**Файл Figure.h:**

#ifndef TETRIS\_FIGURE\_H

#define TETRIS\_FIGURE\_H

#include "Picture.h"

#include "header.h"

// Функция для генерации случайного числа в определенном диапазоне

int generateRandomNumber(int a, int b);

// Класс Block, представляет собой отдельный блок тетриса

class Block {

public:

int x; // x-координата блока

int y; // y-координата блока

Block(int x, int y);// Конструктор с параметрами координат

~Block(); // Деструктор класса

};

// Базовый класс Figure, представляет одну фигурку в тетрисе

class Figure {

protected:

// Вектор блоков, из которых состоит фигурка

// Общая картина зависит от положения и формы этих блоков

std::vector <Block> status;

// Номер текущего положения фигурки (поворот)

int rotationStatus;

// Цвет фигурки

int color;

// Установленные размеры ячейки, смещения по оси x и y,

// высота блока и расстояние до столкновения

int cellSize;

int offsetX;

int offsetY;

int heightOfBlock;

int distanceToCollision;

// Образы фигурки и тени

Picture cubeImage;

Picture shadowCube;

// Определяет тип фигуры (для специфических поведений)

int type;

public:

Figure();

int getType() const;

int getColor() const;

void setDistanceToCollision(int x);

void drawFigure(sf::RenderWindow& window);

void move(int xPos, int yPos);

std::vector<Block> calculateMovedPosition ();

std::vector<Block>& getStatus();

int get\_offset\_x() const;

int get\_offset\_y() const;

virtual void rotateFigure (bool flag) = 0;

virtual ~Figure();

};

// Остальной код является определениями различных конкретных фигур,

// включая J\_Block, Z\_Block, T\_Block, S\_Block, L\_Block, I\_Block и O\_Block.

// Каждый из этих классов предоставляет свои варианты поворота и формы.

class J\_Block: public Figure {

private:

std::map<int, std::vector<Block>> allRotationOptions;

public:

J\_Block() {

type = 1;

allRotationOptions[0] = {Block(0, 0), Block(1, 0), Block(2, 0), Block(2, 1)};

allRotationOptions[1] = {Block(1, 0), Block(1, 1), Block(1, 2), Block(0, 2)};

allRotationOptions[2] = {Block(0, 0), Block(0, 1), Block(1, 1), Block(2, 1)};

allRotationOptions[3] = {Block(0, 0), Block(1, 0), Block(0, 1), Block(0, 2)};

status = allRotationOptions[0];

heightOfBlock = 3; }

void rotateFigure(bool flag) override {

if (flag) {

rotationStatus++;

if (rotationStatus > 3)

rotationStatus = 0;

if (rotationStatus == 0 || rotationStatus == 3)

heightOfBlock = 3;

else

heightOfBlock = 2; }

else {

rotationStatus--;

if(rotationStatus < 0)

rotationStatus = 3;

if (rotationStatus == 0 || rotationStatus == 3)

heightOfBlock = 3;

else

heightOfBlock = 2; }

status = allRotationOptions[rotationStatus]; }

~J\_Block() override {};

};

class Z\_Block: public Figure {

private:

std::map<int, std::vector<Block>> allRotationOptions;

public:

Z\_Block() {

type = 2;

allRotationOptions[0] = {Block(0, 0), Block(1, 0), Block(1, 1), Block(2, 1)};

allRotationOptions[1] = {Block(1, 0), Block(1, 1), Block(0, 1), Block(0, 2)};

heightOfBlock = 2;

status = allRotationOptions[0]; }

void rotateFigure(bool flag) override {

if (flag) {

rotationStatus++;

if (rotationStatus > 1)

rotationStatus = 0;

if(rotationStatus == 0)

heightOfBlock = 2;

else

heightOfBlock = 3; }

else {

rotationStatus--;

if(rotationStatus < 0)

rotationStatus = 1; }

status = allRotationOptions[rotationStatus]; }

~Z\_Block() override {};

};

class T\_Block: public Figure {

private:

std::map<int, std::vector<Block>> allRotationOptions;

public:

T\_Block() {

type = 3;

allRotationOptions[0] = {Block(0, 0), Block(1, 0), Block(2, 0), Block(1, 1)};

allRotationOptions[1] = {Block(1, 0), Block(1, 1), Block(0, 1), Block(1, 2)};

allRotationOptions[2] = {Block(1, 0), Block(0, 1), Block(1, 1), Block(2, 1)};

allRotationOptions[3] = {Block(0, 0), Block(0, 1), Block(0, 2), Block(1, 1)};

status = allRotationOptions[0];

heightOfBlock = 2; }

void rotateFigure(bool flag) override {

if (flag) {

rotationStatus++;

if (rotationStatus > 3)

rotationStatus = 0;

if (rotationStatus == 0 || rotationStatus == 3)

heightOfBlock = 2;

else

heightOfBlock = 3; }

else {

rotationStatus--;

if(rotationStatus < 0)

rotationStatus = 3;

if (rotationStatus == 0 || rotationStatus == 3)

heightOfBlock = 2;

else

heightOfBlock = 3; }

status = allRotationOptions[rotationStatus]; }

~T\_Block() override {};

};

class S\_Block: public Figure {

private:

std::map<int, std::vector<Block>> allRotationOptions;

public:

S\_Block() {

type = 4;

allRotationOptions[0] = {Block(1, 0), Block(2, 0), Block(0, 1), Block(1, 1)};

allRotationOptions[1] = {Block(0, 0), Block(0, 1), Block(1, 1), Block(1, 2)};

status = allRotationOptions[0];

heightOfBlock = 2; }

void rotateFigure(bool flag) override {

if (flag) {

rotationStatus++;

if (rotationStatus > 1)

rotationStatus = 0;

if(rotationStatus == 0)

heightOfBlock = 2;

else

heightOfBlock = 3; }

else {

rotationStatus--;

if(rotationStatus < 0)

rotationStatus = 1; }

status = allRotationOptions[rotationStatus]; }

~S\_Block() override {};

};

class L\_Block: public Figure {

private:

std::map<int, std::vector<Block>> allRotationOptions;

public:

L\_Block() {

type = 5;

allRotationOptions[0] = {Block(0, 0), Block(1, 0), Block(2, 0), Block(0, 1)};

allRotationOptions[1] = {Block(0, 0), Block(1, 0), Block(1, 1), Block(1, 2)};

allRotationOptions[2] = {Block(0, 1), Block(1, 1), Block(2, 1), Block(2, 0)};

allRotationOptions[3] = {Block(0, 0), Block(0, 1), Block(0, 2), Block(1, 2)};

status = allRotationOptions[0];

heightOfBlock = 3; }

void rotateFigure(bool flag) override {

if (flag) {

rotationStatus++;

if (rotationStatus > 3)

rotationStatus = 0;

if (rotationStatus == 0 || rotationStatus == 3)

heightOfBlock = 3;

else

heightOfBlock = 2; }

else {

rotationStatus--;

if(rotationStatus < 0)

rotationStatus = 3;

if (rotationStatus == 0 || rotationStatus == 3)

heightOfBlock = 3;

else

heightOfBlock = 2; }

status = allRotationOptions[rotationStatus]; }

~L\_Block() override {};

};

class I\_Block: public Figure {

private:

std::map<int, std::vector<Block>> allRotationOptions;

public:

I\_Block() {

type = 6;

allRotationOptions[0] = {Block(0, 0), Block(1, 0), Block(2, 0), Block(3, 0)};

allRotationOptions[1] = {Block(0, -3), Block(0, -2), Block(0, -1), Block(0, 0)};

status = allRotationOptions[0];

heightOfBlock = 4; }

void rotateFigure(bool flag) override {

if (flag) {

rotationStatus++;

if (rotationStatus > 1)

rotationStatus = 0;

if(rotationStatus == 0)

heightOfBlock = 4;

else

heightOfBlock = 1; }

else {

rotationStatus--;

if(rotationStatus < 0)

rotationStatus = 1; }

status = allRotationOptions[rotationStatus]; }

~I\_Block() override {};

};

class O\_Block: public Figure {

private:

std::map<int, std::vector<Block>> allRotationOptions;

public:

O\_Block() {

type = 7;

allRotationOptions[0] = {Block(0, 0), Block(0, 1), Block(1, 0), Block(1, 1)};

rotationStatus = 0;

status = allRotationOptions[0];

heightOfBlock = 2; }

void rotateFigure(bool flag) override { }

~O\_Block() override {};

};

#endif //TETRIS\_FIGURE\_H

**Файл Picture.h:**

#ifndef TETRIS\_PICTURE\_H

#define TETRIS\_PICTURE\_H

#include "SFML/Graphics.hpp"

#include "Text.h"

#include "PlayerInfo.h"

// Структура Picture используется для создания изображения, наследуется от Text

struct Picture: public Text {

sf::Image image; // Внутренняя переменная для хранения изображения

sf::Texture texture; // Текстура изображения

sf::Sprite sprite; // Спрайт, который выводится на экран

float x\_coordinate; // Координата x спрайта

float y\_coordinate; // Координата y спрайта

Picture() = delete; // Запрещаем конструктор по умолчанию

explicit Picture(std::string fileName, float x, float y); // Конструктор, принимающий имя файл и координаты

virtual void setPosition (float x, float y) override; // Метод установки позиции спрайта

virtual void draw(sf::RenderWindow& window) override; // Метод отрисовки спрайта в окне

float getPositionX () const; // Геттер для координаты x

float getPositionY () const; // Геттер для координаты y

void updateSprite(std::string fileName); // Метод обновления спрайта

};

