

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 «ТЕТРИС»

БГУИР КП 1-40 02 01 508 ПЗ

Студент: гр. 250505 Гнетецкий Д. Г.

Руководитель: ассистент кафедры ЭВМ  
Марзалиюк А. В.

МИНСК 2023

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	4
1 ПОСТАНОВКА ЗАДАЧИ .....	7
2 ОБЗОР ЛИТЕРАТУРЫ.....	8
2.1 Обзор методов и алгоритмов решения поставленной задачи .....	10
2.1.1 Управление игровыми фигурами.....	10
2.1.2 Обработка столкновений .....	11
2.1.3 Уровни сложности.....	11
3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ .....	12
3.1 Структура входных и выходных данных .....	12
3.2 Разработка диаграммы классов .....	12
3.3 Описание классов .....	12
3.3.1 Класс игры.....	12
3.3.2 Шаблонные классы.....	14
3.3.3 Классы исключений .....	16
3.3.4 Другие классы .....	17
4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ .....	22
4.1 Разработка схем алгоритмов.....	22
4.2 Разработка алгоритмов.....	22
4.3 Исходный текст программы .....	23
5 РЕЗУЛЬТАТЫ РАБОТЫ .....	24
ЗАКЛЮЧЕНИЕ .....	26
ЛИТЕРАТУРА .....	27
ПРИЛОЖЕНИЕ А .....	28
ПРИЛОЖЕНИЕ Б .....	29
ПРИЛОЖЕНИЕ В .....	30
ПРИЛОЖЕНИЕ Г .....	31

## **ВВЕДЕНИЕ**

В современном мире программирования прикладного программного обеспечения имеется множество возможностей для создания эффективных и мощных приложений. Однако важно правильно выбирать инструменты и языки программирования, чтобы достичь желаемых результатов. В данной пояснительной записке я рассмотрю современные возможности разработки прикладного программного обеспечения с использованием языка высокого уровня C++.

Язык C++ является одним из самых мощных и универсальных языков программирования на сегодняшний день. Он предоставляет разработчикам широкие возможности для создания сложных и высокопроизводительных приложений. Возможности C++ включают в себя объектно-ориентированное программирование, стандартную библиотеку шаблонов (STL), а также обширные средства для многопоточного и системного программирования.

Для демонстрации силы и гибкости языка C++ в рамках данного курсового проекта я разработал собственную версию игры "Тетрис" с использованием C++ и графической библиотеки SFML.

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++.

Рассмотрим основные составляющие библиотеки SFML:

**Графика и отображение:** SFML предоставляет простые и эффективные средства для работы с графикой, включая отображение изображений, создание форм и применение текстур. Это делает ее отличным выбором для разработки игр, визуализаций и графических приложений.

**Аудио и звук:** Библиотека обеспечивает поддержку воспроизведения звуков и музыки, что позволяет создавать мультимедийные приложения с высококачественным звуковым сопровождением.

**Оконный менеджмент:** SFML упрощает создание и управление окнами приложения, предоставляя легкий доступ к событиям, таким как нажатие клавиш, движение мыши и другие.

**Переносимость:** SFML является кроссплатформенной библиотекой, что позволяет разрабатывать приложения, способные работать на различных операционных системах без изменений в исходном коде.

**Анимация:** Встроенная поддержка анимации делает SFML отличным выбором для создания динамичных и интерактивных визуальных эффектов.

Игра «Тетрис» – это одна из самых популярных и известных аркадных игр, созданная и разработанная Алексеем Пажитновым в 1984 году. Эта игра, хотя и известна своими простыми правилами, предоставляет отличный пример

приложения, которое требует эффективной обработки данных и оптимизации для создания плавного и увлекательной игры.

Данная игра обрела популярность благодаря следующим факторам:

«Тетрис» прост в понимании. Правила игры легко освоить, и играть в неё можно уже при первом запуске.

Падающие в игре фигуры требуют быстрого принятия решений и точного позиционирования, что развивает быстро мышления.

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

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.

В игре используется система рекордов, которая стимулирует игроков соревноваться между собой и стремиться установить новые рекорды.

Игра «Тетрис» является примером того, как можно использовать C++ для создания высокопроизводительного приложения. Мой проект включает в себя разработку игрового процесса, и графического интерфейса с использованием библиотеки SFML.

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

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

## **1 ПОСТАНОВКА ЗАДАЧИ**

Написать аналог классического игры «Тетрис». Игра должна иметь удобный пользовательский интерфейс.

Основные задачи:

1. Изучение существующих аналогов – анализ наиболее популярных и версий игры «Тетрис»;
2. Определение правил и механик игры – создание четкого и прямолинейного описания правил игры, с акцентом на то, как именно игра будет награждать пользователя за прохождение каждого уровня;
3. Разработка графического интерфейса – построение графического интерфейса, который будет прост для понимания;
4. Реализация игрового процесса – написание функций, реализующих игровую механику;
5. Тестирование приложения – проведение тестирования для обнаружения возможных ошибок и проблем.

Дополнительные функции: отображение текущего счета игрока, постепенное увеличение уровня сложности,

Специфические функции: возможность сохранять прогресс игры и загружать его в любое время, а также продуманную обработку исключительных ситуаций.

Для реализации данного проекта будет использован объектно-ориентированный язык программирования C++. Рекомендовано использовать мультимедийные библиотеки: SFML или SDL, с целью упрощения разработки графического интерфейса и управления.

## 2 ОБЗОР ЛИТЕРАТУРЫ

«Тетрис» был создан и выпущен в 1984 году программистом Алексеем Пажитновым из Советского Союза. Игра была вдохновлена его любовью к пазлам. Пажитнов создал игру, используя программное обеспечение Elektronika 60, а затем перенес ее на IBM PC. Игра была увидена генеральным директором компании Henk Rogers, который привнес игру на рынок США через его компанию, Bullet-Proof Software. Игра стала одной из самых известных и популярных видеоигр в мире.

