

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 каф. ЭВМ
Марзалюк А. В.

Минск 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	5
1 ПОСТАНОВКА ЗАДАЧИ	7
2 ОБЗОР ЛИТЕРАТУРЫ	8
2.1 Обзор методов и алгоритмов решения поставленной задачи.....	8
2.1.1 Управление игровыми фигурами.....	8
2.1.2 Обработка столкновений	9
2.1.3 Уровни сложности.....	9
3. ФУНКЦИАНАЛЬНОЕ ПРОЕКТИРОВАНИЕ	9
3.1 Структура входных и выходных данных	10
3.2 Разработка диаграммы классов	10
3.3 Описание классов.....	10
3.3.1 Класс игры	10
3.3.2 Шаблонные классы	12
3.3.3 Классы исключений	13
3.3.4 Другие классы.....	13
4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ.....	19
4.1 Разработка схем алгоритмов.....	19
4.2 Разработка алгоритмов	19
4.3 Листинг кода	20
5 РЕЗУЛЬТАТЫ РАБОТЫ.....	21
ЗАКЛЮЧЕНИЕ	23
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	24
ПРИЛОЖЕНИЕ А	25
ПРИЛОЖЕНИЕ Б.....	26
ПРИЛОЖЕНИЕ В	27
ПРИЛОЖЕНИЕ Г	28
ПРИЛОЖЕНИЕ Д	29

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.

Полезность библиотеки SFML в C++:

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 увлекательного геймплея.

Игра "Тетрис" является примером того, как можно использовать C++ для создания высокопроизводительного приложения. Мой проект включает в себя разработку игрового движка, 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

Цель данного курсового проекта заключается в создании современного и графически привлекательного аналога классической игры "Тетрис". Основной упор сделан на глубокую погруженность пользователя в игровой процесс, а также на построение интуитивно понятного и удобного пользовательского интерфейса.

Основные задачи проекта:

Изучение существующих аналогов: Анализ наиболее популярных и успешных версий игры "Тетрис", чтобы понять основные механики, которые делают игровой процесс привлекательным, и видеть, какие аспекты можно усовершенствовать.

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.

Разработка графического интерфейса: Построение графического интерфейса, который будет прост для понимания, легко управляем, но при этом обладает глубиной, чтобы держать внимание пользователя.

Реализация игрового процесса: Представление тщательно продуманной игровой механики для достижения подлинного игрового опыта.

Тестирование приложения: Проведение тщательно планированных тестов для гарантирования производительности и надежности приложения, а также урегулирования любых возможных ошибок и проблем.

Специфические функции, которые будут включены в этот проект, включают отображение текущего счета игрока, постепенное увеличение уровня сложности, 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, такие как SFML или SDL, с целью упрощения разработки графического интерфейса и управления.

2 ОБЗОР ЛИТЕРАТУРЫ

Тетрис – это классическая аркадная игра, созданная программистом Алексеем Пажитновым в 1984 году. Она стала одной из самых известных и популярных видеоигр в мире. В игре используются геометрические формы, называемые тетромино, которые состоят из четырех квадратных блоков.

Цель игры – управлять падающими тетромино и формировать горизонтальные линии без промежутков. Когда линия полностью заполняется, она исчезает, принося игроку очки.

В игре используются семь различных форм тетромино, каждая с уникальной комбинацией квадратных блоков. Эти формы могут быть вращены и перемещены игроком для оптимального распределения по игровому полю.

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.

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.

Метод 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.

Управление сенсорами обеспечит адаптацию игры для мобильных платформ. Это будет удобно для игроков, использующих сенсорные устройства, но требуется дополнительная настройка и адаптация для различных устройств.

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.

Метод 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.

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.

Метод позволяет контролировать уровень сложности и усилить динамичность игры. Уровни сложности обеспечивают разнообразие игрового процесса и возможность для игроков улучшать свои навыки. Однако требует балансировки, чтобы сохранить игру интересной и честной.

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. ФУНКЦИАНАЛЬНОЕ 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.

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 ненажатой 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 сожженных 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 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 сожженные 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 Node<T> – шаблонный класс узла бинарного дерева.
Поля:

T data – данные.

Node *next – указатель на следующий элемент.

Методы:

Node(T data = 0, Node *next = nullptr) – конструктор с параметрами по умолчанию.

Класс Queue<T> – шаблонный класс очереди.

Поля:

Node<T> *head – указатель на голову очереди.

Node<T> *tail – указатель на хвост очереди.

int size – размер очереди.

Методы:

Queue() – конструктор.