#endif //TETRIS\_PICTURE\_H

**Файл Figure.cpp:**

#include "Figure.h"

// Конструктор класса Block

Block::Block(int x, int y) {

this->x = x; // Установка значения координаты x для блока

this->y = y; // Установка значения координаты y для блока }

// Функция для генерации случайного числа в заданном диапазоне

int generateRandomNumber(int a, int b) {

std::random\_device rd; // Инициализация генератора случайных чисел

std::mt19937 gen(rd()); // Использование Mersenne Twister алгоритма

std::uniform\_int\_distribution<int> distribution(a, b);

return distribution(gen); // Генерация случайного числа }

// Конструктор класса Figure

Figure::Figure(): shadowCube("./images/shadow\_cube.png",0,0), cubeImage("./images/color\_cubes.png", 0,0), type(0), cellSize(30), offsetX(4), offsetY(0), distanceToCollision(0) {

rotationStatus = 0; // Установка начального статуса вращения фигуры

color = generateRandomNumber(1, 3); // Генерация случайного цвета фигуры }

// Методы получения приватных полей класса

int Figure::getType() const {

return type; }

int Figure::getColor() const {

return color; }

// Установка дистанции до столкновения

void Figure::setDistanceToCollision(int x) {

distanceToCollision = x; }

// Получение текущего статуса фигуры

std::vector<Block>& Figure::getStatus() {

return status; }

// Получение смещения фигуры по осям x и y

int Figure::get\_offset\_x() const {

return offsetX; }

int Figure::get\_offset\_y() const {

return offsetY; }

// Функция отрисовки фигуры

void Figure::drawFigure(sf::RenderWindow& window) {

std::vector<Block> tmp = calculateMovedPosition( );

cubeImage.sprite.setTextureRect(sf::IntRect((color-1) \* cellSize,0,cellSize , cellSize) );

shadowCube.sprite.setTextureRect(sf::IntRect((color-1) \* cellSize,0,cellSize , cellSize) );

for (Block& item: tmp) {

cubeImage.sprite.setPosition(static\_cast<float>(item.x\*cellSize +576), static\_cast<float>(item.y\*cellSize + 175));

window.draw(cubeImage.sprite);

if (distanceToCollision >= heightOfBlock) {

shadowCube.sprite.setPosition(static\_cast<float>(item.x \* cellSize + 576),

static\_cast<float>(item.y \* cellSize + 175 + distanceToCollision \* CELL\_SIZE));

window.draw(shadowCube.sprite); } } }

// Функция перемещения фигуры

void Figure::move(int x, int y) {

this->offsetX += x;

this->offsetY += y; }

// Расчет переместившегося положения фигуры

std::vector<Block> Figure::calculateMovedPosition() {

std::vector<Block> tmp = status;

std::vector<Block> movedCondition;

for (auto & i : tmp) {

Block pos = Block(i.x + offsetX, i.y + offsetY);

movedCondition.push\_back(pos); }

return movedCondition; }

// Деструктор класса Figure

Figure::~Figure() {

status.clear(); }

// Деструктор класса Block

Block::~Block() { }

**Файл Button.cpp:**

#include "Button.h"

// Конструктор кнопки, получает параметры текста, размера, изображения и координат

Button::Button(std::string \_someText, float w, float h, std::string fileName, float x, float y) :

Picture(fileName, x , y), width(w), height(h), isPressed(false) {

// Загрузка звука для кнопки

buffer.loadFromFile("music/click.ogg");

sound.setBuffer(buffer);

// Позиционирование кнопки

sprite.setPosition(x,y); }

// Отрисовка кнопки

void Button::draw(sf::RenderWindow &window) {

sprite.setPosition(x\_coordinate, y\_coordinate);

window.draw(sprite); }

// Функции для получения размеров кнопки

float Button::getWidth() {

return width; }

float Button::getHeight() {

return height; }

// Функция для воспроизведения звука кнопки

void Button::playMusic() {

sound.play(); }

// Функции для работы с состоянием кнопки

bool Button::getIsPressed() {

return isPressed; }

void Button::setIsPressed(bool val) {

isPressed = val; }

// Деструктор кнопки

Button::~Button() { }

**Файл Text.cpp:**

#include "Text.h"

Text::Text(const std::string& \_someText, std::string& fontName, int size, int x, int y): x\_pos(x), y\_pos(y) {

// Загрузить файл шрифта

font.loadFromFile(fontName);

text.setFont(font); // Установить шрифт текста

text.setCharacterSize(size); // Установить размер текста

text.setFillColor(sf::Color::White); // Установить цвет текста

someText = std::move(\_someText); // Назначить текстовую строку

text.setString(someText); // Присвоить строку тексту для отображения }

// Конструктор класса Text с 2-мя аргументами размера и выбора

Text::Text(int size, bool sel, std::string fontName) {

font.loadFromFile(fontName); // Загрузить файл шрифта

text.setFont(font); // Установить шрифт текста

text.setCharacterSize(size); // Установить размер текста

text.setFillColor(sf::Color(20,122,122)); // Установить цвет текста

text.setPosition(670,420); // Установить позицию текста

isSelected = sel; // Установить выбор текста

if (sel) {

text.setString("|"); // Если выбрано, установить строку на "|" }

else

text.setString(""); // В противном случае установить пустую строку }

// Конструктор класса Text с одним аргументом имени шрифта

Text::Text(std::string fontName) {

font.loadFromFile(fontName); // Загрузить файл шрифта

text.setFont(font); // Установить шрифт текста

text.setCharacterSize(48); // Установить размер текста

text.setFillColor(sf::Color::White); // Установить белый цвет текста

x\_pos = 240; // Установить позицию X

y\_pos = 530; // Установить позицию Y }

// Обработка ввода пользователем

void Text::inputLogic(int charTyped) {

// Если символ введенный пользователем не равен ключам DELETE, ESCAPE, или ENTER

if (charTyped != DELETE\_KEY && charTyped != ESCAPE\_KEY && charTyped != ENTER\_KEY) {

txt << static\_cast<char>(charTyped); // Добавить символ в поток текста }

else if (charTyped == DELETE\_KEY) // Если пользователь нажал клавишу DELETE, {

if(txt.str().length() > 0) // и длина строки больше 0, {

deleteLastChar(); // удалить последний символ } }

text.setString(txt.str() + "|"); // Установить строку текста, добавив курсор (|) в конец }

// Удалить последний символ из текста

void Text::deleteLastChar() {

std::string t = txt.str(); // Получить текущий текст

std::string newT = ""; // Создать новую пустую строку

for (int i = 0; i < t.length() - 1; ++i) // Для каждого символа в текущем тексте, кроме последнего, {

newT += t[i]; // Добавить символ в новую строку }

txt.str(""); // Очистить текущий текст

txt << newT; // Установить новый текст как текущий

text.setString(txt.str()); // Установить новый текст в текст для отображения }

// Установить строку как число

void Text::setStrAsNumber(float num) {

// Преобразовать число в строку и установить как текущий текст

someText = std::to\_string(static\_cast<int>(num));

text.setString(someText); // Установить текст для отображения }

// Рисуем текст

void Text::draw(sf::RenderWindow& window) {

window.draw(text); // Нарисовать текст в окне }

// Отрисовка числа

void Text::drawNumber(sf::RenderWindow& window, int number) {

setStrAsNumber(number); // устанавливаем строку как число

// Меняем позицию текста, основываясь на количестве цифр в числе

if (number > 9 && number < 100) {

x\_pos = 230; }

else if (number > 99 && number < 1000) {

x\_pos = 220; }

text.setPosition(x\_pos, y\_pos);

window.draw(text); }

// Функции для установки текста, позиции и размера текста

void Text::setString(std::string str) {

text.setString(str); }

void Text::setPosition(float x, float y) {

text.setPosition(x, y); }

void Text::setCharacterSize(int size) {

text.setCharacterSize(size); }

// Установить ограничение на длину текста

void Text::setLimit(bool x) {

hasLimit = x; }

void Text::setLimit(bool x, int lim) {

hasLimit = x;

limit = lim; }

// Установить статус выбора текста

void Text::setSelected(bool sel) {

isSelected = sel;

if(!sel) {

std::string t = txt.str();

std::string newT = "";

for (int i = 0; i < t.length() - 1; ++i) {

newT += t[i]; }

text.setString(newT); } }

// Вернуть текущий введенный текст

std::string Text::getString() {

return txt.str(); }

// Обрабатывает события ввода после проверки на выбор и лимит символов

void Text::typeOn(sf::Event& event, sf::RenderWindow& window) {

if(isSelected) {

int charTyped = event.text.unicode;

if (charTyped < 128) // Если символ входит в диапазон ASCII {

if (hasLimit) // Если установлено ограничение, {

// и текущий текст короче или равен лимиту

if(txt.str().length() < limit) {

inputLogic(charTyped); // Обработать введенный символ }

else if(txt.str().length() >= limit && charTyped == DELETE\_KEY) // Если текущий текст равен или превышает лимит,