«Тетрис» – это головоломка, в которой игроки должны управлять случайно падающими тетромино (геометрическими формами из четырех квадратных блоков), чтобы создать непрерывную горизонтальную линию. Когда линия формируется, она исчезает, и игроки получают очки. Если блоки достигают верха экрана, игра заканчивается.

Каждый тетромино имеет уникальную форму и может быть повернут, чтобы помочь игрокам заполнять пробелы. На рисунке 2.1 представлены семь тетромино, используемых в классическом Тетрисе.

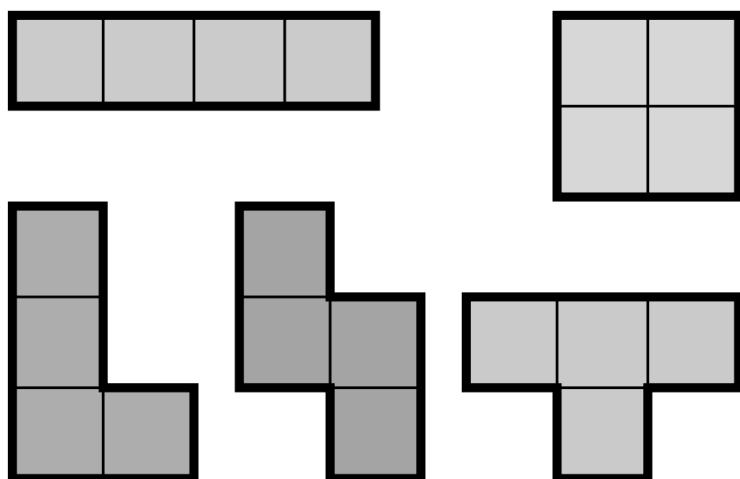


Рисунок 2.1 – Варианты тетромино

Важным аспектом является обработка столкновений между тетромино и игровым полем. Управление тетромино может осуществляться различными методами, такими как управление клавишами клавиатуры или сенсорными устройствами.

Для создания динамичного игрового процесса можно использовать различные методы увеличения сложности, такие как увеличение скорости падения тетромино или изменение набора доступных форм на разных уровнях.

Существует множество вариаций и аналогов Тетриса. Некоторые из самых популярных включают:

«Риуо Риуо Тетрис» – это кроссовер между двумя классическими головоломками: Риуо Риуо и Tetris. В этой игре игроки могут выбирать между

традиционным режимом Tetris и режимом Puyo Puyo, или даже играть в оба одновременно в режиме Fusion. Главная цель в Puyo Puyo - это соединять четыре или более "пуло" одного цвета, чтобы они исчезли. Игра предлагает разнообразные игровые режимы и опции для одного игрока или мультиплеера, что делает ее более разнообразной и интересной, чем классический Тетрис. На рисунке 2.2 показана эта разновидность игры «Тетрис».

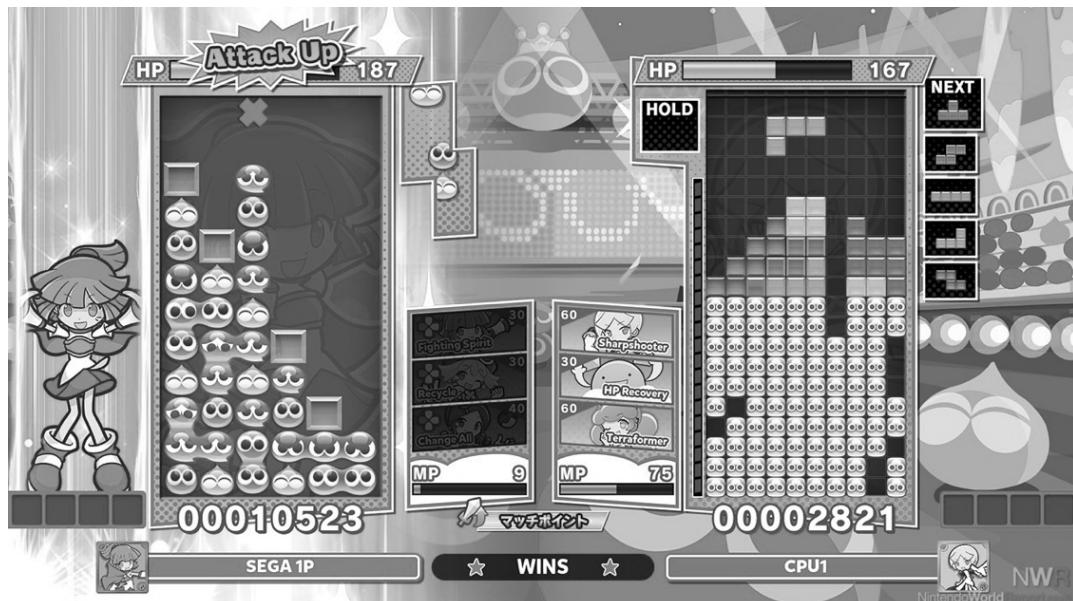


Рисунок 2.2 – «Puyo Puyo Tetris»

«Tetris Effect» – это современная интерпретация классического Тетриса, созданная дизайнером Tetsuya Mizuguchi (Рис 2.3).

