## ВВЕДЕНИЕ

В настоящее время многопользовательские игры занимает отдельную нишу в игровой индустрии. Взаимодействие с настоящими игроками, а не с искусственным интеллектом, делает онлайн-игры очень популярными, а сейчас и доступными благодаря высокому уровню компьютеризации населения. Например онлайн-игра *World Of Tanks* от компании *Wargaming* насчитывает около миллиона игроков [1].

В многоуровневых приложениях презентационный сервис, прикладная логика, и сервис данных отделены друг от друга. Логические уровни не обязательно должны соответствовать их физическому местонахождению в сети. В многоуровневом приложении клиенту предоставляется только пользовательский интерфейс. Прикладная логика реализуется промежуточным уровнем, который размещается между пользовательским интерфейсом и системой хранения данных. Это и позволяет выделять каждый тип сервиса в отдельный уровень.

Преимущество такой модели в том, что прикладная логика сосредоточена в одном месте и может быть легко модифицирована. Презентационный уровень отвечает фактически только за взаимодействие  с пользователем. В многоуровневом приложении клиент не обращается к системе хранения данных напрямую. Разделение всех сервисов позволяет модифицировать любой уровень системы, не внося изменения в другие. **Локальное приложение** представляет собой программу, установленную на жестком диске компьютера и использующую для работы только собственные ресурсы компьютера, на котором она запущена, поэтому изменения одного уровня такой системы зачастую влечёт изменение других уровней.

## 1 АНАЛИЗ ТЕХНОЛОГИЙ РАЗРАБОТКИ

## СЕТЕВЫХ ПРИЛОЖЕНИЙ

## 

## Реализация с помощью *AForge.NET*

*AForge.NET* является открытым *C#* фреймворком, предназначенным для разработчиков и исследователей в области компьютерного зрения и искусственного интеллекта – обработка изображений, нейронные сети, генетические алгоритмы, нечеткая логика, машинного обучения, робототехники и т.д.

Каркас состоит из набора модулей, которые выполняют свои задачи:

* *AForge.Imaging* – модуль для обработки изображений;
* *AForge.Vision* – модуль компьютерного зрения;
* *AForge.Video* – набор библиотек для обработки видео;
* *AForge.Neuro* – модуль для работы с нейронными сетями;
* *AForge.Robotics* – модуль, обеспечивающий поддержку некоторых наборов робототехники;
* *AForge.MachineLearning* – модуль машинного обучения;

Работа по совершенствованию фреймворка находится в непрерывном процессе, что означает, что новые функции и пространство имен постоянно добавляются. Для получения новой информации о фреймворке и его модулях, можно зайти на официальный сайт или в репозиторий фреймворка.

*AForge.NET* содержит не только различные готовые библиотеки, но и содержит большое количество примеров приложений, которые демонстрируют работу этого фреймворка. Также доступна документация, предоставляемая в формате *HTML Help.* Также документация доступна в онлайн режиме.

* 1. **Реализация с помощью** ***Accord.NET***

Существует два широко известных и доказавших свою эффективность подхода к созданию структуры сетевой игры: с использованием авторитарного сервера *Authoritative Server* или с использованием не-авторитарного сервера *Non-Authoritative Server*. Оба подхода основываются на связи сервера с клиентами и обменом информацией между ними. Оба подхода также обеспечивают приватность конечных пользователей, так как клиенты никогда не соединяются напрямую друг с другом и не открывают свой IP адрес другим клиентам.

Использование авторитарного сервера требует от сервера просчёта и выполнения всех действий, затрагивающих мир игры, применение правил игры и обработки ввода от игроков-клиентов. Каждый клиент высылает введенную информацию (в форме нажатия клавиш или запрашиваемых действий) на сервер и постоянно получает от сервера текущее состояние игры. Клиент не может самостоятельно вносить какие-либо изменения в состояние игры. Вместо этого, он сообщает серверу, что конкретно он хочет сделать, сервер обрабатывает этот запрос и отвечает клиенту, объясняя что получилось в результате запроса.

Принципиально важно, что существует разница между тем, что игрок хочет сделать и тем, что в действительности происходит. Это позволяет серверу принимать запросы от каждого клиента, после чего решать, как изменить состояние игры.

Преимущество такого подхода в том, что он значительно усложняет клиентам возможность использования нечестных приёмов. Например, у клиента нет возможности обмануть сервер (и, соответственно, остальных клиентов), что враг был убит, поскольку клиент не принимает таких решений. Он может только лишь сказать серверу, что оружие выстрелило и, начиная с этого момента, только сервер определяет, было ли совершено убийство или нет.

Другим примером применения авторитарного сервера может быть многопользовательская игра, основанная на физике. Если бы каждый клиент просчитывал физику самостоятельно, то незначительные отличия в просчете физики между клиентами постепенно привели бы к рассинхронизации клиентов друг с другом. Однако, если симуляция всех физических объектов управляется на центральном сервере, обновленные состояния игры могут быть высланы клиентам, гарантируя их согласованность.

Возможным недостатком системы с авторитарным сервером можно назвать время, которое требуется для пересылки сигнала через сеть. Если игрок нажал кнопку, чтобы двинуться вперёд, а ответ от сервера о возможности такого движения приходит через десятую долю секунды, то такая задержка будет ощутима для игрока. Одним из возможных решений этой проблемы будет использование так называемых прогнозов движения на стороне клиента *client-side prediction*. Суть этой технологии в том, чтобы позволить клиенту обновлять локальную версию состояния игры, но при этом иметь возможность получать уточняющие данные со стороны сервера по мере необходимости. Как правило, это применяется только для простых действий в игре и не должно оказывать влияние на значительные изменения логики игры. Например, было бы неразумно сообщать игроку, что враг был убит только лишь затем, чтобы сервер в дальнейшем переписал это решение.

Поскольку прогнозы на стороне клиента относятся к продвинутым техникам, мы не будем пытаться их объяснить в этом руководстве, однако вам доступны множество книг и сетевых ресурсов для дальнейшего изучения.

Авторитарный сервер обрабатывает намного больше информации, чем неавторитарный. Если от сервера не требуется управления всеми изменениями состояния игры, значительная часть нагрузки может быть распределена между клиентами.

Неавторитарный сервер не контролирует результаты ввода каждого игрока. Клиенты самостоятельно отслеживают введенные игроком действия и игровую логику локально, после чего высылают результат определенного действия на сервер. После этого сервер синхронизирует все совершенные действия с состоянием мира. Это легче реализовать с точки зрения архитектуры, так как сервер отвечает только лишь за передачу сообщений между клиентами, не делая никаких дополнительных вычислений, которые делают клиенты.

Не требуется никаких специальных методов прогнозирования, так как клиенты сами обрабатывают всю физику и события, после чего отсылают результат на сервер. Они являются хозяевами этих объектов и единственными агентами с разрешением на пересылку по сети локальных изменений этих объектов.

Встроенные в *Unity* способы работы с сетями поддерживают:

1. Создание серверов и подключение клиентов;

2. Обмен данными между подключенными клиентами;

3. Определение, какой игрок, управляет какими объектами;

4. Доступ через различные конфигурации сети.

Перед тем как начать играть в сетевую игру, необходимо определить другие компьютеры, с которым вы будете обмениваться данными. Чтобы это сделать, необходимо создать сервер. Это может быть как машина, на которой запущена игра, так и отдельная выделенная машина, не принимающая участия в игре. Чтобы создать сервер, используется метод *Network.InitializeServer*. Если необходимо подсоединиться к уже существующему серверу как клиент, вызывается метод *Network.Connect.*

Связь игроков между сервером и отображение объектов сторонних пользователей происходит через компонент *NetworkView*.

*Network View* (просмотр сети) это компонент, который отправляет данные через сеть. Компонент *Network View* даёт вашим объектам *GameObject* возможность отправлять данные, используя удаленный вызов процедур *RPC* (*Remote Procedure Calls*) или синхронизацию состояний *State Synchronization.* Способ, которым вы используете *Network View* будет определять, как будут работать сетевые взаимодействия вашей игры. *Network View* имеют несколько вариантов.

Удаленные вызовы процедур (*Remote Procedure Calls*) это функции, объявленные в скриптах, прикрепленных к *GameObject*, который содержит *NetworkView*. *NetworkView* должен указывать на скрипт, содержащий *RPC* функцию. После этого, *RPC* функция может быть вызвана из любого скрипта в этом *GameObject*. Удаленный вызов процедуры используется для вызова на другом компьютере через сеть, хотя «сеть» может подразумевать канал для сообщений между клиентом и сервером, когда они оба запущены на одном компьютере. Клиенты могут отправлять *RPC* на сервер, а сервер может отправлять *RPC* на один или несколько клиентов. Чаще всего, они используются для действий, которые происходят нечасто. Например, если клиент нажимает на кнопку, чтобы открыть дверь, он может отправить на сервер *RPC*, что дверь была открыта. После этого, сервер может отправить другой *RPC* всем клиентам, вызывая их локальные функции, открывающие эту же дверь. Они используются для управления и выполнения отдельных событий.

