Санкт-Петербургский политехнический университет
Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Программирование на языках высокого уровня
Тема: Создание Desktop приложения на C++ с использованием Qt

Выполнил студент гр. 3331506/10401 **Суднеко В.П.**

Преподаватель **Ананьевский М.С.**

«___»_____ 2024 г.

Санкт-Петербург
2024

Содержание

1 Формулировка задачи

В компании Геоскан разрабатывается продукт для учебных учреждений, позволяющий тестировать навыки управления беспилотным летательным аппаратом не с реальными дронами, а с их симуляциями. Этот продукт получил название "Арена"и содержит в себе различные игровые механики. В нём две или более команд выполняют задачу по набору наибольшего количества очков. Для этого помимо квадрокоптеров в игре присутствуют такие элементы как "Стартовая позиция "Фабрика "Огонь"и так далее. Все эти элементы могут взаимодействовать с дроном и как добавлять ему очки, так и лишать их.

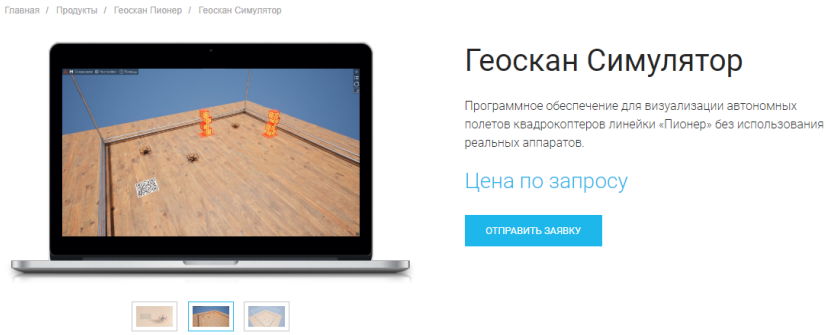

Рис. 1: Симулятор автономных полетов квадрокоптеров линейки "Пионер"

Для этого симулятора есть как собственно 3D часть, написанная на Unreal Engine, так и серверная. Сервер управляет ареной. Команды серверу дает графическая оболочка. Изначально она написана на PyQT и работает довольно медленно, при этом размер этой программы очень большой. Задача заключается в том, чтобы полностью переписать её реализацию на C++ Qt. Собственно, об этом далее.

2 Архитектура

2.1 Микрофронтенд

Архитектура, при которой каждый модуль максимально независим от других и взаимодействует со своей частью графического интерфейса, называется микрофронтенд (Micro-Frontend).

Микрофронтенды заимствуют идеи микросервисной архитектуры, применяя их к фронтенд-разработке. В такой архитектуре приложение делится на независимые модули или компоненты, каждый из которых отвечает за свою часть пользовательского интерфейса и функционирует автономно. Эти модули могут быть разработаны, развернуты и обновлены независимо друг от друга, что упрощает управление сложными приложениями и улучшает масштабируемость и гибкость разработки.

Ключевые характеристики микрофронтендов:

- Независимость модулей: Каждый модуль может быть разработан и развернут отдельно.
- Ясное разграничение ответственности: Модули отвечают за свои конкретные функции и части интерфейса.
- Разнообразие технологий: Модули могут быть написаны на разных языках программирования и использовать разные фреймворки.
- Изоляция данных и состояния: Каждый модуль управляет своим состоянием и данными, минимизируя зависимость от других модулей.
- Такой подход позволяет легче масштабировать и развивать приложения, а также упрощает работу с командами, работающими над разными частями одного проекта.

Именно такой подход был желателен при реализации этого проекта.

2.2 Код на Qt в код на Vue.js

Однако накладывалась специфика c++ и в общем всё свелось к попыткам превратить код на c++ в код на JavaScript, иначе говоря, код на Qt в код на React или Vue.js.

Рис. 2: Общий принцип формирования проекта

Это связано с тем, что до текущей задачи все мои проекты были написаны на JavScript с использованием фреймворков React и Vue. Каждый из них основан на реактивном изменении страницы.

Основной принцип Vue.js заключается в реактивном обновлении пользовательского интерфейса. Это означает, что Vue.js автоматически отслеживает изменения данных и обновляет соответствующие части DOM, чтобы отобразить эти изменения. Вот несколько ключевых аспектов этого принципа:

Декларативное связывание данных (Data Binding): Vue.js использует директивы (например, v-bind, v-model, v-for, v-if), чтобы связывать данные и DOM. Это позволяет разработчикам описывать, как интерфейс должен выглядеть в зависимости от состояния данных, а Vue.js берет на себя обновление DOM, когда данные изменяются.

Реактивность (Reactivity): В Vue.js данные являются реактивными объектами. Когда данные изменяются, Vue.js автоматически отслеживает эти изменения и обновляет DOM-элементы, которые зависят от измененных данных.

Компонентный подход (Component-Based Architecture): Vue.js позволяет создавать приложения, разбивая их на переиспользуемые компоненты. Каждый компонент инкапсулирует свою логику и шаблон, что делает код более организованным и модульным.

Директивы и шаблоны (Directives and Templates): Vue.js использует шаблоны на основе HTML с расширениями, такими как директивы, которые позволяют описывать разметку декларативно. Директивы — это специальные атрибуты с префиксом v-, которые предоставляют функции, управляющие рендерингом.

Простота интеграции (Ease of Integration): Vue.js можно использовать как для создания небольших виджетов, так и для создания полноценных одно-

страничных приложений (SPA). Он легко интегрируется в существующие проекты благодаря своему гибкому дизайну.

Основной принцип Vue.js — это сделать разработку пользовательских интерфейсов интуитивно понятной и удобной за счет реактивного обновления данных и декларативного связывания данных с DOM.

Реализация хотя бы отчасти похожей структуры как одно из требований, которое было поставлено задаче.

