Государственный комитет Российской Федерации по телекоммуникациям

Сибирский государственный университет

телекоммуникаций и информатики

**КУРСОВАЯ РАБОТА**

**По дисциплине «Программирование (часть 2). Язык программирования С++»**

**Вариант № 22**

Работу выполнил

студент 1 курса

группы ИП-912

Лёвкин Н. О.

Работу проверил

Перцев И.В., к.т.н.

Работа защищена

«\_\_\_» \_\_\_\_\_\_\_\_\_\_2019г.

С оценкой «\_\_\_\_\_\_\_»

**Новосибирск 2020**

ЗАДАНИЕ

Разработать программу для игры в "Арканоид". Для реализации графической составляющей была использована библиотека SFML.

СОДЕРЖАНИЕ:

[ВВЕДЕНИЕ 5](#_Toc535081923)

[1. ПОСТАНОВКА КОМПЛЕКСА ЗАДАЧ 6](#_Toc535081924)

[3. ОПИСАНИЕ СТРУКТУРЫ ПРОГРАММЫ](#_Toc535081928) 7

[3. ИСХОДНЫЙ МОДУЛЬ ПРОГРАММЫ](#_Toc535081928) 9

[4. РЕЗУЛЬТАТЫ ПРОГРАММЫ 1](#_Toc535081929)9

Целью данной работы является закрепление основных понятий программирования на языке программирования C/С++, таких как: одномерные массивы, функции, многофайловые проекты, классы а также решение поставленной задачи.

# ПОСТАНОВКА КОМПЛЕКСА ЗАДАЧ

Программа при запуске должна выводить окно с элементами меню – кнопками, как минимум “Играть” и “Выход”.

При нажатии на кнопку “Играть” происходит переход из окна меню в окно выбора уровня. При нажатии на пиктограмму уровня происходит загрузка ресурсов и переход на уровень. В окне игры находятся: платформа, отбивающая шар, сам шар способный наносит урон блокам и блоки расположенные в верхней части окна. При ударе шара об блок последний должен исчезать или менять свойства. При вылете шара за нижнюю границу экрана игрок проигрывает.

Для создания объектов типа “Кнопка”, “Кнопка с изображением”, “Текст”, используется лично написанный класс Form использующий встроенные в SFML элементы “RectangleShape”, “Text”, “Texture”, “Sprite”. Для отображения игровых объектов используются объекты “Sprite” с наложенными текстурами, находящимися по пути images/ и levels/. Для текста используется шрифт “Roboto”.

# ОПИСАНИЕ СТРУКТУРЫ ПРОГРАММЫ

Программа состоит из файла “Main.cpp”, содержащего точку входа в программу – функцию main(), файлов функций меню “MainMenu.cpp”, “LevelMenu.cpp”, “SettingsMenu.cpp”, класса Form (“Form.hpp” и “Form.cpp”) предназначенного для более удобной работы с элементами интерфейса, класса ResourceManager (“ResourceManager.hpp” и “ResourceManager.cpp”) предназначенного для работы с ресурсами игры такими как музыка и текстуры, класса “GameObject (“GameObject.hpp” и “GameObject.cpp”) предназначенного для работы с игровыми объектами “шар” и “платформа”, класса Level (“Level.hpp” и “Level.cpp”) предназначенного для загрузки уровня.

При запуске программы вызывается функция main(), откуда происходит вызов MainMenu(). При нажатии кнопки “Играть” происходит вызов функции LevelMenu(), а при нажатии на кнопку “Настройки” происходит вызов функции SettingsMenu(). При запуске уровня из LevelMenu() происходит загрузка ресурсов уровня и его запуск. Создаются массивы блоков, объект плафтормы, объект шара, объект заднего фона. Шар перемещается, используя вектор направления, скорость и delta времени между кадрами для устранения зависимости от частоты кадров. Платформа перемещается с помощью стрелок “влево” и “вправо” на клавиатуре, присутствуют ограничения как по передвижению шара, так и по передвижению платформы. Когда шар ударяется об блок у него отнимается некоторая прочность, в зависимости от чего он меняет текстуру, а при полной потери прочности исчезает из отрисовки и расчёта движений шара”.

Самые основные использованные функции:

* MainMenu(RenderWindow& window) - отвечает за действия в меню игры: отрисовка кнопок и фона, начало игры, выход из приложения.
* Level::changeLevel(RenderWindow& window) - главная для игрового процесса функция. Отвечает за отрисовку персонажа, вызывает функцию update() для его движения, . Ответственна за отрисовку фона и специальных предметов, вызов паузы.

Функции класса GameObject:

* update(Sprite &paddle, float time\_, float dx\_) – отвечает за проверку пересечения объекта шара с блоками, стенами, платформой и обновлением позиции.
* BorderCollision() – отвечает за проверку пересечения с границами окна.
* BlockCollision() – отвечает за проверку пересечения с границами экрана.
* PaddleCollision() – отвечает за проверку пересечения с платформой.
* kick() – отвечает за выталкивание шара в сторону отражения при ударе для устранения застревания в объектах.
* normalise() – нормализует dx и dy для вектора velocity.

