|  |  |
| --- | --- |
| Министерство образования Республики Беларусь | |
| Учреждение образования | |
| БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ | |
| ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ | |
|  | |
|  | |
| Факультет компьютерных систем и сетей | |
| Кафедра программного обеспечения информационных технологий | |
| Дисциплина: Компьютерные системы и сети (КСИС) | |
|  | |
|  | |
|  | |
|  | |
|  | |
|  | |
|  | |
|  | |
|  | |
| ПОЯСНИТЕЛЬНАЯ ЗАПИСКА | |
| к курсовому проекту на тему | |
|  | |
| «**Морской бой**» | |
|  | |
| БГУИР КП I-40 01 01 020 ПЗ | |
|  | |
|  | |
|  | |
|  | |
|  | |
|  | |
| Выполнил | Лешок К.Д. |
| студент: гр. 351004 |  |
|  |  |
|  |  |
| Проверил: | Базылев Е.Н. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| Минск 2025  СОДЕРЖАНИЕ | |

[Введение 5](#_Toc195718601)

[1 Аналитический обзор литературы 6](#_Toc195718602)

[1.1 Обзор существующих аналогов 6](#_Toc195718603)

[1.2 Перечень функциональных требований 9](#_Toc195718604)

[2 Моделирование предметной области 10](#_Toc195718605)

[2.1 Архитектура «клиент-сервер» 10](#_Toc195718606)

[2.2 Протоколы TCP и UDP 10](#_Toc195718607)

[2.3 Модель клиента на сервере 11](#_Toc195718608)

[3 Проектирование программного средства 12](#_Toc195718609)

[3.1 Структура программы 12](#_Toc195718610)

[3.2 Проектирование интерфейса программного средства 13](#_Toc195718611)

[3.3 Проектирование функционала программного средства 15](#_Toc195718612)

[4 Разработка программного средства 20](#_Toc195718613)

[4.1 Серверная логика: обработка игровых состояний 20](#_Toc195718614)

[4.2 Клиентская часть: расстановка кораблей 20](#_Toc195718615)

[4.3 Игровой интерфейс и сетевое воздействие 22](#_Toc195718616)

[4.4 Сетевая инфраструктура 23](#_Toc195718617)

[5 Тестирование программного средства 24](#_Toc195718618)

[5.1 Запуск, подключение, соединение 24](#_Toc195718619)

[5.2 Игровой процесс 26](#_Toc195718620)

[5.3 Конец игры 28](#_Toc195718621)

[6 Руководство пользователя 29](#_Toc195718622)

[6.1 Минимальные системные требования 29](#_Toc195718623)

[6.2 Установка 29](#_Toc195718624)

[6.3 Лаунчер 29](#_Toc195718625)

[6.4 Игровой процесс 30](#_Toc195718626)

[Заключение 32](#_Toc195718627)

[Список использованной литературы 33](#_Toc195718628)

[Приложение А 34](#_Toc195718629)

# Введение

«Морской бой» – классическая стратегическая игра, известная своей простотой и глубиной. Её история уходит корнями в начало XX века, а популярность закрепилась благодаря настольным версиям и цифровым адаптациям. Правила игры просты: два игрока размещают корабли на своих сетках 10×10 и по очереди атакуют клетки противника, стремясь первым уничтожить все вражеские корабли. Игра развивает логическое мышление, умение прогнозировать действия оппонента и стратегическое планирование.

С развитием сетевых технологий появилась возможность играть в «Морской бой» с противником, находящимся в любой точке мира. Для этого используются протоколы передачи данных, такие как TCP (Transmission Control Protocol), который обеспечивает надёжную доставку информации между устройствами. TCP работает на транспортном уровне модели OSI, используя сокеты для установления соединения между клиентом и сервером. Это позволяет синхронизировать состояние игры и обрабатывать действия игроков в реальном времени.

Данная курсовая работа посвящена разработке сетевой версии игры «Морской бой», включающей TCP-сервер для управления игровой логикой и клиентов с графическим интерфейсом. Основные решаемые задачи:

– изучение принципов работы TCP-сокетов и их применение для обмена данными;

– реализация алгоритмов генерации игровых досок с корректным размещением кораблей;

– проектирование протокола взаимодействия между клиентом и сервером;

– создание интуитивно понятного графического интерфейса на базе библиотеки Tkinter;

– обеспечение многопоточности для одновременной обработки действий двух игроков.