2.3 Архитектура классов

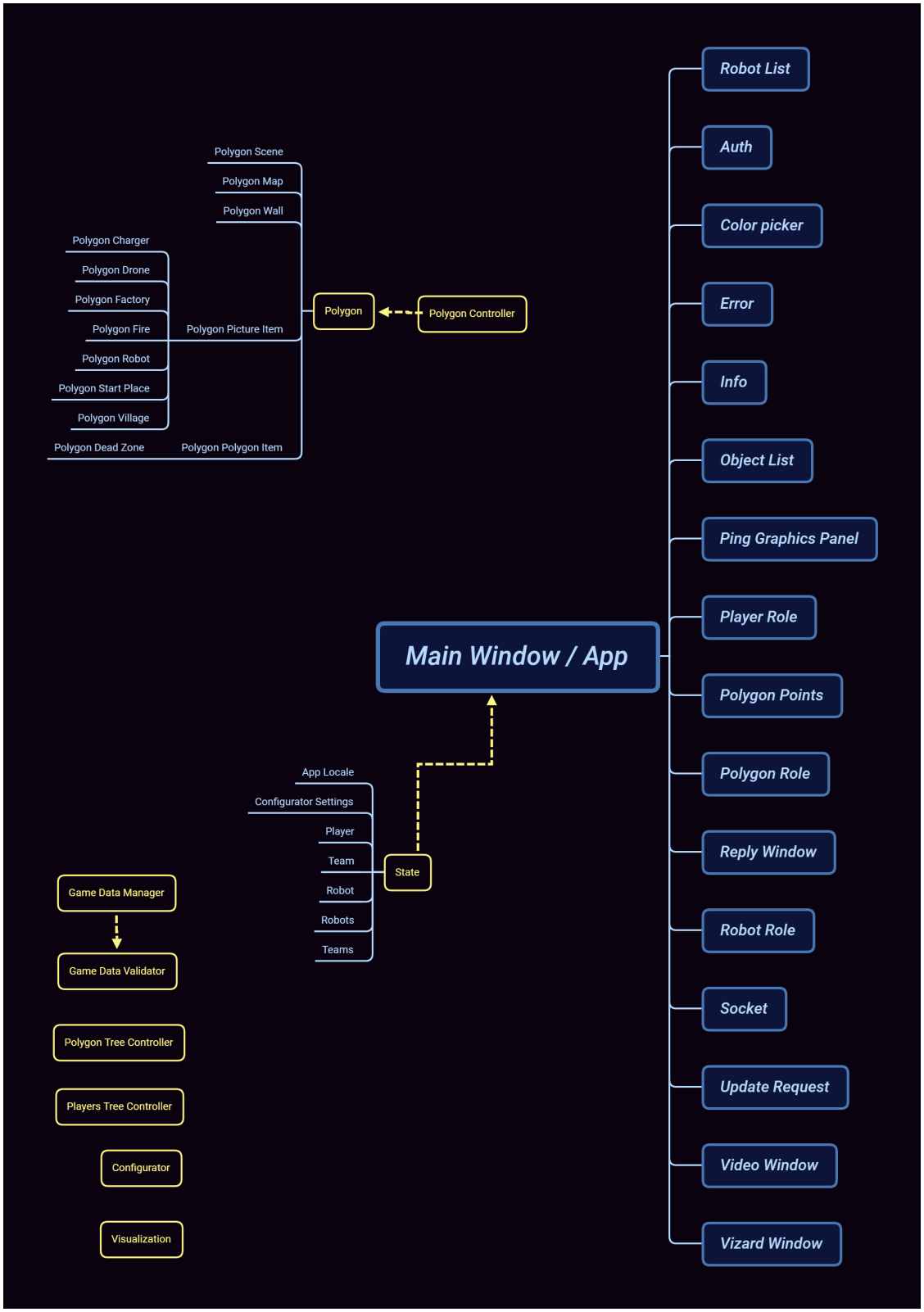

Рис. 3: Архитектура приложения

2.4 Структура проекта

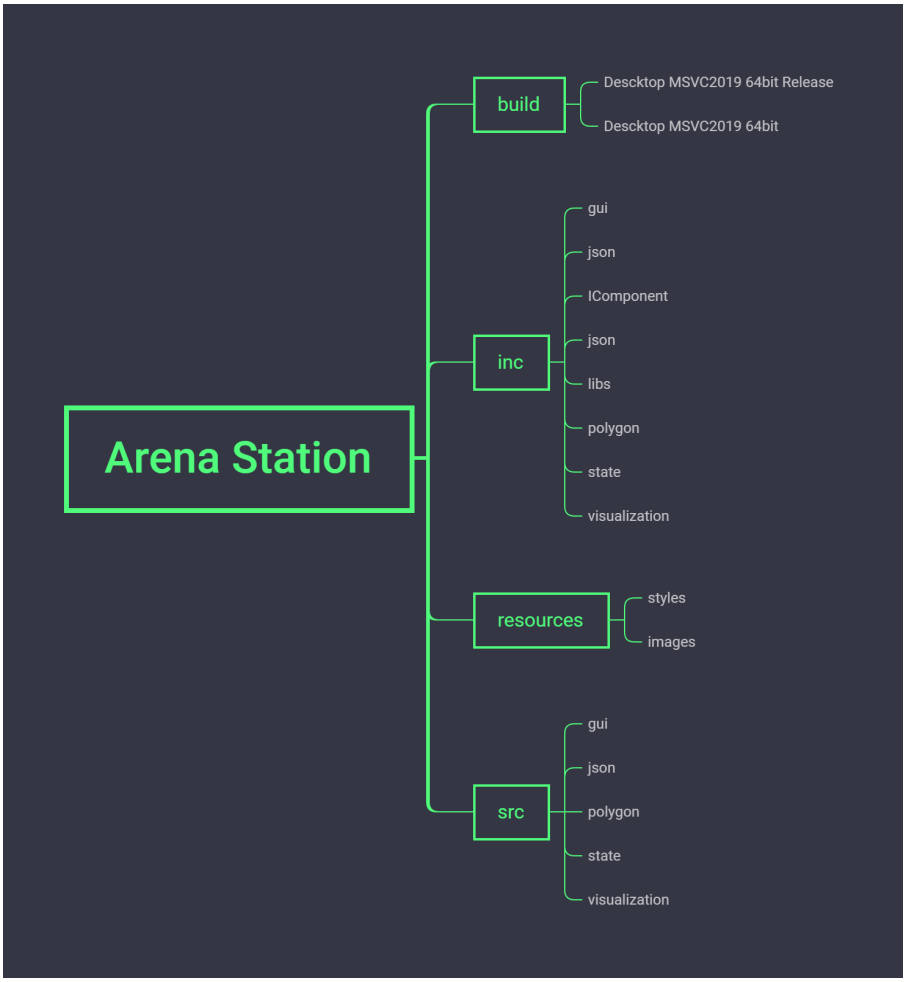

Рис. 4: Структура проекта Arena Station

Проект разделен на 4 папки. В папке build содержатся сборки проекта релизная и для отладки. В папке inc заголовочные файлы. В папке resources стили и используемые изображения. В папке src исходный код.

3 Дизайн

Приложение на текущий момент состоит из главного окна, нескольких вспомогательных окон, нескольких контроллеров data flow и нескольких классов, представляющих из себя локальное хранилище. Его структура будет обсуждена далее.

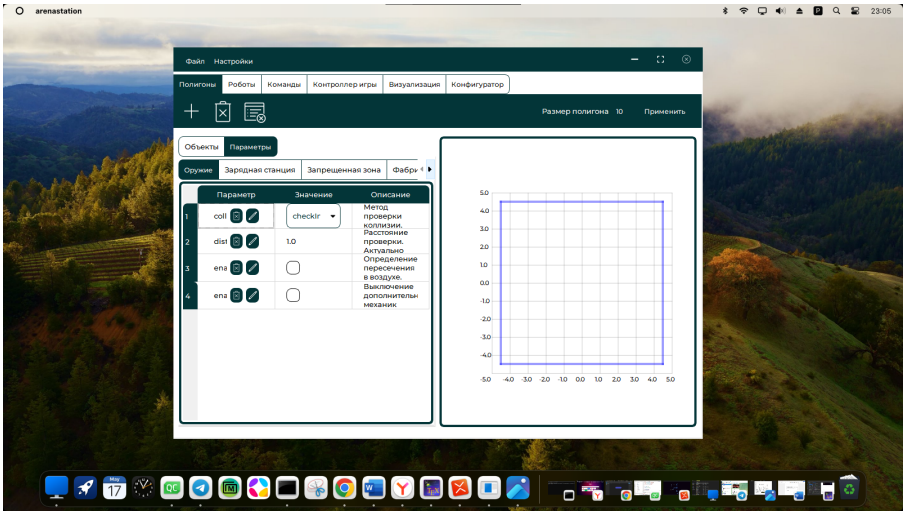

Рис. 5: Общий вид главного окна приложения

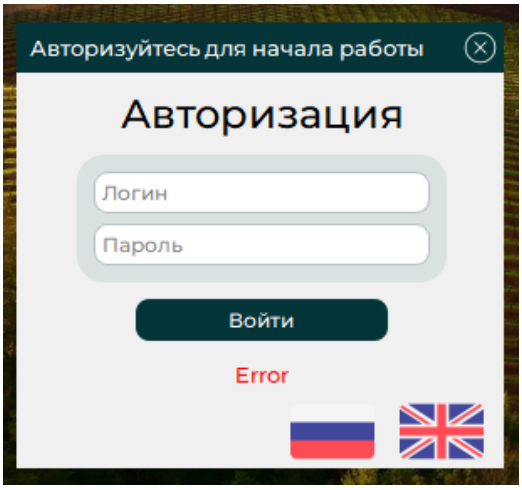

Рис. 6: Окно авторизации

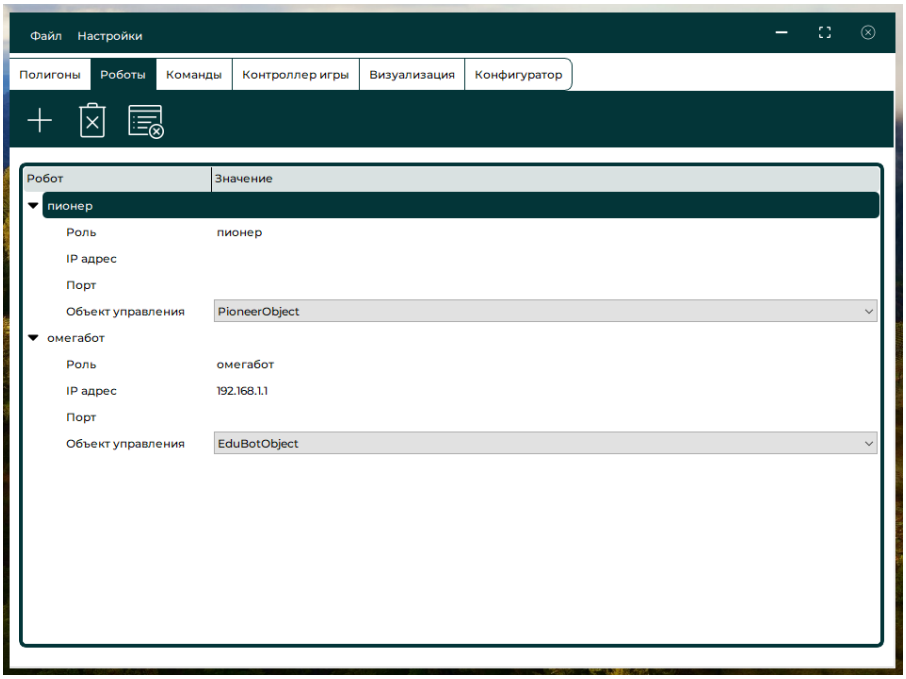

Рис. 7: Вкладка "Роботы"

