|  |  |  |
| --- | --- | --- |
| Санкт-Петербургский политехнический университет Петра Великого  Институт компьютерных наук и технологий  **Высшая школа программной инженерии** | | |
| Курсовая работа | | |
|  | | |
| по дисциплине  «Объектно-ориентированное программирование» | | |
| Выполнили | | |
| студенты гр.23534/7 |  | C.A. Глухов  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  | | |
| Руководитель |  | Ф.М. Иночкин  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| |  | | --- | | Санкт-Петербург | | 2018 | | | |

Оглавление

[Задание 2](#_Toc3910866)

[Код 2](#_Toc3910867)

[Game.cpp 2](#_Toc3910868)

[KeyBuffer.cpp 13](#_Toc3910869)

[Main.cpp 14](#_Toc3910870)

[Pieces.cpp 15](#_Toc3910871)

[Pit.cpp 23](#_Toc3910872)

[Score.cpp 27](#_Toc3910873)

[Скриншоты 27](#_Toc3910874)

# Задание

Создать игру тетрис с использованием графической библиотеки SFML на языке программирования C++.

# Код

## Game.cpp

#include <stdexcept>

#include <iostream>

#include "Game.hpp"

#include "Utility.hpp"

#include "Pieces.hpp"

#include "KeyBuffer.hpp"

//TODO: define canPieceTakeAction()

//TODO: need to unify block locations. First look at line 65 of Pit.cpp

static constexpr float PIECE\_MOVE\_TIME = .5; //seconds

static constexpr float PIT\_WALL\_WIDTH = 5; //how many pixels is the wall of the pit?

static const sf::Vector2f PIT\_OFFSET = sf::Vector2f(275, 45);

static const sf::Vector2f PIECE\_SPAWN\_OFFSET = {PIT\_OFFSET.x + PIT\_WALL\_WIDTH, PIT\_OFFSET.y + PIT\_WALL\_WIDTH};

static const PitCoordinates PIECE\_SPAWN\_COORDINATES = {4,0};

static const sf::Vector2f NEXT\_PIECE\_OFFSET = {PIT\_OFFSET.x - 100, PIT\_OFFSET.y};

static const sf::Vector2f SCORE\_POSITION = {NEXT\_PIECE\_OFFSET.x, NEXT\_PIECE\_OFFSET.y + 100};

static bool isPieceStopped = false;

//prototypes for helper functions

static UPtr<Piece> generateNextPiece(const TextureHolder&);

static bool canPieceMoveDown(const Piece &piece, const Pit &pit);

static bool canPieceMoveLeft(const Piece &piece, const Pit &pit);

static bool canPieceMoveRight(const Piece &piece, const Pit &pit);

static bool canPieceRotateRight(const Piece &piece, const Pit &pit);

static bool canPieceRotateLeft(const Piece &piece, const Pit &pit);

static void dropPiece(Piece& piece, const Pit& pit);

static bool shouldGameEnd(const Piece& piece, const Pit& pit);

Game::Game(const std::string &resourcesPath)

: mIsRunning(true), mIsGameOver(false),

mResourcesPath(resourcesPath), mImagesPath(resourcesPath + "/images/"),

mFontsPath(resourcesPath + "/fonts/") {

loadResources();

initGameState();

}

void Game::initGameState() {

mPit = makeUPtr<Pit>(mTextureHolder.get(TextureID::Pit), PIT\_OFFSET);

auto pitBounds = mPit->boundingRect();

mScore = makeUPtr<Score>(mFontHolder, sf::Vector2f(pitBounds.left + pitBounds.width + 100, NEXT\_PIECE\_OFFSET.y));

mGameOverText = makeUPtr<sf::Text>("\t\t\tGame Over!\nPress Enter to start a new game.", mFontHolder.get(FontID::GameFont));

mGameOverText->setPosition(110,500);

mNextPiece = generateNextPiece(mTextureHolder);

mCurrentPiece = makeUPtr<I>(mTextureHolder, PIECE\_SPAWN\_OFFSET);

mCurrentPiece->moveToCoords(PIECE\_SPAWN\_COORDINATES);