Синхронизация состояний это постоянный обмен данными между всеми клиентами игры. Таким способом позиция игрока может быть синхронизирована со всем клиентами, так что будет казаться, что он управляется локально, когда данные в действительности доставляются через сеть. Для синхронизации состояний внутри объекта *GameObject* вам просто надо добавить компонент *NetworkView* на этот объект и объяснить ему, за чем наблюдать. Наблюдаемые данные после этого синхронизируются со всеми клиентами в игре.

*State Synchronization* (Синхронизация Состояний) используется для обмена данными, которые постоянно меняются. Хорошим примером будут координаты игрока в игре. Игрок постоянно движется, бегает, прыгает и т.д. Все остальные игроки в сети, даже те, кто не контролируют этого игрока, должны знать где именно он сейчас находится и что он делает. При помощи постоянной ретрансляции данных о позиции игрока, игра может точно отображать эту позицию остальным игрокам.

Этот тип данных постоянно и часто передаётся через сеть. Поскольку для этих данных важно время и требуется время для передачи сигнала по сети от одного компьютера к другому, важно снижать количество передаваемой информации, насколько это возможно. Проще говоря, *State Synchronization*, как правило, требует гораздо большей пропускной способности сети, поэтому вам следует стремиться по возможности минимизировать сетевой трафик.

*Network.Instantiate* позволяет создавать экземпляры префабов на всех клиентах естественно и просто. По сути, это вызов функции *Instantiate*, но он выполняет создание экземпляров на всех клиентах.

Внутренне *Network.Instantiate* это простой вызов *RPC*, который выполняется на всех клиентах (также локально). Он распределяет *NetworkViewID* и назначает его созданной копии префаба, что гарантирует его правильную синхронизацию среди всех клиентов.

Поскольку сетевые взаимодействия могут быть относительно медленными по сравнению с другими аспектами игры, важно уменьшить их до минимума. Следовательно, очень важно решить, каким объемом данных обмениваться и как часто производить этот обмен.

Величина используемого сетевого трафика в значительной мере зависит от того, используете ли вы *Unreliable* или *Reliable Delta Compression* режим для синхронизации данных (режим выставляется в компоненте *Network View*).

Компонент *NetworkView* имеет два режима передачи данных: unreliable и reliable delta compressed.

В режиме *Unreliable*, синхронизируемый объект будет целиком передаваться на каждой итерации цикла сетевого обновления. Частота этого обновления определяется величиной *Network.sendRate*, которая по умолчанию равна 15 обновлениям в секунду. Режим *Unreliable* гарантирует частые обновления, но любой потерянный или задерживающийся пакет будет просто игнорироваться. Часто это лучший режим синхронизации для объектов, меняющих своё состояние с большой частотой и когда эффект от утерянного обновления будет заметен очень короткое время. Однако, следует помнить об объемах данных, которые могут передаваться в каждом обновлении. Например, синхронизация компонента *Transform* включает передачу девяти переменных типа *float* (три *Vector3* с тремя *float* в каждом), что равнозначно 36 байтам в каждом обновлении. Если к серверу подключено восемь клиентов и используется частота обновления по умолчанию, то он будет принимать 4.320 Кбайт в секунду (8\*36\*15) или 34.6 Кбит/с и передавать 30.2 Кбайт/с (8\*7\*36\*15) или 242 Кбит/с. Вы можете значительно уменьшить потребление сетевого трафика, уменьшая частоту обновления, но значение по умолчанию 15 обычно подходит играм с быстрыми действиями.

В режиме *Reliable Delta Compressed* гарантируется, что данные будут надёжно отправлены и прибудут в правильном порядке. Если пакеты теряются, они отправляются заново, а если они приходят в неправильном порядке, то они будут помещаться в буфер до тех пор, пока не придут все пакеты последовательности. Хотя это гарантирует, что переданные данные будут получены корректно, ожидания и повторные отправления ведут к увеличению использования сетевого трафика. Однако, данные также дельта-компрессированы (сжаты по критерию изменения, *delta compressed*), что означает, что будут передаваться только отличия по сравнению с прошлым состоянием. Если состояние такое же, ничего не отправляется. Выгоды от дельта-компрессии, таким образом, зависят от того, как сильно изменяется состояние и в каких свойствах [3].

* 1. **Реализация с помощью *TCP/UDP* сокетов**

*UDP* – один из ключевых элементов [*TCP/IP*](https://ru.wikipedia.org/wiki/TCP/IP). С *UDP* компьютерные приложения могут посылать сообщения (в данном случае

браузер не поддерживает технологию, приложение не сможет функционировать. *Unity* же позволяет «подняться» на уровень выше сетевой абстракции и не беспокоиться о деталях сетевого взаимодействия и синхронизации, а также использовать встроенные компоненты для создания 3*D-*графики. Реализация приложения с помощью *TCP/UDP* позволяет выиграть в скорости передачи данных и минимизировать сетевой трафик. Для разработки приложения «*Tanks*» была выбрана технология *TCP/UDP* сокетов.

* 1. **Выбор языка программирования и используемого фреймворка**

При выборе языка программирования учитывались такие факторы, как:

* Возможность быстрого создания удобного пользовательского интерфейса;
* Наличие большой документации, которая позволит быстро изучить область, необходимую для выполнения курсового проекта;
* Наличие удобных фреймворков, необходимых для выполнения курсового проекта;

Выбор языка программирования будет проводится из трех наиболее популярных языков программирования:

* *C#*;
* *Java*;
* *С++*;

Сравнение языков по вышеописанным факторам:

**Возможность быстрого создания удобного пользовательского интерфейса**

C# имеет специальные технологии для создания пользовательского интерфейса приложений:

* Window Forms;
* WPF;

Язык C++ не предназначен для создания пользовательского интерфейса, так что он нам не подойдёт.

Java имеет различные возможности создания графического интерфейса, но менее удобные, чем в c#.

**Наличие большой документации, которая позволит быстро изучить область, необходимую для выполнения курсового проекта**

Все выше перечисленные языки имеют обширную документацию и не менее обширное сообщество.

**Наличие удобных фреймворков, необходимых для выполнения курсового проекта**

C# имеет несколько фреймворков для работы с компьютерным зрением и созданием интерактивных приложений. Так же эти фреймворки имеют хорошую документацию, которая позволит быстрее получить все необходимые знания для выполнения курсового проекта.

Язык Java не имеет фреймворков, для создания интерактивных приложений.

Язык C++ имеет фреймворк OpenCV, с помощью которого можно создавать интерактивные приложения,

Таблица результатов сравнения языков программирования по выбранным факторам

|  |  |  |  |
| --- | --- | --- | --- |
|  | *C#* | *Java* | *C++* |
| Возможность быстрого создания удобного пользовательского интерфейса | + | +- | - |
| Наличие большой документации, которая позволит быстро изучить область, необходимую для выполнения курсового проекта | + | + | + |
| Наличие удобных фреймворков, необходимых для выполнения курсового проекта | + | - | + |

Таблица 1.1 Результаты сравнения языков программирования

Из таблицы 1.1 видно, что наиболее подходящий язык для выполнения курсового проекта является C#.

**Выбор фреймворка для создания интерактивного приложения на языке C#**

Язык C# имеет несколько фреймворков для разработки интерактивного приложения:

* Accord.Net
* Aforge.Net

Данные фреймворке описаны выше в этой главе.

## Для выполнения курсового проекта мы будем использовать оба этих фреймворка. В Aforge.Net будем использовать Aforge.Vision и в Accord.Net будет использоваться Accord.Vision.Detection.

**2 АРХИТЕКТУРА СЕТЕВОГО ПРИЛОЖЕНИЯ «*Tanks*»**

**2.1 Общая архитектура**

Проектирование приложения «*Tanks*» начинается с выделения оновных модулей и схемы взаимодействия между ними. В приложении можно выделить два основных модуля *Client* и *Server*. Модуль *Server* не содержит *GUI* и отвечает за установку и закрытие соединения, обеспечивает синхронизацию подключенных клиентов между собой, сообщает игрокам о завершении игры. Все основные вычисления и отрисовка игрового поля производятся на модуле *Client*. Также на модуле *Client* реализован графический интерфейс. После выделения основных модулей выполняется проектирование внутренней структуры каждого модуля.

