;
  case 4:
    this->score += 750 * this->level;
    break; }}
void Game::updateLevel()  {
  this->level = this->totalRows / 5 + 1; }
void Game::setScore(int score) {
  scoreText.setString(std::to_string(score));
  sf::FloatRect rectangle(90, 100, 120, 50);
  sf::FloatRect textBounds = scoreText.getGlobalBounds();
  float X = rectangle.left + (rectangle.width - textBounds.width) /
2.f;
  float Y = rectangle.top + (rectangle.height - textBounds.height)
/ 2.f;
```

```
  scoreText.setPosition(X, Y); }
void Game::setLines(int lines) {
  linesText.setString(std::to_string(lines));
  sf::FloatRect rectangle(90, 220, 120, 50);
  sf::FloatRect textBounds = linesText.getGlobalBounds();
  float X = rectangle.left + (rectangle.width - textBounds.width) /
2.f;
  float Y = rectangle.top + (rectangle.height - textBounds.height)
/ 2.f;
  linesText.setPosition(X, Y); }
Game::~Game()  {
  delete this->gameBoard;
  delete this->pieceCreating;
  delete this->pauseButton;
  delete this->volumeButton;
  delete this->shadowButton; }
```

Файл Game.h:

```
#pragma once
#include <SFML/Audio.hpp>
#include "Piece.h"
#include "Board.h"
#include "NextPieceBoard.h"
#include "PieceCreating.h"
#include "Button.h"
#include "Enums.h"
#include "GameSound.h"
class Game  {
private:
  int boardWidth;          // ширина поля
  int boardHeight;         // высота поля
  PieceCreating* pieceCreating;   // указатель на объект класса
PieceCreating для создания фигур
  Board* gameBoard;          // указатель на объект класса игрового
поля
  bool gameOver = false;       // флаг конца игры
  int score = 0;           // рейтинг
  int level = 1;           // уровень
  int totalRows = 0;         // количество убранных линий
  bool volume = true;        // флаг звука (on/off)
  bool shadowFlag = true;      // флаг определяющий рисовать или нет
тень фигуры
  sf::Text scoreText;
  sf::Text linesText;
  sf::Font textFont = Resources::getFont("res/Exo-ExtraBold.ttf");
  int* gamePartPtr;          // указатель на выполняемую часть игры
  int* scorePtr;           // указатель на основную переменную
рейтинга
  int* levelPtr;           // указатель на основную переменную
уровня
  int* linesPtr;           // указатель на основную переменную
количества линий
```

```
  sf::Image icon;         // иконка
  sf::RenderWindow* window;     // указатель на окно sfml
  sf::Sprite pauseWindow;
  sf::Texture pauseWindowTexture;
  Button* pauseButton;
  Button* volumeButton;
  Button* shadowButton;
  // движение фигуры
  // true - успешно, false - нет
  bool moveLeft       (Piece* piece, Piece* shadowPiece);
  bool moveRight      (Piece* piece, Piece* shadowPiece);
  // поворот фигуры
  // true - успешно, false - нет
  bool rotate       (Piece* piece, Piece* shadowPiece);
  // опускает фигуру вниз на 1 клетку
  // true - успешно, false - нет
  bool fallDown       (Piece* piece);
  // устанавливает позицию тени
  void setShadowPosition  (Piece* currentPiece, Piece*
shadowPiece);
  // измененяет рейтинг относительно количества удалённых строк
  void
      int* scorePtr, int* levelPtr, int*linesPtr, bool shadowFlag);
  // деструктор
  ~Game();
  // основной цикл игры
  void run();
  // выводит текущий рейтинг на экран
  void setScore(int score);
  // выводит текущее количество убранных линий на экран
  void setLines(int lines);
};
```

Файл GameOver.cpp:

```
п»ї#include "GameOver.h"
GameOver::GameOver(sf::RenderWindow* window, std::string filename,
int* gameStatePtr, int* scorePtr, int* linesPtr)  {
  this->window = window;
  this->gamePartPtr = gameStatePtr;
  this->scorePtr = scorePtr;
  this->linesPtr = linesPtr;
  this->filename = filename;
  nickText.setFont(textFont);
  nickText.setCharacterSize(26);
  nickText.setFillColor(sf::Color(252, 242, 231, 255));
  nickText.setString("start typing...");
  nickText.setPosition(window->getSize().x / 2 -
nickText.getGlobalBounds().width / 2, 565);
  scoreText.setFont(textFont);
  scoreText.setCharacterSize(26);
  scoreText.setFillColor(sf::Color(252, 242, 231, 255));
  scoreText.setString(std::to_string(*scorePtr));
```