}

void Game::loadResources() {

mTextureHolder.load(TextureID::Pit, mImagesPath + "Pit.png");

mTextureHolder.load(TextureID::I, mImagesPath + "I.png");

mTextureHolder.load(TextureID::J, mImagesPath + "J.png");

mTextureHolder.load(TextureID::L, mImagesPath + "L.png");

mTextureHolder.load(TextureID::O, mImagesPath + "O.png");

mTextureHolder.load(TextureID::S, mImagesPath + "S.png");

mTextureHolder.load(TextureID::Z, mImagesPath + "Z.png");

mTextureHolder.load(TextureID::T, mImagesPath + "T.png");

mFontHolder.load(FontID::GameFont, mFontsPath + "simplistic\_regular.ttf");

}

static bool canPieceMoveDown(const Piece &piece, const Pit &pit) {

Piece tmpPiece = piece;

tmpPiece.moveDown();

for (auto& coords : tmpPiece.coordinatesOfAllBlocks()) {

if (!pit.isBlockInBounds(coords) || pit.isBlockAtCoordinates(coords)) {

return false;

}

}

return true;

}

static bool canPieceMoveLeft(const Piece &piece, const Pit &pit) {

Piece tmpPiece = piece;

tmpPiece.moveLeft();

for (auto& coords : tmpPiece.coordinatesOfAllBlocks()) {

if (!pit.isBlockInBounds(coords) || pit.isBlockAtCoordinates(coords)) {

return false;

}

}

return true;

}

static bool canPieceMoveRight(const Piece &piece, const Pit &pit) {

Piece tmpPiece = piece;

tmpPiece.moveRight();

for (auto& coords : tmpPiece.coordinatesOfAllBlocks()) {

if (!pit.isBlockInBounds(coords) || pit.isBlockAtCoordinates(coords)) {

return false;

}

}

return true;

}

static bool canPieceRotateRight(const Piece &piece, const Pit &pit) {

Piece tmpPiece = piece;

tmpPiece.rotateRight();

for (auto& coords : tmpPiece.coordinatesOfAllBlocks()) {

if (!pit.isBlockInBounds(coords) || pit.isBlockAtCoordinates(coords)) {

return false;

}

}

return true;

}

static bool canPieceRotateLeft(const Piece &piece, const Pit &pit) {

Piece tmpPiece = piece;

tmpPiece.rotateLeft();

for (auto& coords : tmpPiece.coordinatesOfAllBlocks()) {

if (!pit.isBlockInBounds(coords) || pit.isBlockAtCoordinates(coords)) {

return false;

}

}

return true;

}

static void dropPiece(Piece& piece, const Pit& pit) {

while (canPieceMoveDown(piece, pit)) {

piece.moveDown();

}

isPieceStopped = true;

}

void Game::resetGameState() {

initGameState();

}

void Game::handleKeyEvent(const sf::Event &e) {

KeyBuffer::processEvent(e);

if (KeyBuffer::hasKeys()) {

if (!mIsGameOver) {

switch(KeyBuffer::popKey()) {

case sf::Keyboard::Left:

if (canPieceMoveLeft(\*mCurrentPiece, \*mPit)) mCurrentPiece->moveLeft();

break;

case sf::Keyboard::Right:

if (canPieceMoveRight(\*mCurrentPiece, \*mPit)) mCurrentPiece->moveRight();

break;

case sf::Keyboard::Down:

if (canPieceMoveDown(\*mCurrentPiece, \*mPit)) mCurrentPiece->moveDown();

break;

case sf::Keyboard::Space:

dropPiece(\*mCurrentPiece, \*mPit);

break;

case sf::Keyboard::Z:

if (canPieceRotateLeft(\*mCurrentPiece, \*mPit)) mCurrentPiece->rotateLeft();

break;

case sf::Keyboard::X:

if (canPieceRotateRight(\*mCurrentPiece, \*mPit)) mCurrentPiece->rotateRight();

break;

case sf::Keyboard::Escape:

mIsRunning = false;

break;

default:

break;

}

}

else {

switch(KeyBuffer::popKey()) {

case sf::Keyboard::Return:

mIsGameOver = false;

initGameState();

break;

default:

break;

}

}

}

}

bool Game::isRunning() const noexcept {

return mIsRunning;