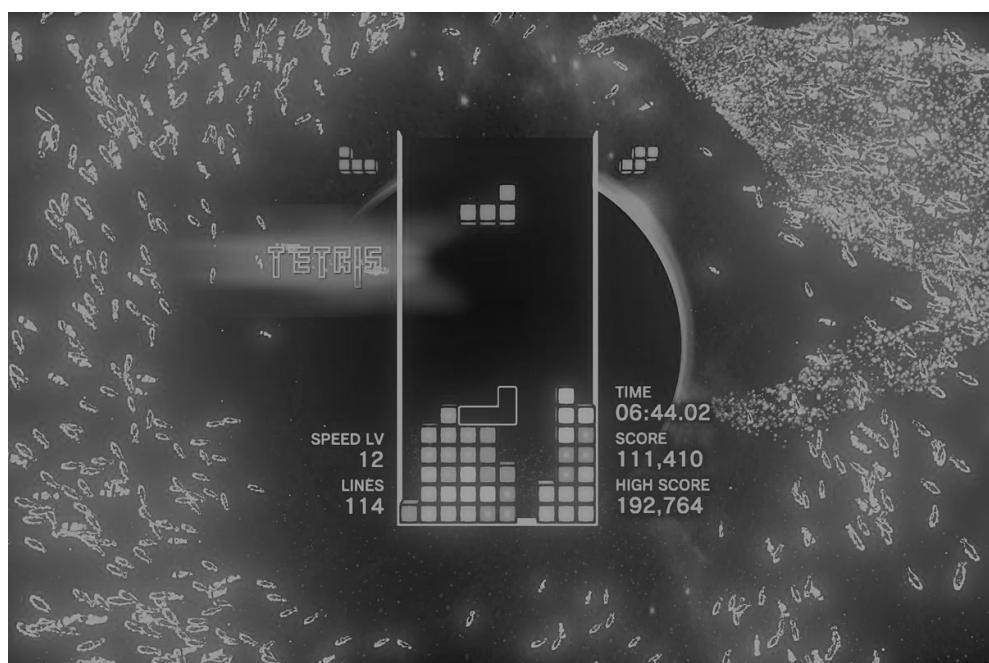


Рисунок 2.3 – «Tetris Effect»

Главным отличием этой игры является то, что каждое действие, которое игрок выполняет (перемещение, вращение, уничтожение линий), синхронизируется с музыкой и визуальными эффектами в игре. Это создает уникальный и увлекательный игровой опыт. Tetris Effect получил высокие оценки за свою инновационную концепцию и красивый дизайн.

«Lumines» – это еще одна головоломка, созданная Tetsuya Mizuguchi. В этой игре игроки управляют квадратами, состоящими из четырех меньших блоков разных цветов. Цель игры - создавать квадраты или прямоугольники одного цвета. Подобно Tetris Effect, Lumines интегрирует музыку и визуальные эффекты в основной геймплей, что создает уникальный и захватывающий игровой опыт. Lumines получил хорошие отзывы за свою музыку, визуальные эффекты и инновационный геймплей. Данная разновидность представлена на рисунке 2.4.



Рисунок 2.4 - «Lumines»

## 2.1 Обзор методов и алгоритмов решения поставленной задачи

### 2.1.1 Управление игровыми фигурами

Метод 1: Управление клавишами клавиатуры.

Для управления игровыми фигурами можно использовать метод, основанный на отслеживании событий клавиш клавиатуры при помощи библиотеки SFML. Это позволяет игроку легко перемещать и вращать фигуры с помощью клавиш клавиатуры.

Данный подход обладает простотой и интуитивностью для игрока. Игрок может мгновенно реагировать на изменения в игре.

Управление клавишами клавиатуры хорошо подходит для игр на компьютере и легко настраивается, однако для мобильных устройств, управление клавишами может быть менее удобным.

Метод 2: Управление сенсорными устройствами.

Возможно управление игровыми фигурами через события сенсорных устройств на мобильных устройствах. Это позволит игрокам легко перемещать и вращать фигуры, касаясь экрана смартфона.

### **2.1.2 Обработка столкновений**

Метод 1: Матричный подход.

Для обработки столкновений между фигурами и игровым полем, можно использовать матричный подход. Игровое поле и фигуры представлены в виде матриц, где каждая ячейка соответствует клетке поля или фигуры. При перемещении и вращении фигур происходит сравнение значений матрицы фигуры с матрицей игрового поля.

Подход обеспечивает точную обработку столкновений и контроль над каждой ячейкой, позволяет точно определять столкновения и взаимодействие между фигурами и полем. Однако матричный подход может потребовать дополнительного объема памяти для хранения матриц.

Метод 2: Использование булевых массивов.

Для каждой клетки поля я использовал булевый массив для отслеживания ее состояния (занята или свободна). Это позволяет быстро определять столкновения без необходимости хранения матриц.

Этот метод обеспечивает быструю обработку столкновений и более экономичное использование памяти. Он позволяет оптимизировать использование памяти и быстро определять столкновения. Однако требуется дополнительный код для обновления булевых массивов при каждом движении фигур.

### **2.1.3 Уровни сложности**

Метод 1: Увеличение скорости падения.

Постепенное увеличение скорости падения тетромино с уровнем сложности. На каждом новом уровне фигуры будут падать быстрее, что делает игру более динамичной и вызывает у игроков больше вызовов.

Данный подход позволяет контролировать уровень сложности и усилить динамичность игры. Уровни сложности обеспечивают разнообразие игрового процесса и возможность для игроков улучшать свои навыки.

Метод 2: Изменение набора тетромино.

Возможно добавление различных наборов тетромино на разных уровнях сложности. Новые фигуры добавят разнообразие и вызовут у игроков новые тактические решения. Однако это требует дополнительной разработки фигур и балансировки уровней.

### **3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ**

В данном разделе описываются входные и выходные данные программы, диаграмма классов, а также приводится описание используемых классов и их методов.

#### **3.1 Структура входных и выходных данных**

Файл Game.bin хранит данные о последней сохраненной партии в бинарном файле.

Для хранения данных игроков используется файл BestPlayersInfo.txt структура которого представлена в таблице 3.1.1.

Таблица 3.1.1 - Файл BestPlayersInfo.txt

Никнейм	Игровой счет
Daniel	87439

Файл color\_cubes.png хранит изображение клеток фигур.

Файл menu.png хранит изображение главного меню.

Файл buttonStart.png хранит изображение нажатой кнопки «Начать игру».