В результате будет разработано приложение, позволяющее двум игрокам сразиться в «Морской бой» через сеть. Сервер будет управлять игровой логикой, а клиенты предоставлять пользователям интерактивный интерфейс. Приложение продемонстрирует практическое применение сетевых технологий и многопоточного программирования в Python.

# Аналитический обзор литературы

## Обзор существующих аналогов

### **Морской бой «Sea Battle» на cardgames.io**

Онлайн-версия игры, изображённая на рисунке 1.1, доступная через браузер. Популярна благодаря простоте и мгновенному запуску без установки. Она занимает лидирующие позиции в поисковой выдаче по запросу «морской бой онлайн».

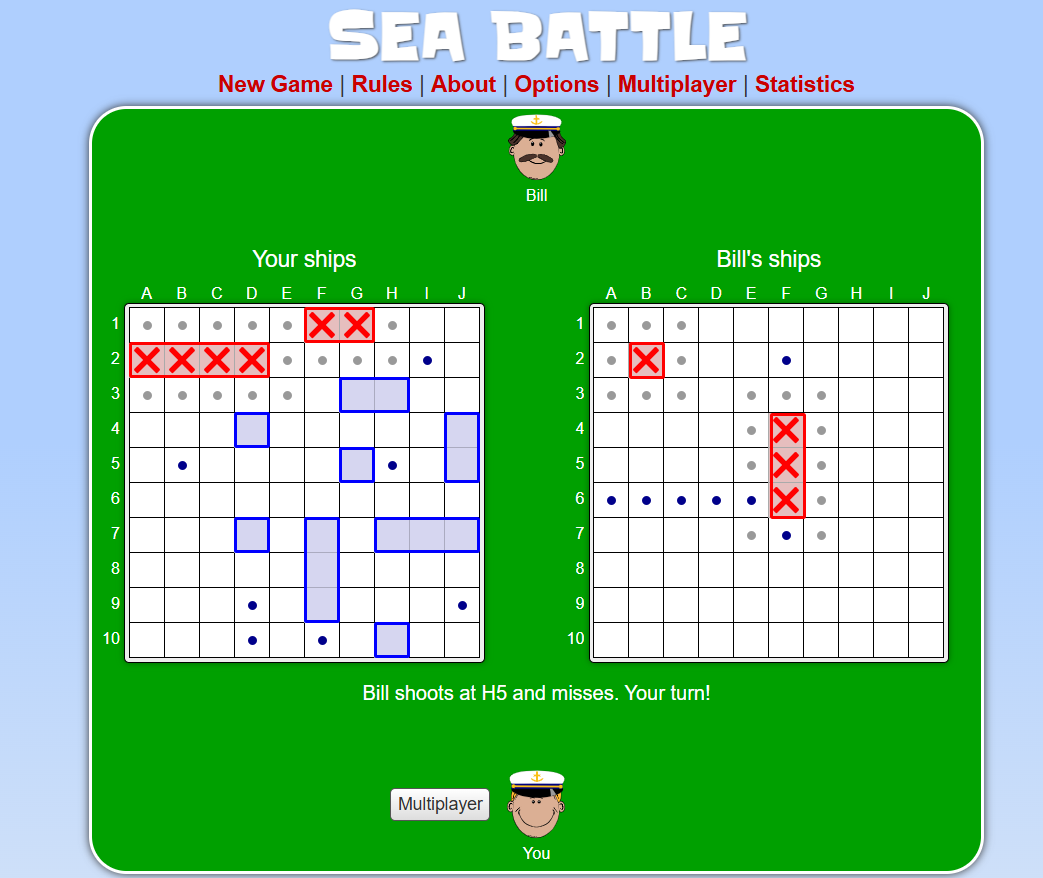


Рисунок 1.1 – Морской бой «**cardgames.io»**

Список выполняемых функций:

* игра против ИИ или реального игрока;
* визуализация попаданий и промахов.