}

void Game::handleEvent(const sf::Event &e) {

switch (e.type) {

case sf::Event::Closed:

mIsRunning = false;

break;

case sf::Event::KeyPressed:

case sf::Event::KeyReleased:

handleKeyEvent(e);

default:

break;

}

}

static bool shouldGameEnd(const Piece& piece, const Pit& pit) {

for (auto& block : piece.blocks()) {

if (pit.isBlockAtCoordinates(block.coordinates())) {

return false;

}

}

return true;

}

static UPtr<Piece> generateNextPiece(const TextureHolder &textureHolder) {

PieceTypes chosenPiece = static\_cast<PieceTypes>(rand() % NUM\_PIECE\_TYPES);

//TODO change to generate correct peices

switch(chosenPiece) {

case PieceTypes::I:

return makeUPtr<I>(textureHolder, NEXT\_PIECE\_OFFSET);

break;

case PieceTypes::O:

return makeUPtr<O>(textureHolder, NEXT\_PIECE\_OFFSET);

break;

case PieceTypes::J:

return makeUPtr<J>(textureHolder, NEXT\_PIECE\_OFFSET);

break;

case PieceTypes::L:

return makeUPtr<L>(textureHolder, NEXT\_PIECE\_OFFSET);

break;

case PieceTypes::Z:

return makeUPtr<Z>(textureHolder, NEXT\_PIECE\_OFFSET);

break;

case PieceTypes::S:

return makeUPtr<S>(textureHolder, NEXT\_PIECE\_OFFSET);

break;

case PieceTypes::T:

return makeUPtr<T>(textureHolder, NEXT\_PIECE\_OFFSET);

break;

default:

assert("This should be unreachable.");

return nullptr;

}

}

void Game::update(const sf::Time &delta) {

static float secondsSinceLastPieceMove = 0;

secondsSinceLastPieceMove += delta.asSeconds();

if(!mIsGameOver) {

//timer is up, or we are waiting for a new block

if (secondsSinceLastPieceMove > PIECE\_MOVE\_TIME || isPieceStopped) {

secondsSinceLastPieceMove = 0;

if (!isPieceStopped && canPieceMoveDown(\*mCurrentPiece, \*mPit)) {

mCurrentPiece->moveDown();

}

else { //cannot move down anymore.

isPieceStopped = false;

//1. add blocks to pit

for (auto& block : mCurrentPiece->blocks()) {

mPit->putBlock(block);

}

int scoreToAdd = 0;

//2. if there are full lines, we want to delete them and add to score

switch(mPit->deleteFullRows()){

case 1:

scoreToAdd = 40;

break;

case 2:

scoreToAdd = 100;

break;

case 3:

scoreToAdd = 300;

break;

case 4:

scoreToAdd = 1200;

break;

}

mScore->setValue(mScore->value() + scoreToAdd);

//3. if there is room, spawn new piece, else lose

mCurrentPiece = std::move(mNextPiece);

mCurrentPiece->moveToOffset(PIECE\_SPAWN\_OFFSET);

mCurrentPiece->moveToCoords(PIECE\_SPAWN\_COORDINATES);

mNextPiece = generateNextPiece(mTextureHolder);

if (!shouldGameEnd(\*mCurrentPiece, \*mPit)) {

mIsGameOver = true;

}

}

}

}

}

void Game::render(sf::RenderTarget &rt) const {

rt.clear(sf::Color(34,34,34));

rt.draw(\*mPit);

rt.draw(\*mScore);

rt.draw(\*mNextPiece);

rt.draw(\*mCurrentPiece);

if (mIsGameOver) {

rt.draw(\*mGameOverText);

}

}

## KeyBuffer.cpp

#include "KeyBuffer.hpp"

#include <queue>

#include <cassert>

namespace KeyBuffer {

static std::queue<sf::Keyboard::Key> keysReleased;

void processEvent(const sf::Event &e) {

switch(e.type) {

case sf::Event::KeyReleased:

keysReleased.push(e.key.code);

break;

default:

break;

}

}

sf::Keyboard::Key popKey() {

assert(!keysReleased.empty() && "KeyBuffer is empty. This function should not be called");

sf::Keyboard::Key key = keysReleased.front();

keysReleased.pop();

return key;