Файл buttonResume.png хранит изображение нажатой кнопки «Продолжить игру».

Файл buttonExit.png хранит изображение нажатой кнопки «Завершить игру».

Файл pause.png хранит изображение не нажатой кнопки паузы.

Файл unpause.png хранит изображение нажатой кнопки паузы.

Файл board.png хранит изображение игрового поля.

Файл rows.png хранит изображение блока, в котором выводится количество удаленных линий.

Файл buttonON.png хранит изображение включенной кнопки музыки.

Файл buttonOFF.png хранит изображение выключенной кнопки музыки.

Файл restart.png хранит изображение кнопки «Начать игру заново».

Файл rules.png хранит изображение правил игры

Файл gameOver.png хранит изображение экрана после проигрыша.

#### **3.2 Разработка диаграммы классов**

Диаграмма классов представлена в приложении А.

#### **3.3 Описание классов**

##### **3.3.1 Класс игры**

Класс Game реализует игровую логику.

Поля:

Board field – игровое поле.

Figure\* currentFigure – текущая фигура.

Figure\* nextFigure – следующая фигура.

std::vector<Figure\*> figures – вектор всех возможных фигур.

Button buttonPause – кнопка паузы.

Button buttonRestart – кнопка для начала игры заново.

Button buttonMusic – кнопка для включения/выключения музыки.

Picture buttonGameOver – экран окончания игры.

Picture buttonRowsCount; – картинка количества сожженных линий.

Picture oneBlock – изображение одного кубика.

Picture\_Sprite pauseBoard – изображение доски во время паузы.

sf::Font font – шрифт текста.

sf::Text text – текст.

sf::Clock gameTime – переменная для вычисления пройденного игрового времени.

sf::Music music – музыка в игре.

int lines\_in\_a\_row – кол-во линий сожженных одновременно.

int score – игровой счет.

int time – игровое время.

int fileTime – игровое время загруженное из файла.

int tmpTime – игровое время во время паузы.

int countLines – количество сожженных линий.

std::string number – вспомогательная строка для перевода в число.

Queue<PlayerInfo> infoQueue – очередь из никнеймов лучших игроков и их счета.

Методы:

Game() – конструктор класса Game.

int keyPressCheck(sf::Event& event, int key) – проверка нажатых клавиш.

int mousePressedCheck(sf::Event& event, sf::RenderWindow& window) – проверка нажатий мышкой.

void buttonAction (int& key) – действие в зависимости от нажатой клавиши.

void fallingFigure (sf::Clock& timer, float pause) – падение фигуры.

Figure\* getRandomFigure() – получить случайную фигуру.

void getAllFigures() – возобновить вектор всех фигур.

void drawBoardImage (sf::RenderWindow& window) – рисование игрового поля.

void draw(sf::RenderWindow& window) – рисование всех кнопок и изображений.

void drawNextFigureBlock(sf::RenderWindow& window) – рисование следующей фигуры.  
bool boundariesIsBroken () – проверка на столкновение.  
void isLocked() – проверка на приземление фигуры.  
int distanceToLocked () – вычисление расстояния до приземления.  
void drawPlacedBlocks(sf::RenderWindow& window) – рисование упавших блоков.  
bool gameOver(sf::RenderWindow& window) – проверка на конец игры.  
void checkAndClearFilledLines () – проверка на заполнение линии.  
void deleteLine (int num, int count) – удаление заполненной линии.  
void readFileBestPlayers(const char\* fileName) – прочитать статистику игроков из текстового файла.  
void writeFileBestPlayers(const char\* fileName) – записать статистику игроков в текстовый файл.  
void showBestPlayersBlock(sf::RenderWindow& window) – вывести статистику пяти лучших игроков на экран.  
void scoreBooster (int& \_lines\_in\_a\_row) – начисление очков за соженные линии.  
void showGameTime(sf::RenderWindow& window) – показ пройденного игрового времени.  
void showScore (sf::RenderWindow& window) – показ количества набранных очков.  
void checkStatisticBeforeSave() – занести информацию о текущем игроке по окончанию игры в файл со статистикой других игроков.  
bool processGameCycle (sf::RenderWindow& window) – обработка игрового цикла.  
void loadGameFromFile(std::string fileName) – прочитать прошлую игру из бинарного файла.  
void saveGameToFile(std::string fileName) – сохранить игру в бинарный файл.  
~Game() – деструктор.

### 3.3.2 Шаблонные классы

Класс List<T, Allocator> – шаблонный класс, реализующий двусвязный список.

Поля:

size\_t size\_ – размер списка.  
FakeListNode fake\_ – фейковый узел списка.  
node\_allocator allocator\_ – аллокатор списка.

Методы:

List(const Allocator& alloc = Allocator()) – конструктор.  
~List() – деструктор.

`push_back(const T& element)` – добавление элемента в конец списка.

`pop_back()` – удаление элемента из конца списка.

`push_front(const T& element)` – добавление элемента в начало списка.

`pop_front()` – удаление элемента из начала списка.

`empty()` – проверка списка на пустоту.

`size() const` – получение размера списка.

`List(const List& other)` – конструктор копирования.

`List<T>& operator=(List<T> other)` – оператор присваивания.

`iterator begin() const` – возвращает итератор, указывающий на начало списка.

`iterator end() const` – возвращает итератор, указывающий на конец списка.

`reverse_iterator rbegin()` – возвращает обратный итератор, указывающий на конец списка.

`reverse_iterator rend()` – возвращает обратный итератор, указывающий на начало списка.

`const_iterator cbegin() const` – возвращает константный итератор, указывающий на начало списка.

`const_iterator cend() const` – возвращает константный итератор, указывающий на конец списка.

`const_reverse_iterator rbegin() const` – возвращает константный обратный итератор, указывающий на конец списка.

`const_reverse_iterator rend() const` – возвращает константный обратный итератор, указывающий на начало списка.