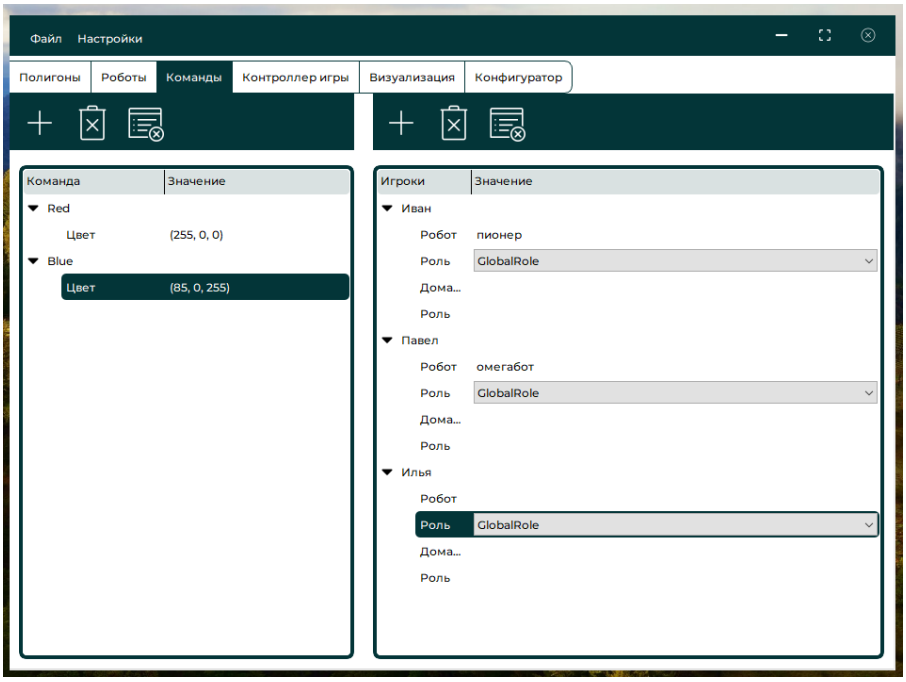

Рис. 8: Вкладка "Команды"

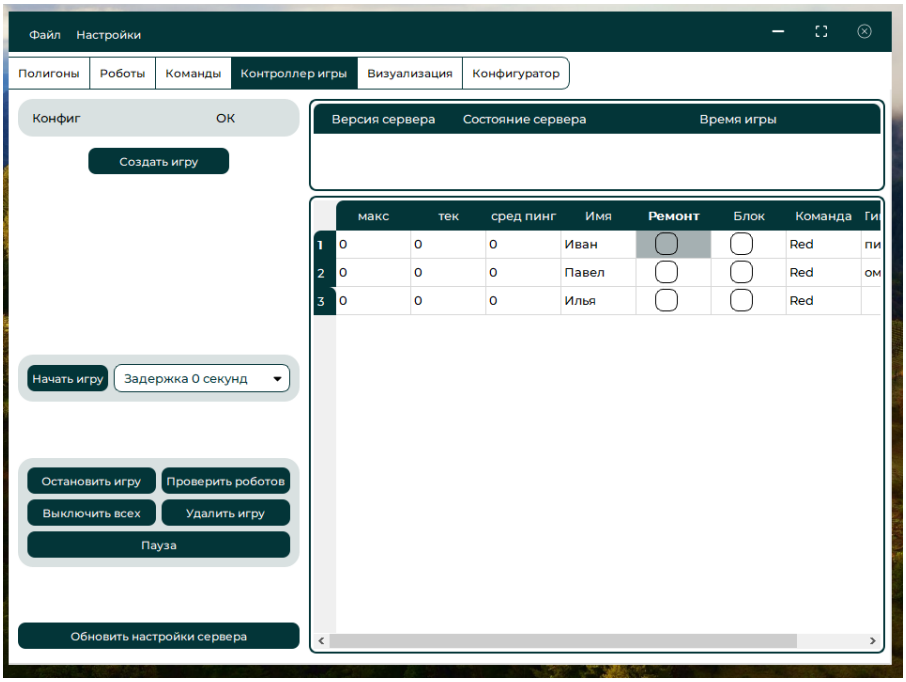

Рис. 9: Вкладка "Контроллер игры"

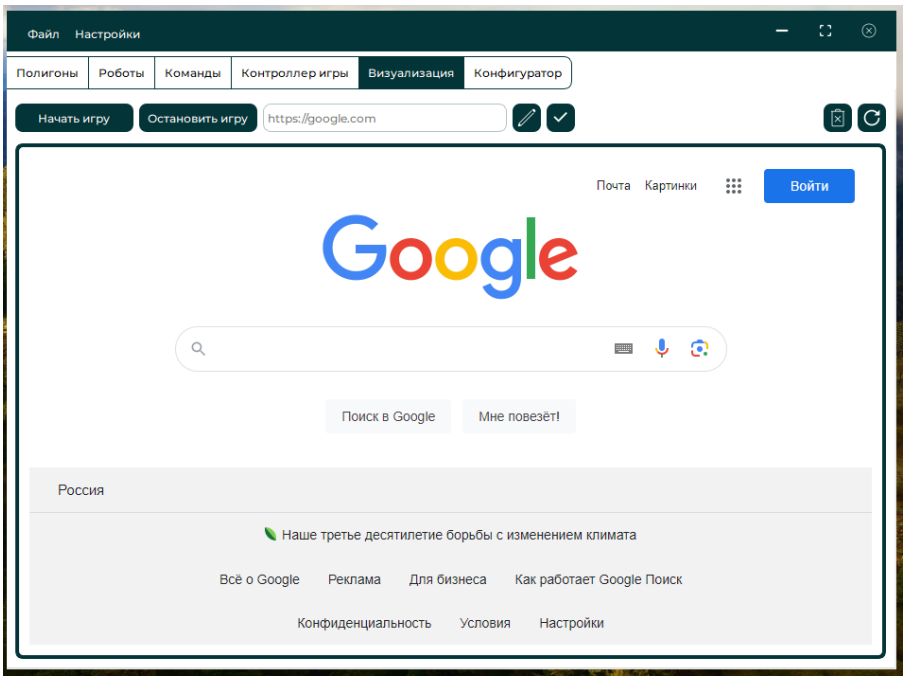

Рис. 10: Вкладка "Визуализация"

На текущий момент в качестве placeholder-а в этой вкладке используется сайт google.com исключительно для отображения функциональности вкладки. В реальной работе в ней отображена 2D визуализация игры, запускаемая на сервере. Подробнее об этой функциональности будет сказано ниже.

