ЛАБОРАТОРНАЯ РАБОТА №5

«ИССЛЕДОВАНИЕ СПОСОБОВ ИНТЕГРАЦИИ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ НА ЯЗЫКЕ QML И ФУНКЦИОНАЛЬНОСТИ НА ЯЗЫКЕ С++»

1. Цель работы

Исследование способов взаимодействия языка С++ и языка разметки QML. Приобретение навыков разработки приложений на основе QML-интерфейса.

2. Постановка задачи

2.1. Изучить способы организации взаимодействия QML и серверных классов на C++ (выполняется в ходе самостоятельной подготовки к лабораторной работе).

2.2. Разработать класс, реализующий функциональность по варианту задания, приведенному в Приложении.

2.3. Определить свойства и методы, необходимые для использования в QML разметке, с помощью соответствующих макросов.

2.4. Добавить класс в контекст Qt Quick приложения.

2.5. Дополнить разметку необходимыми элементами управления с вызовом соответствующих методов.

2.6. Исследовать эффективность работы полученного приложения, имитируя ошибки ввода/вывода.

2.7. Выполнить сравнительный анализ методов построения приложений в данной лабораторной работе и работе №3 по критерию трудоемкости проектирования и программирования.

3. Вариант задания

Вариант задания 2. Добавить две кнопки, по нажатию на первую текущее состояние игрового поля должно сохраняться в файл, по нажатию на вторую – загружаться из файла.

4. Ход работы

Был открыт проект четвертой лабораторной работы. В QML-файл приложения были добавлены две кнопки – save и load (сохранить и загрузить). Измененное приложение показано на рисунке 1.

