Министерство науки и высшего образования РФ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Новгородский государственный университет имени Ярослава Мудрого»

Кафедра информационных технологий и систем

**Сетевая игра «Биатлон»**

Курсовая работа по дисциплине:

«Операционные системы»

Выполнила

Студент группы №9091

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Виноградов Ю.А.

\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2021

Проверил

Преподаватель

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Ананьев В.В.

\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2021

**Великий Новгород**

**2021**

**ОГЛАВЛЕНИЕ**

**ЦЕЛЬ РАБОТЫ**...........................................................................................................................3

**ЗАДАНИЕ**…...…………………………………………………………………………………...3

**ГЛАВА 1. РАЗРАБОТКА ПРОЕКТА**……………….………………………………………..4

1.1 Основной алгоритм решения задачи………………………………………………..4

1.2 Протокол взаимодействия клиента и сервера………………………………….…..6

1.3 Выбор протокола транспортного уровня OSI-модели (TCP/UDP)……………...10

1.4 Решение проблемы блокировки ввода/вывода…………………………………...10

1.5 Решение проблемы нефиксированного размера целочисленного типа на разных платформах…………………………………………………………………………...…11

**ГЛАВА 2. РЕАЛИЗАЦИЯ ПРОЕКТА**……………………………………………………...12

2.1 Структура клиента и сервера………………………………………………………12

2.2 Программный интерфейс сервера…………………………………………………13

2.3 Список основных функций клиента и сервера……………………………………14

2.4 Список системных вызовов операционной системы……………………………..15

2.5 Пример работы приложения……………………………………………………….17

**РЕЗУЛЬТАТ РАБОТЫ**.............................................................................................................21

**ВЫВОД**………………...……………………………………………………………………….21

**ПРИЛОЖЕНИЕ А**…………………………………………………………………………….22

ЛИСТИНГ А1. Файл mainwindow.h…..………...………………………………….….22

ЛИСТИНГ А2. Файл areaforshot.h…………………………………………………….23

ЛИСТИНГ А3. Файл mainwindow.cpp………………………………………………...24

ЛИСТИНГ А4. Файл areaforshot.cpp………...………………………………………...28

ЛИСТИНГ А5. Файл main.cpp…………………………………………………………29

ЛИСТИНГ А6. Файл server.cpp……………………………………………………….29

**ЦЕЛЬ РАБОТЫ**

Целью курсовой работы является получить практические навыки разработки и реализации многопользовательских сетевых приложений под управлением операционной системы семейства Unix/Linux.

**ЗАДАНИЕ**

Заданием данной курсовой работы является создание сетевой игры “Снайпер”. Суть игры заключается в следующем. В игре принимают участие два человека. Каждый из них должен закрыть некоторое количество мишеней, которые будут появляться на поле в различных местах, в течении 30 секунд. Победил тот игрок, который закрыл большее количество мишеней.

В случае, если игрок промазал, мишень все равно пропадает и появляется на новом месте.

Данное приложение должно работать под управлением операционной системы семейства Unix/Linux. Сервер и клиент должны быть выполнены на языке программирования C/C++ с помощью сокетов, предоставляемых системной библиотекой <sys/socket.h>. Клиент и сервер должны быть написаны в качестве двух отдельных программ. Клиент должен иметь графический интерфейс и просто отображать результат обработки сервера. Сервер в свою очередь должен проверять состояния клиентов перед обработкой, чтобы не допустить жульничества. В себе он должен содержать саму логику игры.

**Глава 1**

**РАЗРАБОТКА ПРОЕКТА**

*1.1 Основной алгоритм решения задачи*

Прежде, чем приступать к реализации проекта, необходимо продумать логику работы и взаимодействия Сервера и Клиента.

Общий алгоритм работы сервера на рисунке 1.1:

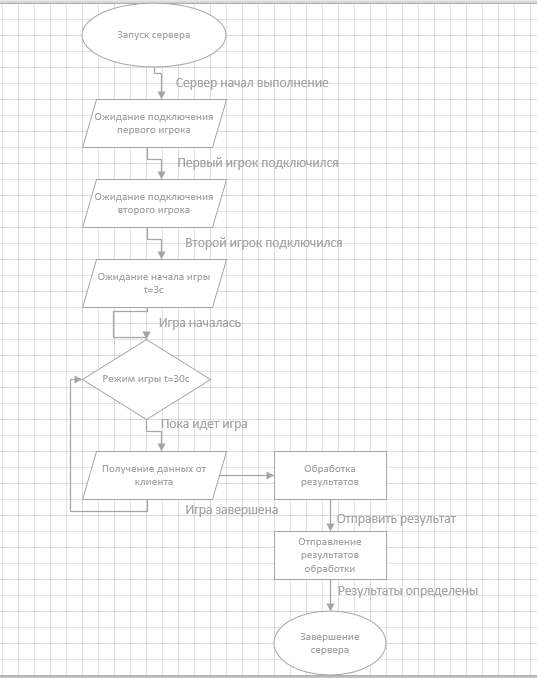


Рисунок 1.1. Общий алгоритм работы сервера.

Общий алгоритм рабоыт клиента на рисунке 1.2:

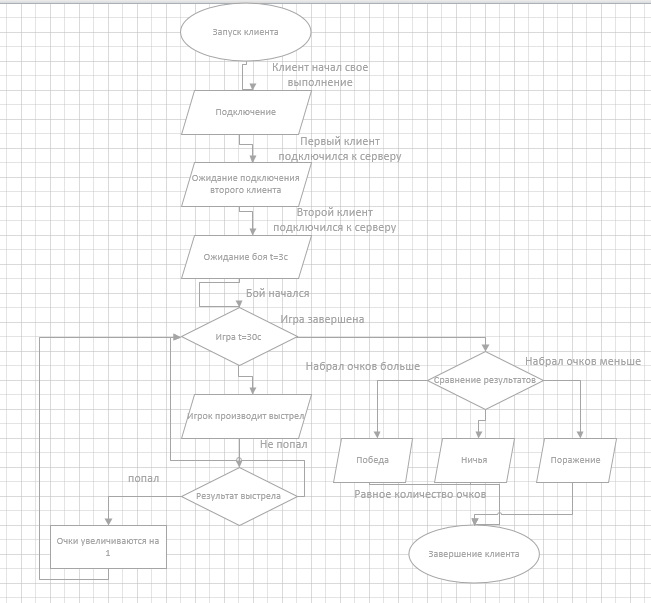


Рисунок 1.2. Общий алгоритм работы клиента.

На представленных схемах в блоках обведены состояния, в которых может находиться клиент/сервер в процессе своей работы. Слова, находящиеся вне блоков, обозначают события, по которым клиент/сервер переходят из одного состояния в другое. Стрелочка, отходящая от одного блока к другому, обозначает переход от одного состояния к другому.

В начале своей работы клиент переходит в состояние «Подключение». В этом состоянии на клиенте отображается главное меню с заставкой. На данном этапе из функционала присутствует только строка для ввода адреса и кнопка подключение. В этот момент сервер находится в состоянии «Ожидание подключения первого игрока», чтобы принять соединение. После подключения первого игрока, клиент переходит в состояние «Ожидание подключения второго игрока», так как игра рассчитана на двух игроков, поэтому она не может начаться. На этом этапе выводится заставка с просьбой подождать другого игрока. В этот момент сервер тоже находиться в состоянии «Ожидание подключения второго игрока» для принятия соединения.