}

bool hasKeys() {

return !keysReleased.empty();

}

}

## Main.cpp

#include <iostream>

#include <SFML/Graphics.hpp>

#include "Game.hpp"

int main(int argc, char \*\*argv) {

sf::RenderWindow window(sf::VideoMode(800,600), "Tetris");

Game game((argc > 1) ? argv[1] : "../res");

sf::Clock clock;

window.setFramerateLimit(60);

while(game.isRunning()) {

sf::Event e;

while(window.pollEvent(e)) {

game.handleEvent(e);

}

game.update(clock.restart());

window.clear();

game.render(window);

window.display();

}

window.close();

return 0;

}

## Pieces.cpp

#include "Pieces.hpp"

#include "Pit.hpp"

#include <iostream>

static void moveBlocksToNewPositions(std::array<Block,4> &blocks, const std::array<PitCoordinates,4> &rotationPosition, const sf::Vector2f& offset){

for(auto i = 0u; i < blocks.size(); i++) {

//DBGMSG("Block " << i);

//DBGMSG("Block original pos: " << blocks[i]);

const PitCoordinates& blockPosition = rotationPosition[i];

blocks[i].moveAbsolute(blockPosition, offset);

//DBGMSG("Rotation Vector: Row:" << blockPosition.row << " Column: " << blockPosition.column);

//DBGMSG("Block new pos: " << blocks[i]);

}

}

std::ostream& operator<<(std::ostream &os, const Block &block)

{

return os << "X: " << block.getPosition().x << " Y: " << block.getPosition().y;

}

Block::Block(const sf::Texture &texture, const sf::Vector2f &pixelPosition, const PitCoordinates &coords)

: mBlockSprite(texture), mCoordinates(coords)

{

mBlockSprite.setPosition(sf::Vector2f(pixelPosition));

}

void Block::moveAbsolute(const PitCoordinates& newPos, const sf::Vector2f& offset ) {

sf::Vector2f texSize(mBlockSprite.getTextureRect().width,

mBlockSprite.getTextureRect().height);

sf::Vector2f relativePos(newPos.column \* texSize.x, newPos.row \* texSize.y);

mBlockSprite.setPosition(relativePos + offset);

mCoordinates = newPos;

}

void Block::moveDown() {

mCoordinates.row++;

const sf::Vector2f& currentPos = mBlockSprite.getPosition();

mBlockSprite.setPosition(currentPos.x, currentPos.y + mBlockSprite.getTextureRect().height);

}

void Block::draw(sf::RenderTarget &target, sf::RenderStates states) const {

target.draw(mBlockSprite, states);

}

const sf::Vector2f& Block::getPosition() const {

return mBlockSprite.getPosition();

}

const PitCoordinates& Block::coordinates() const {

return mCoordinates;

}

PitCoordinates& Block::coordinates() {

return mCoordinates;

}

//piece methods

Piece::Piece(const Array2D<PitCoordinates,4,4> &rotationPositions, const sf::Texture &blockTexture, const sf::Vector2f &offsetPosition)

:mOffset(offsetPosition), mRotatationPositions(rotationPositions),

mBlocks{{

Block(blockTexture, offsetPosition +

sf::Vector2f(rotationPositions[0][0].column \* blockTexture.getSize().x, rotationPositions[0][0].row \* blockTexture.getSize().y), rotationPositions[0][0]),

Block(blockTexture, offsetPosition +

sf::Vector2f(rotationPositions[0][1].column \* blockTexture.getSize().x, rotationPositions[0][1].row \* blockTexture.getSize().y), rotationPositions[0][1]),

Block(blockTexture, offsetPosition +

sf::Vector2f(rotationPositions[0][2].column \* blockTexture.getSize().x, rotationPositions[0][2].row \* blockTexture.getSize().y), rotationPositions[0][2]),

Block(blockTexture, offsetPosition +

sf::Vector2f(rotationPositions[0][3].column \* blockTexture.getSize().x, rotationPositions[0][3].row \* blockTexture.getSize().y), rotationPositions[0][3]),