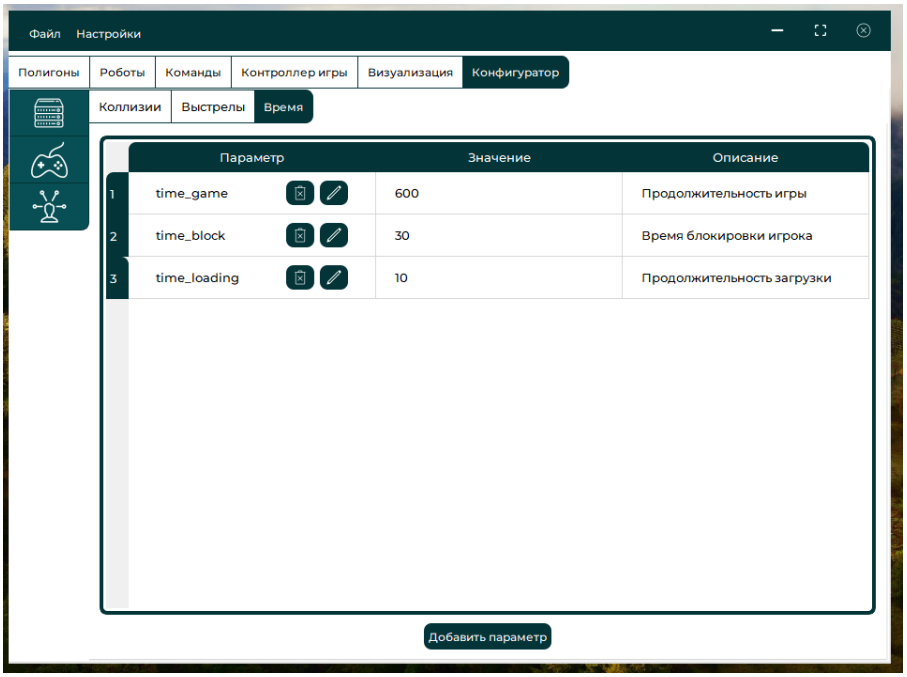

Рис. 11: Вкладка "Конфигуратор"

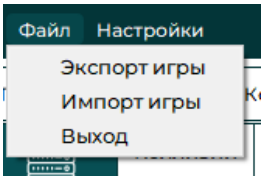

Рис. 12: Содержимое меню "Файл"

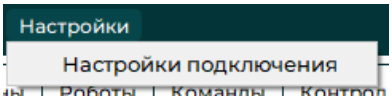

Рис. 13: Содержимое меню "Настройки"

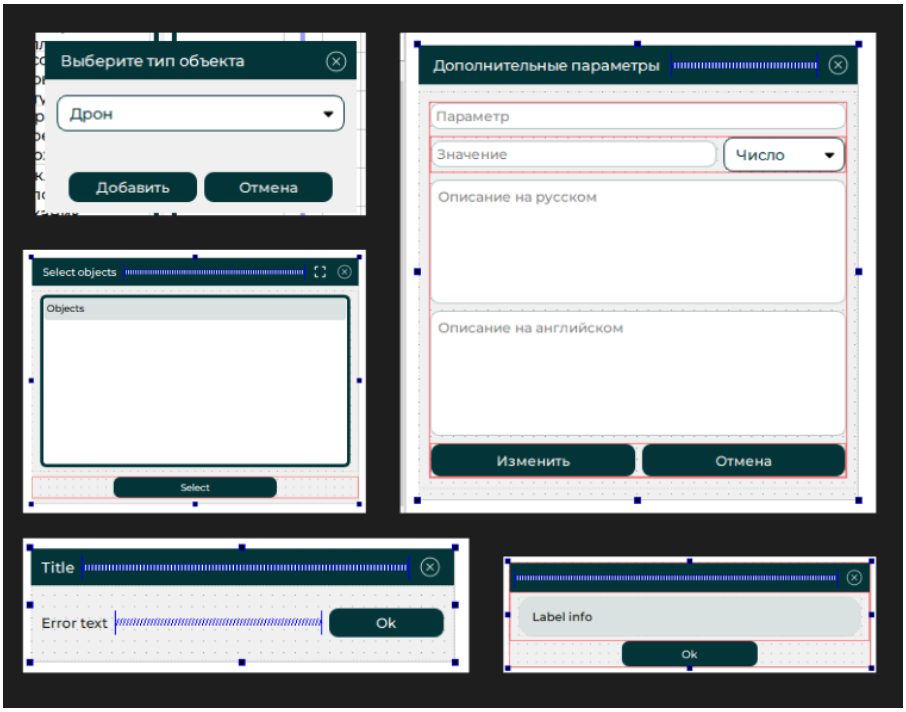

Рис. 14: Примеры других отображаемых окон

4 Код

4.1 Qt

Qt — это мощный кросс-платформенный фреймворк на языке C++ для разработки графических пользовательских интерфейсов (GUI) и приложений, работающих на различных операционных системах, включая Windows, macOS, Linux, iOS и Android. Вот краткий обзор ключевых аспектов Qt:

Qt позволяет писать код один раз и запускать его на разных платформах без необходимости значительных изменений. Это достигается благодаря абстракции платформенных API.

Богатый набор виджетов и графических элементов: Qt предоставляет множество готовых виджетов для создания пользовательских интерфейсов, включая кнопки, текстовые поля, списки, таблицы и многое другое. Также поддерживается создание собственных виджетов.

Система сигналов и слотов: Одна из ключевых особенностей Qt — это механизм сигналов и слотов, который используется для связи между объектами. Сигналы и слоты обеспечивают безопасный и удобный способ обработки событий и взаимодействия компонентов.

```
QObject::connect(senderObject, &SenderObject::signal,
 receiverObject, ReceiverObject::slot);
```

Модульность: Qt состоит из множества модулей, таких как QtCore, QtGui, QtWidgets, QtNetwork, QtMultimedia и других, что позволяет использовать только те компоненты, которые нужны для конкретного проекта.

Поддержка графики и анимации: Qt включает мощные возможности для работы с 2D и 3D графикой, анимациями и эффектами. Модуль QtQuick и язык QML позволяют создавать современные и динамичные интерфейсы.

Интеграция с C++: Qt написан на C++ и предоставляет обширные возможности для интеграции с C++ кодом. Это делает Qt мощным инструментом для разработки производительных приложений.

Инструменты разработки: Qt Creator — это интегрированная среда разработки (IDE), специально созданная для работы с Qt. Она поддерживает создание и редактирование проектов, отладку, анализ кода и многое другое.

4.2 Система сборки

CMake (Cross-platform Make) — это кроссплатформенная система автоматизации сборки программного обеспечения, которая используется для управ-

ления процессом сборки приложений на различных платформах. Она позволяет создавать независимые от платформы файлы для сборки проекта и автоматически генерировать скрипты для различных сред разработки и компиляторов.

Основные особенности CMake:

Кроссплатформенность: CMake позволяет писать единые скрипты сборки, которые могут использоваться на различных операционных системах (Windows, macOS, Linux и другие).

Многоязыковая поддержка: CMake поддерживает сборку проектов на различных языках программирования, включая C, C++, Python, Java и многие другие.

Модульность: Сборка проектов в CMake осуществляется с использованием модулей, что позволяет легко добавлять и управлять зависимостями и компонентами проекта.

Генерация файлов сборки: CMake генерирует файлы сборки для различных систем сборки, таких как Makefiles для Unix-подобных систем или проекты для сред разработки, таких как Visual Studio, Xcode и Eclipse.

Простота использования: CMake использует простой и интуитивно понятный синтаксис для описания процесса сборки, что делает его доступным для широкого круга разработчиков.

Для данного проекта содержимое файла CmakeLists.txt следующее:

```
cmake_minimum_required(VERSION 3.5)

project(arenastation VERSION 0.1 LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

include_directories(resources)
include_directories(inc)
include_directories(inc/IComponent)
include_directories(inc/gui)
include_directories(inc/libs)
include_directories(inc/polygon)
include_directories(inc/visualization)


find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets LinguistTools)
```

```
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets LinguistTools)
find_package(Qt6 REQUIRED COMPONENTS WebEngineWidgets)
find_package(Qt6 REQUIRED COMPONENTS OpenGLWidgets)

set(TS_FILES
arenastation_en_US.ts
arenastation_ru_RU.ts
)
file(GLOB_RECURSE HEADERS inc/*)
file(GLOB_RECURSE SRC src/*)
file(GLOB_RECURSE RESOURCES resources/*)

qt_create_translation(QM_FILES ${CMAKE_SOURCE_DIR} ${TS_FILES})

set(PROJECT_SOURCES
main.cpp
${HEADERS}
${SRC}
${RESOURCES}
${QM_FILES}
${TS_FILES}
)

qt_add_executable(arenastation
MANUAL_FINALIZATION
${PROJECT_SOURCES}
)



target_link_libraries(arenastation PRIVATE Qt${QT_VERSION_MAJOR}::Widgets
    Qt${QT_VERSION_MAJOR}::WebEngineWidgets)
target_link_libraries(arenastation PRIVATE Qt6::OpenGLWidgets)

include(GNUInstallDirs)
install(TARGETS arenastation
BUNDLE DESTINATION .
LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)

if(QT_VERSION_MAJOR EQUAL 6)
qt_finalize_executable(arenastation)
endif()
configure_file(${CMAKE_CURRENT_SOURCE_DIR}/resources/schema.json
${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/data COPYONLY)
configure_file(${CMAKE_CURRENT_SOURCE_DIR}/resources
    /default_game_settings.json
```

```
67 ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/data COPYONLY)
```

Минимальная версия CMake — 3.5. Используется стандарт C++ 2017 года. Далее подключаются папки с заголовочными файлами и дополнительные библиотеки. Для упрощения работы с проектом применяется регулярное выражение, которое находит все вложенные файлы и добавляет их в проект. Следует строка для внедрения перевода, оставшиеся настройки и перенос статических файлов с сохраненными версиями игры и схемой валидации в папки проекта.