После того, как все игроки подключились к серверу, оба клиента переходят в состояние “Ожидание боя”. Через 3 секунды начинается бой, и оба клиента переходят в состояние “бой”. По истечении 30 секунд, бой прекращается и в зависимости от результата, клиент может либо перейти в состояние “победа”, либо перейти в состояние “поражение”, либо перейти в состояние “ничья”. После этого сервер прекращает свою работу.

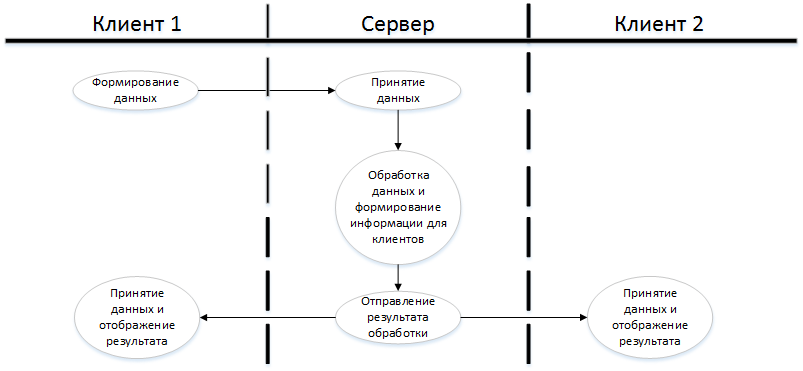
Клиенты завершают свою работу в том случае, если пользователь закрыл приложение.

*1.2 Протокол взаимодействия клиента и сервера*

Для обеспечения взаимодействия между клиентом и сервером между ними происходит обмен сообщениями, которые представляют из себя структуру данных определённого размера. В некоторых случаях обмен данными происходит не целой структурой, а небольшими переменными. Обмен сообщениями происходит следующим образом.

В начале работы приложения серверу необходимо знать сокеты клиентов, чтобы общаться с ними. Поэтому на этапе подключения сервер получает сокеты клиентов.

После подключения игроков наступает стадия боя. На данном этапе обмен сообщениями происходит целыми блоками. Чтобы лучше понять, как происходит обмен сообщениями между клиентом и сервером на стадии боя, рассмотрим схему, приведённую на рис. 1.3.



**Рис. 1.2**. Обмен данными между клиентом и сервером на стадии боя

Подобная схема применима и для второго клиента. Во время боя сервер ждёт только один блок данных – позицию выстрела игрока. Когда сервер получил необходимые ему данные, он обрабатывает их и формирует блок информации для клиентов, в которых содержится позиция выстрела клиента и его результат: попал или промахнулся. Затем сформированные блоки отправляются клиентам как результат обработки.

Помимо стадии боя обмен небольшими блоками информации, которые содержат в себе следующее состояние клиента, происходит на всём протяжении работы приложения. Эти блоки информации необходимы для того, чтобы переводить клиент из одного состояния в другое. Например, когда один из игроков подключается к серверу, то ему необходимо подождать другого игрока, так как игра рассчитана на двоих. Чтобы сообщить, что противник ещё не подключился к игре, клиента, после подключения, переводят в состояние «Ожидание подключения второго игрока», чтобы вывести ему соответствующее сообщение с просьбой подождать.

Так как клиент принимает данные, ему необходимо распознавать вид сообщения, чтобы предпринять те или иные действия. Серверу в свою очередь не нужно знать вид, так как на каждом определённом этапе он точно знает какая информация поступит.

Таким образом, чтобы клиент имел представления, какие данные ему пришли, было решено сформировать структуру сообщения, в которой первые два байта информации содержат в себе тип сообщения, а следующие байты содержат остальную информацию, необходимую для отображения. Каждая переменная, входящая в сообщение равна двум байтам. Были сформированы следующие виды сообщений:

1. CoorOfTarget – вид сообщения, в котором содержится позиция цели, сформированный сервером. Данное сообщение имеет следующую структуру:

**Таблица 1.2**. Структура сообщения Message

|  |  |  |
| --- | --- | --- |
| **Имя переменной** | **Размер, байт** | **Смещение, байт** |
| type | 2 | 0 |
| PosX | 2 | 4 |
| PosY | 2 | 6 |

Переменная type отвечает за тип сообщения. Данная переменная содержит в себе следующее значение: coor\_of\_shot (соответствует значению 2) – координаты цели. PosX – переменная, которая хранит в себе позицию цели по оси абсцисс, а PosY –переменная, которая хранит в себе позицию цели по оси ординат.

1. ResultOfBung – вид сообщения, в котором содержится результат выстрела игрока. Данное сообщение имеет следующий структуру:

**Таблица 1.1**. Структура сообщения Shot

|  |  |  |
| --- | --- | --- |
| **Имя переменной** | **Размер, байт** | **Смещение, байт** |
| type | 2 | 0 |
| result | 2 | 2 |

Переменная type отвечает за тип сообщения. Данная переменная содержит в себе следующее значение: result\_of\_shot (соответствует значению 0) – результат выстрела. Переменная result отвечает за результат выстрела: попал или промахнулся. В данном случае имеет следующие значения: not\_hit (соответствует значению 0) – промахнулся, hit (соответствует значению 1) – попал.

1. ScoreOfEnemy – вид сообщения, в котором содержатся очки, набранные противником. Данное сообщение имеет следующий структуру:

**Таблица 1.1**. Структура сообщения Shot

|  |  |  |
| --- | --- | --- |
| **Имя переменной** | **Размер, байт** | **Смещение, байт** |
| type | 2 | 0 |
| Score | 2 | 2 |

Переменная type отвечает за тип сообщения. Данная переменная содержит в себе следующее значение: score\_of\_enemy (соответствует значению 3) – количество очков, набранное противником. Переменная Score содержит в себе количество очков, которое набрал противник.

1. Shot – вид сообщения, в котором содержатся только координаты выстрела. Данное сообщение имеет следующий структуру:

**Таблица 1.1**. Структура сообщения Shot

|  |  |  |
| --- | --- | --- |
| **Имя переменной** | **Размер, байт** | **Смещение, байт** |
| PosX | 2 | 0 |
| PosY | 2 | 2 |

PosX – переменная, отвечающая за позицию выстрела по оси абсцисс, PosY – переменная, отвечающая за позиция выстрела по оси ординат.

1. StateForClient – вид сообщения, необходимый для перевода клиента из одного состояния в другое. Данное сообщение имеет следующий структуру:

**Таблица 1.1**. Структура сообщения Shot

|  |  |  |
| --- | --- | --- |
| **Имя переменной** | **Размер, байт** | **Смещение, байт** |
| type | 2 | 0 |
| state | 2 | 2 |

Переменная type отвечает за тип сообщения. Данная переменная содержит в себе следующее значение: state\_for\_client (соответствует значению 1) – состояние клиента. Переменная state содержит в себе следующее состояние клиента:

1. WaitingOfConnection (соответствует значению 0) - состояние, отвечающее за вывод промежуточного окна после подключения с просьбой подождать другого игрока, так как игра не может начаться с участием только одного игрока.
2. WaitingOfCombat (соответствует значению 1) – состояние, отвечающее за вывод промежуточного окна после подключения второго игрока с просьбой подождать 3 секунды до начала игры.
3. Win (соответствует значению 2) - состояние, отвечающее за вывод итогового окна со словом «Победа» в случае, если игрок выиграл.
4. Lose (соответствует значению 3) - состояние, отвечающее за вывод итогового окна со словом «Поражение» в случае, если игрок проиграл.
5. Combat (соответствует значению 4) - состояние, отвечающее за отображение окна с игровым полем, где будет происходить сама битва.
6. DeadHeat (соответствует значению 5) – состояние, отвечающее за вывод итогового окна со словом “Ничья” в случае, если игроки сыграли в ничью.