`T& front() и const T& front() const` – доступ к первому элементу

`T& back() и const T& back() const` – доступ к последнему элементу

`static void swap(List& left, List& right)` – обмен содержимым двух списков - `left` и `right`.

Структура `ListNode` – вложенная структура узла списка `List<T, Allocator>`.

Поля:

`T element` – данные элемента списка.

`FakeListNode* prev` – указатель на предыдущий узел в списке.

`FakeListNode* next` – указатель на следующий узел в списке.

Методы:

`ListNode(const T& elem)` – конструктор, принимающий константную ссылку на элемент.

`ListNode(T&& elem)` – конструктор, принимающий `rvalue` ссылку на элемент.

Структура `FakeListNode` – вложенная структура узла списка `List<T, Allocator>`.

Поля:

`FakeListNode* prev` – указатель на предыдущий узел списка.

`FakeListNode* next` – указатель на следующий узел списка.

Класс `BasicIterator` – вложенный класс, реализующий функциональность итератора в `List<T, Allocator>`.

Поля:

`FakeListNode* ptr_` – указатель на текущий узел.

Методы:

`BasicIterator(FakeListNode* node);` – конструктор, принимающий указатель на узел.

`BasicIterator& operator++()` – оператор инкремента (prefix).

`BasicIterator operator++(int)` – оператор инкремента (postfix).

`BasicIterator& operator--()` – оператор декремента (prefix).

`BasicIterator operator--(int)` – оператор декремента (postfix).

`reference operator*()` и `pointer operator->()` – операторы разыменования.

`bool operator==(const BasicIterator& other) const` – оператор сравнения на равенство.

`bool operator!=(const BasicIterator& other) const` – оператор сравнения на неравенство.

### 3.3.3 Классы исключений

Класс `Exceptions` – класс ошибок.

Поля:

`std::string message` – описание ошибки.

Методы:

`Exceptions()` – конструктор по умолчанию.

`const char* what() const noexcept override` – конструктор с параметрами .

Класс `ExceptionFile` – класс ошибок работы с файлами.

Методы:

`explicit ExceptionFile(std::string mes)` – конструктор.

Класс `ExceptionSFML` – класс ошибок работы с библиотекой SFML.

Методы:

`explicit ExceptionSFML(std::string mes)` – конструктор.

Класс `OutOfBoundsException` – класс ошибок выхода за границы.

Методы:

`explicit OutOfBoundsException(std::string mes)` – конструктор.

### 3.3.4 Другие классы

Класс Board – класс игрового поля.

Поля:

`sf::Image gridImage` – изображение игрового поля.

`sf::Texture gridTexture` – текстура игрового поля.

`sf::Sprite gridSprite` – спрайт игрового поля.

`int width` – ширина игрового поля.

`int height` – высота игрового поля.

`int gameBoard[HEIGHT + 1][WIDTH]` – массив чисел

представляющий собой игровое поле.

Методы:

`Board()` – конструктор.

`sf::Sprite& getGridSprite()` – получить спрайт поля.

`void drawGameBoard(sf::RenderWindow& window)` – нарисовать игровое поле.

`Int getGameBoard(int x, int y) const` – получить значение игрового поля.

`void setGameBoard(int x, int y, int value)` – установить значение игрового поля.

`void initializeVector()` – заполнить игровое поле.

`int getWidth() const` – получить ширину игрового поля.

`int getHeight() const` – получить высоту игрового поля.

`~Board()` – деструктор.

Класс Button – класс кнопки.

Поля:

`float width` – ширина кнопки.

`float height` – высота кнопки.

`sf::SoundBuffer buffer` – звуковой буфер.

`sf::Sound sound` – звук нажатия кнопки.

`bool isPressed` – состояние кнопки.

Методы:

`explicit Button(std::string _someText, float w, float h, std::string fileName, float x, float y, std::string fontName, int size, float x_pos, float y_pos)` – конструктор.

`void draw(sf::RenderWindow& window) override` – функция рисования кнопки.

`float getWidth()` – получить ширину кнопки.

`float getHeight()` – получить высоту кнопки.

`void playMusic()` – издать звук нажатия кнопки.

`bool getIsPressed()` – узнать состояние кнопки.

`void setIsPressed(bool val)` – установить состояние кнопки.  
`~Button()` – деструктор.

Класс `Block` – игровой блок.

Поля:

`int x` – координата по x.  
`int y` – координата по y.

Методы:

`Block(int x, int y)` – конструктор.  
`~Block()` – деструктор.

Класс `Figure` – абстрактный класс фигуры.

Поля:

`std::vector <Block> status` – вектор текущего состояния блоков фигуры.

`int rotationStatus` – текущее состояние поворота.  
`int color` – цвет фигуры.  
`int offsetX` – смещение координат фигуры по x.  
`int offsetY` – смещение координат фигуры по y.  
`int heightOfBlock` – высота фигуры.  
`int distanceToCollision` – расстояние до столкновения.  
`Picture cubeImage` – изображение одного блока фигуры.  
`Picture shadowCube` – полупрозрачное изображение блока фигуры.  
`int type` – тип фигуры.

Методы:

`Figure()` – конструктор.  
`int getType() const` – получить тип фигуры.  
`int getColor() const` – получить цвет фигуры.  
`void setDistanceToCollision(int x)` – расстояние до столкновения.  
`sf::Sprite& getCubeSprite()` – получить спрайт блока фигуры.  
`void drawFigure(sf::RenderWindow& window)` – нарисовать фигуру.  
`void move(int xPos, int yPos)` – переместить фигуру.  
`std::vector<Block> calculateMovedPosition ()` – вычисляет и возвращает новое положение блоков фигуры после её перемещения .  
`std::vector<Block>& getStatus()` – получить вектор координат фигуры.  
`int get_offset_x() const` – получить координату по x.  
`int get_offset_y() const` – получить координату по y.  
`virtual void rotateFigure (bool flag) = 0` – виртуальная функция поворота фигуры.  
`virtual ~Figure()` – виртуальный деструктор.