Исходный код программы

**Main.cpp**

#include <iostream>

#include <ctime>

#include <filesystem>

#include <SFML/Graphics.hpp>

#include <SFML/Audio/Music.hpp>

#include "MainMenu.hpp"

#include "ResourceManager.hpp";

using namespace std;

using namespace sf;

int main() {

srand(static\_cast<unsigned int>(time(NULL)));

Clock clock;

ContextSettings settings;

settings.antialiasingLevel = 12;

ResourceManager &gameResources = ResourceManager::getInstance();

Settings &gameSettings = Settings::getInstance();

RenderWindow window(VideoMode(gameSettings.getResolution().x, gameSettings.getResolution().y), "Arkanoid", Style::Close);

//window.setVerticalSyncEnabled(true);

Texture loadingImage;

loadingImage.loadFromFile("images/loading.png");

loadingImage.setSmooth(true);

Sprite loading;

loading.setTexture(loadingImage);

loading.setScale(Vector2f(window.getSize().x / loading.getGlobalBounds().width, window.getSize().y / loading.getGlobalBounds().height));

while (window.isOpen()) {

Event event;

while (window.pollEvent(event)) {

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

window.close();

}

}

window.draw(loading);

if (clock.getElapsedTime().asSeconds() > 1) {

if (Settings::getInstance().successfulLoaded()) {

MainMenu(window);

}

}

window.display();

}

}

**class ResourceManager**

**ResourceManager.hpp**

#pragma once

#include <iostream>

#include <SFML/Audio/Music.hpp>

#include <SFML/Graphics.hpp>

#include <filesystem>

using namespace std;

using namespace sf;

namespace fs = filesystem;

class ResourceManager

{

protected:

bool isFullscreen = false;

bool isAmbientPlay = true;

bool allLoaded = false;

int i = 0;

string pathToMusic = "music/";

vector<string> allMusic;

map<string,Texture> loadedTextures;

Music ambient;

Vector2i resolution = Vector2i(1280, 720);

public:

static ResourceManager& getInstance();

void loadTexture(const string &name, const string &path);

void unloadTexture(const string &name);

Texture& getTexture(const string &name);

};

class Settings : protected ResourceManager

{

private:

Settings() {

for (auto& file : fs::directory\_iterator(pathToMusic)) {

allMusic.push\_back(file.path().filename().string());

}

i = rand() % (allMusic.size() - 1);

ambient.setVolume(20);

allLoaded = true;

}

public:

static Settings& getInstance();

Vector2i getResolution();

bool successfulLoaded();

void switchFullScreen(RenderWindow& window);

float getVolume(string music);

void setVolume(string music, float volume\_);

void switchMusic();

void updateMusic();

};

**ResourceManager.cpp**

#define NOMINMAX

#include<Windows.h>

#include <iostream>

#include <SFML/Audio/Music.hpp>

#include <SFML/Graphics.hpp>

#include "ResourceManager.hpp"

const int FLWIDTH = GetSystemMetrics(SM\_CXSCREEN); //Разрешение экрана по X

const int FLHEIGHT = GetSystemMetrics(SM\_CYSCREEN); //Разрешение экрана по Y

//const int FLWIDTH = 1920;

//const int FLHEIGHT = 1080;

using namespace std;

using namespace sf;

ResourceManager& ResourceManager::getInstance()

{

static ResourceManager instance;

return instance;

}

void ResourceManager::loadTexture(const string &name, const string &path)

{

loadedTextures.insert({ name, Texture() });

loadedTextures[name].loadFromFile(path);

}

void ResourceManager::unloadTexture(const string& name)

{

loadedTextures.erase(name);

}

Texture& ResourceManager::getTexture(const string& name)

{

return loadedTextures[name];

}

Settings& Settings::getInstance() {

static Settings instance;

return instance;

}

Vector2i Settings::getResolution() {

return resolution;

}

void Settings::switchFullScreen(RenderWindow& window) {

if (isFullscreen) {

window.create(VideoMode(1280, 720), "Arkanoid", Style::Default);

resolution = Vector2i(1280, 720);

}

else {

resolution = Vector2i(FLWIDTH, FLHEIGHT);

window.create(VideoMode(FLWIDTH, FLHEIGHT), "Arkanoid", Style::Fullscreen);

}

isFullscreen = !isFullscreen;

}

bool Settings::successfulLoaded() {

return allLoaded;

}

float Settings::getVolume(string music) {

if (music == "ambient") {

return ambient.getVolume();

}

return 100;

}

void Settings::setVolume(string music, float volume\_) {

if (music == "ambient") {

if (ambient.getVolume() + volume\_ >= 0 && ambient.getVolume() + volume\_ <= 100) {

ambient.setVolume(ambient.getVolume() + volume\_);

}

}

}

void Settings::switchMusic() {

isAmbientPlay = !isAmbientPlay;

if (isAmbientPlay)

ambient.play();

}

void Settings::updateMusic() {

if (isAmbientPlay) {

if (ambient.getStatus() == Music::Playing);

else {

ambient.openFromFile(pathToMusic + allMusic[i]);

ambient.play();

i++;

if (i >= allMusic.size()) {

i = 0;

}

}

}

else {

ambient.pause();

}

}

**MainMenu.cpp**

#include <iostream>

#include <SFML/Graphics.hpp>

#include <SFML/Audio/Music.hpp>

#include "Form.hpp"

#include "MainMenu.hpp"

#include "LevelMenu.hpp"

#include "SettingsMenu.hpp"

#include "ResourceManager.hpp";

#include "Level.hpp"

using namespace std;

using namespace sf;

using namespace form;

void MainMenu(RenderWindow& window)

{

const int WIDTH = window.getSize().x, HEIGHT = window.getSize().y;