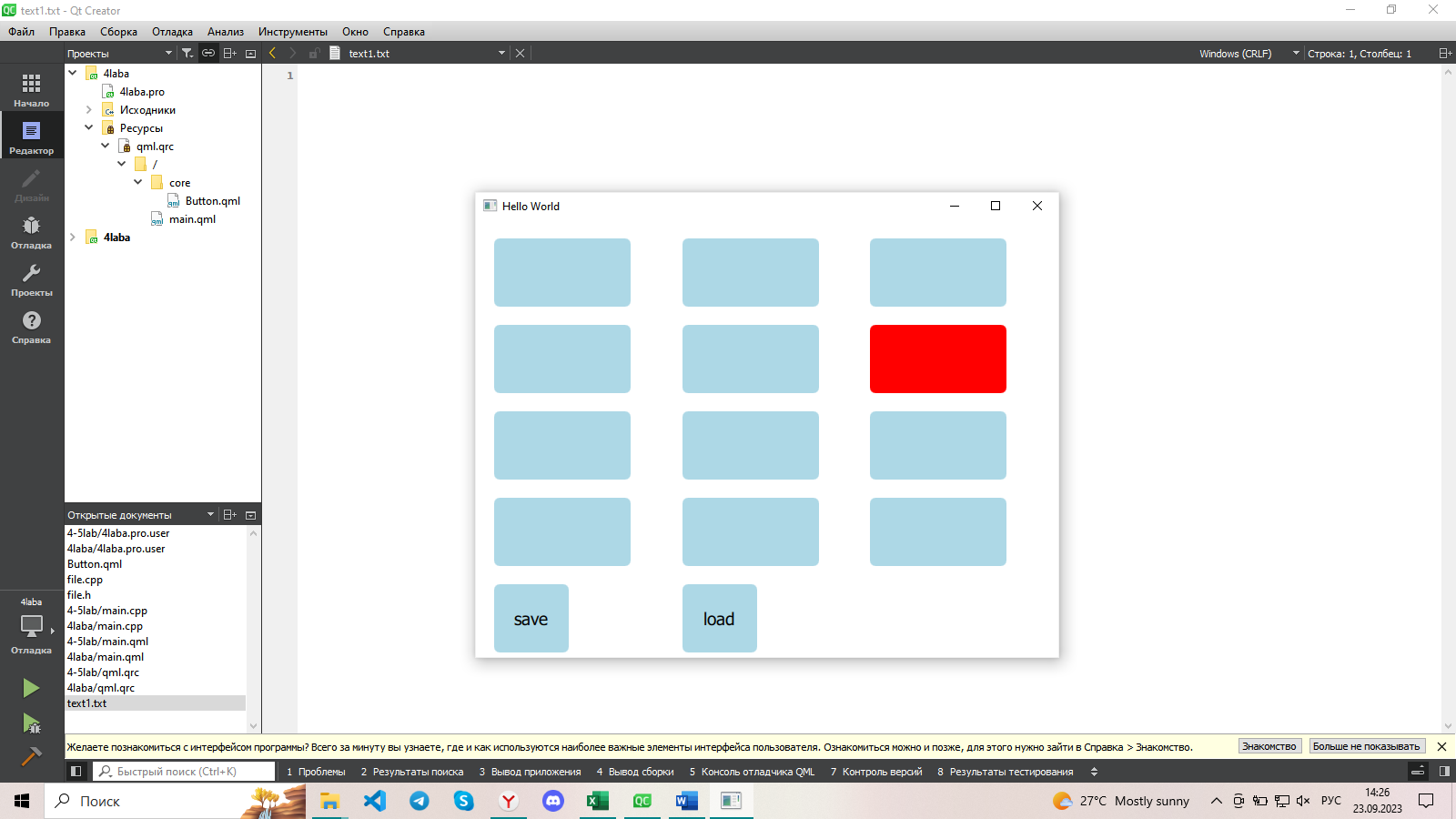


Рисунок 1 – Измененный интерфейс приложения

В проект был добавлен файл класса .h/.cpp и назван «file». В данных файлах описан весь необходимый функционал для сохранения и загрузки состояния игрового поля. В листингах 1-4 представлен измененный код проекта.

Листинг 1 – Файл file.h

#ifndef FILE\_H

#define FILE\_H

#include <QtGui/QGuiApplication>

#include <QFile>

#include <QTextStream>

class **File** : public QObject

{

Q\_OBJECT

public:

**File**(QObject \*parent = 0);

int curResult=0;

Q\_PROPERTY(int curResult READ getState READ loadState WRITE setState)

Q\_INVOKABLE void **saveState**(int state);

Q\_INVOKABLE int **loadState**();

Q\_INVOKABLE int **setState**(int state);

Q\_INVOKABLE int **getState**();

~***File***();

private:

QString FILENAME;

};

#endif // FILE\_H

Листинг 2 – Файл «file.cpp»

#include "file.h"

File::**File**(QObject \*parent):

QObject(parent)

{

this->FILENAME = QString("C:\\Users\\Skory\\Documents\\4laba\\text1.txt");

}

File::~***File***()

{}

void File::**saveState**(int state)

{

QFile file(FILENAME);

//curResult = getResult();

if (file.*open*(QFile::ReadWrite | QIODevice::Truncate)) {

//QTextStream inStream(&file);

//curResult += result.toInt();

QTextStream outStream(&file);

outStream << state;

}

file.*close*();

}

int File::**loadState**()

{

QFile file(FILENAME);

if (file.*open*(QFile::ReadWrite)) {

QTextStream inStream(&file);

curResult = inStream.readAll().toInt();

}

file.*close*();

return curResult;

}

int File::**setState**(int state)

{

curResult = state;

return curResult;

}

int File::**getState**()

{

return curResult;

}

Листинг 3 – Файл main.qml

import QtQuick 2.12

import QtQuick.Window 2.12

import QtQuick.Layouts 1.0

import "core"

Window {

width: 640

height: 480

visible: true

title: *qsTr*("Hello World")

function *buttonClicked*(curentButton)

{

var *buttons* = [*button0*,*button1*,*button2*,*button3*,*button4*,*button5*,*button6*,*button7*,*button8*,*button9*,*button10*,*button11*];

if(*curentButton*.color == "#ff0000"){

*curentButton*.color = "lightblue";

var *index* = *Math*.floor(*buttons*.length \* *Math*.random());

var *btn* = *buttons*[*index*];