Выше перечисленные состояния могут храниться на самом клиенте, так как нужны они лишь для отображения.

Те же состояния хранятся и на сервере.

*1.3 Выбор протокола транспортного уровня OSI-модели (TCP/UDP)*

Для того, чтобы сервер и клиент могли обмениваться информацией, необходимо сначала выбрать по какому из протоколов TCP/UDP будет происходить обмен. Необходимо выяснить их основные различия.

TCP – протокол, который обеспечивает надёжность передаваемых данных. Если данные были не получены или искажены в процессе отправки, то он запрашивает повторную передачу. Также этот протокол обеспечивает правильный порядок принятых данных. Данный протокол требует больших затрат времени, однако он гарантирует точную доставку пакета, что избавляет программиста от написания проверки на получение всех байт информации.

UDP – протокол, который обеспечивает быструю отправку данных. Данный протокол имеет ряд недостатков. Сообщения поступают в любом порядке, и то, которое отправлено первым, может быть получено последним. Доставка сообщений в UDP не гарантируется. Сообщения могут потеряться, или могут быть получены две копии одного и того же сообщения.

Для своей курсовой работы я выбрал протокол TCP, так как мне необходимо, чтобы все сообщения всегда доходили от сервера до клиентов и наоборот. Например, если до сервера не дойдут координаты выстрела, сформированные одним из игроков, или исказятся, то может обратиться не к той ячейке таблицы или вообще выйти за её предел, что приведёт к ошибкам отображения, так как этот результат будет отправлен клиентам, и игры в целом. Используя TCP мне нет необходимости писать дополнительные проверки получения данных, так как надёжность получения всех данных обеспечивает сам протокол TCP. Также учитывая, что игра сама по себе не масштабная по функционалу и графике, лучше обеспечить надёжность доставки данных, чем просто их быструю отправку.

*1.4 Решение проблемы блокировки ввода/вывода*

Обычно приём данных с сокета выполняется в блокирующем режиме. В случае, когда данных на сокете нету, то поток, в котором происходит чтение, блокируется до их появления. Подобная работа представляет проблему для приложения, которое должно проверять данные, которые пришли к нему, без торможения графического интерфейса. Помимо этого, проблемы возникают, когда отправляются данные на сервер. Во время боя игроки ходят параллельно. Таким образом игра наделена следующими свойствами:

1. Сервер работает в многопоточном режиме. Данное свойство даёт возможность сохранить параллельность ходов. Когда данные приходят на сервер, то заранее проверяется состояние клиента, который отправил данные.
2. Клиент читает данные с своего сокета в неблокирующем режиме. Обычно чтение данных с сокета происходит в блокирующем режиме, что создаёт проблемы для графического интерфейса приложения. Приложение должно постоянно рисовать сцену для отображения. Если приложение начнёт считывать данные с сокета, то оно остановиться на этом моменте, пока все данные не придут. А значит приложение не будет отображать графический интерфейс, то есть остановится на одном моменте и дальше этого кадра рисовать не будет. Данное свойство помогает постоянно отображать графический интерфейс и читать данные, которые были отправлены сервером без затормаживания графического интерфейса.

*1.5 Решение проблемы нефиксированного размера целочисленного типа на компьютерах разной архитектуры*

Исторически сложилось так, что спецификация языка С точно не определяет точный размер каждого целочисленного типа. В языке намеренно оставили размер целочисленных типов не фиксированным для того, чтобы компилятор мог самостоятельно подобрать наиболее подходящий размер для определённого типа данных в зависимости от компьютерной архитектуры. Это вызывает проблему при обмене данных между клиентом и сервером, так как они могут быть запущены на компьютерах разной архитектурах, где размер одного и того же типа может быть разным. Соответственно данные при считывании будут неправильно интерпретироваться и это будет плохо влиять на процесс игры и работу вообще всего приложения.

Для решения проблемы была использована библиотека <stdint.h>, которая содержит в себе множество целочисленных типов фиксированного размера, что позволяет считывать данные правильно, так как клиент или сервер будут точно знать, что при чтении им отправится строго определённое количество байт.

**Глава 2**

**РЕАЛИЗАЦИЯ ПРОЕКТА**

*2.1 Структура клиента и сервера*

Клиент содержит в себе следующие классы:

1. MainWindow ­­– класс, который занимается отображением главного окна.
2. AreaForShot – класс, который описывает цель для попадания.

Сервер содержит в своей структуре только главную функцию, с которой начинается выполнение программы, - main(), которая в свою очередь порождает два потока DataFromFirstClient() и DataFromSecondClient(), в которых происходит обработка данных соответственно от первого клиента и от второго.

UML – диаграмму классов клиента можно увидеть на рис. 2.1.

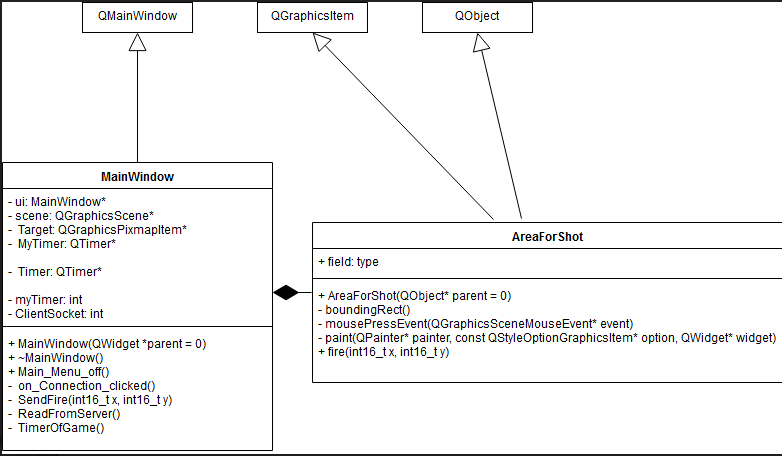


Рисунок 2.1. UML-диаграмма классов клиента

*2.2 Программный интерфейс сервера*

В процессе своей работы клиент формирует сообщения для их последующей обработки на сервере. На стадии боя, когда игроки стреляют, сервер ждёт только одно сообщение – координаты выстрела игрока.

Так как сервер принимает те или иные сообщения, он должен выводить результат их обработки или сообщения о том, что не все данные получены на определённой стадии, например, на стадии подключения, когда второй игрок не подключился к игре, так как сервер не может начать обработку данных без участия второго игрока.

Таким образом, сервер в процессе своей работы формирует и принимает определённый вид сообщения для его общения с клиентами. В самом начале игры, на стадии подключения, сервер ждёт только одного сообщения – дескриптор сокета клиента, с которым он будет в дальнейшем обмениваться данными. В случае если один из игроков подключится раньше другого, то сервер отправит соответствующее сообщение с просьбой подождать противника. По сути это сообщение содержит в себе состояние «Ожидание второго игрока».

После стадии подключения сервер формирует сообщение с следующим состоянием, где игроки должны 3 секунды, чтобы начать игру, и отправляет клиентам сообщение о том, что через 3 секунды начнется бой.