Button bPlay(WIDTH / 2 - 100, (HEIGHT / 2) - 25 - 150 , 200, 50, "Play", 20),

bSettings(WIDTH / 2 - 100, (HEIGHT / 2) - 25, 200, 50, "Settings", 20),

bExit(WIDTH / 2 - 100, (HEIGHT / 2) - 25 + 150, 200, 50, "Exit", 20);

while (window.isOpen()) {

Event event;

Vector2i mouse = Mouse::getPosition(window);

Settings::getInstance().updateMusic();

window.clear(Color(34, 34, 38));

bPlay.setStyle("default");

bSettings.setStyle("default");

bExit.setStyle("default");

if (bPlay.select(mouse)) {

bPlay.setStyle("hover");

}

if (bSettings.select(mouse)) {

bSettings.setStyle("hover");

}

if (bExit.select(mouse)) {

bExit.setStyle("hover");

}

while (window.pollEvent(event)) {

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

window.close();

}

if (event.type == Event::MouseButtonReleased)

{

if (event.mouseButton.button == Mouse::Left)

{

if (bPlay.select(mouse)) {

Level::getInstance().clearLevel();

LevelMenu(window);

}

if (bSettings.select(mouse)) {

SettingsMenu(window);

}

if (bExit.select(mouse)) {

window.close();

}

}

}

}

window.draw(bPlay.displayButton());

window.draw(bPlay.displayText());

window.draw(bSettings.displayButton());

window.draw(bSettings.displayText());

window.draw(bExit.displayButton());

window.draw(bExit.displayText());

window.display();

}

}

**LevelMenu.cpp**

#include <iostream>

#include <SFML/Graphics.hpp>

#include "Form.hpp"

#include "LevelMenu.hpp"

#include "MainMenu.hpp"

#include "ResourceManager.hpp"

#include "Level.hpp"

using namespace std;

using namespace sf;

using namespace form;

void LevelMenu(RenderWindow& window)

{

Level& level = Level::getInstance();

Font font;

font.loadFromFile("fonts/Roboto/Roboto-Regular.ttf");

const int WIDTH = window.getSize().x, HEIGHT = window.getSize().y;

bool isWindow = true;

Label lTop("Levels", WIDTH / 2, 40, true, 36);

Button bMenu(40, HEIGHT - 90, 200, 50, "Menu", 20);

vector<ImageButton> levels;

for (int i = 0; i < 1; i++) {

for (int j = 0; j < 6; j++) {

levels.push\_back(ImageButton(55 + j \* 200, 100 + i \* 180, 177.7, 100, "levels/level\_" + to\_string(i + j + 1) + ".png"));

}

}

while (isWindow) {

Event event;

Vector2i mouse = Mouse::getPosition(window);

Settings::getInstance().updateMusic();

window.clear(Color(34, 34, 38));

for (int i = 0; i < levels.size(); i++) {

levels[i].setStyle("default");

}

bMenu.setStyle("default");

for (int i = 0; i < levels.size(); i++) {

if (levels[i].select(mouse)) {

levels[i].setStyle("hover");

}

}

if (bMenu.select(mouse)) {

bMenu.setStyle("hover");

}

while (window.pollEvent(event)) {

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

window.close();

}

if (event.type == Event::MouseButtonReleased) {

if (event.mouseButton.button == Mouse::Left) {

if (bMenu.select(mouse)) {

MainMenu(window);

}

if (levels[0].select(mouse)) {

isWindow = false;

level.loadLevel(window, "level\_1");

}

if (levels[1].select(mouse)) {

isWindow = false;

level.loadLevel(window, "level\_2");

}

if (levels[2].select(mouse)) {

isWindow = false;

level.loadLevel(window, "level\_3");

}

if (levels[3].select(mouse)) {

isWindow = false;

level.loadLevel(window, "level\_4");

}

if (levels[4].select(mouse)) {

isWindow = false;

level.loadLevel(window, "level\_5");

}

if (levels[5].select(mouse)) {

isWindow = false;

level.loadLevel(window, "level\_6");

}

}

}

if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape))

{

MainMenu(window);

}

}

window.draw(lTop.display());

window.draw(bMenu.displayButton());

window.draw(bMenu.displayText());

for (auto level : levels) {

window.draw(level.displayButton());

window.draw(level.displaySprite());

}

window.display();

}

}

**SettingsMenu.cpp**

#include <iostream>

#include <SFML/Graphics.hpp>

#include "Form.hpp"

#include "SettingsMenu.hpp"

#include "MainMenu.hpp"

#include "ResourceManager.hpp";

#include "Level.hpp"

using namespace std;

using namespace sf;

using namespace form;

void SettingsMenu(RenderWindow& window)

{

Font font;

font.loadFromFile("fonts/Roboto/Roboto-Regular.ttf");

const int WIDTH = window.getSize().x, HEIGHT = window.getSize().y;

Label lTop("Settings", WIDTH / 2, 40, true, 36);

Button bResolution(WIDTH / 2 - 75, HEIGHT / 2 - 120, 150, 40, "");

Button bMusicState(WIDTH / 2 - 75, HEIGHT / 2 - 20, 150, 40, ""),

bMusicLow(WIDTH / 2 - 125, HEIGHT / 2 - 20, 40, 40, "-"),

bMusicHigh(WIDTH / 2 + 85, HEIGHT / 2 - 20, 40, 40, "+");

Button bMenu(40, HEIGHT - 90, 200, 50, "Menu", 20);

while (window.isOpen()) {

Event event;

Vector2i mouse = Mouse::getPosition(window);