```
  scoreText.setPosition(window->getSize().x / 2 -
scoreText.getGlobalBounds().width / 2, 335);
  linesText.setFont(textFont);
  linesText.setCharacterSize(26);
  linesText.setFillColor(sf::Color(252, 242, 231, 255));
  linesText.setString(std::to_string(*linesPtr));
  linesText.setPosition(window->getSize().x / 2 -
linesText.getGlobalBounds().width / 2, 435);
  menuButton =    new Button(sf::Vector2f(220, 644),
Resources::getTexture("res/menuNormal.png"),
Resources::getTexture("res/menuPressed.png"));
  playButton =    new Button(sf::Vector2f(380, 644),
Resources::getTexture("res/playNormal.png"),
Resources::getTexture("res/playPressed.png"));
  leaderboardButton = new Button(sf::Vector2f(540, 644),
Resources::getTexture("res/leadersNormal.png"),
Resources::getTexture("res/leadersPressed.png")); }
GameOver::~GameOver()  {
  delete this->menuButton;
  delete this->playButton;
  delete this->leaderboardButton; }
void GameOver::run()  {
  sf::Sprite background;
  sf::Texture backgroundText =
Resources::getTexture("res/gameoverBackground.png");
  background.setTexture(backgroundText);
  while (window->isOpen())  {
    if (*gamePartPtr != GAMEOVER) {
      addScore();
      break; }
    sf::Event event;
    while (window->pollEvent(event)) {
      if (event.type == sf::Event::Closed) {
        addScore();
        window->close(); }
      if (event.type == sf::Event::TextEntered) {
        if (event.text.unicode == '\b') {
          if (nick.length() > 0)
            nick.erase(nick.length() - 1, 1); }
       else if (event.text.unicode != ' ' && (event.text.unicode >=
'A' && event.text.unicode <= 'z')) {
          if (nick.length() <= 15)
            nick += event.text.unicode; }
        nickText.setString(nick);
        nickText.setPosition(window->getSize().x / 2 -
nickText.getGlobalBounds().width / 2, 565); } }
    sf::Vector2f mousePos = window-
>mapPixelToCoords(sf::Mouse::getPosition(*window));
    if (menuButton->updateButton(mousePos))  {
      GameSound::play(1);
      *gamePartPtr = MENU; }
    if (playButton->updateButton(mousePos))  {
```

```
      GameSound::play(1);
      *gamePartPtr = GAME; }
    if (leaderboardButton->updateButton(mousePos))  {
      GameSound::play(1);
      *gamePartPtr = LEADERBOARD; }
    window->clear();
    window->draw(background);
    window->draw(scoreText);
    window->draw(linesText);
    window->draw(nickText);
    window->draw(*menuButton);
    window->draw(*playButton);
    window->draw(*leaderboardButton);
    window->display(); } }
void GameOver::addScore()  {
  std::ofstream outputStream;
  outputStream.open(filename, std::fstream::out |
std::fstream::app);
  if (nick == "")
    nick = "NoName";
  outputStream << nick << " " << std::to_string(*scorePtr) << "\n";
  outputStream.close(); }
```

Файл GameOver.h:

```
#pragma once
#include <SFML/Graphics.hpp>
#include <SFML/Audio/Sound.hpp>
#include "Button.h"
#include <iostream>
#include <fstream>
#include "Enums.h"
#include "GameSound.h"
class GameOver  {
private:
  sf::RenderWindow* window;
  sf::Font textFont = Resources::getFont("res/Exo-ExtraBold.ttf");
  sf::Text nickText;
  sf::Text scoreText;
  sf::Text linesText;
  Button* menuButton;
  Button* playButton;
  Button* leaderboardButton;
  int* gamePartPtr;
  int* scorePtr;
  int* linesPtr;
  std::string filename;
  std::string nick;
public:
  // контруктор
  GameOver(sf::RenderWindow* window, std::string filename, int*
gameStatePtr, int* scorePtr, int* linesPtr);
  // деструктор
```

```
  ~GameOver();
  // главный цикл конца игры (ввод никнейма, чтобы сохранить
рейтинг)
  void run();
  // ввод нового рейтинга в файл
  void addScore();
};
```

Файл GameSound.cpp:

```
#include "GameSound.h"
GameSound* GameSound::sInstance = nullptr;
GameSound::GameSound() {
     assert(sInstance == nullptr);
     sInstance = this;
     std::array<std::string, n> namefilebuf{ "res/background.ogg",
"res/button.ogg", "res/cleared.ogg"};     // массив названий файлов
     for (int i = 0; i < n; i++)
     // присвоение объекту звука
     gSounds[i].setBuffer(Resources::getSound(namefilebuf[i]));
     gSounds[0].setLoop(true);// самый первый звук - фон, он
воспроизводится циклично
};
void GameSound::play(int index) {
     if (sInstance->gSounds[index].getStatus() ==
sf::SoundSource::Status::Stopped)
          sInstance->gSounds[index].play(); }
void GameSound::stop(int index) {
     if (sInstance->gSounds[index].getStatus() ==
sf::SoundSource::Status::Playing)
          sInstance->gSounds[index].stop(); }
```

Файл GameSound.h:

```
#pragma once
#include <array>
#include <SFML/Audio.hpp>
#include "Resources.h"
class GameSound {
     static const int n = 3;                              //
количество звуковых эффектов
     std::array<sf::Sound,n> gSounds;               // массив
объектов звуковых эффектов
     static GameSound* sInstance;                   // во всём
приложении следует чтобы был лишь один объект класса музыки
public:
     GameSound();
     static void play(int index);         // включение звука
     static void stop(int index);         // выключение звука
};
```

Файл IPiece.cpp:

```
#include "IPiece.h"
#include "Enums.h"
```

```
Cell IPiece::cells[4][4] = { {
    Cell(1, 0),  Cell(1, 1),  Cell(1, 2),  Cell(1, 3)
  }, {
    Cell(0, 1),  Cell(1, 1),  Cell(2, 1),  Cell(3, 1)
  }, {
    Cell(2, 0),  Cell(2, 1),  Cell(2, 2),  Cell(2, 3)
  }, {
    Cell(0, 2),  Cell(1, 2),  Cell(2, 2),  Cell(3, 2) }};
IPiece::IPiece(int rotation, int spawnX) : Piece(rotation, IP,
(Cell*)cells, spawnX) {}
```

Файл IPiece.h:

```
#pragma once
#include "Piece.h"
class IPiece : public Piece  {
  static Cell cells[4][4];
public:
  IPiece(int rotation, int spawnX);
};
```

Файл JPiece.cpp:

```
#include "JPiece.h"
#include "Enums.h"
Cell JPiece::cells[4][4] = { {
    Cell(1, 0),  Cell(1, 1),  Cell(1, 2),  Cell(0, 2)
  }, {
    Cell(0, 0),  Cell(0, 1),  Cell(1, 1),  Cell(2, 1)
  }, {
    Cell(2, 0),  Cell(1, 0),  Cell(1, 1),  Cell(1, 2)
  }, {
    Cell(0, 1),  Cell(1, 1),  Cell(2, 1),  Cell(2, 2) }};
JPiece::JPiece(int rotation, int spawnX) : Piece(rotation, JP,
(Cell*)cells, spawnX) {}
```

Файл JPiece.h:

```
#pragma once
#include "Piece.h"
class JPiece : public Piece {
  static Cell cells[4][4];
public:
  JPiece(int rotation, int spawnX);
};
```

Файл LeaderBoard.cpp:

```
#include <fstream>
#include <sstream>
#include "LeaderBoard.h"
using namespace std;
LeaderBoard::LeaderBoard(sf::RenderWindow* window, string filename,
int* gamePartPtr, int* scorePtr) {
  this->window = window;
  this->gamePartPtr = gamePartPtr;
```

```
  this->scorePtr = scorePtr;
  this->filename = filename;
  menuButton = new Button(sf::Vector2f(380, 744),
Resources::getTexture("res/menuNormal.png"),
Resources::getTexture("res/menuPressed.png"));
  // открытие файла с рейтингами
  ifstream inputStream;
  inputStream.open(filename, ios::in);
  // добавление рейтингов в вектор
  string line;
  if (inputStream.is_open())  {
    while (getline(inputStream, line))  {
      stringstream ss(line);
      string nick;
      int score;
      ss >> nick >> score;
      scores.push_back(new SingleScore(nick, score)); } }
  else  {
    // если невозможно открыть файл, то создать его
    ofstream outputStream;
    outputStream.open(filename, ios::out);
    outputStream.close(); }
  inputStream.close();
  // сортировка вектора
  sortScores();
  // оставить только первые 10 рейтингов в векторе, остальные
удаляет
  while (scores.size() > 10) {
    scores.pop_back(); }
  // позиция в окне
  for (int i = 0; i < scores.size(); i++)  {
    scores[i]->update(245, 202 + i * 50); } }
LeaderBoard::~LeaderBoard()  {
  for (SingleScore* single : scores)
    delete single;
  delete menuButton; }
void LeaderBoard::sortScores()  {
  // сортировка рейтингов используя STLвскую сортировку и лямбда
выражение
  sort(scores.begin(), scores.end(),
    [](SingleScore* first, SingleScore* second) {return first-
>getScore() > second->getScore();}); }
void LeaderBoard::run() {
  sf::Sprite background;
  sf::Texture backgroundTexture =
Resources::getTexture("res/LeaderboardBackground.png");
  background.setTexture(backgroundTexture);
  while (window->isOpen())  {
    if (*gamePartPtr != LEADERBOARD)
      break;
    sf::Event event;
    while (window->pollEvent(event))  {
```

```
      if (event.type == sf::Event::Closed)
        window->close();
      else if (event.key.code == sf::Keyboard::Escape)
        *gamePartPtr = MENU; }
    sf::Vector2f mousePos = window-
>mapPixelToCoords(sf::Mouse::getPosition(*window));
    if (menuButton->updateButton(mousePos)) {
      GameSound::play(1);
      *gamePartPtr = MENU; }
    window->clear();
    window->draw(background);
    window->draw(*menuButton);
    for (SingleScore* single : scores)
      window->draw(*single);
    window->display(); } }
```

Файл LeaderBoard.h:

```
#include <SFML/Graphics.hpp>
#include <vector>
#include "SingleScore.h"
#include "Enums.h"
#include "Button.h"
#include "GameSound.h"
class LeaderBoard  {
private:
  sf::RenderWindow* window;
  Button* menuButton;
  std::vector<SingleScore*> scores;
  std::string filename;
  int* gamePartPtr;
  int* scorePtr;
public:
  // конструктор
  LeaderBoard(sf::RenderWindow* window, std::string filename, int*
gamePartPtr, int* scorePtr);
  // деструктор
  ~LeaderBoard();
  // сортировка рейтингов
  void sortScores();
  // основной цикл таблицы лидеров
  void run();
};
```

Файл LPiece.cpp:

```
#include "LPiece.h"
#include "Enums.h"
Cell LPiece::cells[4][4] = { {
    Cell(1, 0),  Cell(1, 1),  Cell(1, 2),  Cell(2, 2)
  }, {
    Cell(0, 1),  Cell(1, 1),  Cell(2, 1),  Cell(0, 2)
  }, {
    Cell(0, 0),  Cell(1, 0),  Cell(1, 1),  Cell(1, 2)
```

```
  }, {
    Cell(0, 1),  Cell(1, 1),  Cell(2, 1),  Cell(2, 0) } };
LPiece::LPiece(int rotation, int spawnX) : Piece(rotation, LP,
(Cell*)cells, spawnX) {}
```

Файл LPiece.h:

```
#pragma once
#include "Piece.h"
class LPiece : public Piece {
  static Cell cells[4][4];
public:
  LPiece(int rotation, int spawnX);
};
```

Файл main.cpp:

```
#include "Game.h"
#include "Button.h"
#include "Menu.h"
#include "GameOver.h"
#include "LeaderBoard.h"
#include "GameSound.h"
int main()  {
  bool volume = true;
  int score = 0;
  int level = 1;
  int lines = 0;
  int boardWidth = 10;
  int boardHeight = 20;
  int gamePart = MENU;
  bool shadowFlag = true;
  sf::VideoMode desktop = sf::VideoMode::getDesktopMode();
                                    // разрешение экрана
  sf::RenderWindow window(sf::VideoMode(900, 900), L"TETRA PEACE",
sf::Style::Close);
  window.setPosition(sf::Vector2i((desktop.width - 900) / 2,
(desktop.height - 900) / 2 - 40));   // позиция окна относительно
экрана
  window.setVerticalSyncEnabled(true);
  // загрузка иконки
  sf::Image icon;
  icon.loadFromFile("res/icon.png");
  window.setIcon(icon.getSize().x, icon.getSize().y,
icon.getPixelsPtr());
  // менеджер ресурсов
  Resources resourceManager;
  // музыка
  GameSound music;
  while (window.isOpen())  {
    if (gamePart == EXIT)
      break;
    switch (gamePart)  {
    case MENU:  {
```

```
      Menu menu(&window, &gamePart);
      menu.run();
      break; }
    case GAME:  {
      Game game(&window, boardWidth, boardHeight, volume,
&gamePart, &score, &level, &lines, shadowFlag);
      game.run();
      break; }
    case GAMEOVER:  {
      GameOver gameOver(&window, "scores.txt", &gamePart, &score,
&lines);
      gameOver.run();
      break; }
    case LEADERBOARD:  {
       LeaderBoard leaderboard(&window, "scores.txt", &gamePart,
&score);
         leaderboard.run();
      break; }
    default:
      gamePart = EXIT;
      break; } }
  return 0; }
```

Файл Menu.cpp:

```
#include "Menu.h"
Menu::Menu(sf::RenderWindow* window, int* gamePartPtr) {
  this->window    = window;
  this->gamePartPtr   = gamePartPtr;
  startButton =     new Button(sf::Vector2f(380, 400),
Resources::getTexture("res/playNormal.png"),
Resources::getTexture("res/playPressed.png"));
  leaderboardButton = new Button(sf::Vector2f(380, 500),
Resources::getTexture("res/leadersNormal.png"),
Resources::getTexture("res/leadersPressed.png"));
  exitButton =    new Button(sf::Vector2f(380, 600),
Resources::getTexture("res/exitNormal.png"),
Resources::getTexture("res/exitPressed.png")); }
void Menu::run() {
  sf::Sprite background;
  sf::Texture backgroundTexture =
Resources::getTexture("res/menuBackground.png");
  background.setTexture(backgroundTexture);
  while (window->isOpen())  {
    // если на данный момент должно выводиться не меню - прервать
    if (*gamePartPtr != MENU)
      break;
    sf::Event event;
    while (window->pollEvent(event))  {
      if (event.type == sf::Event::Closed)
        window->close(); }
    // позиция мыши
```

```
    sf::Vector2f mousePos = window-
>mapPixelToCoords(sf::Mouse::getPosition(*window));
    if (startButton->updateButton(mousePos) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::Space)) {
      GameSound::play(1);
      *gamePartPtr = GAME; }
    if (leaderboardButton->updateButton(mousePos) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::L)) {
      GameSound::play(1);
      *gamePartPtr = LEADERBOARD; {
    if (exitButton->updateButton(mousePos)) {
      GameSound::play(1);
      *gamePartPtr = EXIT; }
    window->clear();
    window->draw(background);
    window->draw(*startButton);
    window->draw(*leaderboardButton);
    window->draw(*exitButton);
    window->display(); } }
Menu::~Menu() {
  delete this->startButton;
  delete this->leaderboardButton;
  delete this->exitButton; }
```

Файл Menu.h:

```
#pragma once
#include "Button.h"
#include "Enums.h"
#include "GameSound.h"
class Menu  {
private:
  sf::RenderWindow* window;
  Button* startButton;
  Button* exitButton;
  Button* leaderboardButton;
  int* gamePartPtr;
public:
  Menu(sf::RenderWindow* window, int* gamePartPtr);
  ~Menu();
  void run();
};
```

Файл NextPieceBoard.cpp:

```
#include "NextPieceBoard.h"
NextPieceBoard::NextPieceBoard(Piece* piece, int tileSize)  {
  this->tileSize = tileSize;
  this->piece = piece;
  // определение позиции фигуры в окне NEXT
  int type = 0;
  type = this->piece->getCurrentShapeInt();
  switch (type) {
  case IP:
```

```
    this->piece->setPiecePosition(17, 2, false);
    break;
  case JP:
    this->piece->setPiecePosition(17, 3, false);
    break;
  case LP:
    this->piece->setPiecePosition(16, 3, false);
    break;
  case OP:
    this->piece->setPiecePosition(16, 2, false);
    break;
  case SP:
    this->piece->setPiecePosition(16, 3, false);
    break;
  case TP:
    this->piece->setPiecePosition(17, 3, false);
    break;
  case ZP:
    this->piece->setPiecePosition(16, 3, false);
    break;
  default:
    break; } }
void NextPieceBoard::setPiece(Piece* piece) {
  this->piece = piece;
  switch (piece->getCurrentShapeInt()) {
  case IP:
    this->piece->setPiecePosition(17, 2, false);
    break;
  case JP:
    this->piece->setPiecePosition(17, 3, false);
    break;
  case LP:
    this->piece->setPiecePosition(16, 3, false);
    break;
  case OP:
    this->piece->setPiecePosition(16, 2, false);
    break;
  case SP:
    this->piece->setPiecePosition(16, 3, false);
    break;
  case TP:
    this->piece->setPiecePosition(17, 3, false);
    break;
  case ZP:
    this->piece->setPiecePosition(16, 3, false);
    break;
  default:
    break; } }
void NextPieceBoard::draw(sf::RenderTarget& target,
sf::RenderStates states) const {
  target.draw(*piece); }
```

Файл NextPieceBoard.h:
```
#pragma once
#include <SFML/Graphics.hpp>
#include "Piece.h"
#include "Enums.h"
class NextPieceBoard : public sf::Drawable, sf::Transformable {
private:
  // указатель на фигуру
  Piece* piece;
  // размер одной плитки
  int tileSize;
public:
  // конструктор класса
  NextPieceBoard(Piece* piece, int tileSize);
  // устанавливает позицию следующей фигуры в окне NEXT
  void setPiece(Piece* piece);
protected:
  // отрисовка поля и фигуры
  void draw(sf::RenderTarget& target, sf::RenderStates states)
const override;
};
```

Файл OPiece.cpp:
```
#include "OPiece.h"
#include "Enums.h"
Cell OPiece::cells[4][4] = { {
    Cell(1, 1),  Cell(2, 1),  Cell(1, 2),  Cell(2, 2)
  }, {
    Cell(1, 1),  Cell(2, 1),  Cell(1, 2),  Cell(2, 2)
  }, {
    Cell(1, 1),  Cell(2, 1),  Cell(1, 2),  Cell(2, 2)
  }, {
    Cell(1, 1),  Cell(2, 1),  Cell(1, 2),  Cell(2, 2) } };
OPiece::OPiece(int rotation, int spawnX) : Piece(rotation, OP,
(Cell*)cells, spawnX) {}
```

Файл OPiece.h:
```
#pragma once
#include "Piece.h"
class OPiece : public Piece {
  static Cell cells[4][4];
public:
  OPiece(int rotation, int spawnX);
};
```

Файл Piece.cpp:
```
#include "Piece.h"
#include "Board.h"
#include <random>
#include <chrono>
// генерирует случайное число от 1 до 7
int getRandomNumber() {
```

```
  unsigned seed =
std::chrono::system_clock::now().time_since_epoch().count();
  std::default_random_engine gen(seed);
  std::uniform_int_distribution<int> dis(1, 7);
  return dis(gen); }
Piece::Piece(int rotation, int currentPiece, Cell* shapes, int
spawnX) : rotation(rotation), currentPiece(currentPiece) {
  // копируем возможные варианты расположения фигуры в массив
  setShapes(shapes);
  // устанавливаем стартовую позицию
  this->piecePosition.setPos(spawnX, 0);
  int random = getRandomNumber();
  // представление фигуры четырьмя спрайтами
  for (int i = 0; i < 4; i++)  {
    // если фигура это тень
    if (currentPiece == 8) {
      // установить текстуру тени
      tileSprite[i].setTextureRect(sf::IntRect(40 * currentPiece,
0, 40, 40));
      this->color = 8; }
    else {
      // установить случайную текстуру
      tileSprite[i].setTextureRect(sf::IntRect(40 * random, 0, 40,
40));
      this->color = random; }
    tileSprite[i].setTexture(tiles);
    // установить позицию спрайта на экране
    tileSprite[i].setPosition((piecePosition.getX() +
shape[i].getX()) * 40 + BOARD_OFFSET_X,
                  (piecePosition.getY() + shape[i].getY()) * 40 +
BOARD_OFFSET_Y); } }
void Piece::setShapes(Cell* newShapes)  {
  // копируем в поле класса варианты расположения фигуры
  for (int i = 0; i < 4; i++)  {
    for (int j = 0; j < 4; j++)  {
      this->shapes[i][j] = newShapes[i * 4 + j]; } }
  setCurrentShape(); }
void Piece::setCurrentShape()  {
  // устанавливаем 4 спрайта в порядке, установленном текущим
поворотом фигуры
  for (int i = 0; i < 4; i++)  {
    this->shape[i] = shapes[rotation][i]; } }
void Piece::draw(sf::RenderTarget& target, sf::RenderStates states)
const  {
  // для каждого из четырёх спрайтов
  for (int i = 0; i < 4; i++) {
    target.draw(tileSprite[i], states); { {
const Cell Piece::getPiecePosition() const  {
  return piecePosition; }
// перегрузка для окна следующей фигуры
void Piece::setPiecePosition(int x, int y, bool fixedToBoard)  {
  piecePosition.setX(x);
```