На стадии боя сервер принимает и формирует блоки данных, которые уже были ранее описаны в первой главе в «Протоколе взаимодействия клиента и сервера». Как говорилось ранее, серверу не нужно знать тип сообщения, так как на каждом этапе он точно знает, какие данные ему придут. На этапе боя сервер принимает блок информации, в котором содержится координаты выстрела игрока, координату x и координату y, то есть принимает блок ***Shot***. Когда сервер получает координаты выстрела от игрока, то он начинает обработку этих данных, которые потом отобразятся на клиентах. Таким образом, данные от игрока, который играет, будут обработаны и отправлены клиентам для отображения.

Когда сервер примет данные для обработки, он начнёт формировать сообщение для отображения. В данном случае будет сформирован блок с информацией о результате выстрела: попал ли данный игрок или промахнулся. Для того, чтобы клиент понимал, что это результат выстрела того или иного игрока, в первые два байта он сохраняет тип сообщения, то есть это сообщение с результатом выстрела самого игрока или противника, чтобы отобразить на той или иной игровой сетке, в следующие два байта – результат выстрела, затем опять в следующие два байта – координату x, а в последние – координату y. Таким образом происходит формирование блока ***ResultOfShot***. Затем сформированный блок отправляется клиентам для отображения.

После стадии боя, когда истекло время игры, сервер отправляет сообщения, в которых хранятся результат боя: победил, проиграл или ничья, – то есть сервер формирует сообщение ***StateForClient*** с состоянием «Победа», «Поражение» или «Ничья» для отображения на клиентах. После отправки данных о результате боя сервер прекращает принимать какие – либо сообщения и завершает свою работу.

*2.3 Список основных функций клиента и сервера.*

Основной функционал клиента и сервера приведёны в табл. 2.1 и в табл. 2.2.

**Таблица 2.1**. Список основных методов клиента

|  |  |  |  |
| --- | --- | --- | --- |
| **Название метода** | **Описание** | **Входные данные** | **Выходные данные** |
| void MainWindow::ReadFromServer() | Метод, предназначенный для считывания данных, отправленных с сервера, с сокета клиента. | – | – |
| void MainWindow::SendFire(int16\_t x, int16\_t y) | Слот, предназначенный для отправки координаты выстрела на сервер для его дальнейшей обработки. Первый параметр – позиция выстрела на оси абсцисс, второй параметр – позиция выстрела по оси ординат. | 5 8 | – |
| void MainWindow::TimerOfGame() | Слот, необходимый для обработки таймера, в течении которого идет игра. | – | – |
| void MainWindow::on\_Connection\_clicked() | Слот, предназначенный для подсоединения клиента к серверу. | – | – |

**Таблица 2.2**. Список основных функций сервера.

|  |  |  |  |
| --- | --- | --- | --- |
| **Название функции** | **Описание** | **Входные данные** | **Выходные данные** |
| void DataFromFirstClient(void\* NullData) | Поток, в котором происходит обработка входящих данных от первого клиента. Параметр необходим для корректного функционирования потока и ничего не содержит. | – | – |
| void DataFromSecondClient(void\* NullData) | Поток, в котором происходит обработка входящих данных от второго клиента. Параметр необходим для корректного функционирования потока и ничего не содержит. | – | – |
| void\* Timer(void\* AllThreads) | Поток, в котором происходит обработка игрового времени и результатов игры. Параметр необходим для корректного функционирвоания потока и |  |  |

*2.4 Список системных вызовов операционной системы*

Для реализации проекта были использованы следующие системные вызовы, представленные в табл. 2.3.

**Таблица 2.3**. Список системных вызовов ОС, используемых при реализации

|  |  |
| --- | --- |
| **Название** | **Описание системного вызова** |
| socket() | Служит для создания сокета в адресном пространстве ОС. Имеет три параметра. Первый параметр указывает к какому семейству протоколов относится создаваемый сокет, второй параметр определяет способ передачи данных по сети, а третий параметр определяет протокол, используемый для передачи данных. |
| connect() | Служит для установления соединения со стороны клиента. Имеет три параметра. Первый параметр указывает на сокет клиента, который будет использоваться для обмена данными с сервером, второй параметр содержит указатель на структуру с адресом сервера, а третий параметр содержит длину этой структуры. |
| recv() | Служит для чтения данных. Всего имеет четыре параметра. Первый параметр – дескриптор сокета, через который мы будет считывать данные. Второй параметр – буфер памяти, в который записываются данные. Третий параметр – размер данных в байтах, который мы хотим считать, а четвёртый параметр – набор битовых флагов, управляющих работой функции. |
| send() | Служит для отправки данных. Всего имеет четыре параметра. Первый параметр – дескриптор сокета, через который мы будет отправлять данные. Второй параметр – буфер памяти, который мы будем отправлять. Третий параметр – размер данных в байтах, который мы хотим отправить, а четвёртый параметр – набор битовых флагов, управляющих работой функции. |
| bind() | Явно связывает сокет с адресом в выбранном домене. Первый параметр – дескриптор сокета, который мы хотим привязать к заданному адресу. Второй параметр содержит указатель на структуру с адресом, третий параметр – длину этой структуры. |
| listen() | Прослушивает соединения на сокете. Позволяет показать готовность принимать соединения и задать лимит входящих соединений. Первым параметром указывается дескриптор сокета. Второй параметр определяет максимальную длину очереди входящих сообщений. |
| accept() | Функция создаёт для общения с клиентом новый сокет и возвращает его дескриптор. Первый параметр – слушающий сокет, на который приходят соединения. Второй параметр содержит структуру, в которую записывается адрес сокета клиента, который установил соединение с сервером, а в третий параметр записывается размер структуры. |
| fcntl() | Выполняет управляющие операции над открытыми файлами. Используется в режиме O\_NONBLOCKING для включения неблокирующего режима сокета. |
| pthread\_create() | Функция служит для создания нового потока управления. Всего имеет четыре параметра. Первый параметр ­– указатель на переменную, в которую будет помещен идентификатор созданного потока. Второй параметр – указатель на описатель атрибутов. Если указано NULL, то используются значения по умолчанию. Третий параметр – указатель на функцию, которая будет исполняться в рамках создаваемого потока управления. Четвёртый параметр – указатель на единственный аргумент функции, определённой третьим параметром. |
| pthread\_join() | Функция ожидания завершения потока управления. Имеет всего два параметра. Первый параметр – идентификатор потока управления, до завершения приостановлено выполнение данного потока. Второй параметр – указатель на переменную, в которую будет помещен статус окончания потока. |
| pthread\_cancel() | Функция, которая создает запрос на удаление потока. Имет один параметр – идентификатор потока. |
| pthread\_setcancelstate() | Функция, которая используется для временного запрета/разрешения завершения потока. Параметр PTHREAD\_CANCEL\_ENABLE указывает на разрешение досрочного завершения потока. |

*2.5 Пример работы приложения*

Первое, что увидят игроки при запуске игры – главное меню (рисунок 2.2). На данном этапе отображается задний фон, строка ввода адреса и кнопка для подключения к серверу. На данном этапе игроки просто подключаются к серверу.

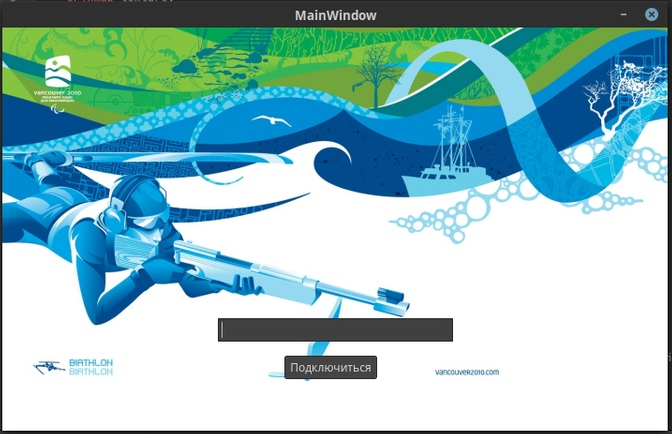


Рисунок 2.2. Главное меню.