Settings::getInstance().updateMusic();

bResolution.setText(to\_string(window.getSize().x) + "x" + to\_string(window.getSize().y));

bMusicState.setText(to\_string(static\_cast<int>(ceil(Settings::getInstance().getVolume("ambient")))));

window.clear(Color(34, 34, 38));

bResolution.setStyle("default");

bMusicState.setStyle("default");

bMusicHigh.setStyle("default");

bMusicLow.setStyle("default");

bMenu.setStyle("default");

if (bResolution.select(mouse)) {

bResolution.setStyle("hover");

bResolution.setText("Fullscreen n/f");

}

if (bMusicState.select(mouse)) {

bMusicState.setStyle("hover");

bMusicState.setText("Pause/Play");

}

if (bMusicHigh.select(mouse)) {

bMusicHigh.setStyle("hover");

}

if (bMusicLow.select(mouse)) {

bMusicLow.setStyle("hover");

}

if (bMenu.select(mouse)) {

bMenu.setStyle("hover");

}

while (window.pollEvent(event)) {

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

window.close();

}

if (event.type == Event::MouseButtonReleased) {

if (event.mouseButton.button == Mouse::Left) {

if (bResolution.select(mouse)) {

Settings::getInstance().switchFullScreen(window);

return SettingsMenu(window);

}

if (bMusicState.select(mouse)) {

Settings::getInstance().switchMusic();

}

if (bMusicHigh.select(mouse)) {

Settings::getInstance().setVolume("ambient", 5);

}

if (bMusicLow.select(mouse)) {

Settings::getInstance().setVolume("ambient", -5);

}

if (bMenu.select(mouse)) {

MainMenu(window);

}

}

}

if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape))

{

MainMenu(window);

}

}

window.draw(lTop.display());

window.draw(bResolution.displayButton());

window.draw(bResolution.displayText());

window.draw(bMusicState.displayButton());

window.draw(bMusicState.displayText());

window.draw(bMusicHigh.displayButton());

window.draw(bMusicHigh.displayText());

window.draw(bMusicLow.displayButton());

window.draw(bMusicLow.displayText());

window.draw(bMenu.displayButton());

window.draw(bMenu.displayText());

window.display();

}

}

**class Level**

**Level.hpp**

#pragma once

#include <iostream>

#include <fstream>

#include "Form.hpp"

#include "ResourceManager.hpp"

#include "json.hpp"

using json = nlohmann::json;

class Level

{

protected:

bool isPaused = false;

json loadedLevel;

map<string, int> loadedBlocks;

string level;

Clock clock;

public:

Level(const string& level\_ = "level\_1") {

void preloadLevel(const string & level\_);

time = 0;

clock.restart();

}

void clearLevel() {

time = 0;

clock.restart();

}

float time;

static Level& getInstance();

void loadLevel(RenderWindow &window, const string& level\_);

void preloadLevel(const string& level\_);

void changeLevel(RenderWindow& window);

int getBlocksCount(const string& block\_);

bool getPaused();

void clockRestart();

void run() {

time = static\_cast<float> (clock.restart().asSeconds() \* 0.06);

}

class EscMenu

{

private:

form::Label lLevel;

EscMenu() {

Settings& settings = Settings::getInstance();

rBlackout = RectangleShape(Vector2f(settings.getResolution().x, settings.getResolution().y));

rBlackout.setFillColor(Color(0, 0, 0, 120));

rBlackoutMenu = RectangleShape(Vector2f(settings.getResolution().x / 3, settings.getResolution().y / 2));

rBlackoutMenu.setPosition(settings.getResolution().x / 2 - rBlackoutMenu.getSize().x / 2, settings.getResolution().y / 2 - rBlackoutMenu.getSize().y / 2);

rBlackoutMenu.setFillColor(Color(201, 178, 151));

rBlackoutMenu.setOutlineThickness(2);

rBlackoutMenu.setOutlineColor(Color(66, 66, 66));

}

public:

static EscMenu& getInstance();

void escMenuToggle();

void drawEscMenu(RenderWindow& window);

form::Button bResume = form::Button(Settings::getInstance().getResolution().x / 2 - 110, Settings::getInstance().getResolution().y / 2 - 100, 220, 50, "Resume", 20);

form::Button bBackToMenu = form::Button(Settings::getInstance().getResolution().x / 2 - 110, Settings::getInstance().getResolution().y / 2 + 50, 220, 50, "Menu", 20);

RectangleShape rBlackout;

RectangleShape rBlackoutMenu;

};

};

**Level.cpp**

#include <iostream>

#include "Level.hpp"

#include "GameObject.hpp"

#include "MainMenu.hpp"

#include "Form.hpp"

Level& Level::getInstance()

{

static Level instance;

return instance;

}

void Level::loadLevel(RenderWindow &window, const string& level\_)

{

if (level != level\_) {

preloadLevel(level\_);

}

changeLevel(window);

}

void Level::preloadLevel(const string& level\_)

{

level = level\_;

ResourceManager& inst = ResourceManager::getInstance();

ifstream levelFile("levels/" + level\_ + ".json");

levelFile >> loadedLevel;

levelFile.close();

for (auto texture : loadedLevel["textures"]) {

inst.loadTexture(texture[0], texture[1]);

}

for (auto block : loadedLevel["blocks"]) {

loadedBlocks.insert(make\_pair(block[0], block[1]));

}

}

void Level::changeLevel(RenderWindow &window)

{

ResourceManager& resource = ResourceManager::getInstance();