Список достоинств данного программного средства:

* кроссплатформенность (браузер, iOS, Android);
* минималистичный и интуитивно понятный интерфейс.

Список недостатков данного программного средства:

* ограниченная настройка правил;
* ограниченная кастомизация доски.

### Мобильное приложение «Морской бой»

Программное средство, изображённое на рисунке 1.2, является одним из самых популярных приложений среди интерпретаций «Морского боя» для мобильных телефонов.

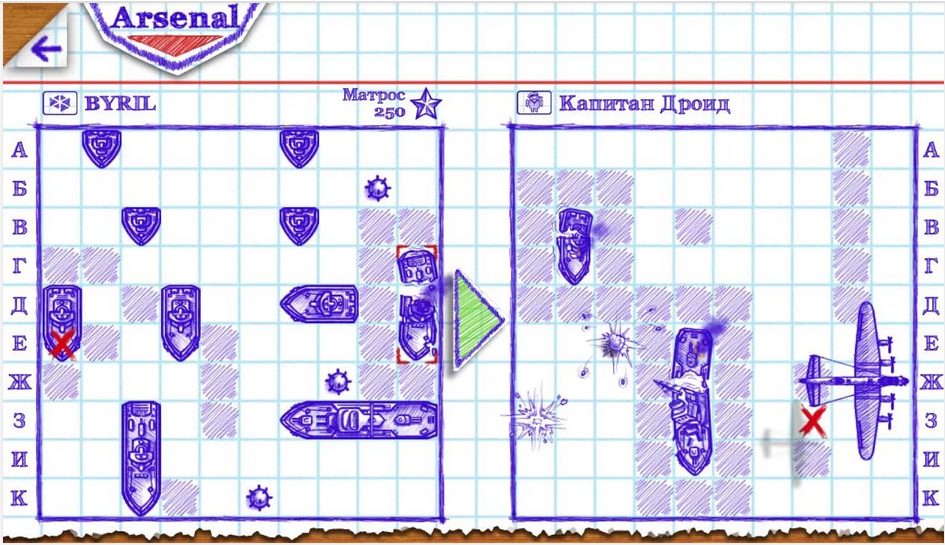


Рисунок 1.2 – Мобильное приложение «Морской бой»

Список выполняемых функций:

* игра против реального игрока;
* анимация выстрелов и разрушения кораблей.

Рассмотрим достоинства данного приложения:

* высокий уровень визуализации;
* регулярные обновления и события.

Рассмотрим недостатки данного приложения:

* наличие рекламы и микроплатежей;
* обязательная регистрация;
* высокие требования к устройству.

### Морской бой на battleship-game.org

Онлайн-игра, изображенная на рисунке 1.3, с современным интерфейсом, доступная через браузер. Является одной из популярнейших реализаций игры, доступная в Интернете. Отличается интуитивным управлением и поддержкой мультиплеера.