```
  piecePosition.setY(y);
  for (int i = 0; i < 4; i++)  {
    // относительно поля (доска начинается с блока границы + отступ
в определённое количество пикселей)
    // + отступ до поля игрового по Y
    if (fixedToBoard)
      tileSprite[i].setPosition((piecePosition.getX() +
shape[i].getX()) * 40 + BOARD_OFFSET_X,
                   (piecePosition.getY() + shape[i].getY()) * 40 +
BOARD_OFFSET_Y);
    // к полю не привязаны фигуры в окне next, для разных фигур
разные отступы
    else {
      if (currentPiece == 4)
        tileSprite[i].setPosition((piecePosition.getX() +
shape[i].getX()) * 40 + 30,
              (piecePosition.getY() + shape[i].getY()) * 40 + 30);
      else if (currentPiece == 1)
        tileSprite[i].setPosition((piecePosition.getX() +
shape[i].getX()) * 40 + 10,
              (piecePosition.getY() + shape[i].getY()) * 40 + 30);
      else
        tileSprite[i].setPosition((piecePosition.getX() +
shape[i].getX()) * 40 + 30,
               (piecePosition.getY() + shape[i].getY()) * 40 +
10);}}}
// перегрузка для основного поля
void Piece::setPiecePosition(int x, int y)  {
  setPiecePosition(x, y, true); }
// перегрузка для тени
void Piece::setPiecePosition(Cell x)  {
  setPiecePosition(x.getX(), x.getY(), true);
  setCurrentShape(); }
void Piece::rotate()  {
  int nextRotation = (this->rotation + 1) % 4;
  setRotation(nextRotation);
  setCurrentShape();
  setPiecePosition(this->getPiecePosition()); }
int Piece::getRotation() const {
  return rotation; }
void Piece::setRotation(int rotation)  {
  if (rotation >= 0 && rotation < 4)
    Piece::rotation = rotation;
  else
    Piece::rotation = 0; {
Cell* Piece::getCurrentShape()  {
  return this->shape; {
Cell* Piece::getRotationShape()  {
  int nextRotation = this->rotation - 1;
  if (nextRotation < 0)  {
    nextRotation = 3; }
  return this->shapes[nextRotation]; {
```

```
int Piece::getCurrentColor() const {
  return color; }
int Piece::getCurrentShapeInt() const  {
  return currentPiece; }
Cell* const Piece::getShapes() const  {
  return (Cell*)shapes; }
```

Файл Piece.h:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "Cell.h"
#include "Resources.h"
/* класс отдельной фигуры */
class Piece : public sf::Drawable, public sf::Transformable  {
private:
  // позиция фигуры на поле
  Cell piecePosition;
  // текущий поворот фигуры
  // у каждой фигуры есть 4 варианта вращения, каждый вариант
описан в массиве Cell shapes[4][4]
  int rotation = 0;
  // цвет текущей фигуры (число определяет смещение в tileset)
  // 0/9 - прозрачный, 1-7 - цветные блоки, 8 - тень
  int color = 0;
  // текущая фигура (её представление в перечислении pieceEnum)
  int currentPiece = 0;
  // текущая форма описывается 4 клетками, каждый поворот
определяетя одним из четырёх вариантов массива Cell shapes[4][4]
  Cell shape[4];
  // основной массив, хранящий положение всех клеток фигуры при
разных вариантах вращений
  Cell shapes[4][4];
  // текстура (набор плиток)
  sf::Texture tiles = Resources::getTexture("res/tiles.png");
  // массив 4 спрайтов для представления одной фигуры
  sf::Sprite tileSprite[4];
  // устанавливает текущую форму фигуры (положение каждого из 4
блоков) в зависимости от поворота
  void setCurrentShape();
  // отрисовка фигуры
  void draw(sf::RenderTarget& target, sf::RenderStates states)
const override;
public:
  // конструктор класса
  // rotation    текущий поворот фигуры
  // currentPiece  номер фигуры согласно pieceEnum
  // Cell* shapes  повороты
  // spawnX    позиция появления фигуры
  Piece(int rotation, int currentPiece, Cell* shapes, int spawnX =
4);
  // копирует все формы фигуры в массив
  void setShapes(Cell* newShapes);
```

```
  // возвращает массив всех поворотов
  Cell* const getShapes() const;
  // возвращает текущий поворот фигуры (0-3)
  int getRotation() const;
  // устанавливает желаемый поворот фигуры
  void setRotation(int rotation);
  // возвращает текущий цвет фигуры
  int getCurrentColor() const;
  // возвращает текущий тип фигуры (согласно перечислению)
  int getCurrentShapeInt() const;
  // возвращает текущую позицию фигуры на поле
  const Cell getPiecePosition() const;
  // устанавливает позицию фигуры. Устанавливает 4 спрайта (блока)
на желаемую позицию
  // 3 перегрузки
  void setPiecePosition(int x, int y, bool fixedToBoard);
  void setPiecePosition(int x, int y);
  void setPiecePosition(Cell x);
  // поворот фигуры (изменение rotation)
  void rotate();
  // возвращает текущую форму фигуры
  Cell* getCurrentShape();
  // возвращает форму фигуры после поворота
  Cell* getRotationShape();
};
```