4.3 Работа с виджетами

Работа с виджетами реализована в двух формах. Как посредством Qt Designer-a, так и их ручным проектированием.

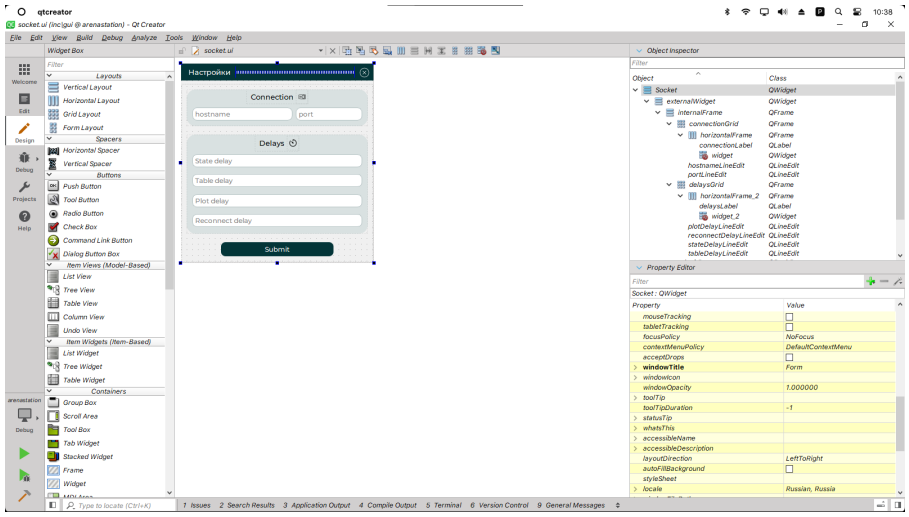

Рис. 15: Создание виджетов через Qt Designer

Пример ручного проектирования виджетов.

```
1  QWidget *titleBarBack = new QWidget();
2  QHBoxLayout *titleBarLayout = new QHBoxLayout(titleBarBack);
3  titleBarBack->setLayout(titleBarLayout);
4  titleBarBack->setStyleSheet("* {background: #033538; margin-bottom: 5px;}");
5
6  titleBarLayout->addWidget(menuBar);
7  titleBarLayout->addItem(new QSpacerItem(menuBar->geometry().width(), 1));
8  titleBarLayout->addWidget(minimize_button);
9  titleBarLayout->addWidget(maximize_button);
10 titleBarLayout->addWidget(close_button);
```

```
11
12 ui->titleBar->addWidget(titleBarBack);
```

4.4 Сигналы и слоты

В Qt, сигналы и слоты представляют собой механизм связи между объектами, который позволяет объектам общаться друг с другом, передавая сигналы и обрабатывая их с помощью слотов. Это ключевая часть системы событий в Qt.

Сигналы — это методы, которые используются для извещения других объектов о том, что произошло какое-то событие. Сигналы объявляются с помощью ключевого слова signals: в классе и вызываются, когда необходимо передать информацию о событии.

Слоты — это методы, которые могут быть подключены к сигналам и будут вызваны в ответ на соответствующий сигнал. Слоты объявляются с помощью ключевого слова slots: в классе.

Механизм слотов и сигналов не является частью c++ и представляет собой расширение функциональности через фреймворк. Вся обработка событий в проекте построена на применении этого принципа. Например, как здесь.

```
1  signals:
2  void onItemCountChanged(QTreeWidgetItem *item, int column);
3
4  private slots:
5  void on_maximize_button_clicked();
6
7  void on_close_button_clicked();
8
9  void on_minimize_button_clicked();
10
11 void on_exit_game_button_clicked();
12
13 void on_import_game_button_clicked();
14
15 void on_game_connect_settings_button_clicked();
```

В коде выше показано, как добавляются сигналы и слоты к классам c++. Ниже показан пример действий при их срабатывании.

```
1  //object position changed in tree
2  connect(this, &PolygonController::somethingChangedInTree,
3  [this](QTreeWidgetItem *item, int column){
```

```
        if (item->text(0)==tr("Позиция") && item->parent()->child(0)->text(0)
        ↪  != tr("Запрещенная зона")) {
                int id = polygonTree->indexOfTopLevelItem(item->parent());
                auto [x, y] = getCoordinates(item->text(1));
                emit objectPositionChangedInTree(id,
                x*polygonScene->getStepSize(),
                y*polygonScene->getStepSize()*-1);
        } else if (item->text(0)==tr("Позиция")) {
                int id = polygonTree->indexOfTopLevelItem(item->parent());
                QString position = item->text(1);
                emit polygonPositionChangedInTree(id, position);
        }
});
```

4.5 Реализация поведения окна

Поскольку в проекте не использовалось стандартное оконное окружение Windows, то в нём пришлось дополнительно реализовывать поведение окна при перетаскивании и масштабировании.

Покажем это на примере одного из вспомогательных окон - для выбора объекта, добавляемого на полигон.

Заголовочный файл:

```
#ifndef POLYGON_ROLE_H
#define POLYGON_ROLE_H

#include <QWidget>
#include <QtWidgets>
#include <QtCore>

namespace Ui {
        class PolygonRole;
}

class PolygonRole : public QWidget
{
        Q_OBJECT

        public:
        PolygonRole(QWidget *parent = nullptr);
        PolygonRole(QMainWindow* main_window, QWidget *parent = nullptr);
        ~PolygonRole();

        private:
```

```
22          void hideWindowFrame();
23
24          protected:
25          virtual void mousePressEvent(QMouseEvent* event) override;
26          virtual void mouseMoveEvent(QMouseEvent* event) override;
27
28          signals:
29          void send_role(QString role_name);
30
31          private slots:
32          void on_cancelButton_clicked();
33
34          void on_closeButton_clicked();
35
36          void on_selectButton_clicked();
37
38          private:
39
40          Ui::PolygonRole *ui;
41
42          QPoint m_lastClickPoint;
43  };
44
45  #endif // POLYGON_ROLE_H
46
```

Исходный код:

```
1   #include "polygon_role.h"
2   #include "../../inc/gui/ui_polygon_role.h"
3
4   PolygonRole::PolygonRole(QWidget *parent)
5   : QWidget(parent)
6   , ui(new Ui::PolygonRole)
7   {
8           ui->setupUi(this);
9           hideWindowFrame();
10  }
11
12  PolygonRole::PolygonRole(QMainWindow* main_window, QWidget *parent)
13  : QWidget(parent)
14  , ui(new Ui::PolygonRole)
15  {
16          ui->setupUi(this);
17          hideWindowFrame();
18
19          connect(this, SIGNAL(send_role(QString)),
```

```
20          main_window, SLOT(add_polygon(QString)));
21  }
22
23  PolygonRole::~PolygonRole()
24  {
25          delete ui;
26  }
27
28  void PolygonRole::hideWindowFrame()
29  {
30          setWindowFlags(Qt::Window | Qt::FramelessWindowHint);
31  }
32
33  void PolygonRole::mousePressEvent(QMouseEvent* event)
34  {
35          m_lastClickPoint = event->pos();
36  }
37
38  void PolygonRole::mouseMoveEvent(QMouseEvent* event)
39  {
40          if (event->buttons() & Qt::LeftButton)
41          {
42                  QPoint Difference = event->pos() - m_lastClickPoint;
43                  window()->move(window()->pos()+Difference);
44          }
45  }
46
47  void PolygonRole::on_cancelButton_clicked()
48  {
49          this->close();
50  }
51
52
53  void PolygonRole::on_closeButton_clicked()
54  {
55          this->close();
56  }
57
58
59  void PolygonRole::on_selectButton_clicked()
60  {
61          emit send_role(ui->rolesComboBox->currentText());
62          this->close();
63  }
64
65
```

4.6 Подключение статических ресурсов

В программе необходимы два статических ресурса, которые пользователь может иметь право изменит - это default game settings.json для представления начального положения при входе в программу и schema.json - для валидации конфига игры. Сохранение их обоих происходит через cmake

```
configure_file(${CMAKE_CURRENT_SOURCE_DIR}/resources/schema.json
${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/data COPYONLY)
configure_file(${CMAKE_CURRENT_SOURCE_DIR}/resources/
    default_game_settings.json
${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/data COPYONLY)
```