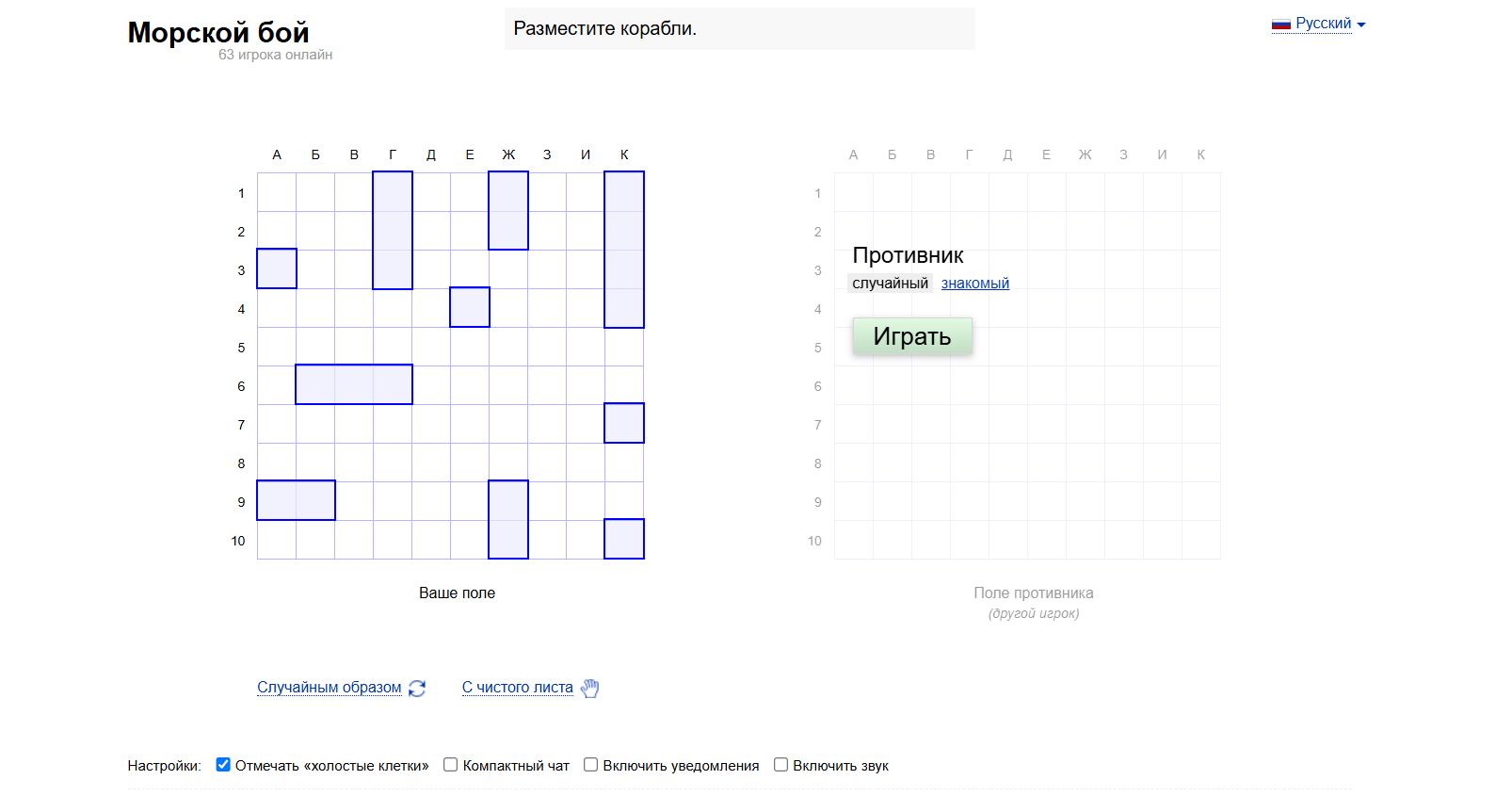


Рисунок 1.3 – Морской бой «battleship-game.org»

Список выполняемых функций:

* игра против ИИ или реального игрока;
* ручное и автоматическое размещение кораблей;
* поддержка разных языков.

Рассмотрим достоинства данного приложения:

* адаптивный дизайн для ПК и мобильных устройств;
* простое и интуитивно понятное управление;
* возможность игры со знакомыми и случайными игроками.

Рассмотрим недостатки данного приложения:

* нет оффлайн-режима;
* отсутствие кастомизации элементов игры;
* долгое ожидание случайных игроков.

## Перечень функциональных требований

После анализа аналогов был составлен следующий список требуемых функций:

– Приложение должно позволять двум игрокам сразиться в «Морской бой» по сети через TCP-соединение.

– Должны быть реализованы сервер (управляющий логикой игры) и клиенты (с графическим интерфейсом).

– Возможность выбора IP-адреса и порта для подключения через графическое меню.

– Мультиплатформенность: сервер и клиенты работают на Windows, Linux, macOS. Игроки на разных ОС могут играть друг с другом.

Данные требования обеспечивают создание кроссплатформенного сетевого приложения, соответствующего правилам классического «Морского боя» и ориентированного на удобство локального использования.

Для разработки программного средства будет использоваться язык программирования Python и среда разработки PyCharm, библиотека Tkinter, а также стандартные модули Python.