Файл PieceCreating.cpp:

```
#include "PieceCreating.h"
PieceCreating::PieceCreating(int spawnX) {
  this->spawnX = spawnX;
  fillVector(); }
void PieceCreating::fillVector()  {
  // заполнение вектора всеми возможными фигурами (их
представлениями в перечислении)
  this->pieces.push_back(IP);
  this->pieces.push_back(JP);
  this->pieces.push_back(LP);
  this->pieces.push_back(OP);
  this->pieces.push_back(SP);
  this->pieces.push_back(TP);
  this->pieces.push_back(ZP);
  // перемешивание фигур с помощью генератора случайных чисел
  unsigned seed =
std::chrono::system_clock::now().time_since_epoch().count();
  std::shuffle(this->pieces.begin(), this->pieces.end(),
std::default_random_engine(seed)); }
Piece* PieceCreating::getPiece()  {
  // если вектор пуст, то заполнить его заново
  if (pieces.empty())
    fillVector();
  // получение случайной фигуры
  int random = pieces[0];
```

```
  // удаление её из вектора
  pieces.erase(pieces.begin(), pieces.begin() + 1);
  Piece* figure = NULL;
  switch (random)  {
  case 1:
    figure = new IPiece(0, spawnX);
    break;
  case 2:
    figure = new JPiece(0, spawnX);
    break;
  case 3:
    figure = new LPiece(0, spawnX);
    break;
  case 4:
    figure = new OPiece(0, spawnX);
    break;
  case 5:
    figure = new SPiece(0, spawnX);
    break;
  case 6:
    figure = new TPiece(0, spawnX);
    break;
  case 7:
    figure = new ZPiece(0, spawnX);
    break; {
  return figure; }
Piece* PieceCreating::getShadowPiece(Piece* currentPiece)  {
  return new ShadowPiece(currentPiece->getRotation(), currentPiece-
>getShapes()); }
```

Файл PieceCreating.h:

```
#pragma once
#include <vector>
#include <random>
#include <chrono>
#include "Enums.h"
#include "Piece.h"
#include "IPiece.h"
#include "JPiece.h"
#include "LPiece.h"
#include "OPiece.h"
#include "TPiece.h"
#include "SPiece.h"
#include "ZPiece.h"
#include "ShadowPiece.h"
class PieceCreating {
  std::vector <int> pieces;
  int spawnX;
  // заполняет вектор всеми возможными фигурами, а затем
перемешивает их
  void fillVector();
public:
```

```
  // конструктор класса
  PieceCreating(int spawnX);
  // возвращает случайную фигуру из вектора, удаляет её из вектора
  // если вектор при этом опустошается полностью, то вызывается
метод fillVector()
  Piece* getPiece();
  // возвращает тень нынешней фигуры
  Piece* getShadowPiece(Piece* currentPiece);
};
```

Файл Resources.cpp:

```
#include "Resources.h"
Resources* Resources::sInstance = nullptr;
Resources::Resources() {
     assert(sInstance == nullptr);
     sInstance = this; }
sf::Texture& Resources::getTexture(std::string const& filename)
          // filename - ключ {
     auto& Map = sInstance->textures;
     auto value = Map.find(filename);
               // value - значение
     if (value != Map.end()) {
          return value->second; {
     else {
          auto& texture = Map[filename];
          texture.loadFromFile(filename);
          return texture; } }
sf::Font& Resources::getFont(std::string const& filename) {
     auto& Map = sInstance->fonts;
     auto value = Map.find(filename);
     if (value != Map.end()) {
          return value->second;     }
     else {
          auto& font = Map[filename];
          font.loadFromFile(filename);
          return font; { {
sf::SoundBuffer& Resources::getSound(std::string const& filename) {
     auto& Map = sInstance->sounds;
     auto value = Map.find(filename);
     if (value != Map.end()) {
          return value->second; }
     else {
          auto& sound = Map[filename];
          sound.loadFromFile(filename);
          return sound; }}
```

Файл Resources.h:

```
#pragma once
#include <SFML/Audio.hpp>
#include <SFML/Graphics.hpp>
#include <cassert>
#include <map>
```

```
#include <memory>
class Resources {
public:
     Resources();
     static sf::Texture& getTexture(std::string const& filename);
     static sf::SoundBuffer& getSound(std::string const& filename);
     static sf::Font& getFont(std::string const& filename);
private:
     static Resources* sInstance;         // во всём приложении
следует чтобы был лишь один объект класса ресурсов
     // переменные ключ - значение
     std::map< std::string, sf::Texture> textures;

     std::map <std::string, sf::Font> fonts;
     std::map <std::string, sf::SoundBuffer> sounds;
};
```