Термины «клиент» и «сервер» используются для обозначения не только программных модулей, но и компьютеров, подключенных к сети. Если компьютер преимущественно предоставляет свои ресурсы другим компьютерам сети, то он называется сервером, а если он их потребляет – клиентом. Иногда один и тот же компьютер может играть одновременно роль и сервера, и клиента [5].

В качестве схемы взаимодействия килента и сервера выбрана концепция «толстого» клиента и «тонкого» сервера. Это подразумевает что все основные вычисления происходит на модуле «*Client*», модуль «*Server*» отвечает только за подключение и синхронизацию. Синхронизация происходит следующим образом: каждый клиент отсылает своё состояние (координаты, направление, информацию о произведении выстрела) на сервер, сервер формирует информацию о состоянии каждого игрока в виде строки с разделителями и рассылает клиентам, после чего клиенты на основании полученных данных производят перерисовку игрового поля.

Далее выбирается технология реализации сетевого приложения, язык программирования и среда разработки. После этого можно приступать непосредственно к написанию кода. Функциональная схема проектирования приложения представлена на рисунке 2.1.



Рисунок 2.1 – Функциональная схема проектирования приложения

С точки зрения сокетов клиенту необходимо знать две вещи, чтобы найти сервер и подключиться к нему: имя сервера и номер порта. Номер порта – это идентификатор, который помогает различать разные клиентские и серверные приложения, запущенные на одном компьютере. Когда сервер ждёт подключения, прослушивается заранее выбранный порт. Номер порта, который назначен серверу, используется клиентами для соединения. После установления соединения можно получить входящий и исходящий потоки, воспользовавшись методами *getInputStream()* и *getOutputStream()* и объект *Socket*. Для передачи серверу одного байта используется метод *writeUTF()* из *OutputStream*. Чтобы отправить вместо байта текстовую строки, лучше заключить поток *OutputStream* внутри объекта *DataOutputStream*. Для получения сообщения используется метод *readUTF()* из потока *InputStream*, заключенный в *DataInputStream*. Разрыв соединения происходит с помощью метода *Close()*.

После установки соединения требуемого количества клиентов, сервер приложения «*Tanks*» начинает непрерывный обмен данными между ними. В случае если по танку одного из игроков произошло попадание пули, сервер сообщает ему о проигрыше и закрывает соединение. Главный цикл на сервере выполняется до тех пор, пока не останется одно активное соединение. Это подразумевает, что остался один игрок, который перестрелял остальных, сервер сообщает ему о победе и закрывает соединение.

**2.2 Структура модулей *Client* и *Server***

В приложении «*Tanks*» модуль *Server* реализован одним классом. Сервер содержит следующую информацию: номер порта, серверный сокет, массив клиентских сокетов, максимально-допустимое количество подключенных игрок, количество подключенных игроков в данный момент, массивы входных и выходных потоков, массив буферов, содержащих данные о подключенных игроках. Структура модуля *Server* представлена на рисунке 2.2.



Рисунок 2.2 – Структура модуля *Server*

В основном методе *main*() сервер ждёт подключения требуемого числа клиентов, назначает им потоки ввода/вывода для принятия и отправки данных, создаются буферы для хранения информации о каждом игроке, после этого запускается основной цикл, который работает пока игра не окончена. Схема работы основного метода сервера после установки соединения представлена на рисунке 2.3.



Рисунок 2.3 – Схема работы основного метода модуля *Server*

Клиент реализован в нескольких классах, классы для отображения графики и класс для сетевого взаимодействия. В основном методе клиента создаётся три потока, которые работают параллельно. Первый поток считывает данные с клавиатуры, второй поток рисует графику, третий поток выполняет взаимодействие с сервером. Структура клиента представлена на рисунке 2.4.



Рисунок 2.4 – Структура модуля *Client*

Клиент хранит и способен передавать информацию о местоположении своего танка, также может отображать танки других игроков, основываясь на данных полученных от сервера. Для имитации движения пули создаётся отдельный поток, который отображает пулю. При запуске основного метода клиента создаётся окно *JFrame* с кнопкой подключения к заданному *ip*-адресу, после того как произошло подключение вышеперечисленные потоки начинают свою работу. Работа потоков продолжается до тех пор, пока на поле не останется один игрок. Схема данных представлена в приложении Г.

**2.3 Структура классов приложения «*Tanks*»**

Класс *Server*, отвечает за реализацию сервера в прложении «*Tanks*».

Класс содержит статические поля: \_*port(int)* – порт сервера; \_*socket*(*ServerSocket*) – сокет сервера; *maxCountPlayer*(*int*) – максимальное количество игрок; *playerCon*(*int*) – количество подключенных игроков; *sockPlayer*(*Socket*[]) – клиентские сокеты; *dataInputStreams*(*DataInputStream*[]) – входные потоки для клиентских сокетов; *dataOutputStreams* (*DataOutputStream*[]) – выходные потоки для клиентских сокетов; *isAlive*(*boolean*[]) – логическая переменная, которая указывает на статус активности танка; *strBuff*(*String*[]) – данные об игроке(положение и направление); *sizeBlockWidth/ sizeBlockWidth* (*int*) – ширина и высота блока на игровом поле. Не имеет конструкторов.

Методы класса *Server*:

1. *public* *static* *void* *main*(*String* *args*[]) – создаёт серверный и клентские сокеты, буферы для хранение данных игроков. Принимает подключения клиентов, считывает начальные координаты подключенного игрока. До тех пор пока не остался один игрок отправляет данные обо всех игроках. Если в игрока попали, закрывает его сокет. В случае если остался один игрок, сообщает ему о выигрыше. Закрывает все соединения.
2. *public* *static* *boolean* *isDie*(*int* *num*) ­– принимает индекс подключенного игрока, проверяет на попадание пули по игроку, возвращает логическое значение *true* или *false*.

За реализацию клиента в приложении «*Tanks*» отвечают четыре класса: *DisplayGraphics, Area, SocketData, MainFrame*. Класс *MainFrame* содержит метод *main()* и служит входной точкой при старте программы.

Класс *DisplayGraphics* содержит следующие поля: *xPlayer, yPlayer, xPlayerDirection, yPlayerDirection (int)* – координаты и направление игрока; *xBullet, yBullet, xDirection, yDirection (int) –* координаты и направление пули; *bullet* (*boolean*) – флаг, сообщающий о наличии пули на поле; *areaWidth*, *areaHeight* (*int*) – длина и ширина поля; *sizeWidth*, *sizeHeight*, *sizeBlockWidth*, *sizeBlockHeight* (*int*) – количество блоков на поле, длина и ширина блока. Конструктор с параметрами *DisplayGraphics(int areaWidth, int areaHeight, int sizeWidth, int sizeHeight).*

Методы класса *DisplayGraphics*:

1. *public* *void* *setBulletPosition*() – присвает начальную позицию пуле.

2. *public* *void* *moveBullet*(*int* *speed*) – принимает скорость пули в качестве аргумента, перемещает пулю в зависимости от направления.

3. *private* *void* *setSizeBlock*() – задаёт размеры блока на поле.

4. *public* *void* *setPositionPlayer*(*int* *x*, *int* *y*) – задаёт координаты игрока.

5. *public* *Graphics2D* *getGraphBullet*(*Graphics2D* *graphics*, *int* *x*, *int* *y*) – отрисовка пули других игроков.

6. *public* *Graphics2D* *getGraphBullet*(*Graphics2D* *graphics*) – отрисовка своей пули.

7. *public* *Graphics2D* *drawArea*(*Graphics2D* *graphics*) – отрисовка игрового поля.

8. *public* *Graphics2D* *getGraphPlayer*(*Graphics2D* *graphics*) – отрисовка своего танка.

9. *public* *Graphics2D* *getGraphPlayer*(*Graphics2D* *graphics*, *int* *x,* *int* *y*, *int* *xDir*, *int* *yDir*) – отрисовка танка других игроков.

10. *public* *void* *movePlayer*(*int* *speed*, *int* *xPos*, *int* *yPos*) – метод передвижения клиента в зависимости от направления.

11. *public* *String* *getPlayerInfo*() – формирование строки данных о клиенте для передачи на сервер.

12. *public* *Graphics2D* *setPlayerInfo*(*String* *data*, *Graphics2D* *g2d*) – получения состояния других игроков с сервера и их отрисовка.

Класс *Area* наследуется от класса *JPanel* и служит адаптером для панели, содержит следующие поля: *displayGraphics*(*DisplayGraphics*) – переменная для отрисовки; *str*(*String*) – строка с информацией о других игроках. Содержит конструктор без параметров *Area*().

Методы класса *Area*:

1. *public* *void* *moveBullet*() – создаёт поток для отрисовки пули.
2. *public* *void* *movePlayer*() – перемещение танка на панели.
3. *public* *void* *paint*(*Graphics* *g)* – метод отрисовки панели.