Класс `J_Block` – фигура типа J.

Методы:

`J_Block()` – конструктор для инициализации типа фигуры и возможных ее координат при разных поворотах.

`void rotateFigure(bool flag) override` – функция поворота фигуры.  
`~J_Block() override` – деструктор.

Класс `S_Block` – фигура типа S.

Методы:

`S_Block()` – конструктор для инициализации типа фигуры и возможных ее координат при разных поворотах.

`void rotateFigure(bool flag) override` – функция поворота фигуры.  
`~S_Block() override` – деструктор.

Класс `Z_Block` – фигура типа Z.

Методы:

`Z_Block()` – конструктор для инициализации типа фигуры и возможных ее координат при разных поворотах.

`void rotateFigure(bool flag) override` – функция поворота фигуры.  
`~Z_Block() override` – деструктор.

Класс `L_Block` – фигура типа L.

Методы:

`L_Block()` – конструктор для инициализации типа фигуры и возможных ее координат при разных поворотах.

`void rotateFigure(bool flag) override` – функция поворота фигуры.  
`~L_Block() override` – деструктор.

Класс `O_Block` – фигура типа O.

Методы:

`O_Block()` – конструктор для инициализации типа фигуры и возможных ее координат при разных поворотах.

`void rotateFigure(bool flag) override` – функция поворота фигуры.  
`~O_Block() override` – деструктор.

Класс `I_Block` – фигура типа I.

Методы:

`I_Block()` – конструктор для инициализации типа фигуры и возможных ее координат при разных поворотах.

`void rotateFigure(bool flag) override` – функция поворота фигуры.  
`~I_Block() override` – деструктор.

Класс `T_Block` – фигура типа T.

Методы,

T\_Block() – конструктор для инициализации типа фигуры и возможных ее координат при разных поворотах.

void rotateFigure(bool flag) override – функция поворота фигуры.

~T\_Block() override – деструктор.

Класс GameMenu – класс игрового меню.

Поля:

int selectedMenuItem – текущая активная кнопка в меню.

int key – нажатая клавиша.

Button buttonStart – кнопка старта.

Button buttonResume – кнопка рестарт.

Button buttonExit – кнопка выхода из игры.

Picture mainMenu – изображение главного меню.

bool isMenu – отслеживает видимость меню в игре.

Методы:

GameMenu() – конструктор.

void showMenu(sf::RenderWindow& window, Game& game) – показать меню.

void keyPressCheck(sf::Event& event) – проверка нажатия клавиш.

void buttonAction(Game& game) – действие в зависимости от нажатой клавиши.

bool getIsMenu() const – получить значение isMenu.

void setIsMenu(bool x) – установить значение isMenu.

~GameMenu() – деструктор.

Класс Picture – класс игрового изображения.

Поля:

sf::Image image – изображение.

sf::Texture texture – текстура.

sf::Sprite sprite – спрайт.

float x\_coordinate – координата изображения по x.

float y\_coordinate – координата изображения по y.

Методы:

explicit Picture(std::string fileName, float x, float y) – конструктор.

virtual void setPosition (float x, float y) override – установить позицию изображения.

virtual void draw(sf::RenderWindow& window) override – нарисовать изображение.

float getPositionX () const – получить координату по x.

float getPositionY () const – получить координату по y.

void updateSprite(std::string fileName) – поменять спрайт изображения.

Класс PlayerInfo – класс игрока.

Поля:

std::string nickName – имя игрока.  
int score – игровой счет игрока.

Методы:

PlayerInfo(const std::string& name = "", int \_score = 0) – конструктор с параметрами по умолчанию.

PlayerInfo& operator=(const PlayerInfo& obj) – перегрузка оператора равно.

std::string getNickName() const – получить значение имени игрока.  
int getScore() const – получить значение счета игрока.  
void setNickName(const std::string& name) – установить имя игрока.  
void setScore(int \_score) – установить счет игрока.

Класс Text – класс текста.

Поля

std::string someText – текст.  
sf::Font font – шрифт.  
sf::Text text – текст отображаемый на игровом экране.  
bool isSelected – переменная для типа отображения текста.  
bool hasLimit – есть ли ограничения по вводу текста.  
float x\_pos – позиция по x.  
float y\_pos – позиция по y.  
int limit – ограничения длины текста.

Методы:

explicit Text(const std::string& \_someText, std::string& fontName, int size, int x, int y) – конструктор.

Text (int size, bool sel, std::string fontName) – конструктор.  
Text (std::string fontName) – конструктор.  
void inputLogic (int charTyped) – обработка ввода пользователем.  
void deleteLastChar () – удалить последний символ строки.  
void setStrAsNumber (float num) – перевести число в строку.  
virtual void draw(sf::RenderWindow& window) – нарисовать текст.  
void drawNumber (sf::RenderWindow& window, int number) – нарисовать число на экране.

void setString (std::string str) – установить текст.  
virtual void setPosition(float x, float y) – установить позицию.  
void setCharacterSize (int size) – установить размер текста.  
void setLimit(bool x) – установить ограничение по длине текста.  
std::string getString() – получить значение текста.  
void typeOn (sf::Event& event, sf::RenderWindow& window) – управление процессом ввода текста.

## **4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ**

### **4.1 Разработка схем алгоритмов**

Метод `checkAndClearFilledLines()` класса `Game` выполняет проверку и удаление полностью заполненных линий на игровом поле. Схема представлена в приложении Б.

Метод `deleteLines()` класса `Game` выполняет удаление полностью заполненных линий на игровом поле. Схема представлена в приложении В.

### **4.2 Разработка алгоритмов**

Метод `swap(List& left, List& right)` класса `List` (обмен содержимым между двуми списками):

Шаг 1: Начало.

Шаг 2:

Входные данные:

`List& left, List& right` – объекты которые надо поменять местами.