}}

{}

void Piece::rotateRight() {

mCurrentRotation++;

mCurrentRotation %= mRotatationPositions.size();

moveBlocksToNewPositions(mBlocks, mRotatationPositions[mCurrentRotation], mOffset);

}

void Piece::rotateLeft() {

mCurrentRotation--;

mCurrentRotation %= mRotatationPositions.size();

moveBlocksToNewPositions(mBlocks, mRotatationPositions[mCurrentRotation], mOffset);

}

void Piece::moveLeft() {

for(auto& blockPositions : mRotatationPositions) {

for(auto& blockPosition : blockPositions) {

blockPosition.column--;

}

}

moveBlocksToNewPositions(mBlocks, mRotatationPositions[mCurrentRotation], mOffset);

}

void Piece::moveRight() {

for(auto& blockPositions : mRotatationPositions) {

for(auto& blockPosition : blockPositions) {

blockPosition.column++;

}

}

moveBlocksToNewPositions(mBlocks, mRotatationPositions[mCurrentRotation], mOffset);

}

void Piece::moveDown() {

for(auto& blockPositions : mRotatationPositions) {

for(auto& blockPosition : blockPositions) {

blockPosition.row++;

}

}

moveBlocksToNewPositions(mBlocks, mRotatationPositions[mCurrentRotation], mOffset);

}

void Piece::moveToOffset(const sf::Vector2f& offset) {

mOffset = offset;

moveBlocksToNewPositions(mBlocks, mRotatationPositions[mCurrentRotation], mOffset);

}

const std::array<Block, 4> Piece::blocks() const {

return mBlocks;

}

void Piece::moveToCoords(const PitCoordinates &coordsOfTopLeftBlock) {

PitCoordinates coordOffset;

coordOffset.column = coordsOfTopLeftBlock.column - mRotatationPositions[mCurrentRotation][0].column;

coordOffset.row = coordsOfTopLeftBlock.row - mRotatationPositions[mCurrentRotation][0].row;

for (auto& rotation : mRotatationPositions) {

for (auto& coords : rotation) {

coords.column += coordOffset.column;

coords.row += coordOffset.row;

}

}

moveBlocksToNewPositions(mBlocks, mRotatationPositions[mCurrentRotation], mOffset);

}

const std::array<PitCoordinates, 4>& Piece::coordinatesOfAllBlocks() const {

return mRotatationPositions[mCurrentRotation];

}

void Piece::draw(sf::RenderTarget &target, sf::RenderStates states) const {

for(auto &block : mBlocks){

target.draw(block, states);

}

}

I::I(const TextureHolder &textures, const sf::Vector2f &pos)

:Piece({{

{{{0,0},{0,1},{0,2},{0,3}}},

{{{1,1},{0,1},{-1,1},{-2,1}}},

{{{0,2},{0,1},{0,0},{0,-1}}},

{{{-1,1},{0,1},{1,1},{2,1}}},

}},

textures.get(TextureID::I), pos)

{}

J::J(const TextureHolder &textures, const sf::Vector2f &pos)

:Piece({{

{{{1,0},{1,1},{1,2},{0,2}}},

{{{2,1},{1,1},{0,1},{0,0}}},

{{{1,2},{1,1},{1,0},{2,0}}},

{{{0,1},{1,1},{2,1},{2,2}}},

}},

textures.get(TextureID::J), pos)

{}

O::O(const TextureHolder &textures, const sf::Vector2f &pos)

:Piece({{

{{{0,0},{1,0},{0,1},{1,1}}},

{{{0,0},{1,0},{0,1},{1,1}}},

{{{0,0},{1,0},{0,1},{1,1}}},

{{{0,0},{1,0},{0,1},{1,1}}},

}},

textures.get(TextureID::O), pos)

{}

L::L(const TextureHolder &textures, const sf::Vector2f &pos)

:Piece({{

{{{0,0},{0,1},{0,2},{1,2}}},

{{{-1,1},{0,1},{1,1},{-1,2}}},

{{{0,0},{0,1},{0,2},{-1,0}}},

{{{-1,1},{0,1},{1,1},{1,0}}},

}},

textures.get(TextureID::L), pos)

{}

Z::Z(const TextureHolder &textures, const sf::Vector2f &pos)