Settings& settings = Settings::getInstance();

EscMenu& esc = EscMenu::getInstance();

Clock blockClock;

Clock ballClock;

Thread gameTime(&Level::run, &Level::getInstance());

gameTime.launch();

const int WIDTH = window.getSize().x, HEIGHT = window.getSize().y;

bool isDrawable = true, isPaused = false;

int Lose = 0;

form::Label lWin("YOU WON!", WIDTH / 2, 100, true, 64);

form::Label lLose("You Lose", WIDTH / 2, 100, true, 64);

form::Label lCount("", 50, 50, false, 32);

Sprite sBackground(resource.getTexture("Background")), sPaddle(resource.getTexture("Paddle"));

sBackground.scale(WIDTH / sBackground.getLocalBounds().width, HEIGHT / sBackground.getLocalBounds().height);

sPaddle.setPosition(static\_cast<float> (WIDTH / 2), static\_cast<float> (HEIGHT - 50));

FloatRect fRect = sPaddle.getLocalBounds();

sPaddle.setOrigin(fRect.left + fRect.width / 2.0f,

fRect.top + fRect.height / 2.0f);

vector<Block> blocksBase(getBlocksCount("Center"), Block(2, resource.getTexture("BlockGreen"), blockClock));

int bBase = 0;

vector<Block> blocksSideL(getBlocksCount("Left"), Block(2, resource.getTexture("BlockRed"), blockClock));

int bSideL = 0;

vector<Block> blocksSideR(getBlocksCount("Right"), Block(2, resource.getTexture("BlockBlue"), blockClock));

int bSideR = 0;

vector<Ball> balls(1, Ball(sPaddle, resource.getTexture("Ball"), blocksSideL, blocksBase, blocksSideR, ballClock));

int n = 0;

for (unsigned int i = 1; i <= ceil(blocksBase.size() / 10); i++)

for (int j = 1; j <= ceil(blocksBase.size() / 10); j++) {

blocksBase[n].sprite.setPosition(i \* blocksBase[n].sprite.getLocalBounds().width + WIDTH/5, j \* blocksBase[n].sprite.getLocalBounds().height);

FloatRect fRect = blocksBase[n].sprite.getLocalBounds();

blocksBase[n].sprite.setOrigin(fRect.left + fRect.width / 2.0f,

fRect.top + fRect.height / 2.0f);

n++;

}

n = 0;

for (unsigned int i = 1; i <= 1; i++)

for (int j = 1; j <= 10; j++) {

blocksSideL[n].sprite.setPosition(i \* 42 \* 1.5, j \* 30);

FloatRect fRect = blocksSideL[n].sprite.getLocalBounds();

blocksSideL[n].sprite.setOrigin(fRect.left + fRect.width / 2.0f,

fRect.top + fRect.height / 2.0f);

n++;

}

n = 0;

for (unsigned int i = 1; i <= 1; i++)

for (int j = 1; j <= 10; j++) {

blocksSideR[n].sprite.setPosition(i \* 42 + WIDTH - (3 \* 42), j \* 30);

FloatRect fRect = blocksSideR[n].sprite.getLocalBounds();

blocksSideR[n].sprite.setOrigin(fRect.left + fRect.width / 2.0f,

fRect.top + fRect.height / 2.0f);

n++;

}

while (window.isOpen()) {

Event event;

Clock clock;

Vector2i mouse = Mouse::getPosition(window);

settings.updateMusic();

while (window.pollEvent(event)) {

if (event.type == Event::Closed) {

window.close();

}

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

esc.escMenuToggle();

}

}

if (!getPaused()) {

float dx = 0;

if (sPaddle.getPosition().x - sPaddle.getOrigin().x > 0 && (Keyboard::isKeyPressed(Keyboard::Left))) {

sPaddle.move(-10 \* time, 0);

dx = -static\_cast<float> (rand() % 5 + 2)/10.0f;

}

if (sPaddle.getPosition().x + sPaddle.getOrigin().x < WIDTH && (Keyboard::isKeyPressed(Keyboard::Right))) {

sPaddle.move(10 \* time, 0);

dx = static\_cast<float> (rand() % 5 + 2) / 10.0f;

}

for (auto& ball : balls) {

ball.update(sPaddle, time, dx);

}

}

else {

esc.bResume.setStyle("default");

esc.bBackToMenu.setStyle("default");

if (Lose != 0) {

if (esc.bResume.select(mouse)) {

esc.bResume.setStyle("hover");

}

}

else

{

esc.bResume.setStyle("disabled");

}

if (esc.bBackToMenu.select(mouse)) {

esc.bBackToMenu.setStyle("hover");

}

if (event.type == Event::MouseButtonReleased)

{

if (event.mouseButton.button == Mouse::Left)

{

if (esc.bResume.select(mouse)) {

esc.escMenuToggle();

}

if (esc.bBackToMenu.select(mouse)) {

esc.escMenuToggle();

MainMenu(window);

}

}

}

}

window.clear();

if (isDrawable) {

window.draw(sBackground);

window.draw(sPaddle);

Lose = 0;

for (auto& ball : balls) {

if (ball.drawable) {

ball.draw(window);

Lose++;

}

lCount.setText(to\_string(ball.count));

}

if (Lose == 0) {

Level::getInstance().isPaused = true;

}

bBase = 0;

for (auto &blockBase : blocksBase) {

blockBase.draw(window);

if (blockBase.ableToDraw) {

bBase++;