Если IP адрес не будет введен, то игрок увидит сообщение (рисунок 2.3).

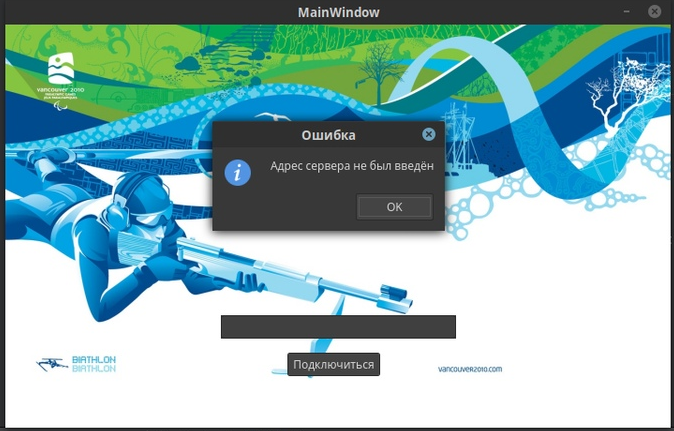


Рисунок 2.3. Сообщение в случае, если IP адрес не введен.

Если IP адресс введен неправильно, или сервер не запущен, то игрок увидит сообщение (рисунок 2.4).

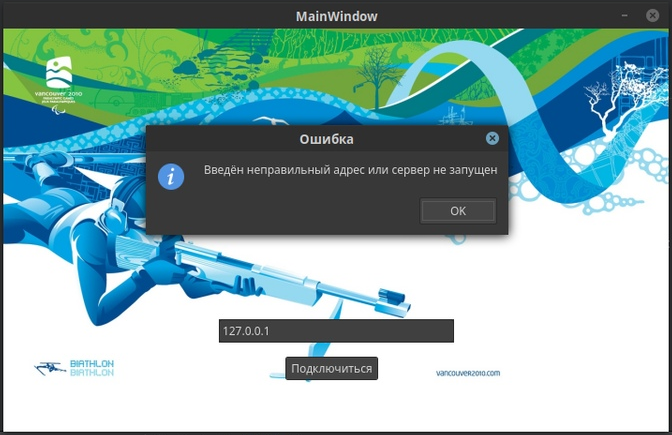


Рисунок 2.4. Сообщение в случае, если IP введен неправильно, или сервер не запущен.

Если какой-то из игроков подключился раньше, то он перейдет на стадию ожидания второго игрока, так как для того, чтобы играть в эту игру, необходимо наличие двух игроков (рисунок 2.5). На данном этапе на клиенте просто выводится сообщение с просьбой подождать второго ирока, пока он не подключился.

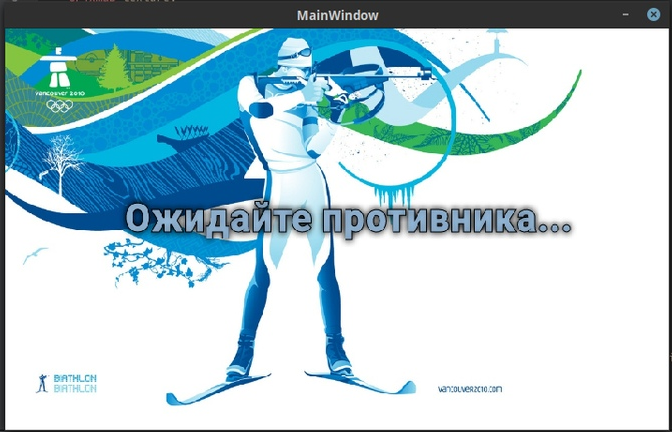


Рисунок 2.5. Ожидание второго игрока.

После того, как все клиенты подключились к серверу, они видят на экране сообщение о том, что игра начнется через 3 секунды (рисунок 2.6).

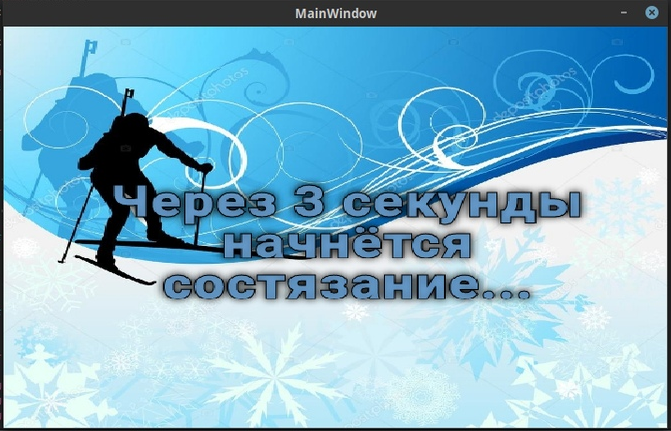


Рисунок 2.6. Сообщение о том, что через 3 секунды начнется состязание.

По истечении 3-х секунд начинается стадия боя (рисунок 2.7). На данном этапе в случайных частях экрана появляются мишени, по которым игроки должны выстреливать. После того, как игрок совершает выстрел, мишень исчезает и появляется на новом месте. Игра длится 30 секунд. Каждый игрок видит, сколько осталось времени и количество очков, которое набрал сам игрок, а также количество очков, которое набирает соперник.

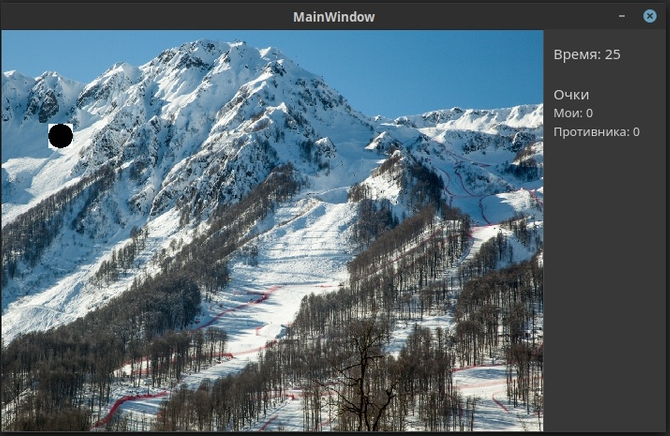


Рисунок 2.7. Стадия боя.

По истечении боя сервер меняет состояния клиентов на “Победа” (рисунок 2.8), “Поражение” (рисунок 2.9) или “Ничья” (рисунок 2.10), в зависимости от исхода игры, чтобы вывести соответствующие сообщения игрокам. После отправки сообщений сервером, клиенты переходят на стадию результата боя. На данном этапе на клиентах выводятся соответствующие сообщения.



Рисунок 2.8. Победа.