Помимо этого используются различные изображения. Их встраивание происходит через файл icons.qrc

```
<RCC>
<qresource prefix="/create">
<file>add.png</file>
</qresource>
<qresource prefix="/font">
<file>Montserrat-Medium.ttf</file>
</qresource>
<qresource prefix="/json_schema">
<file>schema.json</file>
</qresource>
<qresource prefix="/flags">
<file>EN.png</file>
<file>RU.png</file>
</qresource>
<qresource prefix="/power">
<file>power.png</file>
</qresource>
<qresource prefix="/next">
<file>next_step.png</file>
</qresource>
<qresource prefix="/prev">
<file>prev_step.png</file>
</qresource>
<qresource prefix="/reload">
<file>reload.png</file>
</qresource>
<qresource prefix="/edit">
<file>edit.png</file>
</qresource>
<qresource prefix="/timer">
<file>timer.png</file>
```

```
</qresource>
<qresource prefix="/ethernet">
<file>ethernet.png</file>
</qresource>
<qresource prefix="/role">
<file>role.png</file>
</qresource>
<qresource prefix="/accept">
<file>accept.png</file>
</qresource>
<qresource prefix="/triamgle_white">
<file>cb_triangle_w.png</file>
</qresource>
<qresource prefix="/server_black">
<file>server_black.png</file>
</qresource>
<qresource prefix="/server_white">
<file>server_white.png</file>
</qresource>
<qresource prefix="/controller_black">
<file>controller_black.png</file>
</qresource>
<qresource prefix="/controller_white">
<file>controller_white.png</file>
</qresource>
<qresource prefix="/triangle_r">
<file>cb_triangle_r.png</file>
</qresource>
<qresource prefix="/close">
<file>close.png</file>
</qresource>
<qresource prefix="/maximize">
<file>maximize.png</file>
</qresource>
<qresource prefix="/minimize">
<file>minimize.png</file>
</qresource>
<qresource prefix="/triangle">
<file>cb_triangle.png</file>
</qresource>
<qresource prefix="/import">
<file>imprt.png</file>
</qresource>
<qresource prefix="/export">
<file>exprt.png</file>
</qresource>
<qresource prefix="/remove_all">
```

```
<file>del_all.png</file>
</qresource>
<qresource prefix="/remove">
<file>del.png</file>
</qresource>
<qresource prefix="/diagonal_line">
<file>line.png</file>
</qresource>
<qresource prefix="/">
<file>polygon_images/charger.png</file>
<file>polygon_images/drone.png</file>
<file>polygon_images/factory.png</file>
<file>polygon_images/fire.png</file>
<file>polygon_images/loading.gif</file>
<file>polygon_images/robot.png</file>
<file>polygon_images/startPlace.png</file>
<file>polygon_images/village.png</file>
<file>styles/playersTree.qss</file>
<file>styles/ContextMenu.qss</file>
<file>styles/DefaultText.qss</file>
<file>styles/NotRoundedButton.qss</file>
<file>styles/QuickControlButton.qss</file>
<file>styles/RoundedButton.qss</file>
<file>styles/RoundedCheckBox.qss</file>
<file>styles/RoundedProgressBar.qss</file>
<file>styles/RoundedToggledButton.qss</file>
<file>styles/RoundedToolButton.qss</file>
<file>styles/SquareRoundedButton.qss</file>
<file>styles/SquareRoundedCheckBox.qss</file>
<file>styles/TabBase.qss</file>
<file>styles/TableBase.qss</file>
<file>styles/TableHorizontalHeader.qss</file>
<file>styles/TableVerticalHeader.qss</file>
<file>styles/WhiteComboBox.qss</file>
</qresource>
<qresource prefix="/styles"/>
</RCC>

```

Тогда использовать их в коде можно следующим образом.

```
PolygonPictureItem(x, y, stepSize, ":/polygon_images/drone.png")
```

Помимо изображений здесь также происходит встраивание путей к шрифту приложения и файлам стилей. О них подробнее будет сказано ниже.

4.7 Валидация сохраненной игры через JSON Validator

Игру можно сохранять в виде конфига и загружать после сохранения. Однако конфиг является обычным текстовым файлом формата .JSON и пользователь может внести в него недопустимые изменения. Чтобы этого избежать, необходимо применить либо множественные проверки каждого поля, либо использовать специально разработанный валидатор, который сверяет формат текущего файла с номинальным.

Сам валидатор имеет всего один метод и реализован следующим образом.

```
bool GameDataValidator::validate(const char* schema_path,
const char* doc_path) {
        using valijson::Schema;
        using valijson::SchemaParser;
        using valijson::Validator;
        using valijson::adapters::RapidJsonAdapter;

        // Load JSON document using RapidJSON with Valijson helper function
        rapidjson::Document mySchemaDoc;
        if (!valijson::utils::loadDocument(schema_path, mySchemaDoc)) {
                return false;
        }

        // Parse JSON schema content using valijson
        Schema mySchema;
        SchemaParser parser;
        RapidJsonAdapter mySchemaAdapter(mySchemaDoc);
        parser.populateSchema(mySchemaAdapter, mySchema);

        rapidjson::Document myTargetDoc;
        if (!valijson::utils::loadDocument(doc_path, myTargetDoc)) {
                return false;
        }

        Validator validator;
        RapidJsonAdapter myTargetAdapter(myTargetDoc);
        if (!validator.validate(mySchema, myTargetAdapter, NULL)) {
                return false;
        } else {
                return true;
        }
}
```

Как видим, он использует дополнительные библиотеки. Сам же номинальный конфиг имеет следующую структуру:

```
{
        "comment": "Json Schema for Arena Station",
        "type": "object",
        "properties": {
                "Game_settings": {
                        "type": "object",
                        "properties": {
                                "time_game": {
                                        "type": "integer"
                                },
                                "time_block": {
                                        "type": "number"
                                },
                                "time_loading": {
                                        "type": "integer"
                                },
                                "time_action": {
                                        "type": "integer"
                                },
                                "max_bullet": {
                                        "type": "integer"
                                },
                                "shooting_radius": {
                                        "type": "number"
                                },
                                "method_check_collision_players": {
                                        "type": "string"
                                },
                                "method_check_collision_polygon": {
                                        "type": "string"
                                },
                                "dist_collision_circle": {
                                        "type": "number"
                                },
                                "dist_collision_square": {
                                        "type": "number"
                                },
                                "game_description": {
                                        "type": "string"
                                }
                        },
                        "additionalProperties": false,
        ...
        "definitions": {
                "color_code": {
                        "type": "number",
                        "minimum": 0,
```

```
48                          "maximum": 255
49                  }
50          }
51  }
```

Как видим, в нем представлены типы используемых объектов, структура вложенности, а также кастомные типы, как, например, цвет.

4.8 Подключение шрифта

Шрифт подключается в icons.qrc. Внедряется в проект следующим образом.

Start point проекта (строки 14, 15).

```
1   #include "inc/gui/auth_window.h"
2
3   #include <QApplication>
4   #include <QLocale>
5   #include <QTranslator>
6   #include <QFontDatabase>
7
8
9   int main(int argc, char *argv[])
10  {
11          QApplication a(argc, argv);
12
13
14          QFontDatabase::addApplicationFont(":/font/Montserrat-Medium.ttf");
15          qApp->setFont(QFont("Montserrat Medium", 10, QFont::Normal, false));
16
17          QTranslator translator;
18          if (translator.load("arenastation_ru_RU.qm")) {
19                  a.installTranslator(&translator);
20          }
21
22          AuthWindow w;
23          w.show();
24          return a.exec();
25  }
26
```

Аналогично внедряются переводы.