}

}

bSideL = 0;

for (auto &blockL : blocksSideL) {

blockL.draw(window);

if (blockL.ableToDraw) {

bSideL++;

}

}

bSideR = 0;

for (auto& blockR : blocksSideR) {

blockR.draw(window);

if (blockR.ableToDraw) {

bSideR++;

}

}

if (bBase == 0 && bSideL == 0 && bSideR == 0)

{

Level::getInstance().isPaused = true;

}

window.draw(lCount.display());

if (getPaused()) {

esc.drawEscMenu(window);

if (Lose == 0) {

window.draw(lLose.display());

}

if (bBase == 0 && bSideL == 0 && bSideR == 0) {

window.draw(lWin.display());

}

}

}

window.display();

}

}

int Level::getBlocksCount(const string& block\_)

{

return loadedBlocks[block\_];

}

bool Level::getPaused() {

return isPaused;

}

void Level::clockRestart()

{

clock.restart();

}

Level::EscMenu& Level::EscMenu::getInstance()

{

static EscMenu instance;

return instance;

}

void Level::EscMenu::escMenuToggle()

{

Level::getInstance().isPaused = !Level::getInstance().isPaused;

}

void Level::EscMenu::drawEscMenu(RenderWindow& window)

{

window.draw(rBlackout);

window.draw(rBlackoutMenu);

window.draw(bResume.displayButton());

window.draw(bResume.displayText());

window.draw(bBackToMenu.displayButton());

window.draw(bBackToMenu.displayText());

}

**class GameObject**

**GameObject.hpp**

#pragma once

#include <iostream>

#include <random>

#include <SFML/Graphics.hpp>

#include "Form.hpp"

//float normalise(const float value);

class Block {

public:

Sprite sprite;

bool ableToDraw;

bool isDamaged;

Clock clock;

Time damaged\_timer;

Block();

Block(int type\_, Texture& texture, Clock clock\_);

int getHp();

void update();

void draw(RenderWindow& window);

void damaged(int damage);

private:

int hp;

int type;

};

class Ball {

public:

float dx = static\_cast<float>(rand() % 3 - 6);

float dy = -5;

float speed = 8;

Sprite sprite;

Clock clock;

Vector2f velocity;

vector<Block>\* blocksSideL;

vector<Block>\* blocksBase;

vector<Block>\* blocksSideR;

int count = 3;

bool drawable = true;

Ball(const Sprite& paddle\_, Texture& texture\_, vector<Block>& blocksLeft\_, vector<Block>& blocksCenter\_, vector<Block>& blocksRight\_, Clock clock\_);

Ball(Vector2f startPosition\_, Texture& texture\_, vector<Block> &blocksLeft\_, vector<Block>& blocksCenter\_, vector<Block>& blocksRight\_, Clock clock\_);

void lifedown();

float normalise(float value);

void kick(float x, float y);

void PaddleCollision(Sprite &paddle, float dx\_);

void collisionReflecting(Block &block);

void BlockCollision();

void BorderCollision();

void update(Sprite &paddle, float time\_, float dx\_);

void draw(RenderWindow& window);

};

**GameObject.hpp**

#include <iostream>

#include "GameObject.hpp"

#include "ResourceManager.hpp"

using namespace std;

Block::Block() {

ableToDraw = true;

damaged\_timer = {};

isDamaged = false;

hp = 1;

type = 1;

}

Block::Block(int type\_, Texture& texture, Clock clock\_) {

ableToDraw = true;

isDamaged = false;

clock = clock\_;

clock.restart();

hp = type\_;

type = type\_;

sprite.setTexture(texture);

}

int Block::getHp() {

return hp;

}

void Block::update() {

ResourceManager &res = ResourceManager::getInstance();

if (hp < type) {

sprite.setTexture(res.getTexture("BlockWhite"));

}

if (hp < 1) {

ableToDraw = false;

}

else {

ableToDraw = true;

}

}

void Block::draw(RenderWindow& window) {

if (ableToDraw)

window.draw(sprite);

}

void Block::damaged(int damage) {

if (clock.getElapsedTime().asMilliseconds() > 200) {

hp -= damage;

update();

clock.restart();

}

}

Ball::Ball(const Sprite& paddle\_, Texture& texture\_, vector<Block>& blocksLeft\_, vector<Block>& blocksCenter\_, vector<Block>& blocksRight\_, Clock clock\_)

{

blocksSideL = &blocksLeft\_;

blocksBase = &blocksCenter\_;

blocksSideR = &blocksRight\_;

sprite.setPosition(paddle\_.getPosition() + Vector2f(0, -50));

FloatRect fRect = sprite.getLocalBounds();

sprite.setOrigin(fRect.left + fRect.width / 2.0f,

fRect.top + fRect.height / 2.0f);

sprite.setTexture(texture\_);

velocity = Vector2f(normalise(dx), normalise(dy));

clock = clock\_;

clock.restart();

}

Ball::Ball(Vector2f startPosition\_, Texture& texture\_, vector<Block>& blocksLeft\_, vector<Block>& blocksCenter\_, vector<Block>& blocksRight\_, Clock clock\_)

{

blocksSideL = &blocksLeft\_;

blocksBase = &blocksCenter\_;

blocksSideR = &blocksRight\_;

sprite.setPosition(startPosition\_ + Vector2f(rand() % 200 - 100, rand() % 20 - 10));

FloatRect fRect = sprite.getLocalBounds();

sprite.setOrigin(fRect.left + fRect.width / 2.0f,

fRect.top + fRect.height / 2.0f);

sprite.setTexture(texture\_);

velocity = Vector2f(normalise(dx), normalise(dy));

clock = clock\_;

clock.restart();