Шаг 3: Объявление переменных:

`FakeListNode* tmpl` – предыдущий узел правого списка.

`FakeListNode* tmpr` – следующий узел правого списка.

Шаг 4: Если свойство `propagate_on_container_swap::value` равно `true`, обмен владеющими указателями `allocator_` списков `left` и `right`.

Шаг 5: Временные узлы сохраняют указатели на предыдущий (`tmpl`) и следующий (`tmpr`) узел из списка `right`.

Шаг 6: Проверка, если размер списка `left` не равен 0, то узлы между `left.fake_.prev` и `left.fake_.next` вставляются после узла `&right.fake` с помощью метода `insert_nodes()`, иначе – устанавливается связь между узлами `&right.fake_` с помощью метода `connect_nodes()`.

Шаг 7: Если размер списка `right` не равен 0, то узлы между `tmpl` и `tmpr` вставляются после узла `&left.fake_` с помощью метода `insert_nodes`, иначе – устанавливается связь между узлами `&left.fake_` с помощью метода `connect_nodes()`.

Шаг 8: Обмен размерами списков с пощью `std::swap()`.

Шаг 9: Конец.

Метод `inputLogic(int charTyped)` класса `Text` (обработка ввода пользователя в объекте класса `Text`):

Шаг 1: Начало.

Шаг 2: Входные данные: `int charTyped` – символ, введенный пользователем.

Шаг 3: Если введенный символ не является клавишами Backspace или Enter, то добавить его к полю `txt` класса `Text`, иначе переход на шаг 4.

Шаг 4: Если введенный символ является клавишей Backspace, то

переходим к шагу 5, иначе переход на шаг 6.

Шаг 5: Если длина больше 0, удаляем последний символ с помощью метода deleteLastChar(), иначе переходим на шаг 6, т. к строка пустая.

Шаг 6: Устанавливаем строку текста, добавив символ курсора ("|") в конец (text.setString(txt.str() + "|").

Шаг 7: Конец.

### **4.3 Исходный текст программы**

Код программы представлен в приложении Г.

## 5 РЕЗУЛЬТАТЫ РАБОТЫ

На рисунке 5.1 представлено главное меню программы, на котором написаны краткие правила игры и рассказано про систему управления в игре. В меню присутствуют кнопки начала, продолжения игры и выхода из нее.

На рисунке 5.2 показан процесс игры. Во время игры отображается игровое время, счет, количество сожженных линий. В левом верхнем углу представлена таблица пяти лучших игроков. Имеются кнопки для рестарта игры, паузы, отключения музыки.

На рисунке 5.3 показано сжигание четырех линий и начисление за это игровых очков.

На рисунке 5.4 показан пример работы паузы в игре.

На рисунке 5.5 показан экран окончания игры.