*btn*.color = "#ff0000";

//appClass.saveState(index);

*appClass*.setState(*index*);

}

}

function *restoreLastSaving*(index)

{

var *buttons* = [*button0*,*button1*,*button2*,*button3*,*button4*,*button5*,*button6*,*button7*,*button8*,*button9*,*button10*,*button11*];

for(let *i*=0;*i*<*buttons*.length;*i*++){

var *btn* = *buttons*[*i*];

if(*i*==*index*){

*btn*.color = "#ff0000";

}else{

*btn*.color = "lightblue";

}

}

}

GridLayout {

columns: 3

anchors.fill: *parent*

anchors.margins: 20

rowSpacing: 20

columnSpacing: 20

flow: *width* > *height* ? GridLayout.LeftToRight : GridLayout.TopToBottom

Button {

id: *button0*

x: 19

y: 68

color:"#ff0000"

onButtonClick:

{

*buttonClicked*(*button0*);

}

}

Button {

id: *button1*

x: 177

y: 68

onButtonClick:

{

*buttonClicked*(*button1*);

}

}

Button {

id: *button2*

x: 334

y: 68

onButtonClick:

{

*buttonClicked*(*button2*);

}

}

Button {

id: *button3*

x: 19

y: 171

onButtonClick:

{

*buttonClicked*(*button3*);

}

}

Button {

id: *button4*

x: 177

y: 171

onButtonClick:

{

*buttonClicked*(*button4*);

}

}

Button {

id: *button5*

x: 334

y: 171

onButtonClick:

{

*buttonClicked*(*button5*);

}

}

Button {

id: *button6*

x: 19

y: 273

onButtonClick:

{

*buttonClicked*(*button6*);

}

}

Button {

id: *button7*

x: 177

y: 273

onButtonClick:

{

*buttonClicked*(*button7*);

}

}

Button {

id: *button8*

x: 334

y: 273

onButtonClick:

{

*buttonClicked*(*button8*);

}

}

Button {

id: *button9*

x: 19

y: 375

onButtonClick:

{

*buttonClicked*(*button9*);

}

}

Button {

id: *button10*

x: 177

y: 375

onButtonClick:

{

*buttonClicked*(*button10*);

}

}

Button {

id: *button11*

x: 334

y: 375

onButtonClick:

{

*buttonClicked*(*button11*);

}

}

Button {

id: *saveButton*

x: 531

y: 68

width: 82

height: 75

label: "save"

onButtonClick:

{

*appClass*.saveState(*appClass*.getState());

}

}

Button {

id: *loadButton*

x: 531

y: 171

width: 82

height: 75

label: "load"

onButtonClick:

{

*restoreLastSaving*(*appClass*.loadState());

}

}

}

}

Листинг 4 – Файл main.cpp

#include <QGuiApplication>

#include <QQmlApplicationEngine>

#include <QQmlContext>

#include "file.h"

int main(int argc, char \*argv[])

{

#if QT\_VERSION < QT\_VERSION\_CHECK(6, 0, 0)

QCoreApplication::setAttribute(Qt::AA\_EnableHighDpiScaling);

#endif

QGuiApplication app(*argc*, argv);

File appClass;

QQmlApplicationEngine engine;

const QUrl url(QStringLiteral("qrc:/main.qml"));

QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,

&app, [url](QObject \*obj, const QUrl &objUrl) {

if (!obj && url == objUrl)

QCoreApplication::exit(-1);

}, Qt::QueuedConnection);

engine.load(url);

QQmlContext\* rootContext = engine.rootContext();

rootContext->setContextProperty("appClass",&appClass);

return app.exec();

}

ВЫВОДЫ

В ходе лабораторной работы было доработано приложение, написанное в предыдущей работе, а именно были добавлены кнопки «Сохранить» и «Загрузить». Была добавлена реализация логики работы кнопок с помощью серверного класса и определенных в нем свойств и методов с использованием соответствующих макросов.