}

void Ball::lifedown()

{

if (count > 1) {

count--;

sprite.setPosition(Vector2f(600, 600) + Vector2f(0, -50));

}

else {

drawable = false;

}

}

float Ball::normalise(float value) {

return value / sqrt(dx \* dx + dy \* dy);

}

void Ball::kick(float x, float y) {

sprite.move(0, 0);

sprite.move(x \* 1, y \* 1);

}

void Ball::collisionReflecting(Block &block) {

if (block.ableToDraw && sprite.getGlobalBounds().intersects(block.sprite.getGlobalBounds())) {

float bx = velocity.x;

float by = velocity.y;

if (sprite.getPosition().x < (block.sprite.getPosition().x - block.sprite.getOrigin().x))

bx = -bx;

if (sprite.getPosition().x + sprite.getOrigin().x > (block.sprite.getPosition().x + block.sprite.getOrigin().x))

bx = -bx;

if (sprite.getPosition().y < (block.sprite.getPosition().y - block.sprite.getOrigin().y))

by = -by;

if (sprite.getPosition().y + sprite.getOrigin().y > (block.sprite.getPosition().y + block.sprite.getOrigin().y)) {

by = -by;

}

kick(normalise(bx), normalise(by));

velocity = Vector2f(normalise(bx), normalise(by));

dx = bx;

dy = by;

if (clock.getElapsedTime().asSeconds() \* 1000 > 20) {

block.damaged(1);

clock.restart();

}

}

}

void Ball::PaddleCollision(Sprite& paddle, float dx\_)

{

if (sprite.getGlobalBounds().intersects(paddle.getGlobalBounds())) {

float pX = velocity.x;

float pY = velocity.y;

if (sprite.getPosition().x < (paddle.getPosition().x - paddle.getOrigin().x + 2) || sprite.getPosition().x > (paddle.getPosition().x + paddle.getOrigin().x - 2)) {

pX = -pX;

}

pY = -pY;

if ((sprite.getPosition().y + sprite.getOrigin().y) > (paddle.getPosition().y - paddle.getOrigin().y)) {

kick(pX, (pY + 100));

}

kick(pX, pY);

velocity = Vector2f(pX, pY);

dx\_ != 0 ? dx = pX + dx\_ : dx = pX;

dy = pY;

}

}

void Ball::BlockCollision()

{

for (auto& blockBase : \*blocksBase) {

collisionReflecting(blockBase);

}

for (auto& blockL : \*blocksSideL) {

collisionReflecting(blockL);

}

for (auto& blockR : \*blocksSideR) {

collisionReflecting(blockR);

}

}

void Ball::BorderCollision()

{

float borderX = velocity.x;

float borderY = velocity.y;

bool wallHit = false;

if (sprite.getPosition().x < 0) {

borderX = -borderX;

wallHit = true;

}

if (sprite.getPosition().x + sprite.getOrigin().x > Settings::getInstance().getResolution().x) {

borderX = -borderX;

wallHit = true;

}

if (sprite.getPosition().y < 0) {

borderY = -borderY;

wallHit = true;

}

if (sprite.getPosition().y + sprite.getOrigin().y > Settings::getInstance().getResolution().y) {

borderY = -borderY;

wallHit = true;

lifedown();

}

if (wallHit) {

kick(normalise(borderX), normalise(borderY));

velocity = Vector2f(normalise(borderX), normalise(borderY));

dx = borderX;

dy = borderY;

}

}

void Ball::update(Sprite &paddle\_, float time\_, float dx\_)

{

if (drawable) {

velocity = Vector2f(normalise(dx), normalise(dy));

BlockCollision();

PaddleCollision(paddle\_, dx\_);

BorderCollision();

sprite.move(velocity.x \* speed \* time\_, velocity.y \* speed \* time\_);

}

}

void Ball::draw(RenderWindow& window)

{

window.draw(sprite);

}

**class Form**

**Form.hpp**

#pragma once

#include <iostream>

#include <string>

#include <SFML/Graphics.hpp>

using namespace std;

using namespace sf;

namespace form {

class Label {

public:

Label(string text\_ = "Text", float x\_ = 0, float y\_ = 0, bool center\_ = false, size\_t size\_ = 20, Color color\_ = Color::White) {

font.loadFromFile("fonts/Roboto/Roboto-Bold.ttf");

x = x;

y = y;

text = text\_;

TXT.setFont(font);

TXT.setCharacterSize(size\_);

TXT.setFillColor(color\_);

TXT.setPosition(x\_, y\_);

TXT.setString(text\_);

if (center\_) {

updateTextOrigin();

}

}

string getText();

void setText(string text\_);

void updateTextOrigin();

Text& display();

protected:

float x;

float y;

Font font;

string text;

Text TXT;

};

class Button: public Label {

public:

Button(float x\_ = 0, float y\_ = 0, float width\_ = 200, float height\_ = 50, string text\_ = "", size\_t size\_ = 20, Color buttonColor\_ = Color(133, 133, 133), Color color\_ = Color::White) {

font.loadFromFile("fonts/Roboto/Roboto-Bold.ttf");

x = x\_;

y = y\_;

width = width\_;

height = height\_;

text = text\_;

TXT.setFont(font);

TXT.setCharacterSize(size\_);

TXT.setFillColor(color\_);

TXT.setString(text\_);

FloatRect fRect = TXT.getLocalBounds();