Класс *SocketData* служит для подключения клиента к серверу, и обмена данными, содержит следующие поля: \_*port*(*int*) – порт; \_*ipAddress*(*String*) – *ip*-адрес сервера в формате строки; \_*socket*(*Socket*) – сокет клиента; *isConnect*(*boolean*) – флаг отражающий подключение к серверу; *inetAddress*(*InetAddress*) – *ip*-адрес сервера в формате *InetAddress*; *iStream*(*DataInputStream*) – входной поток; *oStream*(*DataOutputStream*) – выходной поток; *isWin*(*boolean*) – флаг отражающий выигрыш игрока. Содержит конструктор с параметрами *SocketData*(*int* \_*port*, *String* \_*ipAddress*).

Методы класса *SocketData*:

1. *public* *void* *close*() – метод для закрытия соединения.
2. *public* *void* *Сonnect*() – метод для подключения к серверу.
3. *public* *String* *daraServer*(*String* *dataSend*) – метод для обмена данными с сервером.

Класс *MainFrame* наследуется от *JFrame*, создает игровое окно, и запускает основные потоки приложения, содержит следующие поля: *panel*(*Area*) – панель для рисования; *flag*(*boolean*) – флаг отражающий состояние подключения к серверу; *rhDraw*(*Thread*) – поток для рисования; *playerButPress*(*Thread*) – поток для передачи даннх; *sockData*(*SocketData*) – переменная для сетевого взаимодействия; *keyListener*(*KeyListener*) – переменная для считывания данных с клавиатуры; *textField*(*JTextField*) – текстовое поле для ввода *ip*-адреса. Содержит конструктор без параметров *MainFrame*().

Методы класса *MainFrame*:

1. *public* *void* *keyTyped*(*KeyEvent* *arg0*) – метод для чтения нажатых клавиш клавиатуры.
2. *public* *static* *void* *main*(*String* *args*[]) – метод для запуска основных потоков.

Листинг всех классов представлен в приложении А.

**3 ОПЫТНАЯ ЭКСПЛУАТАЦИЯ СЕТЕВОГО ПРИЛОЖЕНИЯ «*Tanks*»**

* 1. **Пользовательский интерфейс**

Графический интерфейс был реализован с помощью библиотеки *Swing*, библиотека включена в *Java Runtime Environment.*

Swing это набор для создания графического интерфейса пользователя (*GUI*) для *Java* программ и апплетов. В сравнение с ранее использовавшейся библиотекой *AWT*, библиотека *SWING* имеет ряд преимуществ:

1. Богатый набор интерфейсных примитивов;

2. Настраивающийся внешний вид на различных платформах ;

3. Раздельная архитектура модель-вид;

4. Встроенная поддержка *HTML.*

Для реакции на события нажатия кнопок на клавиатуре, используются классы, реализующие интерфейс *KeyListener()*. Окно клиента представлено на рисунке 3.1.

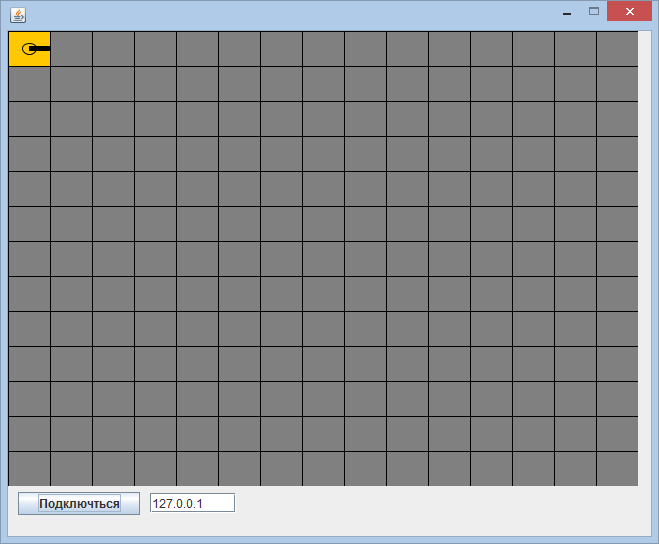


Рисунок 3.1 – Главное окно клиента

Как видно из рисунка 3.1 на главном окне, есть кнопка подключения к серверу по-заданному *ip*-адресу. Окно реализуется с помощью объекта *JFrame*. *JFrame* содержит ряд свойств которые можно менять вызывая для него соответствующие методы, например размер окна, возможность изменения размера и тд. Кнопку подключения реализует объект *JButton*, обработчик события для кнопки добавляется методом *addActionListener*(*ActionListener* *al*). При нажатии на кнопку клиент подключается к серверу по адресу указанному в текстовом поле(по умолчанию указан адрес *localhost*). Текстовое поле реализовано с помощью объекта *JTextArea*, метод который понадобиться в данном приложении *getText*() достаёт значение из текстового поля в формате строки. После нажатия на кнопку, если подключение прошло успешно, в правом нижнем углу экрана выводится сообщении об успешном подключении. Демонстрация представлена на рисунке 3.2.

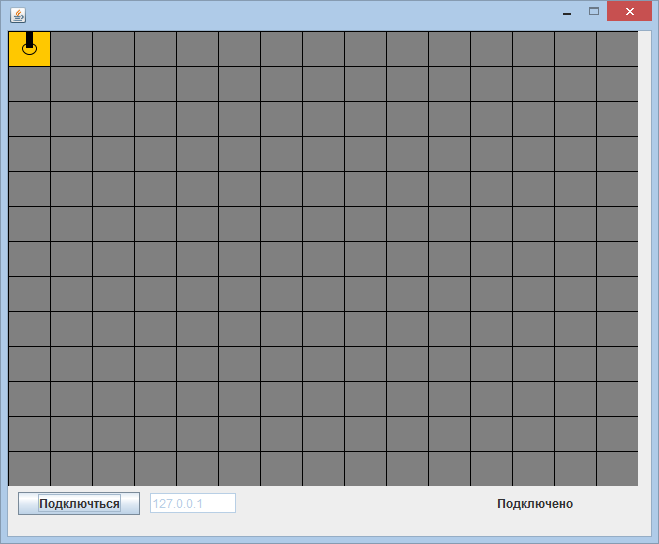


Рисунок 3.2 – Вывод сообщения о подключении

Свой танк отображается оранжевым цветом, а вражеский – желтым. При попадании по вражескому танку в правом нижнем углу отображается строка, сообщающая о победе. Демонстрация представлена на рисунке 3.3.

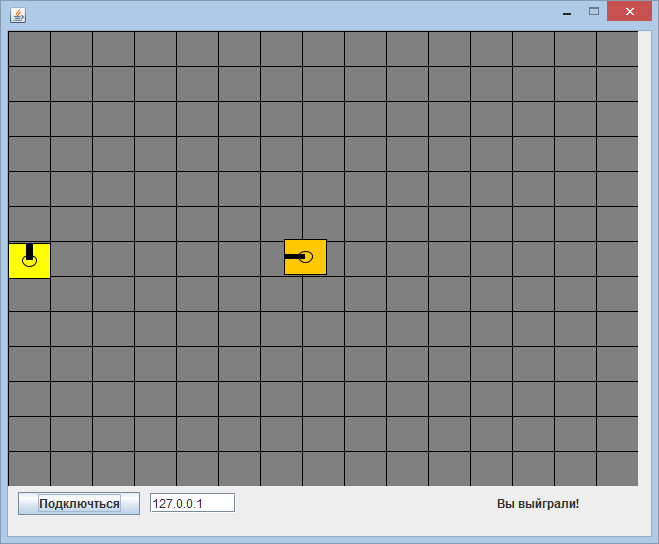


Рисунок 3.3 – Сообщение о победе

Проигравшему игроку выводится сообщение о том, что игра окончена.

* 1. **Запуск и отладка**

Приложение «*Tanks*» запускается в два этапа:

1. Запуск сервера. Сервер может запускаться как на компьютере одного из игроков, так и удалённо. В случае удалённого запуска игрокам необходимо знать *ip*-адрес сетевого узла, на котором запущен сервер.
2. Запуск клиента. При запуске клиента необходимо ввести в текстовое поле *ip*-адрес узла, на котором запущен сервер и нажать кнопку «Подключиться».

Сведения по настройке программы представлены в приложении В.

Исключение в *Java* – это объект, который описывает исключительное состояние, воз­никшее в каком-либо участке программного кода. Когда возникает ис­ключительное состояние, создается объект класса *Exception*. Этот объект пересылается в метод, обрабатывающий данный тип исключительной ситуации. Исключения могут возбуждаться и «вруч­ную» для того, чтобы сообщить о некоторых нештатных ситуациях.