`~Queue()` – деструктор.
`void enqueue(T data)` – добавление элемента в конец.
`void dequeue()` – удаление элемента из начала очереди.
`T front() const` – получение элемента из начала очереди без удаления.
`bool isEmpty() const` – проверка очереди на пустоту.
`int getSize() const` – получение размера очереди.
`void printQueue()` – печать элементов очереди, начиная с головы.
`Queue(const Queue<T>& other)` – конструктор копирования.
`Queue<T>& operator=(Queue<T> other)` – перегрузка оператора присваивания.
`bool operator==(const Queue<T>& other) const` – перегрузка оператора равенства.

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 высотку 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 selectedMenuOption – 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 клавиши.

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

на экране.

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 enqueue() шаблонного класса Queue (добавление в конец)

Шаг 1: Начало.

Шаг 2:

Входные данные:

T data - добавляемые данные

Переменные:

Node *node - новый узел.

head, tail, size - члены класса, где

head - указатель на начало очереди

tail - конец очереди

size - размер очереди.

Шаг 3: Создание нового узла node с данными data. Указатель next нового узла устанавливается на nullptr, показывая, что данный узел будет последним в очереди.

Шаг 4: Проверка, существует ли tail (конец очереди). Если tail существует, то его следующим элементом становится node. Это означает, что новый узел добавляется в конец очереди.

Шаг 5: Если tail не существует, то head (начало очереди) устанавливается на node. Это означает, что новый узел является единственным в очереди.

Шаг 6: Увеличиваем size на 1, обновляя размер очереди.

Шаг 7: Конец.

Метод checkStatisticBeforeSave() класса Game(обновляет статистику игрока, проверяя текущий счет, и если он больше счета в очереди, добавляет его в правильном порядке.)

Шаг 1: Начало.

Шаг 2:

Переменные:

Queue<PlayerInfo> tempQueue - временная очередь для хранения информации об играх.

bool isScoreAdded - булева переменная, указывающая, добавлен ли новый показатель счета.

Шаг 3: Запускается цикл, который продолжается, пока очередь infoQueue не опустеет.

Шаг 4: Извлекается первый элемент из infoQueue в переменную current и удаляется из infoQueue.

Шаг 5: Если новый счет еще не добавлен (!isScoreAdded) и текущий счет (score) больше счета текущего игрока (current.getScore()), то создается новый объект PlayerInfo со значениями nickName и score, и помещается в tempQueue. Переменная isScoreAdded устанавливается в true.

Шаг 6: Текущий объект игрока (current) помещается в tempQueue.

Шаг 7: Если после завершения цикла новый счет еще не добавлен (!isScoreAdded), он добавляется в конец tempQueue.

Шаг 8: Запускается новый цикл, который продолжается, пока tempQueue не опустеет. В этом цикле все элементы из tempQueue перемещаются обратно в infoQueue.

Шаг 9: Конец

4.3 Листинг кода

Код программы представлен в приложении Г.

5 РЕЗУЛЬТАТЫ РАБОТЫ

На рисунке 5.1 представлено главное меню программы. Написаны краткие правила игры и рассказано про систему управления в игре. В Меню присутствуют кнопки начал, 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