Приложение было протестировано, все ошибки исправлены.

Контрольные вопросы:

5.1. Что понимается под макросом в C++?

В C++ макросы - это инструкции препроцессора, которые позволяют вставлять или изменять текст программы до ее компиляции. Макросы обычно используются для автоматизации и упрощения написания кода, объявления констант, определения условной компиляции и других задач.

5.2. Что необходимо сделать, чтобы обеспечить QML доступ к методам и свойствам серверных классов?

Объявить серверный класс наследником от QObject.

Использовать макрос Q\_OBJECT в определении класса для поддержки системы метаобъектов Qt.

Объявить методы и свойства класса с использованием соответствующих макросов, таких как Q\_INVOKABLE и Q\_PROPERTY.

5.3. Какие макросы используются для предоставления QML доступа к свойствам серверного класса?

Для предоставления QML доступа к свойствам серверного класса используется макрос Q\_PROPERTY.

5.4. Какие макросы используются для предоставления QML доступа к методам серверного класса?

Для предоставления QML доступа к методам серверного класса используется макрос Q\_INVOKABLE.

5.5. Назовите основные параметры макроса Q\_PROPERTY. Для чего они используются?

Тип свойства: определяет тип данных свойства.

Имя свойства: задает имя свойства, по которому оно будет доступно в QML.

Чтение (READ) и запись (WRITE): указывают методы класса, которые будут использоваться для чтения и записи значения свойства.

Уведомление (NOTIFY): указывает метод класса, который будет вызываться при изменении значения свойства.

Макрос Q\_PROPERTY используется для создания связи между свойствами серверного класса и их представлением в QML

5.6. Какие параметры имеет макрос Q\_INVOKABLE?

Макрос Q\_INVOKABLE не имеет параметров. Он просто помечает метод класса как доступный для вызова из QML.

5.7. Как добавить объект класса в контекст Qt Quick приложения?

Создать экземпляр класса, который требуется добавить.

Получить указатель на корневой контекст (QQuickView или QQmlApplicationEngine).

Вызвать метод setContextProperty() на корневом контексте, передав имя, под которым объект будет доступен в QML, и указатель на созданный экземпляр класса.

Пример:

QObject\* myObject = new MyObject();

engine.rootContext()->setContextProperty("myObject", myObject);

Теперь объект класса MyObject будет доступен в QML по имени myObject.

5.8. Какие действия нужно выполнить для передачи данных из QMLинтерфейса приложения на серверную часть?

5.9. Для чего необходимо взаимодействие QML с серверными классами, почему нельзя реализовать всю функциональность приложения в рамках QML?

5.8. Для передачи данных из QML-интерфейса приложения на серверную часть необходимо выполнить следующие действия:

Определить свойство или сигнал в QML-интерфейсе.

Привязать свойство или сигнал к серверному классу, используя соответствующие макросы, например, Q\_PROPERTY или Q\_SIGNAL.

Реализовать логику обработки свойств или сигналов в серверном классе.

При изменении свойства в QML или вызове сигнала, связанного с серверным классом, будет выполнена соответствующая логика на сервере.

5.9. Взаимодействие QML с серверными классами необходимо для реализации более сложной бизнес-логики, обработки данных, выполнения операций, которые лучше подходят для реализации на стороне C++ или требуют доступа к низкоуровневым функциям или библиотекам. Некоторые причины, почему нельзя реализовать всю функциональность приложения в рамках QML, включают ограничения QML в отношении работы с низкоуровневыми операциями, сложные вычисления, доступ к системным ресурсам, многопоточность и другие аспекты, которые могут быть лучше контролируемыми или эффективно реализованными с использованием C++. Кроме того, разделение логики между QML и C++ позволяет лучше организовать код, повысить его читаемость и облегчить сопровождение и расширение приложения в будущем.