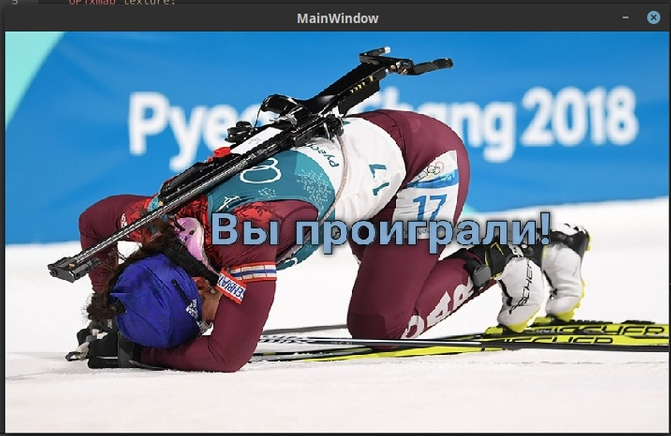


Рисунок 2.9. Поражение.



Рисунок 2.10. Ничья.

**РЕЗУЛЬТАТ РАБОТЫ**

В ходе выполнения данной курсовой работы удалось разработать сетевую игру “Снайпер”.

Клиент и Сервер удалось выполнить в качестве двух отдельных программ, где клиент просто отображает результат обработки Сервера, а Сервер содержит в себе саму логику игры. Удалось реализовать проверку состояний перед обработкой данных.

Данное приложение работает под управлением операционной системы семейства Unix/Linux. Сервер и клиент выполнены на языке программирования С++ с использованием компилятора G++ с помощью сокетов, предоставляемых системной библиотекой <sys/socket.h>.

Таким образом, исходя из задания курсовой работы, результат полностью соответствует заданию.

**ВЫВОД**

Таким образом, в результате выполнения курсовой работы, мы приобрели практические навыки разработки многопользовательских сетевых приложений под управлением операционной системы семейства Unix/Linux.

**ПРИЛОЖЕНИЕ А**

*Листинг А1. Файл mainwindow.h*

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QMessageBox>

#include <QDebug>

#include <QTimer>

#include <QPixmap>

#include <QGraphicsScene>

#include <unistd.h>

#include <areaforshot.h>

//-------------Работа с сетью---------

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <netinet/in.h>

#include <fcntl.h>

//------------------------------------

namespace Ui {

class MainWindow;

}

struct CoorOfTarget

{

int16\_t type;

int16\_t PosX;

int16\_t PosY;

};

struct ResultOfBung

{

int16\_t type;

int16\_t result;

};

struct ScoreOfEnemy

{

int16\_t type;

int16\_t Score;

};

struct Shot

{

int16\_t PosX;

int16\_t PosY;

};

struct StateForClient

{

int16\_t type;

int16\_t state;

};

enum States { WaitingOfConnection, WaitingOfCombat, Win, Lose, Combat, DeadHeat };

enum Msg\_type { result\_of\_shot, state\_for\_client, coor\_of\_taget, score\_of\_enemy };

enum ResultOfShot { not\_hit, hit };

class MainWindow : public QMainWindow

{

Q\_OBJECT

public:

explicit MainWindow(QWidget \*parent = 0);

~MainWindow();

void Main\_Menu\_off();

private slots:

void on\_Connection\_clicked();

void SendFire(int16\_t x, int16\_t y);

void ReadFromServer();

void TimerOfGame();

private:

Ui::MainWindow\* ui;

QGraphicsScene\* scene;

QGraphicsPixmapItem\* Target;

QTimer\* MyTimer;

QTimer\* Timer;

int myTimer = 30;

int ClientSocket;

};

#endif // MAINWINDOW\_H

*Листинг А2. Файл areaforshot.h*

#ifndef AREAFORSHOT\_H

#define AREAFORSHOT\_H

#include <QGraphicsItem>

#include <QGraphicsSceneMouseEvent>

#include <QCursor>

#include <stdint.h>

#define WidthOfFrame 580

#define HeightOfFrame 415

class AreaForShot : public QObject, public QGraphicsItem

{

Q\_OBJECT

public:

explicit AreaForShot(QObject\* parent = 0);

private:

QRectF boundingRect() const;

void mousePressEvent(QGraphicsSceneMouseEvent\* event);

void paint(QPainter\* painter, const QStyleOptionGraphicsItem\* option, QWidget\* widget);

signals:

void fire(int16\_t x, int16\_t y); //Сигнал отправляет координаты x и y в слот SendFire(), описанный в mainwindow.cpp, для отправки на сервер.

public slots:

private slots:

};

#endif // AREAFORSHOT\_H

*Листинг А3. Файл mainwindow.cpp*

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#define WidthOfFrame 710

#define HeightOfFrame 430

MainWindow::MainWindow(QWidget \*parent) :

QMainWindow(parent),

ui(new Ui::MainWindow)

{

ui->setupUi(this);

ui->MainPicture->setFixedSize(QSize(WidthOfFrame, HeightOfFrame));

setFixedSize(QSize(WidthOfFrame, HeightOfFrame));

ui->Frame->setFixedSize(QSize(WidthOfFrame, HeightOfFrame));

ui->Frame->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

ui->Frame->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

ui->ScorePlayers->hide();

scene = new QGraphicsScene;

MyTimer = new QTimer;

Timer = new QTimer;

MyTimer->setInterval(50);

}

void MainWindow::Main\_Menu\_off()

{

ui->MainPicture->hide();

ui->InputLine->hide();

ui->Connection->hide();

}

MainWindow::~MainWindow()

{

delete ui;

}

void MainWindow::on\_Connection\_clicked()

{

QString IPaddress = ui->InputLine->text();

if (IPaddress == NULL)

{

QMessageBox::information(this, "Ошибка", "Адрес сервера не был введён");

}

else

{

QByteArray tmp = IPaddress.toLatin1();

const char\* IP = tmp.data();

ClientSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

struct sockaddr\_in ServAddr;

ServAddr.sin\_family = AF\_INET;

ServAddr.sin\_port = htons(25678);

inet\_pton(AF\_INET, IP, &(ServAddr.sin\_addr));

if (::connect(ClientSocket, (struct sockaddr\*) &ServAddr, sizeof(ServAddr)) != -1)

{

connect(MyTimer, &QTimer::timeout, this, &MainWindow::ReadFromServer);

fcntl(ClientSocket, F\_SETFL, O\_NONBLOCK);

MyTimer->start();

}

else

{

QMessageBox::information(this, "Ошибка", "Введён неправильный адрес или сервер не запущен");

}

}

}

void MainWindow::SendFire(int16\_t x, int16\_t y)

{

Shot ShotForServer;

ShotForServer.PosX = x;

ShotForServer.PosY = y;

send(ClientSocket, &ShotForServer, sizeof(Shot), MSG\_NOSIGNAL);

}

void MainWindow::TimerOfGame()

{

--myTimer;

qDebug() << "Осталось " << myTimer;

if (myTimer >= 0)

{

ui->Timer->setText("Время: " + QString::number(myTimer));

if (myTimer == 0)

{

delete Timer;

}

}

}

void MainWindow::ReadFromServer()

{

int16\_t buffer[3];

if (recv(ClientSocket, &buffer, sizeof(int16\_t), MSG\_NOSIGNAL) > 0)

{

Msg\_type type = static\_cast<Msg\_type>(buffer[0]);

qDebug() << "type = " << type;

if (type == state\_for\_client)

{

recv(ClientSocket, &buffer[1], sizeof(int16\_t), MSG\_NOSIGNAL);

States state = static\_cast<States>(buffer[1]);

qDebug() << "state = " << state;

if (state == WaitingOfConnection)

{

Main\_Menu\_off();

QPixmap frame;

frame.load(":/img/WaitingOfConnection.jpg");

frame = frame.scaled(WidthOfFrame, HeightOfFrame);

scene->addPixmap(frame);

ui->Frame->setScene(scene);

ui->Frame->show();

}

else if (state == WaitingOfCombat)