{ // и был введен символ удаления,

deleteLastChar(); // Удалить последний символ } }

else {

// Если ограничение не установлено

inputLogic(charTyped); // Обработать введенный символ } } } }

**Файл Text.h:**

#ifndef TETRIS\_TEXT\_H

#define TETRIS\_TEXT\_H

#include "header.h" // Включить другие необходимые заголовки

#include <sstream> // Включить заголовочный файл для работы со строковыми потоками

// Макросы для определения клавиш (по ASCII-коду)

#define DELETE\_KEY 8

#define ENTER\_KEY 13

#define ESCAPE\_KEY 27

// Класс Text для отображения и обработки текста

class Text {

protected:

std::string someText; // Стандартная строка для работы с текстом

sf::Font font; // Шрифт текста

sf::Text text; // Текст для отображения его на экране

std::ostringstream txt; // Строковый поток для работы с текстом

bool isSelected = true; // Проверка, выбран ли текст

bool hasLimit = true; // Ограничение на количество символов

float x\_pos; // X позиция текста

float y\_pos; // Y позиция текста

int limit = 10; // Лимит символов для текста

public:

// Контструкторы Text

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

Text (int size, bool sel, std::string fontName);

Text (std::string fontName);

// Методы для обработки текста и его отображения

void inputLogic(int charTyped); // Обрабатывает вводимые символы

void deleteLastChar(); // Удалить последний символ

void setStrAsNumber(float num); // Установить строку в виде числа

virtual void draw(sf::RenderWindow& window); // Отрисовывает текст в окне

void drawNumber(sf::RenderWindow& window, int number); // Рисует числа в окне

void setString(std::string str); // Установить строку

virtual void setPosition(float x, float y); // Установить позицию текста

void setCharacterSize(int size); // Установить размер символа

void setLimit(bool x); // Установить флаг имеет ли строка лимит

void setLimit(bool x, int lim); // Установить флаг и максимум символов

void setSelected(bool sel); // Выбрано ли текстовое поле

std::string getString(); // Получить строку

void typeOn(sf::Event& event, sf::RenderWindow& window); // Обрабатывает события ввода данных от пользователя

};

#endif // TETRIS\_TEXT\_H

**Файл header.h:**

#ifndef TETRIS\_HEADER\_H

#define TETRIS\_HEADER\_H

const int WIDTH = 10;

const int HEIGHT = 20;

const int CELL\_SIZE = 30;

#include <SFML/Graphics.hpp>

#include <iostream>

#include <random>

#include <fstream>

#include <map>

#include <thread>

int generateRandomNumber (int a, int b);

#endif //TETRIS\_HEADER\_H

**Файл Picture.cpp:**

#include "Picture.h"

// Конструктор структуры Picture

Picture::Picture(std::string fileName, float x, float y) : x\_coordinate(x), y\_coordinate(y), Text("./fonts/D.ttf") {

image.loadFromFile(fileName); // Загрузка изображения из файла

texture.loadFromImage(image); // Создание текстуры из изображения

sprite.setTexture(texture); // Установка текстуры для спрайта }

// Метод устанавливает позицию спрайта

void Picture::setPosition (float x, float y) {

x\_coordinate = x;

y\_coordinate = y;

sprite.setPosition(x, y); }

// Метод отрисовки спрайта в окне

void Picture::draw(sf::RenderWindow& window) {

setPosition(x\_coordinate, y\_coordinate); // Устанавливаем позицию спрайта

window.draw(sprite); // Отрисовываем спрайт }

// Геттеры для координат спрайта

float Picture::getPositionX () const {

return x\_coordinate; }

float Picture::getPositionY () const {

return y\_coordinate; }

// Метод обновляет спрайт, загрузив новое изображение из файла

void Picture::updateSprite(std::string fileName) {

image.loadFromFile(fileName); // Загрузка нового изображения

texture.loadFromImage(image); // Создание новой текстуры

sprite.setTexture(texture); // Установка новой текстуры для спрайта }

**Файл Exceptions.h:**

#ifndef TETRIS\_EXCEPTIONS\_H

#define TETRIS\_EXCEPTIONS\_H

#include <exception>

#include <string>

#include <utility>

// Класс Exceptions представляет базовые исключения в этом проекте.

// Наследуется от стандартного класса std::exception

class Exceptions: public std::exception {

protected:

std::string message; // сообщение исключения

public:

Exceptions() = default;

// what() возвращает сообщение об исключении

[[nodiscard]] const char\* what() const noexcept override{return message.c\_str();}

};

// Класс ExceptionFile представляет исключения, связанные с файлами.

class ExceptionFile: public Exceptions {

public:

explicit ExceptionFile(std::string mes) {

message = std::move(mes); }

ExceptionFile() = delete;

};

// Класс ExceptionSFML представляет исключения, связанные с библиотекой SFML.

class ExceptionSFML: public Exceptions {

public:

explicit ExceptionSFML(std::string mes) {

message = std::move(mes); }

ExceptionSFML() = delete;

};

// Класс OutOfBoundsException представляет исключения для случаев, когда

// происходит выход за пределы массива или списка.

class OutOfBoundsException: public Exceptions {

public:

explicit OutOfBoundsException(std::string mes) {

message = std::move(mes); }

OutOfBoundsException() = delete;

};

#endif //TETRIS\_EXCEPTIONS\_H

**Файл PlayerInfo.h:**

#ifndef TETRIS\_PLAYERINFO\_H

#define TETRIS\_PLAYERINFO\_H

#include <iostream>

// класс PlayerInfo содержит информацию об игроке

class PlayerInfo {

private:

std::string nickName; // никнейм игрока

int score; // счет игрока

public:

// Конструктор PlayerInfo принимает строку никнейма и целое число счета

PlayerInfo(const std::string& name = "", int \_score = 0);

// Переопределение оператора присваивания для объектов класса PlayerInfo

PlayerInfo& operator=(const PlayerInfo& obj);

// Геттер для имени пользователя

std::string getNickName() const;

// Геттер для очков пользователя

int getScore() const;

// сеттер для имени пользователя

void setNickName(const std::string& name);

// сеттер для очков пользователя

void setScore(int \_score);

};

#endif //TETRIS\_PLAYERINFO\_H

**Файл Board.cpp:**

#include "Board.h"

#include "Game.h"

// Конструктор по умолчанию.

// Задает ширину и высоту доски значениями по умолчанию (WIDTH, HEIGHT).

// Пытается загрузить изображение для игрового поля. Если неудачно, генерирует исключение.

// Загружает текстуру из изображения и задает её для спрайта.

// Инициализирует вектор игрового поля.

Board::Board(): width(WIDTH), height(HEIGHT) {

if(!gridImage.loadFromFile("images/board.png"))

throw ExceptionSFML("Ошибка загрузки картинки игрового поля");

gridTexture.loadFromImage(gridImage);

gridSprite.setTexture(gridTexture);

initializeVector(); }

// Возвращает спрайт игрового поля (как ссылку).

sf::Sprite& Board::getGridSprite() {

return gridSprite; }

// Возвращает значение соответствующей клетки игрового поля.

int Board::getGameBoard(int x, int y) const {

return gameBoard[x][y]; }

// Устанавливает значение для соответствующей клетки игрового поля.

void Board::setGameBoard(int x, int y, int value) {

gameBoard[x][y] = value; }

// Возвращает ширину игрового поля.

int Board::getWidth() const {

return width; }

// Возвращает высоту игрового поля.

int Board::getHeight() const {

return height; }

// Инициализирует вектор игрового поля заданными значениями.

// Все клетки в пределах заданной высоты и ширины устанавливаются в 0.

// Клетки в нижнем ряду устанавливаются в 1 (обозначение границы поля).

void Board::initializeVector() {

for (int i = 0; i < height; ++i) {

for (int j = 0; j < width; ++j) {

gameBoard[i][j] = 0; } }

for(int i = 0; i < width; ++i)

gameBoard[height][i] = 1; }

// Рисует игровое поле в окне.

void Board::drawGameBoard(sf::RenderWindow &window) {

window.draw(gridSprite); }

// Деструктор класса "Доска" (или "Игровое Поле").

Board::~Board() { }

**Файл List.h:**

#ifndef TETRIS\_LIST\_H

#define TETRIS\_LIST\_H

#include <initializer\_list>

#include <iostream>

#include <iterator>

#include <memory>

template <typename T, typename Allocator = std::allocator<T>>

class List {

private:

size\_t size\_;

struct FakeListNode {

FakeListNode\* prev;

FakeListNode\* next;

};

FakeListNode fake\_;

struct ListNode : public FakeListNode {

T element;

ListNode(const T& elem) : element(elem) {}

ListNode(T&& elem) : element(std::move(elem)) {}

};

using allocator\_traits = std::allocator\_traits<Allocator>;

using node\_allocator = typename allocator\_traits::template rebind\_alloc<ListNode>;

using node\_allocator\_traits = typename allocator\_traits::template rebind\_traits<ListNode>;

static void connect\_nodes(FakeListNode\* left, FakeListNode\* last) {

left->next = last, last->prev = left; }

node\_allocator allocator\_;

template <typename... Args>

void insert(FakeListNode\* left, FakeListNode\* last, Args&&... args) {

++size\_;

insert\_nodes(create\_node(std::forward<Args>(args)...), left, last); }

static void insert\_nodes(FakeListNode\* node, FakeListNode\* left, FakeListNode\* last) {

connect\_nodes(left, node);

connect\_nodes(node, last); }

void destroy\_node(FakeListNode\* ptr) {

if (ptr != nullptr) {

node\_allocator\_traits::destroy(allocator\_, static\_cast<ListNode\*>(ptr));

node\_allocator\_traits::deallocate(allocator\_, static\_cast<ListNode\*>(ptr),

1); } }

void erase\_node(FakeListNode\* node) {

if (empty()) {

return; }

connect\_nodes(node->prev, node->next);