Файл ShadowPiece.cpp:

```
#include "ShadowPiece.h"
#include "Enums.h"
ShadowPiece::ShadowPiece(int rotation, Cell* shapes) :
Piece(rotation, SHADOW, shapes) {}
```

Файл ShadowPiece.h:

```
#pragma once
#include "Piece.h"
class ShadowPiece : public Piece  {
public:
  ShadowPiece(int rotation, Cell* shapes);
};
```

Файл SingleScore.cpp:

```
#include "SingleScore.h"
SingleScore::SingleScore(const std::string& nick, int score)  {
  this->nick = nick;
  this->score = score;
  nickText.setFont(font);
  nickText.setCharacterSize(26);
  nickText.setFillColor(sf::Color(252, 242, 231, 255));
  scoreText.setFont(font);
  scoreText.setCharacterSize(26);
  scoreText.setFillColor(sf::Color(252, 242, 231, 255)); {
long SingleScore::getScore() const  {
  return score; {
void SingleScore::update(int x, int y)  {
  nickText.setString(nick);
  nickText.setPosition(x, y);
  scoreText.setString(std::to_string(score));
  sf::FloatRect rectangle(555, 0, 140, 0);
  sf::FloatRect textBounds = scoreText.getGlobalBounds();
  float X = rectangle.left + (rectangle.width - textBounds.width) /
2.f;
```

```
  scoreText.setPosition(X , y); }
void SingleScore::draw(sf::RenderTarget& target, sf::RenderStates
states) const  {
  target.draw(nickText);
  target.draw(scoreText); {
```

Файл SingleScore.h:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "Resources.h"
#include <string>
class SingleScore : public sf::Drawable {
private:
  sf::Font font = Resources::getFont("res/Exo-ExtraBold.ttf");
  sf::Text nickText;
  sf::Text scoreText;
  std::string nick;
  int score = 0;
protected:
  void draw(sf::RenderTarget& target, sf::RenderStates states)
const override;
public:
  // конструктор
  SingleScore(const std::string& nick, int score);
  // возвращает значение рейтинга
  long getScore() const;
  // устанавливает позицию рейтинга на экране
  void update(int x, int y);
};
```

Файл SPiece.cpp:

```
#include "SPiece.h"
#include "Enums.h"
Cell SPiece::cells[4][4] = { {
    Cell(1, 0),  Cell(1, 1),  Cell(2, 1),  Cell(2, 2)
  }, {
    Cell(0, 1),  Cell(1, 1),  Cell(1, 0),  Cell(2, 0)
  }, {
    Cell(1, 0),  Cell(1, 1),  Cell(2, 1),  Cell(2, 2)
  }, {
    Cell(0, 1),  Cell(1, 1),  Cell(1, 0),  Cell(2, 0) }};
SPiece::SPiece(int rotation, int spawnX) : Piece(rotation, SP,
(Cell*)cells, spawnX) {}
```

Файл SPiece.h:

```
#pragma once
#include "Piece.h"
class SPiece : public Piece {
  static Cell cells[4][4];
public:
  SPiece(int rotation, int spawnX);
};
```

Файл TPiece.cpp:

```
п»ї#include "TPiece.h"
#include "Enums.h"
Cell TPiece::cells[4][4] = { {
    Cell(0, 1),  Cell(1, 1),  Cell(1, 0),  Cell(1, 2)
  }, {
    Cell(0, 1),  Cell(1, 1),  Cell(1, 0),  Cell(2, 1)
  }, {
    Cell(1, 0),  Cell(1, 1),  Cell(1, 2),  Cell(2, 1)
  }, {
    Cell(0, 1),  Cell(1, 1),  Cell(2, 1),  Cell(1, 2) }};
TPiece::TPiece(int rotation, int spawnX) : Piece(rotation, TP,
(Cell*)cells, spawnX) {}
```

Файл TPiece.h:

```
#pragma once
#include "Piece.h"
class TPiece : public Piece {
  static Cell cells[4][4];
public:
  TPiece(int rotation, int spawnX);
};
```

Файл ZPiece.cpp:

```
п»ї#include "ZPiece.h"
#include "Enums.h"
Cell ZPiece::cells[4][4] = { {
    Cell(2, 0),  Cell(2, 1),  Cell(1, 1),  Cell(1, 2)
  }, {
    Cell(0, 0),  Cell(1, 0),  Cell(1, 1),  Cell(2, 1)
  }, {
    Cell(2, 0),  Cell(2, 1),  Cell(1, 1),  Cell(1, 2)
  }, {
    Cell(0, 0),  Cell(1, 0),  Cell(1, 1),  Cell(2, 1) }};
ZPiece::ZPiece(int rotation, int spawnX) : Piece(rotation, ZP,
(Cell*)cells, spawnX) {}
```

Файл ZPiece.h:

```
#pragma once
#include "Piece.h"
class ZPiece : public Piece {
  static Cell cells[4][4];
public:
  ZPiece(int rotation, int spawnX);
};
```

ПРИЛОЖЕНИЕ Д
(обязательное)

Ведомость документов