TXT.setOrigin(fRect.left + fRect.width / 2.0f,

fRect.top + fRect.height / 2.0f);

TXT.setPosition(x\_ + width\_ / 2, y\_ + height\_ / 2);

BOX.setSize(Vector2f(width\_, height\_));

BOX.setPosition(x\_, y\_);

BOX.setFillColor(buttonColor\_);

BOX.setOutlineThickness(2);

BOX.setOutlineColor(Color(66, 66, 66));

}

bool select(Vector2i mouse\_);

void setStyle(string style);

void setPosition(float x\_, float y\_);

Text& displayText();

RectangleShape& displayButton();

protected:

float width;

float height;

RectangleShape BOX;

};

class ImageButton : public Button {

public:

ImageButton(float x\_ = 0, float y\_ = 0, float width\_ = 0, float height\_ = 0, string image\_ = "", string text\_ = "", size\_t size\_ = 20, Color buttonColor\_ = Color(133, 133, 133), Color color\_ = Color::White) {

x = x\_;

y = y\_;

width = width\_;

height = height\_;

text = text\_;

BOX.setSize(Vector2f(width\_, height\_));

BOX.setPosition(x\_, y\_);

BOX.setFillColor(buttonColor\_);

BOX.setOutlineThickness(2);

BOX.setOutlineColor(Color(66, 66, 66));

TEXTURE.loadFromFile(image\_);

SPRITE.setTexture(TEXTURE, true);

SPRITE.setScale(width\_ / SPRITE.getLocalBounds().width, height\_ / SPRITE.getLocalBounds().height);

SPRITE.setPosition(x\_, y\_);

}

Sprite& displaySprite();

private:

Texture TEXTURE;

Sprite SPRITE;

};

class Input : public Button {

public:

Input(float x\_ = 0, float y\_ = 0, float width\_ = 200, float height\_ = 50, string text\_ = "", size\_t size\_ = 20, Color buttonColor\_ = Color(133, 133, 133), Color color\_ = Color::White) {

x = x\_;

y = y\_;

width = width\_;

height = height\_;

text = text\_;

focus = false;

TXT.setFont(font);

TXT.setCharacterSize(size\_);

TXT.setFillColor(color\_);

TXT.setPosition(x\_ + 10, y\_ + height\_ / 4);

TXT.setString(text\_);

FloatRect fRect = TXT.getLocalBounds();

TXT.setOrigin(fRect.left + fRect.width / 2.0f,

fRect.top + fRect.height / 2.0f);

BOX.setSize(Vector2f(width\_, height\_));

BOX.setPosition(x, y);

BOX.setFillColor(buttonColor\_);

BOX.setOutlineThickness(2);

BOX.setOutlineColor(Color(66, 66, 66));

}

void reText(char tmp\_);

bool select();

bool select(Vector2i mouse\_);

Text& displayText();

RectangleShape& displayBOX();

string readText();

protected:

bool focus;

};

} // namespace form

**Form.cpp**

#include <iostream>

#include <string>

#include <SFML/Graphics.hpp>

#include "Form.hpp"

using namespace std;

using namespace sf;

string form::Label::getText()

{

return text;

}

void form::Label::setText(string text\_)

{

text = text\_;

}

void form::Label::updateTextOrigin()

{

FloatRect fRect = TXT.getLocalBounds();

TXT.setOrigin(fRect.left + fRect.width / 2.0f,

fRect.top + fRect.height / 2.0f);

}

Text& form::Label::display() {

updateTextOrigin();

TXT.setString(text);

return TXT;

}

bool form::Button::select(Vector2i mouse\_)

{

if ((mouse\_.x > x && mouse\_.x < x + width) &&

mouse\_.y > y && mouse\_.y < y + height) {

return true;

}

else {

return false;

}

}

void form::Button::setStyle(string style\_)

{

if (style\_ == "default") {

BOX.setFillColor(Color(133, 133, 133));

}

else if (style\_ == "hover") {

BOX.setFillColor(Color(158, 158, 158));

}

else if (style\_ == "disabled") {

BOX.setFillColor(Color(200, 200, 200));

}

}

void form::Button::setPosition(float x\_, float y\_)

{

BOX.setPosition(x\_, y\_);

}

Text& form::Button::displayText()

{

updateTextOrigin();

TXT.setString(text);

return TXT;

}

RectangleShape& form::Button::displayButton()

{

return BOX;

}

void form::Input::reText(char tmp\_)

{

text.erase(text.size() - 1);

if (tmp\_ != 8) {

text += tmp\_;

}

else {

if (text.size() > 0) {

text.erase(text.size() - 1);

}

}

text += "|";

TXT.setString(text);

}

bool form::Input::select()

{

return focus;

}

bool form::Input::select(Vector2i mouse\_)

{

if ((mouse\_.x > x && mouse\_.x < x + width) &&

(mouse\_.y > y && mouse\_.y < y + height)) {

focus = true;

text += "|";

}

else {

if (text.size() > 0) {

if (text[text.size() - 1] == 124) {

text.erase(text.size() - 1);

}

}

focus = false;

}

return true;

}

Text& form::Input::displayText()

{

TXT.setString(text);

return TXT;

}

RectangleShape& form::Input::displayBOX()

{

return BOX;

}

string form::Input::readText()

{

if (text[text.size() - 1] == '|') {

text.erase(text.size() - 2);

}

return text;

}

Sprite& form::ImageButton::displaySprite()

{

return SPRITE;

}