К механизму обработки исключений в *Java* имеют отношение 5 клю­чевых слов: – *try*, *catch*, *throw,* *throws* и *finally*. Схема работы этого механизма следующая. Вы пытаетесь (*try*) выполнить блок кода, и если при этом возникает ошибка, система возбуждает (*throw*) исключение, ко­торое в зависимости от его типа вы можете перехватить (*catch*) или пере­дать умалчиваемому (*finally*) обработчику.

Во время выполнения игрового приложения могу возникать различные исключительные ситуации, которые могут пагубно сказаться на игровом процессе, поэтому эти ошибки необходимо предусмотреть и сделать их обработку.

Самая частая ошибка возникает тогда, когда клиент пытается осуществить соединение к несуществующему серверу. Это происходит, когда клиент ввёл неправильный *ip*-адрес сервера текстовую область.

В разработанном игровом приложении данная ситуация обрабатывается исключением которое выдаёт сообщение клиенту. На рисунке 3.4 продемонстрирована ошибка.

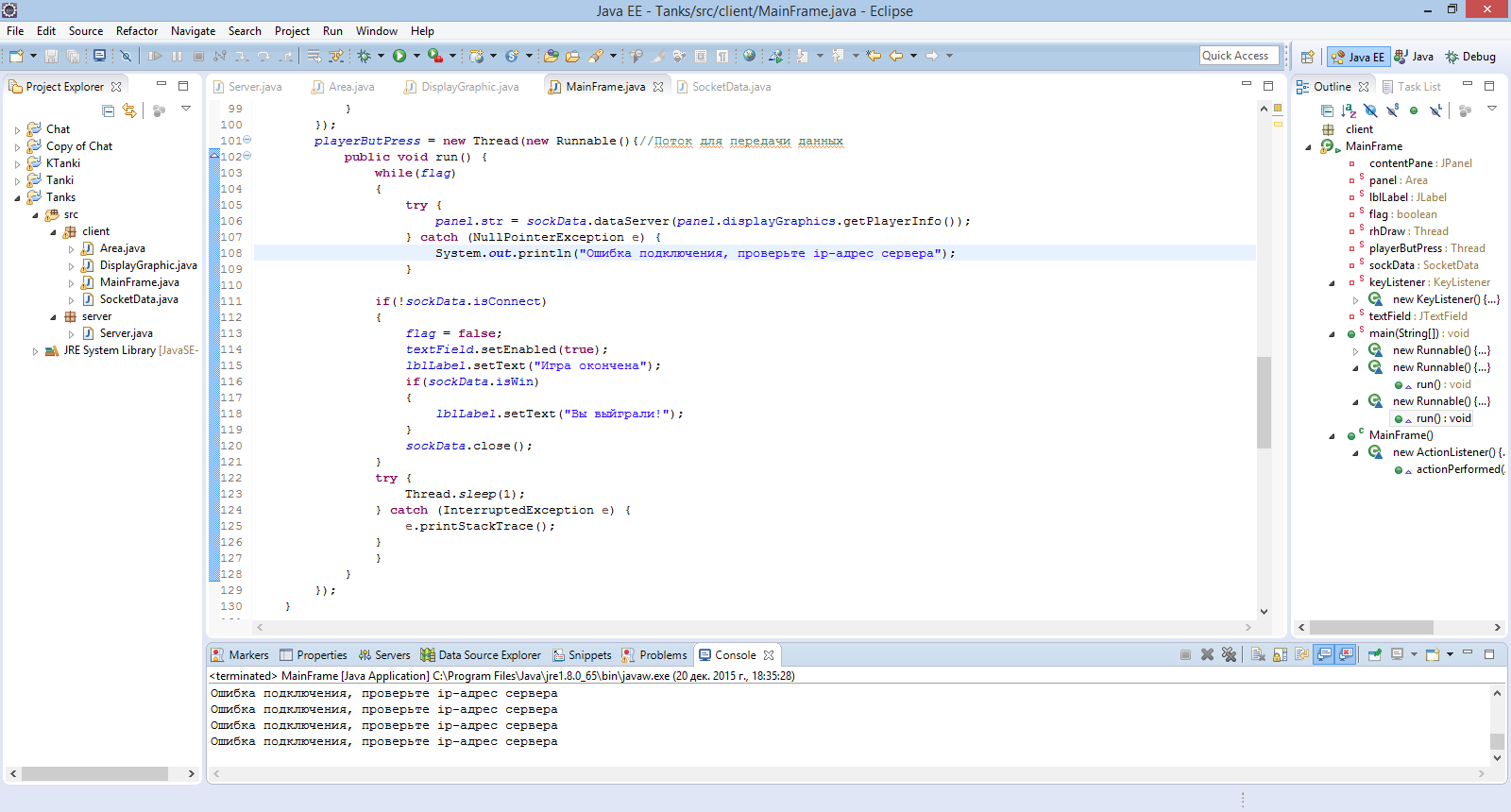


Рисунок 3.4 – Сообщение об ошибке

Во всем приложении идут обработки критических исключений, в скрытом для пользователя режиме, в случае критической ошибки приложение закроется. Полное руководство пользователя представлено в приложении Б.

**ЗАКЛЮЧЕНИЕ**

В ходе выполнения курсового проекта был проведен сравнительный анализ технологий, позволяющих разрабатывать сетевые приложения. На основе анализа всех преимуществ и недостатков рассматриваемых технологий для решения поставленной задачи была выбрана технология *TCP* сокетов. Для выделения этапов проектирования применялась схема функционального проектирования приложений. Разработанное программное обеспечение очень легко в освоении и понимании принципа работы по сравнению с существующими многопользовательскими игровыми приложениями и наглядно демонстрирует технологию сетевого взаимодействия по средствам *TCP* сокетов.

Итогом выполнения курсового проекта является сетевое игровое приложение «*Tanks*» для совместного многопользовательского взаимодействия. Был разработан графический интерфейс, позволяющий пользователю управлять танком и взаимодействовать с другими пользователями. Приложение имеет простой и понятный графический интерфейс, который позволяет пользователю быстро освоиться в игре и начать играть. Рекомендуется использовать приложение только в рамках руководства пользователя. Дальнейшая разработка может быть связана с улучшением графического интерфейса, оптимизации приложения при большом количестве пользователей, добавления дополнительных средств GUI для взаимодействия с пользователем.

Данный курсовой проект даёт возможность глубже изучить пройденный материал, позволяет закрепить навыки решения поставленных задач и обучиться поиску необходимой для этого информации, а также освоить на практике все теоретические знания, полученные по дисциплине программирование сетевых приложений.

**Список использованных источников**

1. Статистика проекта *World Of Tanks* [Электронный ресурс]. – <https://ru.wikipedia.org/wiki/World_of_Tanks/> – Дата доступа: 13.12.2015.
2. Основы технологии *Websockets* [Электронный ресурс]. – http://www.html5rocks.com/ru/tutorials/websockets/basics/– Дата доступа: 13.12.2015.
3. *Unity Game Development Essentials / Will Goldstone*. – *Birmingham*: *Packt* *Publishing*, 2010. – 266 с.
4. *TCP*/*IP* *Illustrated*, *Vol*. 1: *The* *Protocols* / *W.* *Richard* *Stevens*. ­– *Boston*: *Addison*-*Wesley* *Professional*, 1994. – 600 с.
5. Компьютерные сети / Эндрю Таненбаум. – СПб: Питер, 2007. – 992 с.

**ПРИЛОЖЕНИЕ А**

(обязательное)

**Распечатка программных модулей**

Класс Server.java:

**package** server;

**import** java.io.DataInputStream;

**import** java.io.DataOutputStream;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**public** **class** Server {

**private** **static** **int** *\_port* = 8082;

**private** **static** ServerSocket *\_socket*;

**private** **static** **int** *countPlayer* = 2;//Количество игроков

**private** **static** **int** *playerCon*;//количество подключенных игроков

**private** **static** Socket[] *sockPlayer*;//Клиентские сокеты

**private** **static** DataInputStream[] *dataInputStreams*;//Входные потоки

**private** **static** DataOutputStream[] *dataOutputStreams*;//Выходные потоки

**private** **static** **boolean**[] *isAlive*;//true если жив, иначе false

**private** **static** String[] *strBuff*;//Данные о игроках

**private** **static** **int** *sizeBlockWidth* = 42, *sizeBlockHeight* = 35;//длина и ширина блоков на клиенте

**public** **static** **void** main(String[] args) {

**try**

{

*playerCon* = 0;

*\_socket* = **new** ServerSocket(*\_port*);

*sockPlayer* = **new** Socket[*countPlayer*];

*strBuff* = **new** String[*countPlayer*];

*isAlive* = **new** **boolean**[*countPlayer*];

*dataInputStreams* = **new** DataInputStream[*countPlayer*];