4.9 Подключение стилей CSS

Стили CSS внедряются в виде файлов .qss, применяемых к определенным элементам настранице. К примеру, содержимое файла WhiteComboBox.qss

```
QComboBox {
        padding-right: 10px;
        border-radius: 8px;
        color: #033538;
        padding-top: 5px;
        padding-bottom: 5px;
        padding-left: 10px;
        font: 10pt "Montserrat Medium";
        border: 1px solid #033538;
}

QComboBox::item {
        height: 30px;
}

QComboBox::hover {
        background: #e5e5e5;
}

QComboBox::drop-down {
        padding-left: 5px;
        padding-right: 5px;
        subcontrol-origin: padding;
        subcontrol-position: top right;
        width: 15px;
        image: url(:/triangle/cb_triangle.png);
        border-left-width: 0px;
        border-left-color: darkgray;
}

QComboBox QAbstractItemView {
        color: #033538;
        background: white;
        padding: 2px;
        outline: 0px;
        border: 1px solid black;
        border-radius: 7px;
}

QListView::item {
        height: 25px;
        font-size: 12pt;
```

```
}

QListView::item:selected {
        height: 15px;
        background-color: rgba(0, 0, 0, 15%);
        border-radius: 4px;
        color: #033538;
        padding-left: 3px;
}
```

Применение его к элементу происходит следующим образом.

```
QComboBox *comboBox = new QComboBox();
setStyle(comboBox, ":/styles/WhiteComboBox.qss");
```

4.10 Подключение дополнительных бибилотек

В проекте используются две библиотеки - valijson и rapidjson. Первая используется для валидации файла JSON, вторая для его чтения. Обе библиотеки являются header-only, то есть не требуют зависимости от дополнительных предкомпилируемых файлов. Это требование связано с тем, что приложение используется как в Windows, так и в Linux, так что внесение изменений должно быть как можно менее затратным.

Использование же библиотек в программе реализуется их простым подключением через include, как представлено в предыдущем пункте.

4.11 Вариант с использованием JSON как хранилища локальных данных

Открытый файл JSON хранится в поле data класса GameDataManager.

```
GameDataManager *GameDataManager::open_doc(QWidget* parent)
{
        QString json_filter = "JSON (*.json)";
        QString json_file = QFileDialog::getOpenFileName(parent,
            QObject::tr("Открыть сохраненную игру"),
        "/", json_filter, &json_filter);

        QString schema_path = "data/schema.json";
        if (!GameDataValidator::validate(schema_path.toStdString().c_str(),
            json_file.toStdString().c_str())) {
                ErrorWindow *data_not_correct = new
                    ErrorWindow(QObject::tr("Неверный файл"),
```

```
                QObject::tr("Выберите корректную конфигурацию игры!"));
                data_not_correct->show();
                return this;
        }
        is_data_correct = true;
        QByteArray data_json;
        QFile input(json_file);
        if (input.open(QIODevice::ReadOnly | QIODevice::Text)) {
                data_json = input.readAll();
        }
        doc = doc.fromJson(data_json);
        data = doc.object();
        return this;
}
```

Возникает соблазн использовать его как хранилище локальных данных. То есть постоянно все вносимые изменения сразу переносить в JSON.

Однако этот путь оказался тупиковым, ведь для перезаписи всего одного поля нужно обновлять полностью все поля в JSON файле. Возможно, это проблема конкретной библиотеки, однако остальные вынуждают заново перекомпилировать себя для каждой системы при изменениях. Поэтому был применен другой подход.

4.12 Вариант с использованием интерфейса как хранилища данных

Второй подход связан с хранением данных в скрытых от пользователя полях интерфейса. Однако этот подход нарушает принцип MVC и уже в скором времени каждое новое дополнение превращало такую реализацию во всё более не рабочую схему. Ведь компоненты разбросаны по всему интерфейсу и связь между ними порой проходила через несколько классов. От этого метода пришлось отказаться.

4.13 Реализация хранилища с помощью статических классов

Изначально была попытка создания абстрактного класса IComponent.

```
#ifndef ICOMPONENT_H
#define ICOMPONENT_H

class IComponent {
```

```
        public:
        virtual void updateState() = 0;
        virtual void updateComponent() = 0;
};

#endif // ICOMPONENT_H

```

Все компоненты должны были наследоваться от него и реализовывать возможность обновления интерфейса в зависимости от состояния приложения и наоборот. Однако этот метод сработал только для некоторых окон, в которых реализация обновления компонента возможна через один метод. В остальных случаях это несколько разных методов для каждой отдельной части компонента.

В итоге всё пришло к системе, похожей на Vuex в vue.js, однако реализующей только часть функциональности, ведь статические классы не поддерживают сигналы и слоты. Ответственность за обновление компонентов легла на них самих. Очевидно, при изначальной реализации проекта по Clean Architecture этих проблем можно было избежать, однако всё было настроено на то, чтобы ставить во главу угла именно фреймворк Qt и стараться реализовывать всё через него.

Вот пример реализации полученного state-а для работы с конфигурацией (настройкой) проекта.

configurator settings.h

```
#ifndef CONFIGURATOR_SETTINGS_H
#define CONFIGURATOR_SETTINGS_H

#include <QList>
#include <QMap>
#include <QString>

#define SETTINGS \
public:\
static QList<SettingItem> settings; \
static SettingItem at(int index) { \
        return settings.at(index); \
} \
static void addParameter(SettingItem parameter) { \
        settings.append(parameter); \
} \
static void changeParameter(const QString& parameter, const QMap<QString,
 ↪  QString>& newParameterRow) { \
        for (int i = 0; i < settings.size(); ++i) { \
```

```
                SettingItem& item = settings[i]; \
                if (item.getParameter("parameter") == parameter) { \
                        for (auto iter = newParameterRow.begin(); iter !=
                         ↪  newParameterRow.end(); ++iter) { \
                                const QString& key = iter.key(); \
                                const QString& value = iter.value(); \
                                item.setParameter(key, value); \
                        } \
                        break; \
                } \
        } \
} \
static void removeParameter(const QString& parameter) { \
        for (int i = 0; i < settings.size(); ++i) { \
                SettingItem& item = settings[i]; \
                if (item.getParameter("parameter") == parameter) { \
                        settings.removeAt(i); \
                        break; \
                } \
        } \
}

struct SettingItem {
        QMap<QString, QString> data;

        SettingItem(QMap<QString, QString> initialData) : data(initialData)
         ↪  {}

        QString getParameter(const QString& key) const {
                return data.value(key);
        }

        void setParameter(const QString& key, const QString& value) {
                data[key] = value;
        }
};

class ConfiguratorSettings {
        public:
        class ServerSettings {
        };

        class GameSettings {
                public:
                class CollisionsSettings  {
                        SETTINGS
                };
```

```
                class Shots {
                        SETTINGS
                };
                class Time {
                        SETTINGS
                };
        };

        class RoleSettings {
                public:
                class PlayerRolesSettings {
                        SETTINGS
                };
                class PolygonRolesSettings {
                        public:
                        class WeaponRolesSettings {
                                SETTINGS
                        };
                        class PowerStationSettings {
                                SETTINGS
                        };
                        class DeadZoneSettings {
                                SETTINGS
                        };
                        class FactorySettings {
                                SETTINGS
                        };
                        class VillageSettings {
                                SETTINGS
                        };
                        class FireSettings {
                                SETTINGS
                        };
                };
        };
};

#endif // CONFIGURATOR_SETTINGS_H
```

configurator settings.cpp

```
#include "state/configurator_settings.h"

QList<SettingItem>
    ↪ ConfiguratorSettings::GameSettings::CollisionsSettings::settings;
QList<SettingItem> ConfiguratorSettings::GameSettings::Shots::settings;
QList<SettingItem> ConfiguratorSettings::GameSettings::Time::settings;
```

```
QList<SettingItem>
    ConfiguratorSettings::RoleSettings::PlayerRolesSettings::settings;

// Polygon roles
QList<SettingItem> ConfiguratorSettings::RoleSettings::PolygonRolesSettings::
    WeaponRolesSettings::settings;
QList<SettingItem> ConfiguratorSettings::RoleSettings::PolygonRolesSettings::
    PowerStationSettings::settings;
QList<SettingItem> ConfiguratorSettings::RoleSettings::PolygonRolesSettings::
    DeadZoneSettings::settings;
QList<SettingItem> ConfiguratorSettings::RoleSettings::PolygonRolesSettings::
    FactorySettings::settings;
QList<SettingItem> ConfiguratorSettings::RoleSettings::PolygonRolesSettings::
    VillageSettings::settings;
QList<SettingItem> ConfiguratorSettings::RoleSettings::PolygonRolesSettings::
    FireSettings::settings;
```

Как видим, содержимое каждого статического класса представлено как макрос и тем самым облегчает написание. Каждый элемент имеет свое собственное локальное хранилище. Это позволяет работать с элементами в коде довольно удобно.

```
ConfiguratorSettings::RoleSettings::PolygonRolesSettings::
    FireSettings::addParameter(fireParameter)
```

4.14 Добавление многоязычности

При входе в приложение есть возможность выбора языка. Это реализовано через Qt Linquist. Для этого необходимо прописать в cmake, с какими переводами мы будем работать.

```
find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets LinguistTools)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets LinguistTools)

set(TS_FILES
arenastation_en_US.ts
arenastation_ru_RU.ts
)

qt_create_translation(QM_FILES ${CMAKE_SOURCE_DIR} ${TS_FILES})

set(PROJECT_SOURCES
main.cpp
${HEADERS}
```

```
14 ${SRC}
15 ${RESOURCES}
16 ${QM_FILES}
17 ${TS_FILES}
18 )
19
20 qt_add_executable(arenastation
21 MANUAL_FINALIZATION
22 ${PROJECT_SOURCES}
23 )
```

После этого собрать проект. Qt автоматические сгенерирует файлы перевода. Каждый файл нужно открыть в программе Qt Linguist и вручную написать переводы для всех элементов.