--size\_;

destroy\_node(node); }

public:

List(const Allocator& alloc = Allocator()) : size\_(0), allocator\_(alloc) {

connect\_nodes(&fake\_, &fake\_); }

List(size\_t count, const T& element, const Allocator& alloc = Allocator()): List(alloc) {

size\_t ind = 0;

try {

for (; ind < count; ++ind) {

push\_back(element); }

} catch (...) {

for (size\_t j = 0; j < ind; ++j) {

pop\_front(); }

throw; } }

template <typename... Args>

FakeListNode\* create\_node(Args&&... args) {

FakeListNode\* ptr = node\_allocator\_traits::allocate(allocator\_, 1);

try {

node\_allocator\_traits::construct(allocator\_, static\_cast<ListNode\*>(ptr), std::forward<Args>(args)...); }

catch (...) {

node\_allocator\_traits::deallocate(allocator\_, static\_cast<ListNode\*>(ptr), 1);

throw; }

return ptr; }

template <bool IsConst>

class BasicIterator {

private:

FakeListNode\* ptr\_;

public:

using value\_type = T;

using type = std::conditional\_t<IsConst, const T, T>;

using iterator\_category = std::bidirectional\_iterator\_tag;

using difference\_type = std::ptrdiff\_t;

using pointer = type\*;

using reference = type&;

BasicIterator(FakeListNode\* node) : ptr\_(node){};

BasicIterator(const BasicIterator& other) : ptr\_(other.ptr\_) {}

BasicIterator& operator=(const BasicIterator& other) { ptr\_ = other.ptr\_; }

BasicIterator& operator++() {

ptr\_ = ptr\_->next;

return \*this; }

BasicIterator operator++(int) {

ptr\_ = ptr\_->next;

return BasicIterator(ptr\_->prev); }

BasicIterator& operator--() {

ptr\_ = ptr\_->prev;

return \*this; }

BasicIterator operator--(int) {

ptr\_ = ptr\_->prev;

return BasicIterator(ptr\_->next); }

reference operator\*() { return static\_cast<ListNode\*>(ptr\_)->element; }

pointer operator->() { return &(this->operator\*()); }

const type& operator\*() const {

return static\_cast<ListNode\*>(ptr\_)->element; }

const type\* operator->() const { return &(this->operator\*()); }

bool operator==(const BasicIterator& other) const {

return ptr\_ == other.ptr\_; }

bool operator!=(const BasicIterator& other) const {

return ptr\_ != other.ptr\_; }

};

using allocator\_type = Allocator;

using value\_type = T;

using iterator = BasicIterator<false>;

using const\_iterator = BasicIterator<true>;

using reverse\_iterator = std::reverse\_iterator<iterator>;

using const\_reverse\_iterator = std::reverse\_iterator<const\_iterator>;

List(const List& other) : List(allocator\_traits::select\_on\_container\_copy\_construction(other.allocator\_)) {

try {

for (auto it = other.begin(), end = other.end(); it != end; ++it) {

push\_back(T(\*it)); }

} catch (...) {

while (!empty()) {

pop\_back(); }

throw; } }

List& operator=(List&& other) {

List copy(std::move(other));

if (node\_allocator\_traits::propagate\_on\_container\_move\_assignment::value) {

allocator\_ = other.allocator\_; }

swap(\*this, copy);

return \*this; }

static void swap(List& left, List& rigth) {

if (node\_allocator\_traits::propagate\_on\_container\_swap::value) {

std::swap(left.allocator\_, rigth.allocator\_); }

FakeListNode\* tmpl = rigth.fake\_.prev;

FakeListNode\* tmpr = rigth.fake\_.next;

if (left.size\_ != 0) {

insert\_nodes(&rigth.fake\_, left.fake\_.prev, left.fake\_.next); }

else {

connect\_nodes(&rigth.fake\_, &rigth.fake\_); }

if (rigth.size\_ != 0) {

insert\_nodes(&left.fake\_, tmpl, tmpr); }

else {

connect\_nodes(&left.fake\_, &left.fake\_); }

std::swap(left.size\_, rigth.size\_); }

List(List&& other): List(allocator\_traits::select\_on\_container\_copy\_construction(other.allocator\_)) {

if (this == &other) {

return; }

swap(\*this, List());

swap(other, \*this); }

List& operator=(const List& other) {

List copy(other);

if (node\_allocator\_traits::propagate\_on\_container\_copy\_assignment::value) {

allocator\_ = other.allocator\_; }

swap(\*this, copy);

return \*this; }

iterator begin() const {

return iterator(const\_cast<FakeListNode\*>(fake\_.next)); }

iterator end() const { return iterator(const\_cast<FakeListNode\*>(&fake\_)); }

const\_iterator cbegin() const {

return const\_iterator(const\_cast<FakeListNode\*>(fake\_.next)); }

const\_iterator cend() const {

return const\_iterator(const\_cast<FakeListNode\*>(&fake\_)); }

void push\_back(const T& element) { insert(fake\_.prev, &fake\_, element); }

void push\_front(const T& element) { insert(&fake\_, fake\_.next, element); }

void push\_back(T&& element) { insert(fake\_.prev, &fake\_, element); }

void push\_front(T&& element) { insert(&fake\_, fake\_.next, element); }

void pop\_back() { erase\_node(fake\_.prev); }

void pop\_front() { erase\_node(fake\_.next); }

bool empty() const { return size\_ == 0; }

size\_t size() const { return size\_; }

Allocator get\_allocator() const { return allocator\_; }

reverse\_iterator rbegin() { return std::make\_reverse\_iterator(end()); }

reverse\_iterator rend() { return std::make\_reverse\_iterator(begin()); }

const\_reverse\_iterator rcbegin() const {

return std::make\_reverse\_iterator(cend()); }

const\_reverse\_iterator rcend() const {

return std::make\_reverse\_iterator(cbegin()); }

T& front() { return reinterpret\_cast<ListNode\*>(fake\_.next)->element; }

T& back() { return reinterpret\_cast<ListNode\*>(fake\_.prev)->element; }

const T& front() const {

return reinterpret\_cast<ListNode\*>(fake\_.next)->element; }

const T& back() const {

return reinterpret\_cast<ListNode\*>(fake\_.prev)->element; }

~List() {

while (!empty()) {

pop\_back(); } }

};

#endif //TETRIS\_LIST\_H

**Файл Game.h:**

#ifndef TETRIS\_GAME\_H

#define TETRIS\_GAME\_H

#include "Exceptions.h"

#include "header.h"

#include "Board.h"

#include "Figure.h"

#include "Button.h"

#include "Picture.h"

#include "List.h"

#include "GameMenu.h"

class Game // Определение класса Game {

// Защищенные (доступны только внутри данного класса и классов-наследников) атрибуты класса:

protected:

Board field; // Игровое поле

Figure\* currentFigure; // Текущая фигура

Figure\* nextFigure; // Следующая фигура

std::vector<Figure\*> figures; // Вектор фигур

Button buttonPause; // Кнопка паузы

Button buttonRestart; // Кнопка перезагрузки

Button buttonMusic; // Кнопка музыки

Picture buttonGameOver; // Кнопка окончания игры

Picture buttonRowsCount; // Кнопка подсчета счёта

Picture oneBlock; // Один блок

Picture pauseBoard; // Изображение доски при паузе

sf::Font font; // Шрифт

sf::Text text; // Текст

sf::Clock gameTime; // Часы игрового времени

sf::Music music; // Музыка

int lines\_in\_a\_row; // Количество линий в ряду

int score; // Счет

int time; // Время

int fileTime; // Временной файл

int tmpTime; // Временное время

int countLines; // Счётчик линий

std::string number; // Номер

std::string nickName; // Имя игрока

List<PlayerInfo> infoList; // Очередь с информацией об игроках

public:

public:

Game(); // Конструктор класса Game

// Основные функции класса:

int keyPressCheck(sf::Event& event, sf::RenderWindow& window, int& key, GameMenu& menu); // Функция проверки нажатия клавиши

int mousePressedCheck(sf::Event& event, sf::RenderWindow& window); // Функция проверки нажатия мыши

void buttonAction (int& key); // Функция действия кнопки

void fallingFigure (sf::Clock& timer, float pause); // Функция падения фигуры

Figure\* getRandomFigure(); // Функция получения случайной фигуры

void getAllFigures(); // Функция получения всех фигур

void drawBoardImage (sf::RenderWindow& window); // Функция отрисовки изображения доски

void draw(sf::RenderWindow& window); // Основная функция отрисовки

void drawNextFigureBlock(sf::RenderWindow& window); // Функция отрисовки блока следующей фигуры

bool boundariesIsBroken (); // Функция проверки нарушения границ

void isLocked(); // Функция проверки блокировки

int distanceToLocked (); // Функция расчёта расстояния до блокировки

void drawPlacedBlocks(sf::RenderWindow& window); // Функция отрисовки размещенных блоков

bool gameOver(sf::RenderWindow& window, sf::Event& event); // Функция окончания игры

void checkAndClearFilledLines (); // Функция проверки и очистки заполненных линий

void deleteLine (int num, int count); // Функция удаления линии

void readFileBestPlayers(const char\* fileName); // Функция чтения файла с лучшими игроками

void writeFileBestPlayers(const char\* fileName); // Функция записи в файл лучших игроков

void showBestPlayersBlock(sf::RenderWindow& window); // Функция отображения блока с лучшими игроками

void scoreBooster (int& \_lines\_in\_a\_row); // Функция увеличения счета

void showGameTime(sf::RenderWindow& window); // Функция отображения игрового времени

void showScore (sf::RenderWindow& window); // Функция отображения счета

void checkStatisticBeforeSave(); // Функция проверки статистики перед сохранением

bool processGameCycle (sf::RenderWindow& window, GameMenu& menu); // Функция обработки игрового цикла

void loadGameFromFile(std::string fileName); // Функция загрузки игры из файла

void saveGameToFile(std::string fileName); // Функция сохранения игры в файл

void getStartBoxOfFigure(); // Функция получения начальной позиции фигуры

~Game(); // Деструктор класса Game

};

#endif //TETRIS\_GAME\_H

**Файл Button.h:**

#ifndef TETRIS\_BUTTON\_H

#define TETRIS\_BUTTON\_H

#include "Picture.h"

#include "header.h"

#include <SFML/Audio.hpp>

// Класс Button наследуется от класса Picture.