:Piece({{

{{{0,0},{1,0},{1,1},{2,1}}},

{{{1,-1},{1,0},{0,0},{0,1}}},

{{{0,-1},{1,-1},{1,0},{2,0}}},

{{{1,1},{1,0},{2,0},{2,-1}}},

}},

textures.get(TextureID::Z), pos)

{}

S::S(const TextureHolder &textures, const sf::Vector2f &pos)

:Piece({{

{{{2,0},{1,0},{1,1},{0,1}}},

{{{1,1},{1,0},{0,0},{0,-1}}},

{{{2,-1},{1,-1},{1,0},{0,0}}},

{{{2,1},{2,0},{1,0},{1,-1}}},

}},

textures.get(TextureID::S), pos)

{}

T::T(const TextureHolder &textures, const sf::Vector2f &pos)

:Piece({{

{{{0,0},{1,0},{2,0},{1,1}}},

{{{1,-1},{1,0},{1,1},{0,0}}},

{{{0,0},{1,0},{2,0},{1,-1}}},

{{{1,1},{1,0},{1,-1},{2,0}}},

}},

textures.get(TextureID::T), pos)

{}

/\*

case PieceTypes::T:

return makeUPtr<I>(textureHolder, NEXT\_PIECE\_OFFSET);

break;

\*/

## Pit.cpp

#include "Pit.hpp"

#include "Pieces.hpp"

Pit::Pit(const sf::Texture &pitTexture, const sf::Vector2f &pos)

:mPitSprite(pitTexture)

{

mPitSprite.setPosition(pos);

}

void Pit::draw(sf::RenderTarget &target, sf::RenderStates states) const {

target.draw(mPitSprite, states);

for (auto& blockRow : mBlocks) {

for (auto& block : blockRow) {

if (block) {

target.draw(\*block, states);

}

}

}

}

sf::FloatRect Pit::boundingRect() const {

return mPitSprite.getGlobalBounds();

}

bool Pit::isBlockInBounds(const PitCoordinates &coords) const {

return coords.column < Pit::WIDTH && coords.column >= 0 && coords.row >= 0 &&

coords.row < Pit::HEIGHT;

}

bool Pit::isBlockAtCoordinates(const PitCoordinates &coords) const {

return mBlocks.at(coords.row).at(coords.column) != nullptr;

}

static bool isRowFull(const std::array<UPtr<Block>, Pit::WIDTH>& row) {

for (auto& block : row) {

if (!block) {

return false;

}

}

return true;

}

void Pit::putBlock(const Block& block) {

mBlocks.at(block.coordinates().row).at(block.coordinates().column) = makeUPtr<Block>(block);

if (isRowFull(mBlocks[block.coordinates().row])) {

mLocationOfFullRows.push\_back(block.coordinates().row);

}

}

static void deleteRow(int rowNum, Array2D<UPtr<Block>, Pit::HEIGHT, Pit::WIDTH>& blocks) {

//delete row

std::for\_each(blocks[rowNum].begin(), blocks[rowNum].end(), [](UPtr<Block>& block){block.reset();});

//move rows down

for (auto it = blocks.begin() + rowNum; it != blocks.begin(); it--) {

DBGMSG("Before:" << \*it);

\*it = std::move(\*(it - 1));

for (auto& block : \*it) {

if (block) {

block->moveDown();

}

}

DBGMSG("After:" << \*it);

}

DBGMSG("Pit:" << blocks);

}

int Pit::deleteFullRows() {

if (mLocationOfFullRows.size() == 0)

return 0;

int numFullRows = 0;

for (auto& rowNum : mLocationOfFullRows) {

deleteRow(rowNum, mBlocks);

numFullRows++;

}

mLocationOfFullRows.clear();

return numFullRows;

}

## Score.cpp

#include "Score.hpp"

Score::Score(const FontHolder &fontHolder, const sf::Vector2f& position)

:mText("0", fontHolder.get(FontID::GameFont))

{

mText.setPosition(position);

}

int Score::value() const {

return mValue;

}

void Score::setValue(int value) {

mValue = value;

mText.setString(std::to\_string(value));

}

void Score::draw(sf::RenderTarget &target, sf::RenderStates states) const {

target.draw(mText, states);

}

# Скриншоты