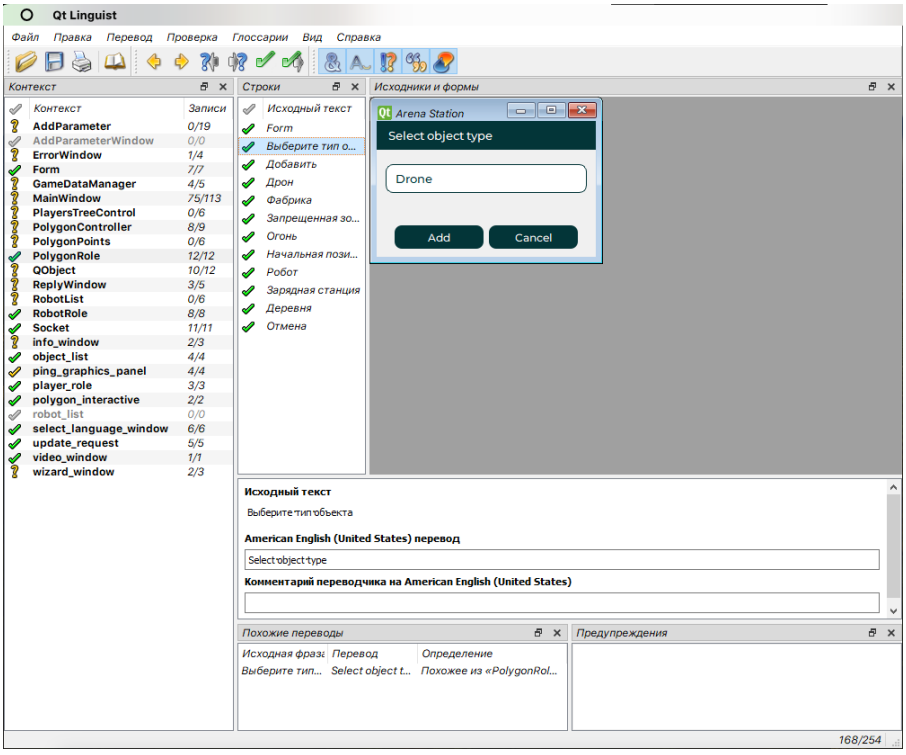

Рис. 16:

Для того, чтобы виджеты были переводимы, нужно либо создать их через Qt Designer и назначить их переводимыми (значение по умолчанию)

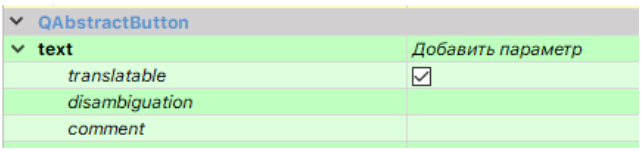

Рис. 17: Переводимый виджет в Qt Designer

Для переводов виджетов, добавленных динамически, текст в них должен быть обрамлен в макрос tr()

```
QMenu *fileMenu = new QMenu();
fileMenu->setTitle(tr("Файл"));

QAction *export_game = new QAction();
export_game->setText(tr("Экспорт игры"));
fileMenu->addAction(export_game);

QAction *import_game = new QAction();
import_game->setText(tr("Импорт игры"));
fileMenu->addAction(import_game);

QAction *exit_game = new QAction();
exit_game->setText(tr("Выход"));
fileMenu->addAction(exit_game);
```

Сама же смена языка происходит так.

```
void AuthWindow::on_ruLangButton_clicked()
{
        if (translator.load("arenastation_ru_RU.qm")) {
                QApplication::instance()->installTranslator(&translator);
        }
        ArenaLocale::setRu();
}


void AuthWindow::on_enLangButton_clicked()
{
        if (translator.load("arenastation_en_US.qm")) {
                QApplication::instance()->installTranslator(&translator);
        }
        ArenaLocale::setEng();
}

void AuthWindow::changeEvent(QEvent *event)
{
        if (event->type() == QEvent::LanguageChange) {
                ui->retranslateUi(this);
        }
        QWidget::changeEvent(event);
}
```

4.15 Встраивание браузерных взаимодействий

Окно визуализации в приложении отображает JavaScript приложение, передаваемое по адресу в локальной сети. Для его отображения используется встраиваемый в приложения отдельный браузер через Qt Web Engine

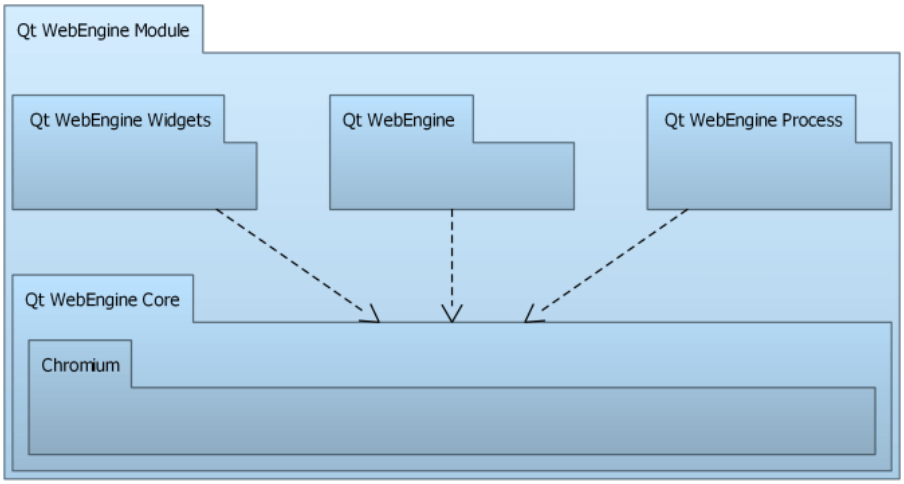

Рис. 18: Qt Web Engine

Он запускает браузер отдельным процессом.

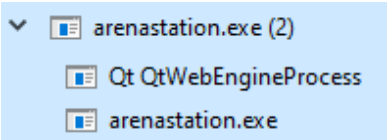

Рис. 19: Браузер запускается отдельным процессом

В коде подключение и использование Qt Web Engine реализовано следующим образом.

В cmake

```
find_package(Qt6 REQUIRED COMPONENTS OpenGLWidgets)
```

В методах классов

```
Visualization::Visualization(QFrame* visualizationFrame,
QLineEdit* urlInput,
QPushButton* allowEditUrl,
QPushButton* makeChanges)
: visualizationFrame(visualizationFrame),
urlInput(urlInput),
allowEditUrl(allowEditUrl),
```

```
8  makeChanges(makeChanges),
9  visualizationUrl(""),
10 isEditUrlLineEnabled(false),
11 webView(new QWebEngineView(visualizationFrame)),
12 webPage(new QVBoxLayout())
13 {
14         webView->setUrl(QUrl(visualizationUrl));
15         webView->setAttribute( Qt::WA_OpaquePaintEvent, true );
16         webView->setAttribute( Qt::WA_PaintOnScreen, true );
17         webView->setAttribute( Qt::WA_DontCreateNativeAncestors, true );
18         webView->setAttribute( Qt::WA_NativeWindow, true );
19         webView->setAttribute( Qt::WA_NoSystemBackground, true );
20         webView->setAutoFillBackground( false );
21
22         webPage->addWidget(webView);
23         visualizationFrame->setLayout(webPage);
24 }
```

4.16 Готовящиеся к реализации возможности

Сам интерфейс реализован. Осталось лишь наладить работу с сервером и обновлять данные в зависимости от принятых и отправленных запросов.

5 Заключение

Хотя реализация клиентской части проекта на C++ в Qt все же возможна, однако его применение всё же более желательно для встраиваемых систем и систем, в которых нужно крайне высокая производительность. А для подобных приложений всё же использовать Java (JavaFX или Swing).