// Это позволяет создавать кнопки с изображением и звуком.

class Button: public Picture {

private:

// Параметры кнопки

float width;

float height;

// Звуковые параметры кнопки

sf::SoundBuffer buffer;

sf::Sound sound;

// Состояние кнопки

bool isPressed;

public:

// Конструктор и деструктор

Button() = delete;

explicit Button(std::string \_someText, float w, float h, std::string fileName, float x, float y);

~Button();

// Функции для работы с кнопкой

void draw(sf::RenderWindow& window) override;

float getWidth();

float getHeight();

void playMusic();

bool getIsPressed();

void setIsPressed(bool val);

};

#endif //TETRIS\_BUTTON\_H

**Файл PlayerInfo.cpp:**

#include "PlayerInfo.h"

// Конструктор PlayerInfo принимает имя и счет игрока в качестве параметров

PlayerInfo::PlayerInfo(const std::string& name, int \_score)

: nickName(name), score(\_score) {}

// Оператор присваивания

PlayerInfo& PlayerInfo::operator=(const PlayerInfo& obj) {

// Проверка на самоприсваивание

if (&obj != this) {

this->score = obj.score;

this->nickName = obj.nickName; }

return \*this; }

// Геттеры

std::string PlayerInfo::getNickName() const {

return nickName; }

int PlayerInfo::getScore() const {

return score; }

// сеттеры

void PlayerInfo::setNickName(const std::string& name) {

nickName = name; }

void PlayerInfo::setScore(int \_score) {

score = \_score; }

**Файл Board.h:**

#ifndef TETRIS\_BOARD\_H

#define TETRIS\_BOARD\_H

#include "header.h"

// Игровое поле для Тетриса

class Board {

protected:

sf::Image gridImage; // Изображение для игрового поля

sf::Texture gridTexture; // Текстура, созданная из этого изображения

sf::Sprite gridSprite; // Спрайт, созданный из этой текстуры

int width; // Ширина игрового поля

int height; // Высота игрового поля

int gameBoard[HEIGHT + 1][WIDTH]; // Массив, представляющий игровое поле

public:

// Конструктор и деструктор

Board();

~Board();

// Методы для управления спрайтом игрового поля

sf::Sprite& getGridSprite();

void drawGameBoard (sf::RenderWindow& window);

// Методы для управления игровым полем

int getGameBoard (int x, int y) const;

void setGameBoard(int x, int y, int value);

void initializeVector ();

// Методы для получения размеров игрового поля

int getWidth() const;

int getHeight() const;

};

#endif //TETRIS\_BOARD\_H

**Файл GameMenu.cpp:**

#include "GameMenu.h"

#include "Game.h"

// Конструктор класса GameMenu. В нем инициализируются изображения для главного меню и кнопок, а также устанавливаются начальные значения переменных

GameMenu::GameMenu():

mainMenu("images/mainMenu.png", 0, 0),

buttonStart("Start", 286,127,"images/buttonStart.png", 579,233),

buttonResume("Resume", 286, 127,"images/buttonResume.png", 579,395),

buttonExit("Exit", 286, 127,"images/buttonExit.png", 579, 554),

selectedMenuOption(0), key(0), isMenu(true) { }

// Деструктор класса GameMenu

GameMenu::~GameMenu() { }

// Функция проверяет, была ли нажата кнопка на клавиатуре

void GameMenu::keyPressCheck(sf::Event& event) {

if (event.type == sf::Event::KeyPressed) {

if (event.key.code == sf::Keyboard::W || event.key.code == sf::Keyboard::Up) {

key = 1; // Если нажата клавиша "вверх", устанавливается key = 1 }

if (event.key.code == sf::Keyboard::S || event.key.code == sf::Keyboard::Down) {

key = 2; // Если нажата клавиша "вниз", ключ будет равен 2 }

if (event.key.code == sf::Keyboard::Enter) {

key = 3; // Если нажат enter, ключ устанавливается равным 3 } } }

// Функция отвечает за отображение главного меню

void GameMenu::showMenu(sf::RenderWindow& window, Game& game) {

// Отображение картинок для каждой кнопки и главного меню

window.draw(mainMenu.sprite);

window.draw(buttonStart.sprite);

window.draw(buttonResume.sprite);

window.draw(buttonExit.sprite);

// Выполняет действие, связанное с нажатием кнопки

buttonAction(game);

// В зависимости от текущего выбранного пункта меню обновляем спрайты кнопок

if (selectedMenuOption == 0) {

buttonStart.updateSprite("images/selectedStart.png");

buttonResume.updateSprite("images/buttonResume.png");

buttonExit.updateSprite("images/buttonExit.png"); }

else if (selectedMenuOption == 1) {

buttonStart.updateSprite("images/buttonStart.png");

buttonResume.updateSprite("images/selectedResume.png");

buttonExit.updateSprite("images/buttonExit.png"); }

else {

buttonStart.updateSprite("images/buttonStart.png");

buttonResume.updateSprite("images/buttonResume.png");

buttonExit.updateSprite("images/selectedExit.png"); } }

// Функция определяет какое действие нужно выполнить при нажатии кнопки

void GameMenu::buttonAction(Game& game) {

if (key == 1) {

selectedMenuOption--; // Перемещаемся вверх по меню

if (selectedMenuOption < 0)

selectedMenuOption = 2;

buttonResume.playMusic(); // Воспроизводим музыку }

if (key == 2) {

selectedMenuOption++; // Перемещаемся вниз по меню

if (selectedMenuOption > 2)

selectedMenuOption = 0;

buttonResume.playMusic(); // Воспроизводим музыку }

if (key == 3) // Если нажат enter {

buttonResume.playMusic(); // Воспроизводим музыку

if (selectedMenuOption == 0)

isMenu = false; // Выходим из меню, если выбран пункт "Start"

else if (selectedMenuOption == 1) {

game.loadGameFromFile("Game"); // Загружаем игру из файла, если выбран пункт "Resume"

isMenu = false; }

else {

exit(EXIT\_SUCCESS); // Выходим из игры, если выбран пункт "Exit" } }

key = 0; // Сбрасываем ключ }

// Геттер для флага isMenu

bool GameMenu::getIsMenu() const {

return isMenu; }

// Сеттер для флага isMenu

void GameMenu::setIsMenu(bool x) {

isMenu = x; }

**Файл GameMenu.h:**

#ifndef TETRIS\_GAMEMENU\_H

#define TETRIS\_GAMEMENU\_H

#include "Picture.h"

#include "header.h"

#include "Button.h"

class Game;

// Класс GameMenu объявляет главное меню игры

class GameMenu {

private:

// Здесь хранится выбранный пользователем пункт меню

int selectedMenuOption;

// Указатель на кнопку в меню

int key;

// Объекты класса Button для каждой кнопки на главном экране

Button buttonStart;

Button buttonResume;

Button buttonExit;

// Объект Picture для отображения главного меню

Picture mainMenu;

// Флаг, указывающий, активно ли меню

bool isMenu;

public:

// Конструктор

GameMenu();

// Отображает меню в окне

void showMenu(sf::RenderWindow& window, Game& game);

// Проверяем, нажата ли клавиша

void keyPressCheck(sf::Event& event);

// Действие, которое нужно выполнить после нажатия кнопки

void buttonAction(Game& game);

// Геттер для поля isMenu

bool getIsMenu() const;

// Сеттер для поля isMenu

void setIsMenu(bool x);

// Деструктор

~GameMenu();

};

#endif //TETRIS\_GAMEMENU\_H

**Файл main.cpp:**

#include "Game.h"

#include <filesystem>

[[noreturn]] void gameRunning(GameMenu& menu) {

while (true) {

Game game;

sf::RenderWindow window(sf::VideoMode::getFullscreenModes()[0], "Tetris", sf::Style::Fullscreen);

game.processGameCycle(window, menu); } }

int main() {

std::filesystem::path exePath = std::filesystem::absolute(std::filesystem::path("/Users/fozboom/CLionProjects/GameTetris/cmake-build-debug/"));

std::filesystem::current\_path(exePath.parent\_path());