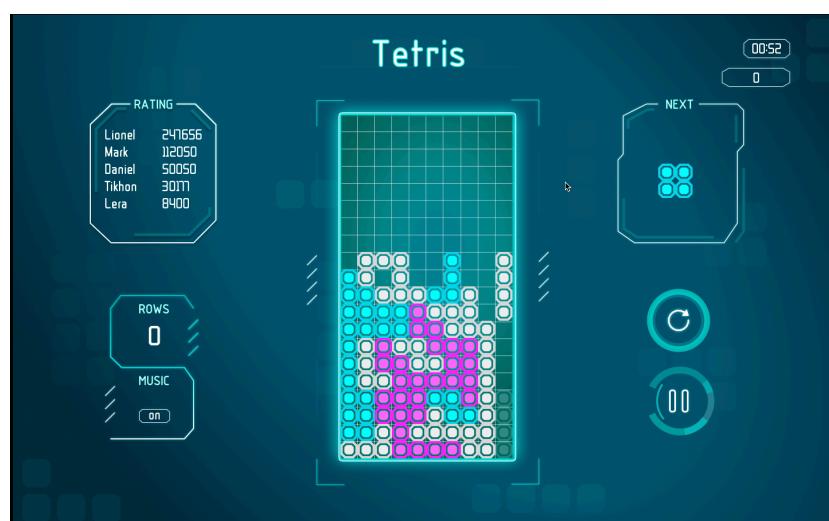


Рисунок 5.2 – Пример работы программы



Рисунок 5.3 – Удаление 4-х линий

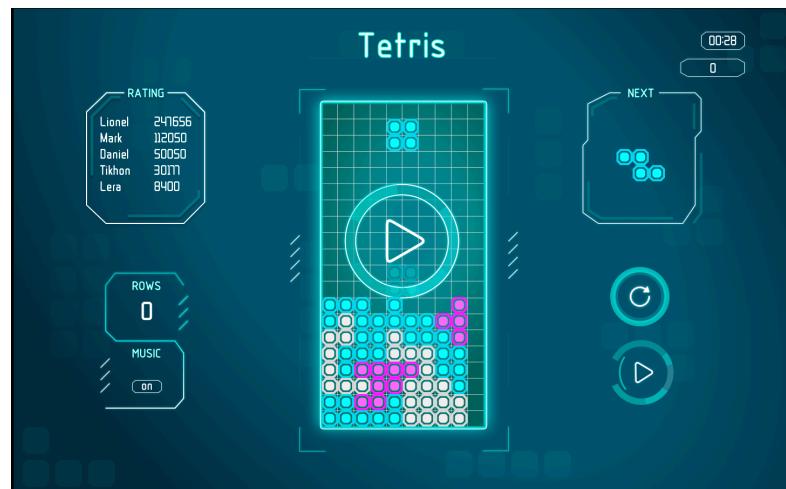


Рисунок 5.4 – Работа паузы в игре

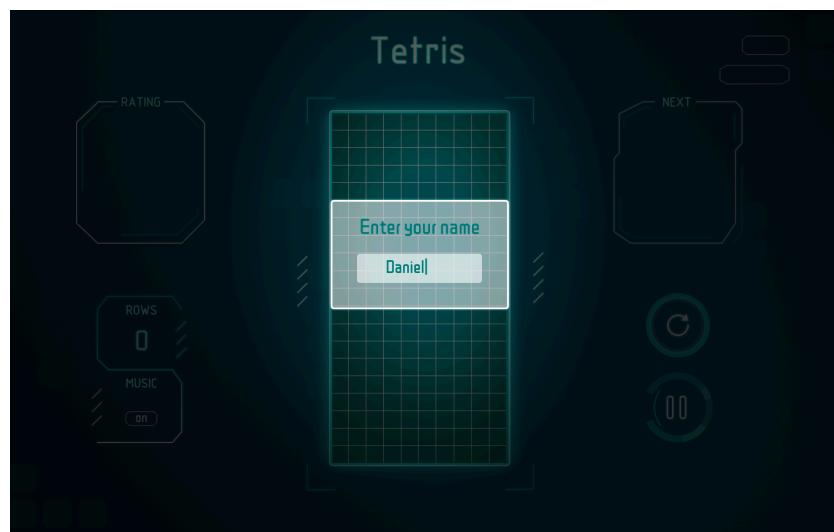


Рисунок 5.5 – Конец игры (поражение)

ЗАКЛЮЧЕНИЕ

В процессе выполнения курсового проекта были выполнены первоначально заданные цели, а именно реализована игра "Тетрис" с удобным интерфейсом. Были также реализованы различные алгоритмы, включая проверку и удаление заполненных линий, обработку ввода пользователя и генерацию новых фигур, продолжение прошлой игры.

Приложения для игр, 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.

Программа сохраняет игровой прогресс пользователя, что дает игроку возможность отслеживать свой прогресс и ставить перед собой новые цели.

Системные требования:

- Среда разработки – CLion 2023.2.2
- Минимальные системные требования:
- Занимаемая память процессора – 256 мб

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Луцик Ю. А. Объектно-ориентированное программирование на языке C++: учеб. пособие /Ю. А. Луцик, В. Н. Комличенко. – Минск: БГУИР, 2008.–266 с.
- [2] Дейтел, Х.М. Как программировать на С++ / Х.М. Дейтел, П.Д. Дейтел; пер. с англ. – М. : Бином, 2007. – 1152 с..
- [3] Страуструп, Б. Язык программирования С++ / Б. Страуструп; специальное издание. Пер. с англ. – СПб. : БНВ, 2008. – 1098 с.
- [4] Лафоре Р. Объектно-ориентированное программирование с С++/ 4-е издание М.:Питер, 2004. – 923 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)

Листинг программы

ПРИЛОЖЕНИЕ Д
(обязательное)

Ведомость документов