{

Main\_Menu\_off();

QPixmap frame;

frame.load(":/img/WaitingOfCombat.jpg");

frame = frame.scaled(WidthOfFrame, HeightOfFrame);

scene->addPixmap(frame);

ui->Frame->setScene(scene);

ui->Frame->show();

}

else if (state == Combat)

{

qDebug() << "It's a BATTLE, baby!\n";

QPixmap frame;

frame.load(":/img/BattleArea.jpg");

frame = frame.scaled(580, HeightOfFrame);

scene->addPixmap(frame);

AreaForShot\* area = new AreaForShot;

area->setPos(0, 0);

connect(area, &AreaForShot::fire, this, &MainWindow::SendFire);

scene->addItem(area);

ui->Frame->setFixedSize(QSize(580, HeightOfFrame));

ui->Frame->setScene(scene);

ui->Frame->show();

connect(Timer, &QTimer::timeout, this, &MainWindow::TimerOfGame);

Timer->setInterval(1000);

Timer->start();

ui->Timer->setText("Время: 30");

ui->ScorePlayers->show();

ui->Player1->setText("Мои: 0");

ui->Player2->setText("Противника: 0");

}

else if (state == Win)

{

QPixmap win;

win.load(":/img/Win.jpg");

win = win.scaled(WidthOfFrame, HeightOfFrame);

scene->clear();

scene->addPixmap(win);

ui->Frame->setScene(scene);

ui->Frame->setFixedSize(WidthOfFrame, HeightOfFrame);

ui->Frame->show();

ui->Timer->hide();

ui->ScorePlayers->hide();

ui->Player1->hide();

ui->Player2->hide();

delete MyTimer;

shutdown(ClientSocket, SHUT\_RDWR);

::close(ClientSocket);

qDebug() << "Я выиграл";

}

else if (state == Lose)

{

QPixmap lose;

lose.load(":/img/Lose.jpg");

lose = lose.scaled(WidthOfFrame, HeightOfFrame);

scene->clear();

scene->addPixmap(lose);

ui->Frame->setScene(scene);

ui->Frame->setFixedSize(WidthOfFrame, HeightOfFrame);

ui->Frame->show();

ui->Timer->hide();

ui->ScorePlayers->hide();

ui->Player1->hide();

ui->Player2->hide();

delete MyTimer;

shutdown(ClientSocket, SHUT\_RDWR);

::close(ClientSocket);

qDebug() << "Блiн, проiграл";

}

else if (state == DeadHeat)

{

QPixmap Draw;

Draw.load(":/img/DeadHeat.jpg");

Draw = Draw.scaled(WidthOfFrame, HeightOfFrame);

scene->clear();

scene->addPixmap(Draw);

ui->Frame->setScene(scene);

ui->Frame->setFixedSize(WidthOfFrame, HeightOfFrame);

ui->Frame->show();

ui->Timer->hide();

ui->ScorePlayers->hide();

ui->Player1->hide();

ui->Player2->hide();

delete MyTimer;

shutdown(ClientSocket, SHUT\_RDWR);

::close(ClientSocket);

qDebug() << "Ничья";

}

}

else if (type == result\_of\_shot)

{

recv(ClientSocket, &buffer[1], 2 \* sizeof(int16\_t), MSG\_NOSIGNAL);

ResultOfShot result = static\_cast<ResultOfShot>(buffer[1]);

if (result == hit)

{

qDebug() << "Я попал!";

ui->Player1->setText("Мои: " + QString::number(buffer[2]));

}

else if (result == not\_hit)

{

qDebug() << "Молоко!";

}

delete Target;

}

else if (type == coor\_of\_taget)

{

recv(ClientSocket, &buffer[1], 2 \* sizeof(int16\_t), MSG\_NOSIGNAL);

qDebug() << "PosX = " << buffer[1];

qDebug() << "PosY = " << buffer[2];

QPixmap target;

target.load(":/img/Target.jpg");

target = target.scaled(70, 70);

Target = scene->addPixmap(target);

Target->setPos(buffer[1], buffer[2]);

}

else if (type == score\_of\_enemy)

{

recv(ClientSocket, &buffer[1], sizeof(int16\_t), MSG\_NOSIGNAL);

ui->Player2->setText("Противника: " + QString::number(buffer[1]));

}

}

}

*Листинг А4. Файл areaforshot.cpp*

#include "areaforshot.h"

AreaForShot::AreaForShot(QObject \*parent) : QObject(parent)

{

QPixmap texture;

texture.load(":/img/cursortarget.png");

texture = texture.scaled(43, 43);

QCursor MyCursor = QCursor(texture);

this->setCursor(MyCursor);

}

void AreaForShot::mousePressEvent(QGraphicsSceneMouseEvent \*event)

{

/\*

\* посылаем сигнал о том, что был произведён выстрел,

\* и отправляем координаты выстрела в слот SendFire(), описанный в файле mainwindow.cpp,

\* для дальнейшей отправки на сервер.

\*/

emit fire(event->pos().x(), event->pos().y());

}

QRectF AreaForShot::boundingRect() const

{

//Формируем квадрат поля противника для вычисления координат.

return QRectF(0, 0, WidthOfFrame, HeightOfFrame);

}

void AreaForShot::paint(QPainter\* painter, const QStyleOptionGraphicsItem\* option, QWidget\* widget)

{

Q\_UNUSED(painter);

Q\_UNUSED(option);

Q\_UNUSED(widget);

}

*Листинг А5. Файл main.cpp*

#include "mainwindow.h"

#include <QApplication>

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

{

QApplication a(argc, argv);

MainWindow w;

w.show();

return a.exec();

}

*Листинг А6. Файл server.cpp*

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <unistd.h>

#include <stdint.h>

#include <pthread.h>

#include <iostream>

#include <clocale>

#include <fcntl.h>

#include <time.h>

#include <cmath>

#define TimeOfGame 30

#define MaxX 480

#define MaxY 330

#define Radius 35

using namespace std;

enum states { WaitingOfConnection, WaitingOfCombat, Win, Lose, Combat, DeadHeat };

enum Msg\_type { result\_of\_shot, state\_for\_client, coor\_for\_target, score\_of\_enemy };

enum ResultOfShot { not\_hit, hit };

struct ResultOfBung

{

int16\_t type = static\_cast<int16\_t>(result\_of\_shot);

int16\_t result;

int16\_t ScoreOfPlayer;

};

struct ScoreOfEnemy

{

int16\_t type = static\_cast<int16\_t>(score\_of\_enemy);

int16\_t Score;

};

struct Message

{

int16\_t type = static\_cast<int16\_t>(coor\_for\_target);

int16\_t PosX;

int16\_t PosY;

};

struct Shot

{

int16\_t PosX;

int16\_t PosY;

};

struct StateForClient

{

int16\_t type = static\_cast<int16\_t>(state\_for\_client);

int16\_t state;

};

long start;

int FirstPlayer, SecondPlayer;

int ScoreOfFP = 0, ScoreOfSP = 0;

states stateOfFirstPlayer = WaitingOfConnection, stateOfSecondPlayer = WaitingOfConnection;

struct Threads

{

pthread\_t\* firstThread;

pthread\_t\* secondThread;

};

void\* Timer(void\* AllThreads)

{

Threads\* GameThreads = static\_cast<Threads\*>(AllThreads);

while (time(NULL) <= start + TimeOfGame);

cout << "Время вышло" << endl;

pthread\_cancel(\*GameThreads->firstThread);

pthread\_cancel(\*GameThreads->secondThread);

sleep(3);