Рисунок 5.1 – Главное меню программы

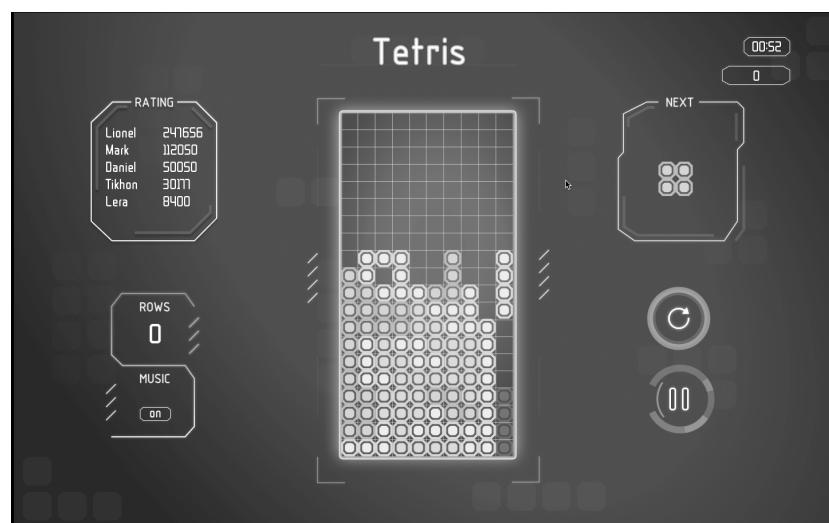


Рисунок 5.2 – Пример работы программы

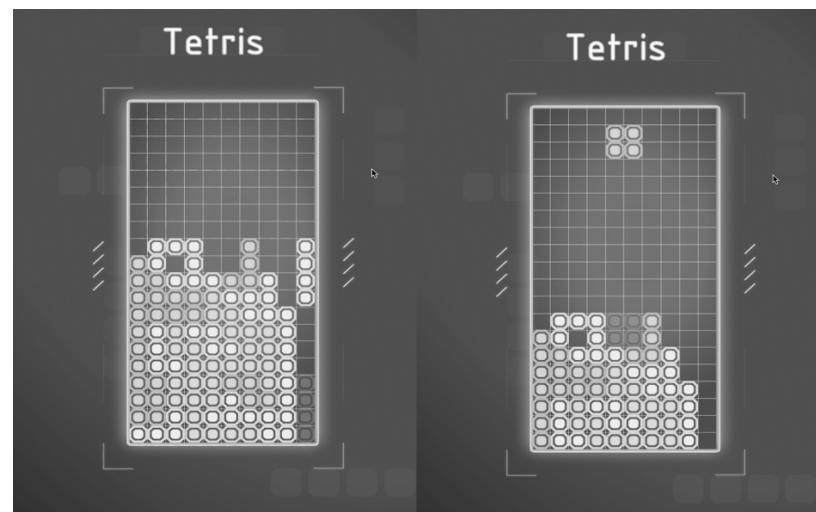


Рисунок 5.3 – Удаление 4-х линий

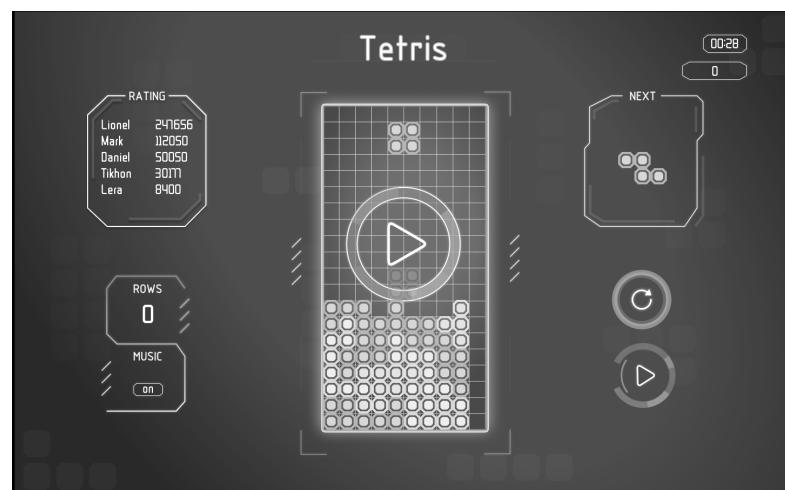


Рисунок 5.4 – Работа паузы в игре

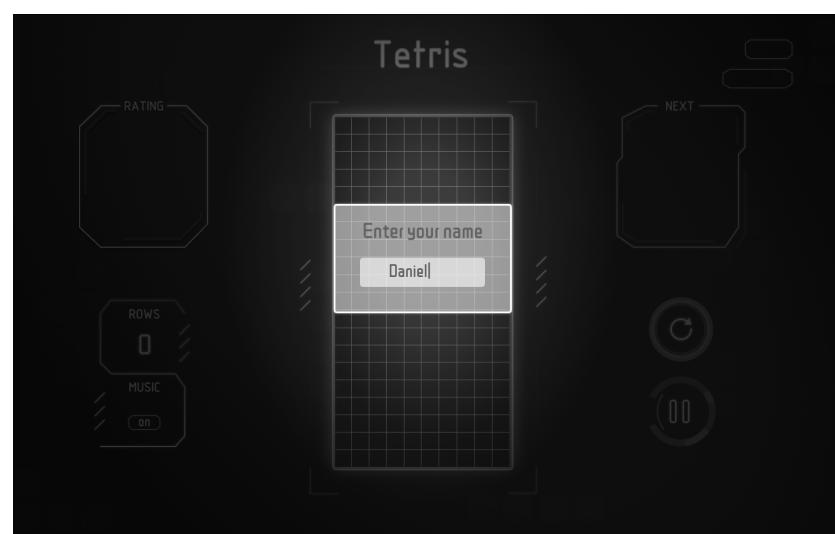


Рисунок 5.5 – Конец игры (поражение)

## **ЗАКЛЮЧЕНИЕ**

В процессе выполнения курсового проекта были выполнены первоначально заданные цели, результатом которых является игра «Тетрис» с графическим интерфейсом. Были также реализованы различные алгоритмы, включая проверку и удаление заполненных линий, обработку ввода пользователя и генерацию новых фигур, продолжение прошлой игры.

Игровые приложения, такие как «Тетрис», являются прекрасной демонстрацией возможностей информационных систем в области развлечений и отдыха. Они обеспечивают удобное и увлекательное средство для прохождения времени, обучения или увлечения. «Тетрис», в частности, заставляет игроков мыслить стратегически и быстро принимать решения, стимулируя таким образом развитие когнитивных навыков.

Есть возможность сохранения игрового прогресса пользователя, что дает игроку возможность отслеживать свой прогресс и ставить перед собой новые цели.

Программа может быть улучшена в дальнейшем:

Можно добавить различные уровни сложности: обычно в играх «Тетрис» уровни сложности нарастают по мере продвижения игрока. Можно предложить разнообразные варианты сложности или даже интересные "тематические" уровни.

Добавление соревновательного режима также положительно повлияет на игру, так как игроки смогут сражаться друг с другом в реальном времени.

Настройки пользовательского интерфейса предоставим игрокам возможности настроить внешний вид и управление игры также может улучшить пользовательский опыт.

## ЛИТЕРАТУРА

- [1] Роджерс Н. Тетрис: Игры, в которые играют люди / Н. Роджерс. - First Second, 2016. – 256 с.
- [2] Шейнин Д. Тетрис: История самой популярной в мире игры / Д. Шейнин. - Омега-Л, 2016. – 416 с.
- [3] Луцик Ю. А. Объектно-ориентированное программирование на языке C++: учеб. пособие /Ю. А. Луцик, В. Н. Комличенко. – Минск: БГУИР, 2008.–266 с.
- [4] Дейтел, Х.М. Как программировать на C++ / Х.М. Дейтел, П.Д. Дейтел; пер. с англ. – М. : Бином, 2007. – 1152 с.
- [5] Страуструп, Б. Язык программирования C++ / Б. Страуструп; специальное издание. Пер. с англ. – СПб. : BHV, 2008. – 1098 с.
- [6] Лафоре Р. Объектно-ориентированное программирование с C++ / Р. Лафоре. - 4-е издание М.:Питер, 2004. – 923 с.
- [7] Майерс С. Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14 / С. Майерс. - O'Reilly Media, 2014. – 334 с.
- [8] Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих / А. Бхаргава. - М.: ДМК Пресс, 2017. – 288 с.

## **ПРИЛОЖЕНИЕ А**

(обязательное)

Диаграмма классов

**ПРИЛОЖЕНИЕ Б**  
(обязательное)

Схема алгоритма checkAndClearFilledLines() класса Game

**ПРИЛОЖЕНИЕ В**  
(обязательное)  
Схема алгоритма deleteLine() класса Game

**ПРИЛОЖЕНИЕ Г**  
(обязательное)  
Исходный текст программы