GameMenu menu;

gameRunning(menu); }

**Файл Game.cpp:**

#include "Game.h"

#include "Exceptions.h"

Game::Game(): // Конструктор по умолчанию класса Game инициализирует следующие члены

buttonRowsCount("./images/rows.png",160, 485), // Инициализация кнопки, отображающей количество строк

buttonRestart("RESTART",111,112, "./images/restart.png",1110,478), // Инициализация кнопки перезапуска игры

buttonPause("PAUSE", 120, 120, "./images/pause.png", 1108, 615), // Инициализация кнопки паузы

buttonGameOver("./images/gameOver.png", 0, 0), // Инициализация кнопки Game Over

buttonMusic("Music",62,34, "./images/buttonON.png", 220,684), // Инициализация кнопки для управления музыкой

oneBlock("./images/color\_cubes.png", 0, 0), // Инициализация блока для игры

pauseBoard("./images/shadowBoard.png",0,0), // Инициализация доски для паузы

lines\_in\_a\_row(0), score(0), time(10), countLines(0), fileTime(0), tmpTime(0) // Инициализация переменных игры {

music.openFromFile("./music/music.ogg");

music.setVolume(30);

music.play();

getStartBoxOfFigure();

readFileBestPlayers("./BestPlayersInfo.txt");

currentFigure = getRandomFigure();

nextFigure = getRandomFigure();

currentFigure->setDistanceToCollision(distanceToLocked());

font.loadFromFile("./fonts/D.ttf");

text.setFont(font);

text.setCharacterSize(24);

text.setFillColor(sf::Color::White);

// В теле конструктора мы загружаем и проигрываем музыку, получаем начальную фигуру и блок,

// загружаем шрифт и устанавливаем его для текста, а также загружаем информацию о лучших игроках. }

void Game::draw(sf::RenderWindow& window) // В этом методе мы отрисовываем все элементы игры {

drawPlacedBlocks(window); // Отрисовка блоков на игровом поле

buttonRowsCount.draw(window); // Отрисовка кнопки количества строк

buttonPause.draw(window); // Отрисовка кнопки "пауза"

buttonRestart.draw(window); // Отрисовка кнопки "перезапуск"

buttonRowsCount.drawNumber(window, countLines); // Отображение числа строк

buttonMusic.draw(window); // Отрисовка кнопки "музыка"

currentFigure->setDistanceToCollision(distanceToLocked()); // Установка расстояния текущей фигуры до коллизии

currentFigure->drawFigure(window); // Отрисовка текущей фигуры

drawNextFigureBlock(window); // Отрисовка следующей фигуры

getAllFigures(); // Получение всех фигур

showGameTime(window); // Отображение времени игры

showScore(window); // Отображение счета

showBestPlayersBlock(window); // Отображение блока лучших игроков }

// Функция получения случайной фигуры для игры

Figure\* Game::getRandomFigure() {

if (figures.empty()) {

figures.push\_back(new J\_Block());

figures.push\_back(new Z\_Block());

figures.push\_back(new T\_Block());

figures.push\_back(new S\_Block());

figures.push\_back(new L\_Block());

figures.push\_back(new I\_Block());

figures.push\_back(new O\_Block()); }

// Генерируем случайный индекс из диапазона от 0 до размера вектора фигур

int randomIndex = generateRandomNumber(0, static\_cast<int>(figures.size()) - 1);

Figure\* fig = figures[randomIndex]; // Выбираем случайную фигуру

// Удаляем выбранную фигуру из вектора

figures.erase(figures.begin() + randomIndex);

return fig; // Возвращаем фигуру }

void Game::getStartBoxOfFigure() {

figures.push\_back(new J\_Block());

figures.push\_back(new T\_Block());

figures.push\_back(new L\_Block());

figures.push\_back(new I\_Block());

figures.push\_back(new O\_Block()); }

void Game::getAllFigures() {

if (figures.empty()) {

figures.push\_back(new J\_Block());

figures.push\_back(new Z\_Block());

figures.push\_back(new T\_Block());

figures.push\_back(new S\_Block());

figures.push\_back(new L\_Block());

figures.push\_back(new I\_Block());

figures.push\_back(new O\_Block()); } }

// Функция для проверки, нарушает ли текущая фигура границы игрового поля.

bool Game::boundariesIsBroken() {

std::vector<Block> object = currentFigure->calculateMovedPosition( );

for (auto & i : object) {

if (i.x< 0 || i.x > field.getWidth() - 1 || field.getGameBoard(i.y, i.x) != 0 || i.y > field.getHeight()- 1)

return true; }

return false; }

void Game::isLocked() // Функция для проверки, зафиксирована ли фигура на игровом поле. {

std::vector<Block> object = currentFigure->calculateMovedPosition(); // Расчет положения блока

for (Block& item: object) {

field.setGameBoard(item.y, item.x, currentFigure->getColor()); // Установка цвета блока на игровое поле }

currentFigure = nextFigure; // Текущей фигурой становится следующая

nextFigure = getRandomFigure(); // Генерируем новую случайную фигуру. }

void Game::drawBoardImage (sf::RenderWindow& window) // Функция для отрисовки изображения доски {

field.drawGameBoard(window); // Отрисовка игрового поля }

void Game::drawPlacedBlocks(sf::RenderWindow& window) // Функция для отрисовки размещенных блоков {

std::vector<Block> tmp = currentFigure->calculateMovedPosition();

for (int i = 0; i < field.getHeight(); ++i) {

for (int j = 0; j < field.getWidth(); ++j) {

if (field.getGameBoard(i,j) != 0) {

oneBlock.sprite.setTextureRect(sf::IntRect((field.getGameBoard(i, j) - 1) \* CELL\_SIZE, 0, CELL\_SIZE , CELL\_SIZE));

oneBlock.sprite.setPosition(static\_cast<float>(j\*CELL\_SIZE+576), static\_cast<float>(i\*CELL\_SIZE+175));

window.draw(oneBlock.sprite); // Отрисовка блока в окне } } } }

void Game::buttonAction(int& key) {

if (key == 1) //влево {

currentFigure->move(-1, 0);

if(boundariesIsBroken())

currentFigure->move(1, 0); }

if (key == 2) //вправо {

currentFigure->move(1, 0);

if(boundariesIsBroken())

currentFigure->move(-1, 0); }

if (key == 3) //вверх {

currentFigure->rotateFigure(false);

if(boundariesIsBroken())

currentFigure->rotateFigure(true); }

if (key == 4) //вниз {

currentFigure->move(0, 1);

if(boundariesIsBroken())

currentFigure->move(0, -1); }

if (key == 5) //пробел {

currentFigure->move(0, distanceToLocked()); }

key = 0; }

void Game::fallingFigure(sf::Clock& timer, float pause) // Метод для обработки падения фигур {

if (timer.getElapsedTime().asSeconds() >= pause) // Если прошедшее время больше или равно паузе {

timer.restart(); // Перезапускаем таймер

currentFigure->move(0,1); // Смещаем текущую фигуру вниз

if(boundariesIsBroken()) // Если границы нарушены {

currentFigure->move(0, -1); // Возвращаем фигуру обратно

isLocked(); // Зафиксируем фигуру на игровом поле

checkAndClearFilledLines(); // Проверяем и очищаем заполненные линии } } }

bool Game::gameOver(sf::RenderWindow& window, sf::Event& event) {

// Создаем вектор блоков, используя текущую фигуру и вычисляя ее смещенное положение

std::vector<Block> object = currentFigure->calculateMovedPosition( );

// Для каждого блока в векторе

for (Block& item: object) {

// Проверяем, нарушены ли границы и находится ли блок на верхней границе поля

if (boundariesIsBroken() && (item.y == 0)) {

// Создаем текстовое поле для ввода имени пользователя

Text nick(30, true, "./fonts/D.ttf");

// Очистка окна

window.clear();

// Отрисовка кнопки завершения игры

buttonGameOver.draw(window);

// Отрисовка текстового поля

nick.draw(window);

// Обновление окна

window.display();

// Ожидание события в окне

while(window.waitEvent(event)) {

// Если событие - это ввод текста

if (event.type == sf::Event::TextEntered) {

// Вводим текст в текстовое поле

nick.typeOn(event, window); }

// Если событие - это нажатие клавиши

if (event.type == sf::Event::KeyPressed) {

// Если нажата клавиша Enter

if (event.key.code == sf::Keyboard::Enter) {

// Запоминаем введенное имя пользователя

nickName = nick.getString();

break; } }

// Очистка окна

window.clear();

// Отрисовка кнопки завершения игры

buttonGameOver.draw(window);

// Отрисовка текстового поля

nick.draw(window);

// Обновление окна

window.display(); }

// Инициализация вектора поля

field.initializeVector();

// Сохранение текущего состояния игры в файл

saveGameToFile("Game");

// Возвращаем, что игра окончена

return true; } }

// Возвращаем, что игра не закончилась

return false; }

// Функция проверки и удаления заполненных линий

void Game::checkAndClearFilledLines() {

// Флаг, чтобы проверить, полностью ли заполнена строка

bool isFull = true;

// Переменная, чтобы отслеживать количество заполненных строк подряд

lines\_in\_a\_row = 0;