StateForClient stateForFP, stateForSP;

if (ScoreOfFP > ScoreOfSP)

{

stateForFP.state = static\_cast<int16\_t>(Win);

stateForSP.state = static\_cast<int16\_t>(Lose);

}

else if (ScoreOfFP < ScoreOfSP)

{

stateForFP.state = static\_cast<int16\_t>(Lose);

stateForSP.state = static\_cast<int16\_t>(Win);

}

else if (ScoreOfFP == ScoreOfSP)

{

stateForFP.state = stateForSP.state = static\_cast<int16\_t>(DeadHeat);

}

send(FirstPlayer, &stateForFP, sizeof(StateForClient), MSG\_NOSIGNAL);

send(SecondPlayer, &stateForSP, sizeof(StateForClient), MSG\_NOSIGNAL);

pthread\_exit(0);

}

void\* DataFromFirstClient(void\* NullData)

{

pthread\_setcancelstate(PTHREAD\_CANCEL\_ENABLE, NULL);

StateForClient State;

State.state = static\_cast<int16\_t>(Combat);

send(FirstPlayer, &State, sizeof(State), MSG\_NOSIGNAL);

while (1)

{

cout << "ScoreFP = " << ScoreOfFP << endl;

Message CoorForTarget;

CoorForTarget.PosX = rand() % MaxX;

CoorForTarget.PosY = rand() % MaxY;

cout << "x1 = " << CoorForTarget.PosX << ", y1 = " << CoorForTarget.PosY << endl;

send(FirstPlayer, &CoorForTarget, sizeof(CoorForTarget), MSG\_NOSIGNAL);

Shot PlayerShot;

recv(FirstPlayer, &PlayerShot, sizeof(Shot), MSG\_NOSIGNAL);

cout << "X1 = " << PlayerShot.PosX << ", Y1 = " << PlayerShot.PosY << endl;

int x2 = (PlayerShot.PosX - (CoorForTarget.PosX + Radius)) \* (PlayerShot.PosX - (CoorForTarget.PosX + Radius));

int y2 = (PlayerShot.PosY - (CoorForTarget.PosY + Radius)) \* (PlayerShot.PosY - (CoorForTarget.PosY + Radius));

ResultOfBung Result;

if (sqrt(x2 + y2) < Radius)

{

++ScoreOfFP;

Result.ScoreOfPlayer = ScoreOfFP;

Result.result = static\_cast<int16\_t>(hit);

send(FirstPlayer, &Result, sizeof(ResultOfBung), MSG\_NOSIGNAL);

ScoreOfEnemy score;

score.Score = ScoreOfFP;

send(SecondPlayer, &score, sizeof(ScoreOfEnemy), MSG\_NOSIGNAL);

}

else

{

Result.result = static\_cast<int16\_t>(not\_hit);

send(FirstPlayer, &Result, sizeof(ResultOfBung), MSG\_NOSIGNAL);

}

}

}

void\* DataFromSecondClient(void\* NullData)

{

pthread\_setcancelstate(PTHREAD\_CANCEL\_ENABLE, NULL);

StateForClient State;

State.state = static\_cast<int16\_t>(Combat);

send(SecondPlayer, &State, sizeof(State), MSG\_NOSIGNAL);

while (1)

{

cout << "ScoreSP = " << ScoreOfSP << endl;

Message CoorForTarget;

CoorForTarget.PosX = rand() % MaxX;

CoorForTarget.PosY = rand() % MaxY;

send(SecondPlayer, &CoorForTarget, sizeof(CoorForTarget), MSG\_NOSIGNAL);

cout << "x2 = " << CoorForTarget.PosX << ", y2 = " << CoorForTarget.PosY << endl;

Shot PlayerShot;

recv(SecondPlayer, &PlayerShot, sizeof(Shot), MSG\_NOSIGNAL);

cout << "X2 = " << PlayerShot.PosX << ", Y2 = " << PlayerShot.PosY << endl;

int x2 = (PlayerShot.PosX - (CoorForTarget.PosX + Radius)) \* (PlayerShot.PosX - (CoorForTarget.PosX + Radius));

int y2 = (PlayerShot.PosY - (CoorForTarget.PosY + Radius)) \* (PlayerShot.PosY - (CoorForTarget.PosY + Radius));

ResultOfBung Result;

if (sqrt(x2 + y2) < Radius)

{

++ScoreOfSP;

Result.result = static\_cast<int16\_t>(hit);

Result.ScoreOfPlayer = ScoreOfSP;

send(SecondPlayer, &Result, sizeof(ResultOfBung), MSG\_NOSIGNAL);

ScoreOfEnemy score;

score.Score = ScoreOfSP;

send(FirstPlayer, &score, sizeof(ScoreOfEnemy), MSG\_NOSIGNAL);

}

else

{

Result.result = static\_cast<int16\_t>(not\_hit);

send(SecondPlayer, &Result, sizeof(ResultOfBung), MSG\_NOSIGNAL);

}

}

}

int main()

{

setlocale(LC\_ALL, "Russian");

StateForClient stateFP, stateSP;

//---------------Создание слушающего сокета---------------------

int MasterSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

struct sockaddr\_in MasterAddr;

MasterAddr.sin\_family = AF\_INET;

MasterAddr.sin\_port = htons(25678);

MasterAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

bind(MasterSocket, (struct sockaddr\*) (&MasterAddr), sizeof(MasterAddr));

listen(MasterSocket, 1);

//-------------------------------------------------------------

//----------------Соединение клиентов---------------------

FirstPlayer = accept(MasterSocket, 0, 0);

if (FirstPlayer > 0)

{

printf("Первый игрок подключился!\n");

stateFP.state = static\_cast<int16\_t>(stateOfFirstPlayer);

send(FirstPlayer, &stateFP, sizeof(stateFP), MSG\_NOSIGNAL);

}

else

{

printf("Соединение провалено!\n");

return 0;

}

cout << "После подключения первого игрока: " << stateOfFirstPlayer << " " << stateOfSecondPlayer << "\n";

SecondPlayer = accept(MasterSocket, 0, 0);

if (SecondPlayer > 0)

{

printf("Второй игрок подключился!\n");

stateOfFirstPlayer = stateOfSecondPlayer = WaitingOfCombat;

stateFP.state = stateOfFirstPlayer;

stateSP.state = stateOfSecondPlayer;

send(FirstPlayer, &stateFP, sizeof(stateFP), MSG\_NOSIGNAL);

send(SecondPlayer, &stateSP, sizeof(stateSP), MSG\_NOSIGNAL);

}

else

{

printf("Соединение провалено!\n");

return 0;

}

//------------------------------------------------------

cout << "После подключения всех игроков: " << stateOfFirstPlayer << " " << stateOfSecondPlayer << "\n";

//----------------------Процесс игры----------------------

void\* Null = NULL;

sleep(3);

srand(time(NULL));

pthread\_t FirstThread, SecondThread, TimerThread;

Threads MyThreads;

MyThreads.firstThread = &FirstThread;

MyThreads.secondThread = &SecondThread;

start = time(NULL);

pthread\_create(&FirstThread, 0, DataFromFirstClient, Null);

pthread\_create(&SecondThread, 0, DataFromSecondClient, Null);

pthread\_create(&TimerThread, 0, Timer, &MyThreads);

pthread\_join(TimerThread, 0);

//--------------------------------------------------------

shutdown(MasterSocket, SHUT\_RDWR);

close(MasterSocket);

cout << "Игра завершилась" << endl;

return 0;

}