# Моделирование предметной области

## Архитектура «клиент-сервер»

Архитектура «клиент-сервер» представляет собой модель взаимодействия в распределённых системах, где роли участников разделены на поставщиков ресурсов (серверы) и потребителей (клиенты). Серверы обрабатывают запросы от клиентов, предоставляя им данные или функциональность через сетевые протоколы. Клиенты, такие как веб-браузеры или мобильные приложения, инициируют запросы и используют полученные ресурсы. Например, при загрузке веб-страницы браузер отправляет запрос серверу, который возвращает необходимые данные.

Преимущества архитектуры:

* возможность одновременного обслуживания множества клиентов;
* централизованное хранение данных, упрощающее управление и безопасность;
* организация взаимодействия между клиентами через сервер.

Недостатки:

* зависимость клиентов от доступности сервера;
* риск перегрузки сервера при высокой нагрузке;
* уязвимость к кибератакам, направленным на серверную инфраструктуру.

## Протоколы TCP и UDP

В игровой индустрии чаще применяется UDP, так как он обеспечивает минимальные задержки передачи данных, что критично для динамичных игр. Однако для данного приложения предпочтительнее TCP, гарантирующий надёжную доставку и целостность информации. TCP работает на транспортном уровне модели OSI, используя механизмы установки соединения, подтверждения получения пакетов и контроля ошибок. Это протокол типа «точка-точка», функционирующий по схеме: установка соединения → передача данных → разрыв связи.

Хотя TCP менее производителен из-за повторной отправки потерянных пакетов, для данной игры, где актуальность данных не требует мгновенного обновления, его надёжность становится ключевым преимуществом. Порт клиента в TCP выбирается из диапазона 2000–65535.

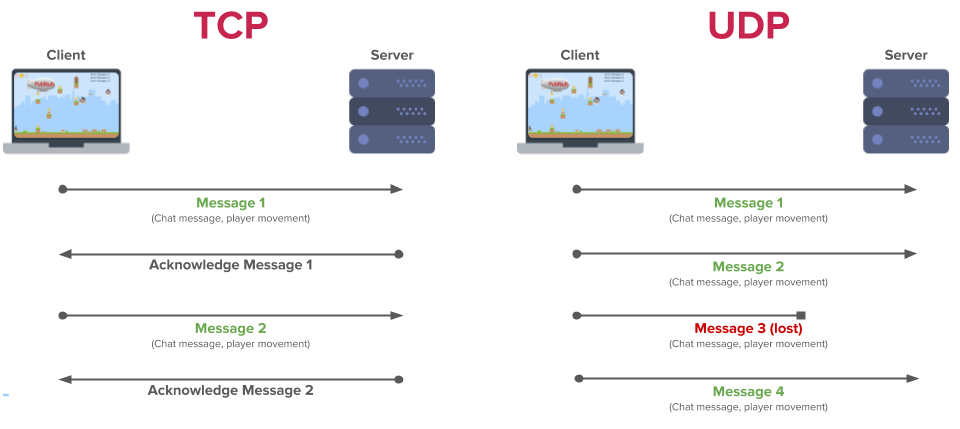


Рисунок 2.1 – Протоколы TCP и UDP

## Модель клиента на сервере

Каждое клиентское подключение на сервере представлено в виде отдельного объекта, инкапсулирующего сокет и метаданные о состоянии пользователя (например, статус «в ожидании», связка с оппонентом). Это позволяет серверу эффективно управлять подключениями, избегая прямого взаимодействия с сокетами. Такая абстракция упрощает обработку данных, маршрутизацию сообщений и отслеживание активности клиентов.

Данный подход обеспечивает гибкость при масштабировании системы и повышает безопасность за счёт централизованного контроля над клиентскими сессиями.

# Проектирование программного средства

## Структура программы

При разработке приложения будут использованы следующие модули:

1. Класс GameServer

Отвечает за серверную логику: обработку подключений игроков, управление игровыми досками, проверку правил боя (попадания, потопления кораблей), определение победителя и синхронизацию состояния между клиентами. Реализует многопоточную обработку клиентов.

1. Класс ShipPlacementWindow

Обеспечивает графический интерфейс для расстановки кораблей игроком. Включает валидацию размещения, визуализацию кораблей на сетке 10x10, выбор направления кораблей и отправку данных на сервер.