// Обходим каждую строку снизу вверх

for (int i = field.getHeight() - 1; i >= 0; --i) {

// Обходим каждый элемент в строке слева направо

for (int j = 0; j < field.getWidth(); ++j) {

// Если элемент свободен (равен 0), значит строка не полностью заполнена

if (field.getGameBoard(i, j) == 0) {

isFull = false;

break; } }

// Если строка полностью заполнена увеличиваем счетчик заполненных строк подряд

if (isFull) {

lines\_in\_a\_row++; }

// Если встретилась незаполненная строка, и до этого были заполненные строки

if (!isFull && lines\_in\_a\_row > 0) {

// Увеличиваем общий счетчик удаленных строк на количество заполненных строк подряд

countLines += lines\_in\_a\_row;

// Удаляем заполненные строки начиная с текущей позиции

deleteLine(i, lines\_in\_a\_row);

// Сдвигаем индекс строки вниз на количество удаленных строк (так как они были удалены)

i += lines\_in\_a\_row;

// Увеличиваем очки пользователя в зависимости от количества одновременно удаленных строк

scoreBooster(lines\_in\_a\_row); }

// Сбрасываем флаг к следующей проверке строки

isFull = true; } }

// Функция для удаления определенной строки

void Game::deleteLine(int num, int count) {

// Повторяем столько раз, сколько строк нужно удалить

for (int k = count; k > 0; --k) {

// Для каждой строки от указанной до верхней границы поля

for (int i = num + count; i > 0; --i) {

// Для каждого блока в строке

for (int j = 0; j < field.getWidth(); ++j) {

// Устанавливаем значение блока в текущей строке равным значению блока в строке выше

// Таким образом, "удаляя" строку, мы сдвигаем все строки выше на одну вниз

field.setGameBoard(i, j, field.getGameBoard(i - 1, j)); } } } }

// Функция для получения расстояния до заблокированных ячеек

int Game::distanceToLocked() {

// Изначально задаем максимально большое расстояние

int minDistance = 20;

// Обходим каждый блок в текущей фигуре

for (Block& item: currentFigure->getStatus()) {

// Счетчик для измерения расстояния

int i = 0;

// Пока не достигнем заблокированной ячейки или края игрового поля

while ((field.getGameBoard(item.y + currentFigure->get\_offset\_y() + i, item.x + currentFigure->get\_offset\_x()) == 0) && i < field.getHeight() - 1) {

// Увеличиваем счетчик

++i; }

// Если текущее расстояние меньше минимального, то обновляем минимальное расстояние

if ((i-1) < minDistance)

minDistance = i-1; }

// Возвращаем минимальное расстояние до заблокированной ячейки

return minDistance; }

// Функция для отрисовки следующей фигуры на экране

void Game::drawNextFigureBlock(sf::RenderWindow &window) {

int offsetX, offsetY; // переменные для коррекции позиции фигуры на экране

// Устанавливаем текстуру и размеры для блока следующей фигуры

oneBlock.sprite.setTextureRect(sf::IntRect((nextFigure->getColor()-1) \* CELL\_SIZE,0,CELL\_SIZE , CELL\_SIZE));

// Проходим по всем блокам следующей фигуры

for (Block& item: nextFigure->getStatus()) {

offsetX = 0, offsetY = 0; // обнуляем смещение для каждого нового блока

// Делаем корректировку смещения в зависимости от типа следующей фигуры

if (nextFigure->getType() == 6)

offsetX = -21; // для фигуры типа 6

else if (nextFigure->getType() == 7)

offsetX = 8; // для фигуры типа 7

// Устанавливаем позицию блока на экране

oneBlock.sprite.setPosition(static\_cast<float>(item.x\*CELL\_SIZE +1122 + offsetX), static\_cast<float>(item.y\*CELL\_SIZE + 261 + offsetY));

// Рисуем блок на экране

window.draw(oneBlock.sprite); } }

// Функция для чтения файла с информацией о лучших игроках

void Game::readFileBestPlayers(const char\* fileName) {

// Открываем файл на чтение

std::ifstream read(fileName);

// Если файл не удалось открыть, бросаем исключение

if (!read.is\_open()) { throw ExceptionFile("Ошибка открытия файла для чтения"); }

std::string tempName; // Временная переменная для хранения имени игрока

int tempScore; // Временная переменная для хранения очков игрока

// Читаем из файла пока есть что читать

while (read >> tempName >> tempScore) {

// Создаем временный объект с информацией об игроке

PlayerInfo tempPlayerInfo(tempName, tempScore);

// Помещаем информацию об игроке в список

infoList.push\_back(tempPlayerInfo); }

// Если конец файла не достигнут, бросаем исключение

if (!read.eof()) {

throw ExceptionFile("Ошибка чтения данных из файла"); } }

// Функция для записи файла с информацией о лучших игроках

void Game::writeFileBestPlayers(const char\* fileName) {

// Проверяем статистику перед сохранением

checkStatisticBeforeSave();

std::ofstream input;

// Открываем файл на запись

input.open(fileName);

// Если файл не удалось открыть, бросаем исключение

if (!input.is\_open()) {throw ExceptionFile("Ошибка открытия файла для записи");}

// Пока список не пуст

while (!infoList.empty()){

// Извлекаем информацию об игроке из списка

PlayerInfo tempPlayerInfo = infoList.front();

// Записываем имя игрока и очки в файл

input << tempPlayerInfo.getNickName() << " " << tempPlayerInfo.getScore() << "\n";

// Удаляем информацию об игроке из списка

infoList.pop\_front(); } }

// Функция для отображения блока с лучшими игроках

void Game::showBestPlayersBlock(sf::RenderWindow& window) {

int offset\_x = 90, offset\_y = 0; // Задаем начальные отступы

List<PlayerInfo> tempList = infoList; // Создаем временный список, копируя имеющийся

// Отображаем информацию о первых пяти игроках из списка

for (int i = 0; i < 5 && !tempList.empty(); ++i) {

// Извлекаем информацию об игроке из списка

PlayerInfo tempPlayerInfo = tempList.front();

// Задаем текстовую строку и позицию для имени игрока

text.setString(tempPlayerInfo.getNickName());

text.setPosition(static\_cast<float>(165), static\_cast<float>(195 + offset\_y));

// Отрисовываем текст

window.draw(text);

// Приводим очки игрока к строковому виду и задаем текстовую строку и позицию для очков

number = std::to\_string(tempPlayerInfo.getScore());

text.setString(number);

text.setPosition(static\_cast<float>(175 + offset\_x), static\_cast<float>(195 + offset\_y));

// Отрисовываем текст

window.draw(text);

// Удаляем информацию об игроке из списка

tempList.pop\_front();

// Увеличиваем вертикальный отступ для следующего игрока

offset\_y += 30; }

// После завершения цикла, временный список автоматически очистится }

// Функция для увеличения очков игрока в зависимости от количества очищенных строк подряд

void Game::scoreBooster(int& \_lines\_in\_a\_row) {

// Если очищена одна строка

if (\_lines\_in\_a\_row == 1) {

// Увеличиваем счет на 40 + 5% от времени, умноженного на 40

score += 40 + static\_cast<int>( 0.05 \* time \* 40);

// Выводим информацию

std::cout << "+80\n"; }

// Если очищено две строки

else if(\_lines\_in\_a\_row == 2) {

// Увеличиваем счет на 100 + 5% от времени, умноженного на 100

score += 100 + static\_cast<int>( 0.05 \* time \* 100);

// Выводим информацию

std::cout << "+200\n"; }

// Если очищено три строки

else if(\_lines\_in\_a\_row == 3) {

// Увеличиваем счет на 300 + 5% от времени, умноженного на 300

score += 300 + static\_cast<int>( 0.05 \* time \* 300);

// Выводим информацию

std::cout << "+600\n"; }

// Если очищено четыре строки

else if(\_lines\_in\_a\_row == 4) {

// Увеличиваем счет на 1200 + 5% от времени, умноженного на 1200

score += 1200 + static\_cast<int>( 0.05 \* time \* 1200);

// Выводим информацию

std::cout << "+2400\n"; }

// Обнуляем счетчик очищенных строк подряд

lines\_in\_a\_row = 0; }

// Функция для отображения количество очков игрока на экране

void Game::showScore(sf::RenderWindow& window) {

float x = 1295, y = 96; // Задаем координаты для отображения

number = std::to\_string(score); // Преобразуем счет в строку

text.setString(number); // Устанавливаем эту строку на текстовое поле

// В зависимости от счета, корректируем координату X для правильного центрирования

if (score < 10)

x -= 0; // Если счет меньше 10, то не корректируем

else if (score < 100)

x -= 5;

else if (score < 1000)

x -= 10;

else if (score < 10'000)

x -= 15;

else if (score < 100'000)

x -= 20;

else if (score < 1'000'000)

x -= 25;

else if (score < 10'000'000)

x -= 30;

// Устанавливаем позицию текстового поля

text.setPosition(x, y);

// Обрисовываем текстовое поле в окне

window.draw(text); }

void Game::showGameTime(sf::RenderWindow &window) {

time = fileTime + tmpTime + gameTime.getElapsedTime().asSeconds();

if(time < 10) {

number = std::to\_string(time);

text.setString("00:0" + number); }

else if (time < 60) {

number = std::to\_string(time);

text.setString("00:" + number); }

else if (time < 600) {

std::string seconds;

number = std::to\_string(time/60);

seconds = std::to\_string(time%60);

if (time%60 < 10)

text.setString("0"+number + ":0" + seconds);

else

text.setString("0"+number + ":" + seconds); }

else {

std::string seconds;

number = std::to\_string(time/60);

seconds = std::to\_string(time%60);

if (time%60 < 10)

text.setString(number + ":0" + seconds);

else

text.setString(number + ":" + seconds); }

text.setPosition(1293,45);

window.draw(text); }

void Game::checkStatisticBeforeSave() {

// Создаем временный список для игроков

List<PlayerInfo> tempList;