*dataOutputStreams* = **new** DataOutputStream[*countPlayer*];

**for** (**int** i = 0; i < *countPlayer*; i++) {

System.***out***.println("wait " + (i + 1));

*sockPlayer*[i] = *\_socket*.accept();

System.***out***.println("connected " + (i + 1));

*dataInputStreams*[i] = **new** DataInputStream(*sockPlayer*[i].getInputStream());

*dataOutputStreams*[i] = **new** DataOutputStream(*sockPlayer*[i].getOutputStream());

*strBuff*[i] = *dataInputStreams*[i].readUTF();

System.***out***.println(*strBuff*[i]);

*playerCon*++;

}

System.***out***.println("Start game");

**while**(*playerCon* > 1)

{

StringBuilder sb;

**for** (**int** i = 0; i < *countPlayer* && *playerCon* > 1; i++) {

**if**(*sockPlayer*[i] != **null**)

{

sb = **new** StringBuilder(String.*valueOf*(*playerCon*));

sb.append(';');

**for** (**int** j = 0; j < *playerCon*; j++) {

**if**( j!= i && *strBuff*[j] != **null**)

{

sb.append(*strBuff*[j]);

sb.append(';');

}

}

*dataOutputStreams*[i].writeUTF(sb.toString());

*dataOutputStreams*[i].flush();

*isAlive*[i] = *isDie*(i);

*dataOutputStreams*[i].writeBoolean(*isAlive*[i]);

*dataOutputStreams*[i].flush();

**if**(!*isAlive*[i])

{

*strBuff*[i] = *dataInputStreams*[i].readUTF();

Thread.*sleep*(1);

}

**else**

{

*strBuff*[i] = **null**;

*dataInputStreams*[i].close();

*dataOutputStreams*[i].close();

*sockPlayer*[i].close();

*sockPlayer*[i] = **null**;

*playerCon*--;

}

}

}

}

**for** (**int** i = 0; i < *countPlayer*; i++) {

**if**(*sockPlayer*[i] != **null**)

{

System.***out***.println("Winer: " + (i + 1));

*dataOutputStreams*[i].writeUTF("0");

*dataInputStreams*[i].close();

*dataOutputStreams*[i].close();

*sockPlayer*[i].close();

*sockPlayer*[i] = **null**;

}

}

*\_socket*.close();

System.***out***.println("Socket closed!");

}

**catch**(Exception ex)

{

ex.printStackTrace();

}

}

**public** **static** **boolean** isDie(**int** num) {

**boolean** result = **false**;

**int** x, y, xEnd, yEnd;

String[] tmp = *strBuff*[num].split(";");

x = Integer.*parseInt*(tmp[0]);

xEnd = x + *sizeBlockWidth*;

y = Integer.*parseInt*(tmp[1]);

yEnd = y + *sizeBlockHeight*;

**for** (**int** i = 0; i < *strBuff*.length && !result; i++) {

**if**(*sockPlayer*[i] != **null**)

{

String[] temp = *strBuff*[i].split(";");

**if**(temp.length > 5 && num != i)

{

**int** xB = Integer.*parseInt*(temp[5]);

**int** yB = Integer.*parseInt*(temp[6]);

**if**(x <= xB && xB <= xEnd && yB <= yEnd && y <= yB)

{

result = **true**;

}

}

}

}

**return** result;

}

}

Класс DisplayGraphics.java:

**package** client;

**import** java.awt.Color;

**import** java.awt.Graphics;

**import** java.awt.Graphics2D;

**import** java.awt.image.BufferedImage;

**import** org.w3c.dom.css.Rect;

**public** **class** DisplayGraphic {

**int** xPlayer, yPlayer, xPlayerDirection, yPlayerDirection;//x и y игрока и направление

**int** sizeWidth, sizeHeight;//Количество блоков на поле

**int** areaWidth, areaHeight;//Ширина и длина поля (px)

**int** sizeBlockWidth, sizeBlockHeight;//Ширина и длина блока (px)

**int** xBullet, yBullet, xDirection, yDirection;

**boolean** bullet;

**public** DisplayGraphic(**int** areaWidth, **int** areaHeight, **int** sizeWidth, **int** sizeHeight)

{

**this**.areaWidth = areaWidth;

**this**.areaHeight = areaHeight;

**this**.sizeWidth = sizeWidth;

**this**.sizeHeight = sizeHeight;

xPlayerDirection = 1;

xDirection = 1;

bullet = **false**;

setSizeBlock();

}

//Присвоение позиции пуле

**public** **void** setBulletPosition() {

**if**(!bullet)

{

xDirection = xPlayerDirection;

yDirection = yPlayerDirection;

xBullet = xPlayer + (sizeBlockWidth / 2);

yBullet = yPlayer + (sizeBlockHeight / 2);

bullet = **true**;

}

}

//Перемещение пули

**public** **void** moveBullet(**int** speed) {

**if**(xDirection > 0)

{

xBullet += 8 \* speed;

}

**else** **if**(xDirection < 0)

{

xBullet -= 8 \* speed;

}

**if**(yDirection > 0)

{

yBullet += 8 \* speed;

}

**else** **if**(yDirection < 0)

{

yBullet -= 8 \* speed;

}

**if**(xBullet < 0 || xBullet > areaWidth || yBullet < 0 || yBullet > areaHeight)

{

bullet = **false**;

}

}

**private** **void** setSizeBlock() {

sizeBlockWidth = areaWidth / sizeWidth;

sizeBlockHeight = areaHeight / sizeHeight;

}

**public** **void** setPositionPlayer(**int** x, **int** y) {

xPlayer = x;

yPlayer = y;

}

//Отрисока пули клиента

**public** Graphics2D getGraphBullet(Graphics2D graphics) {

graphics.setColor(Color.***black***);

graphics.fillRect(xBullet, yBullet, sizeBlockWidth / 8, sizeBlockHeight / 8);

**return** graphics;

}

//Отрисока пули другого игрока

**public** Graphics2D getGraphBullet(Graphics2D graphics, **int** x, **int** y) {

graphics.setColor(Color.***black***);

graphics.fillRect(x, y, sizeBlockWidth / 8, sizeBlockHeight / 8);

**return** graphics;

}

//Отрисовка поля

**public** Graphics2D drawArea(Graphics2D graphics) {

**int** templateX = 0, templateY = 0;

**for** (**int** i = 0; i < sizeWidth; i++) {

**for** (**int** j = 0; j < sizeHeight; j++) {

graphics.setColor(Color.***gray***);

graphics.fillRect(templateX, templateY, sizeBlockWidth, sizeBlockHeight);

graphics.setColor(Color.***black***);

graphics.drawRect(templateX, templateY, sizeBlockWidth, sizeBlockHeight);

templateY += sizeBlockHeight;

}

templateY = 0;

templateX += sizeBlockWidth;

}

**return** graphics;

}

//отображение клиента

**public** Graphics2D getGraphPlayer(Graphics2D/\*Graphics\*/ graphics) {

graphics.setColor(Color.***orange***);

graphics.fillRect(xPlayer, yPlayer, sizeBlockWidth, sizeBlockHeight);

graphics.setColor(Color.***black***);

graphics.drawRect(xPlayer, yPlayer, sizeBlockWidth, sizeBlockHeight);

graphics.drawOval((xPlayer + sizeBlockWidth / 2) - (sizeBlockWidth / 6), (yPlayer + sizeBlockHeight / 2) - (sizeBlockHeight / 6), (sizeBlockWidth / 3), (sizeBlockHeight / 3));

**if**(xPlayerDirection > 0)

{

graphics.setColor(Color.***BLACK***);

graphics.fillRect((xPlayer + sizeBlockWidth / 2), (yPlayer + sizeBlockHeight / 2) - sizeBlockHeight / 12, sizeBlockWidth / 2, (sizeBlockHeight / 6));

}

**else** **if**(xPlayerDirection < 0)

{

graphics.setColor(Color.***BLACK***);

graphics.fillRect((xPlayer + sizeBlockWidth / 2) - sizeBlockWidth / 2, (yPlayer + sizeBlockHeight / 2) - sizeBlockHeight / 12, sizeBlockWidth / 2, (sizeBlockHeight / 6));

}

**else** **if**(yPlayerDirection > 0)

{

graphics.setColor(Color.***BLACK***);

graphics.fillRect((xPlayer + sizeBlockWidth / 2) - sizeBlockWidth / 12, (yPlayer + sizeBlockHeight / 2) , sizeBlockWidth / 6, (sizeBlockHeight / 2));

}

**else** **if**(yPlayerDirection < 0)

{

graphics.setColor(Color.***BLACK***);

graphics.fillRect((xPlayer + sizeBlockWidth / 2) - sizeBlockWidth / 12, (yPlayer + sizeBlockHeight / 2) - sizeBlockHeight / 2, sizeBlockWidth / 6, (sizeBlockHeight / 2));

}

**return** graphics;

}

//Отображение другого игрока

**public** Graphics2D getGraphPlayer(Graphics2D graphics, **int** x, **int** y,**int** xDir,**int** yDir) {

graphics.setColor(Color.***yellow***);

graphics.fillRect(x, y, sizeBlockWidth, sizeBlockHeight);

graphics.setColor(Color.***black***);

graphics.drawRect(x, y, sizeBlockWidth, sizeBlockHeight);

graphics.drawRect(x, y, sizeBlockWidth, sizeBlockHeight);

graphics.drawOval((x + sizeBlockWidth / 2) - (sizeBlockWidth / 6), (y + sizeBlockHeight / 2) - (sizeBlockHeight / 6), (sizeBlockWidth / 3), (sizeBlockHeight / 3));

**if**(xDir > 0)

{

graphics.setColor(Color.***BLACK***);

graphics.fillRect((x + sizeBlockWidth / 2), (y + sizeBlockHeight / 2) - sizeBlockHeight / 12, sizeBlockWidth / 2, (sizeBlockHeight / 6));

}

**else** **if**(xDir < 0)

{

graphics.setColor(Color.***BLACK***);

graphics.fillRect((x + sizeBlockWidth / 2) - sizeBlockWidth / 2, (y + sizeBlockHeight / 2) - sizeBlockHeight / 12, sizeBlockWidth / 2, (sizeBlockHeight / 6));

}

**else** **if**(yDir > 0)

{

graphics.setColor(Color.***BLACK***);

graphics.fillRect((x + sizeBlockWidth / 2) - sizeBlockWidth / 12, (y + sizeBlockHeight / 2) , sizeBlockWidth / 6, (sizeBlockHeight / 2));

}

**else** **if**(yDir < 0)

{

graphics.setColor(Color.***BLACK***);

graphics.fillRect((x + sizeBlockWidth / 2) - sizeBlockWidth / 12, (y + sizeBlockHeight / 2) - sizeBlockHeight / 2, sizeBlockWidth / 6, (sizeBlockHeight / 2));

}

**return** graphics;

}

//Передвижение клиента

**public** **void** movePlayer(**int** speed, **int** xPos, **int** yPos) {

xPlayerDirection = xPos;

yPlayerDirection = yPos;

**int** temp;

**if**(xPos > 0)

{

temp = xPlayer + 4 \* speed;

**if**(temp + sizeBlockWidth <= sizeBlockWidth \* sizeWidth)

{

xPlayer = temp;

}

}

**else** **if**(xPos < 0)

{

temp = xPlayer - 4 \* speed;

**if**(temp >= 0)

{

xPlayer = temp;

}

}

**if**(yPos > 0)

{

temp = yPlayer + 4 \* speed;

**if**(temp + sizeBlockHeight <= sizeBlockHeight \* sizeHeight)

{

yPlayer = temp;

}

}

**else** **if**(yPos < 0)

{

temp = yPlayer - 4 \* speed;

**if**(temp >= 0)

{

yPlayer = temp;

}

}

}

//Запись в строку данных о клиенте

**public** String getPlayerInfo()

{

StringBuilder result = **new** StringBuilder();

result.append(xPlayer);

result.append(';');

result.append(yPlayer);

result.append(';');

result.append(xPlayerDirection);

result.append(';');

result.append(yPlayerDirection);

result.append(';');

**if**(bullet)

{

result.append('0');

result.append(';');

result.append(xBullet);

result.append(';');

result.append(yBullet);

}

**else**

{

result.append('1');

}

**return** result.toString();

}

//Получение информации о игроках с сервера

**public** Graphics2D setPlayerInfo(String data, Graphics2D g2d) {

String[] dataSplit = data.split(";");

**int** countPlayer = Integer.*parseInt*(dataSplit[0]);

**int** index = 0, splitIndex = 1;

**while**(index < (countPlayer - 1) && dataSplit.length > 2)

{

**int** xTemp = Integer.*parseInt*(dataSplit[splitIndex]);

splitIndex++;

**int** yTemp = Integer.*parseInt*(dataSplit[splitIndex]);

splitIndex++;

**int** xDirTemp = Integer.*parseInt*(dataSplit[splitIndex]);

splitIndex++;

**int** yDirTemp = Integer.*parseInt*(dataSplit[splitIndex]);

splitIndex++;

**int** temp = Integer.*parseInt*(dataSplit[splitIndex]);

splitIndex++;

g2d = getGraphPlayer(g2d, xTemp, yTemp, xDirTemp, yDirTemp);

**if**(temp == 0)

{

**int** xBulTemp = Integer.*parseInt*(dataSplit[splitIndex]);

splitIndex++;

**int** yBulTemp = Integer.*parseInt*(dataSplit[splitIndex]);

splitIndex++;

g2d = getGraphBullet(g2d, xBulTemp, yBulTemp);

}

index++;

}

**return** g2d;

}

}

Класс Area.java:

**package** client;

**import** java.awt.Graphics;

**import** java.awt.Graphics2D;

**import** java.awt.RenderingHints;

**import** java.awt.image.BufferedImage;

**import** javax.swing.JPanel;

**public** **class** Area **extends** JPanel{

DisplayGraphic displayGraphics = **null**;

**public** String str;

**public** Area() {

setOpaque(**true**);

str = **null**;

}

//Перемещение игрока

**public** **void** movePlayer(**int** x, **int** y) {

displayGraphics.movePlayer(1, x, y);

//paint(this.getGraphics());

}

//Перемещение пули

**public** **void** moveBullet()

{

Thread th = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**try** {

Thread.*sleep*(50);

displayGraphics.moveBullet(1);

**if**(displayGraphics.bullet)

{

run();

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

});

**if**(!displayGraphics.bullet)

{

displayGraphics.setBulletPosition();

th.start();

}

}

//Метод отрисовки

@Override

**public** **void** paint(Graphics g) {

BufferedImage bi = **new** BufferedImage(**this**.getWidth(), **this**.getHeight(), BufferedImage.***TYPE\_4BYTE\_ABGR***);

Graphics2D g2d = bi.createGraphics();

g2d.setRenderingHint(RenderingHints.***KEY\_ANTIALIASING***, RenderingHints.***VALUE\_ANTIALIAS\_ON***);

**if**(displayGraphics == **null**)

{

displayGraphics = **new** DisplayGraphic(**this**.getBounds().width, **this**.getBounds().height, 15, 13);//, this.getWidth(), this.getHeight());

}

g2d = displayGraphics.drawArea(g2d);

g2d = displayGraphics.getGraphPlayer(g2d);

**if**(displayGraphics.bullet)

{

g2d = displayGraphics.getGraphBullet(g2d);

}

**if**(str != **null**)

{

g2d = displayGraphics.setPlayerInfo(str, g2d);

}

g.drawImage(bi, 0, 0, getParent());

}

}

Класс SocketData.java:

**package** client;

**import** java.io.DataInputStream;

**import** java.io.DataOutputStream;

**import** java.io.IOException;

**import** java.net.InetAddress;

**import** java.net.Socket;

**public** **class** SocketData {

**private** **int** \_port;

**private** String \_ipAddress;

**private** Socket \_socket;

**public** **boolean** isConnect;

InetAddress inetAddress;

DataInputStream iStream;

DataOutputStream oStream;

**boolean** isWin;

**public** SocketData(**int** \_port, String \_ipAddress) {

**this**.\_port = \_port;

**this**.\_ipAddress = \_ipAddress;

isConnect = **false**;

}

**public** **void** close(){

**try** {

iStream.close();

oStream.close();

\_socket.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

**public** **void** Connect()

{

**try**{

isWin = **false**;

isConnect = **true**;

inetAddress = InetAddress.*getByName*(\_ipAddress);

\_socket = **new** Socket(\_ipAddress, \_port);

System.***out***.println("connected");

iStream = **new** DataInputStream(\_socket.getInputStream());

oStream = **new** DataOutputStream(\_socket.getOutputStream());

}

**catch**(Exception ex)

{

System.***out***.println("Ошибка подключения, проверьте ip-адрес сервера");

}

}

**public** String dataServer(String dataSend) {

String result = **null**;

**try** {

**if**(isConnect)

{

oStream.writeUTF(dataSend);

oStream.flush();

result = iStream.readUTF();

**if**(result.length() == 1)

{

isConnect = **false**;

isWin = **true**;

}

**else**

{

isConnect = !iStream.readBoolean();

}

}

} **catch** (IOException e) {

System.***out***.println("Ошибка подключения, проверьте ip-адрес сервера");

}

**return** result;

}

}

Класс MainFrame.java:

**package** client;

**import** java.awt.EventQueue;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.border.EmptyBorder;

**import** java.awt.Color;

**import** javax.swing.JButton;

**import** java.awt.event.ActionListener;

**import** java.awt.event.KeyEvent;

**import** java.awt.event.KeyListener;

**import** java.awt.event.ActionEvent;

**import** javax.swing.JLabel;

**import** javax.swing.JTextField;

**public** **class** MainFrame **extends** JFrame {

**private** JPanel contentPane;

**private** **static** Area *panel*;

**private** **static** JLabel *lblLabel*;

**private** **static** **boolean** *flag* = **false**;

**private** **static** Thread *rhDraw*, *playerButPress*;

**private** **static** SocketData *sockData*;

**private** **static** KeyListener *keyListener* = **new** KeyListener() {

@Override

**public** **void** keyTyped(KeyEvent arg0) {

**boolean** movePlayer = **false**;

**int** x = 0, y = 0;

**if**(arg0.getKeyChar() == 'w' || arg0.getKeyChar() == 'W' || arg0.getKeyChar() == 'ц' || arg0.getKeyChar() == 'Ц')

{

y -= 1;

movePlayer = **true**;

}

**else** **if**(arg0.getKeyChar() == 's' || arg0.getKeyChar() == 'S' || arg0.getKeyChar() == 'ы' || arg0.getKeyChar() == 'Ы')

{

y += 1;

movePlayer = **true**;

}

**else** **if**(arg0.getKeyChar() == 'd' || arg0.getKeyChar() == 'D' || arg0.getKeyChar() == 'в' || arg0.getKeyChar() == 'В')

{

x += 1;

movePlayer = **true**;

}

**else** **if**(arg0.getKeyChar() == 'a' || arg0.getKeyChar() == 'A' || arg0.getKeyChar() == 'ф' || arg0.getKeyChar() == 'Ф')

{

x -= 1;

movePlayer = **true**;

}

**else** **if**(arg0.getKeyChar() == '0')

{

*panel*.moveBullet();

}

**if**(movePlayer)

{

*panel*.movePlayer(x, y);

}

}

@Override

**public** **void** keyReleased(KeyEvent arg0) {

}

@Override

**public** **void** keyPressed(KeyEvent arg0) {

}

};

**private** **static** JTextField *textField*;

/\*\*

\* Launch the application.

\*/

**public** **static** **void** main(String[] args) {

EventQueue.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**try** {

MainFrame frame = **new** MainFrame();

frame.setVisible(**true**);

frame.addKeyListener(*keyListener*);

} **catch** (Exception e) {

e.printStackTrace();

}

}

});

*rhDraw* = **new** Thread(**new** Runnable(){//Поток для отображения графики

**public** **void** run() {

**while**(*flag*)

{

**try** {

Thread.*sleep*(1);

*panel*.paint(*panel*.getGraphics());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

});

*playerButPress* = **new** Thread(**new** Runnable(){//Поток для передачи данных

**public** **void** run() {

**while**(*flag*)

{

**try** {

*panel*.str = *sockData*.dataServer(*panel*.displayGraphics.getPlayerInfo());

} **catch** (NullPointerException e) {

System.***out***.println("Ошибка подключения, проверьте ip-адрес сервера");

}

**if**(!*sockData*.isConnect)

{

*flag* = **false**;

*textField*.setEnabled(**true**);

*lblLabel*.setText("Игра окончена");

**if**(*sockData*.isWin)

{

*lblLabel*.setText("Вы выйграли!");

}

*sockData*.close();

}

**try** {

Thread.*sleep*(1);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

});

}

/\*\*

\* Create the frame.

\*/

**public** MainFrame() {

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

setResizable(**false**);

setBounds(100, 100, 649, 534);

contentPane = **new** JPanel();

contentPane.setBorder(**new** EmptyBorder(5, 5, 5, 5));

setContentPane(contentPane);

contentPane.setLayout(**null**);

*panel* = **new** Area();

*panel*.setBackground(Color.***GRAY***);

*panel*.setBounds(0, 0, 630, 455);

contentPane.add(*panel*);

JButton btnNewButton = **new** JButton("Подключться");

btnNewButton.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent arg0) {//подключение к серверу и запуск потоков

**if**(!*flag*)

{

*flag* = **true**;

*sockData* = **new** SocketData(8082, *textField*.getText());//"127.0.0.1");

*textField*.setEnabled(**false**);

**try**{

*sockData*.Connect();

}**catch**(NullPointerException e){

System.***out***.println("Ошибка подключения, проверьте ip-адрес сервера");

}

*lblLabel*.setText("Подключено");

*rhDraw*.start();

*playerButPress*.start();

}

}

});

btnNewButton.setBounds(10, 461, 122, 23);

contentPane.add(btnNewButton);

/\*final JLabel \*/*lblLabel* = **new** JLabel("");

*lblLabel*.setBounds(489, 465, 130, 14);

contentPane.add(*lblLabel*);

*textField* = **new** JTextField();

*textField*.setText("127.0.0.1");

*textField*.setBounds(142, 462, 86, 20);

contentPane.add(*textField*);

*textField*.setColumns(10);

btnNewButton.addKeyListener(*keyListener*);

}

}

**ПРИЛОЖЕНИЕ Б**

(обязательное)

**Руководство пользователя**

1. **Назначение «*Tanks*»**

Данная программа является развлекательным приложением, играя в которое можно снять напряжение, весело провести время. Достоинством этого приложение в том, что в него может сыграть каждый, кто захочет, потому что на приложение нет никаких возрастных ограничений. И всегда сохраняйте спокойствие, помните, это всего лишь игра. Программа предназначена для игры на разных компьютерах.

1. **Условия выполнения программы**

Приложение не предъявляет требований к операционной системе, для запуска требуется установленная *Java Development Kit v*1.5 или выше.

1. **Выполнение программы**

С приложением «*Tanks*» будут поставляться bat-файлы, в которых будет указываться порядок запуска программ. Для запуска игры необходимо удостовериться, что приложение *Server* запущено и, знать *ip*-адрес узла, на котором запущен сервер. После запуска приложения необходимо ввести *ip*-адрес сервера и нажать кнопку подключить, в нижнем правом углу будет выведено сообщения о подключении либо сообщение об ошибке возникающей при ошибочном вводе *ip*-адреса. Как только все игроки подключены можно управлять танком. С помощью клавиш «*w*», «*a*», «*s*», «*d*» можно двигать танк вверх, влево, вниз, вправо соответственно. Для выхода из игры нужно закрыть игровое окно.

1. **Сообщения пользователю**

При настройке, проверке и в ходе работы программы выводятся различные сообщения:

1. Сообщение об установке соединения – «Подключено»;
2. Сообщение о победе игрока – «Вы выиграли!»;
3. Сообщение о попытке подключения к несуществующему серверу – «Ошибка подключения, проверьте *ip*-адрес сервера!»;
4. Сообщение об окончании игры – «Игра окночена».

При возникновении любых аварийных ситуаций рекомендуется презапустить все модули приложения «*Tanks*».

**ПРИЛОЖЕНИЕ В**

(обязательное)

**Руководство системного программиста**

1. **Общие сведения о «*Tanks*»**

Приложение «*Tanks*» применяется исключительно в развлекательных целях. Для запуска приложения требуется установленная *Java Development Kit v*1.5 или выше, не предъявляет требований к ОС.

1. **Структура программы**

Программа «*Tanks*» состоит из двух основных модулей: *Client* и *Server*. Модуль *Client* выполняет отображение и проводит основные вычисления, модуль *Server* синхронизирует подключенных игроков. Схема взаимодействия между модулями представлена в приложении Г. Программа не рассчитана на взаимодействие с другими программами.

1. **Настройка программы**

Настройка программы производится путём регулирования числа максимальных подключений к серверу. Для установки максимального количества игроков необходимо изменять значение переменной *countPlayer* в классе *Server*.*java*.

1. **Проверка программы**

Для старта программы можно использовать *bat*-файлы поставляемые с приложением либо сгенерировать исполняемый *JAR*-архив. Первым должен запускаться сервер, затем клиенты. Результатом успешного запуска программы является появления окна с возможностью указать *ip*-адрес подключения к серверу.

1. **Сообщения программисту**

При настройке, проверке и в ходе работы программы выводятся различные сообщения:

1. Сообщение об установке соединения – «Подключено»;
2. Сообщение о победе игрока – «Вы выиграли!»;
3. Сообщение о попытке подключения к несуществующему серверу – «Ошибка подключения, проверьте *ip*-адрес сервера!»;
4. Сообщение об окончании игры – «Игра окночена».

**ПРИЛОЖЕНИЕ Г**

(обязательное)

**Схема данных сетевого приложения**