// Создаем новую информацию о счете

PlayerInfo newScore{nickName, score};

// Пока список не пуст

while (!infoList.empty()) {

// Переменная текущего элемента из списка игроков

PlayerInfo current = infoList.front();

// Удаляем текущий элемент из списка

infoList.pop\_front();

// Если счет текущего пользователя меньше, чем у нового пользователя, добавляем нового пользователя в список

if (current.getScore() < newScore.getScore()){

tempList.push\_back(newScore);

newScore = current; // Теперь нашей задачей будет вставка текущего пользователя в правильное место списка

} else {

// иначе просто добавляем текущего пользователя в список

tempList.push\_back(current); } }

// Добавляем последнего пользователя в список.

tempList.push\_back(newScore);

// Позволяем всем элементам изменить позицию в списке

while (!tempList.empty()) {

// Добавляем в начало списка информации первый элемент временного списка

infoList.push\_front(tempList.back());

// Удаляем последний элемент временной информации

tempList.pop\_back(); } }

int Game::keyPressCheck(sf::Event& event, sf::RenderWindow& window, int & key, GameMenu& menu) {

if (event.type == sf::Event::KeyPressed) {

if (event.key.code == sf::Keyboard::Left || event.key.code == sf::Keyboard::A) {

key = 1;

return 3; }

if (event.key.code == sf::Keyboard::Right || event.key.code == sf::Keyboard::D) {

key = 2;

return 3; }

if (event.key.code == sf::Keyboard::Up || event.key.code == sf::Keyboard::W) {

key = 3;

return 3; }

if (event.key.code == sf::Keyboard::Down || event.key.code == sf::Keyboard::S) {

key = 4;

return 3; }

if (event.key.code == sf::Keyboard::Space) {

key = 5;

return 3; }

if (event.key.code == sf::Keyboard::RAlt) {

saveGameToFile("Game");

menu.setIsMenu(true);

return 1; }

if (event.key.code == sf::Keyboard::LControl) {

return 1; }

if (event.key.code == sf::Keyboard::Escape) {

int tmp= time - fileTime;

buttonPause.updateSprite("./images/unpause.png");

buttonPause.playMusic();

while (window.waitEvent(event)) {

window.clear();

window.draw(pauseBoard.sprite);

draw(window);

window.display();

if((event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Escape)||

(sf::IntRect(1104, 619, 120, 120).contains(sf::Mouse::getPosition(window))

&& sf::Mouse::isButtonPressed(sf::Mouse::Left))) {

buttonPause.updateSprite("./images/pause.png");

break; } }

gameTime.restart();

tmpTime = tmp; } }

return mousePressedCheck(event, window); }

// Метод обработки цикла игры

bool Game::processGameCycle(sf::RenderWindow &window, GameMenu& menu) {

// Создаем таймер

sf::Clock timer;

// Устанавливаем начальное время паузы

float pause = 0.27f;

// Инициализируем клавишу

int key = 0;

// Инициализируем переменную для операции, которую нужно выполнить

int toDo;

// Пока окно открыто

while (window.isOpen()) {

// Инициализируем событие

sf::Event event{};

// Пока есть события в очереди

while (window.pollEvent(event)) {

// Если мы в меню

if(menu.getIsMenu())

// Проверяем нажатие клавиш

menu.keyPressCheck(event);

// Иначе

else {

// Проверяем нажатие клавиш и возвращаем действие, которое надо выполнить

toDo = keyPressCheck(event, window, key, menu);

// Если нам надо выйти

if (toDo == 1) return true; } }

// Очищаем окно

window.clear();

// Если мы в меню

if(menu.getIsMenu())

// Отображаем меню

menu.showMenu(window, \*this);

// Иначе

else {

// Падение фигуры

fallingFigure(timer, pause);

// Действие кнопки

buttonAction(key);

// Если игра окончена

if(gameOver(window, event)) {

// Возвращаемся в меню

menu.setIsMenu(true);

// Записываем лучших игроков

writeFileBestPlayers("./BestPlayersInfo.txt");

// Возвращаем истину

return true; }

// Рисуем изображение доски

drawBoardImage(window);

// Рисуем игровое поле

draw(window); }

// Отображаем окно

window.display(); } }

int Game::mousePressedCheck(sf::Event& event, sf::RenderWindow& window) {

if (sf::IntRect(1110, 478, 111, 112).contains(sf::Mouse::getPosition(window))

&& sf::Mouse::isButtonPressed(sf::Mouse::Left)) {

buttonRestart.playMusic();

return 1; }

if ((sf::IntRect(buttonPause.getPositionX(), buttonPause.getPositionY(),

buttonPause.getWidth(), buttonPause.getHeight()).contains(sf::Mouse::getPosition(window))

&& sf::Mouse::isButtonPressed(sf::Mouse::Left))) {

buttonPause.updateSprite("images/unpause.png");

buttonPause.playMusic();

window.clear();

window.draw(pauseBoard.sprite);

draw(window);

window.display();

while (window.waitEvent(event)) {

if((event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Escape)||

(sf::IntRect(1104, 619, 120, 120).contains(sf::Mouse::getPosition(window))

&& sf::Mouse::isButtonPressed(sf::Mouse::Left))) {

buttonPause.updateSprite("images/pause.png");

break; } } }

if (sf::IntRect(220, 684, 62, 34).contains(sf::Mouse::getPosition(window))

&& sf::Mouse::isButtonPressed(sf::Mouse::Left)) {

if(!buttonMusic.getIsPressed()) {

buttonMusic.playMusic();

music.pause();

buttonMusic.updateSprite("images/buttonOFF.png");

buttonMusic.setIsPressed(true); }

else {

buttonMusic.playMusic();

music.play();

buttonMusic.updateSprite("images/buttonON.png");

buttonMusic.setIsPressed(false); } }

return 3; }

// Метод для сохранения текущей игры в файл

void Game::saveGameToFile(std::string fileName) {

// Открываем файл в бинарном режиме

std::ofstream outFile(fileName, std::ios::binary);

// Если файл не открывается, выводим сообщение об ошибке и выходим из функции

if (!outFile.is\_open()) {

std::cerr << "Ошибка открытия файла для сохранения игры" << std::endl;

return; }

// Записываем в файл значение переменной score

outFile.write(reinterpret\_cast<const char\*>(&score), sizeof(int));

// Записываем в файл значение переменной time

outFile.write(reinterpret\_cast<const char\*>(&time), sizeof(int));

// Записываем в файл значение переменной countLines

outFile.write(reinterpret\_cast<const char\*>(&countLines), sizeof(int));

// Записываем в файл значение переменной lines\_in\_a\_row

outFile.write(reinterpret\_cast<const char\*>(&lines\_in\_a\_row), sizeof(int));

// Инициализируем переменную значением ячейки

int cellValue;

// Проходим по всем ячейкам игрового поля

for (int i = 0; i < field.getHeight(); ++i) {

for (int j = 0; j < field.getWidth(); ++j) {

// Получаем значение ячейки

cellValue = field.getGameBoard(i,j);

// Записываем значение ячейки в файл

outFile.write(reinterpret\_cast<const char\*>(&cellValue), sizeof(int)); } }

// Закрываем файл

outFile.close(); }

// Метод для загрузки игры из файла

void Game::loadGameFromFile(std::string fileName) {

// Открываем файл в бинарном режиме

std::ifstream inFile(fileName, std::ios::binary);

// Если файл не открывается, выводим сообщение об ошибке и выходим из функции

if (!inFile.is\_open()) {

std::cerr << "Ошибка открытия файла для чтения данных прошлой игры" << std::endl;

return; }

// Читаем из файла значение переменной score

inFile.read(reinterpret\_cast<char\*>(&score), sizeof(int));

// Читаем из файла значение переменной fileTime

inFile.read(reinterpret\_cast<char\*>(&fileTime), sizeof(int));

// Читаем из файла значение переменной countLines

inFile.read(reinterpret\_cast<char\*>(&countLines), sizeof(int));

// Читаем из файла значение переменной lines\_in\_a\_row

inFile.read(reinterpret\_cast<char\*>(&lines\_in\_a\_row), sizeof(int));

// Перезапускаем таймер игры

gameTime.restart();

// Инициализируем переменную значением ячейки

int cellValue;

// Проходим по всем ячейкам игрового поля

for (int i = 0; i < field.getHeight(); ++i) {

for (int j = 0; j < field.getWidth(); ++j) {

// Читаем значение ячейки из файла

inFile.read(reinterpret\_cast<char\*>(&cellValue), sizeof(int));

// Устанавливаем значение ячейки на игровом поле

field.setGameBoard(i, j, cellValue); } }

// Закрываем файл

inFile.close(); }

// Деструктор класса Game

Game::~Game() {

// Удаляем текущую фигуру

delete currentFigure;

// Удаляем следующую фигуру

delete nextFigure;

// Удаляем все фигуры из вектора фигур

for (Figure \* figure : figures) {

delete figure; }

// Очищаем вектор фигур

figures.clear(); }