1. Класс BattleshipClient

Управляет основным игровым интерфейсом клиента: отображает свою и вражескую доску, обрабатывает атаки игрока, получает обновления состояния от сервера, визуализирует результаты выстрелов (промахи, попадания, потопления) и определяет доступность клеток для атаки.

1. Класс MainMenu

Реализует стартовое меню с выбором режима (сервер/клиент). Обрабатывает ввод сетевых параметров (IP и порт), запускает серверное приложение или инициирует подключение к серверу с последующим переходом к расстановке кораблей.

## Проектирование интерфейса программного средства

### Лаунчер

Лаунчер должен состоять из простого окна с двумя полями для ввода (логический адрес и порт) и двумя кнопками, которые осуществляют запуск cервера и клиента соответственно. Макет главного окна приложения представлен на рисунке 3.1.



Рисунок 3.1 – Окно лаунчера

### Окно расстановки кораблей

Окно расстановки кораблей должно представлять окно, содержащее внутри себя кнопку изменения направления постановки кораблей, поле для установки кораблей и кнопку подтверждения позиций кораблей. Макет главного меню приложения представлен на рисунке 3.2.

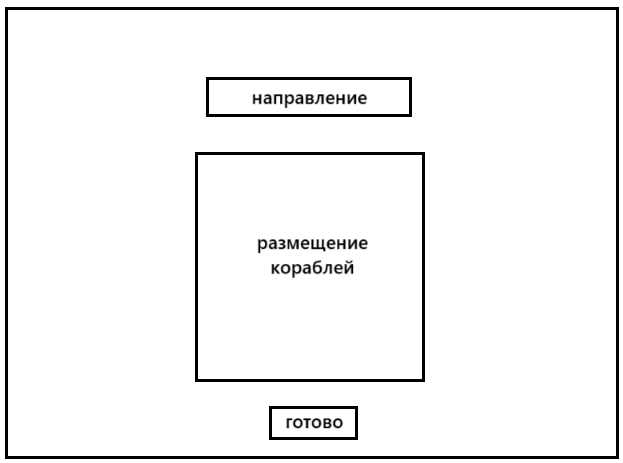


Рисунок 3.2 – Окно расстановки кораблей

### Окно игрового процесса

Окно игры должно состоять из двух частей: поле с кораблями пользователя и поле с кораблями противника. Под этими полями должна быть расположена строка состояния игры. Макет окна игры представлен на рисунке 3.3.

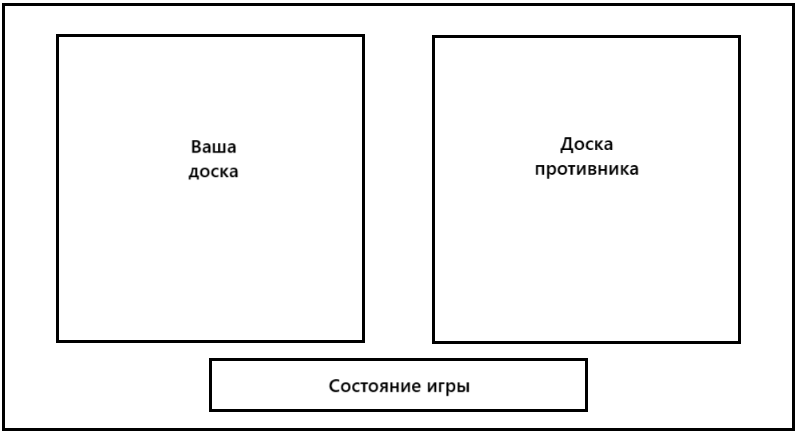


Рисунок 3.3 – Окно игрового процесса

## Проектирование функционала программного средства

При создании игры очень важно сразу определить задачи и цели, а также хорошо составить рабочие алгоритмы. Искать ошибки в коде придётся в любом случае, но хорошо написанный алгоритм упрощает это в разы. В игре должны быть использованы следующие методы:

* проверка уничтожения корабля;
* размещение кораблей на поле;
* проверка возможности размещения корабля;
* создание игрового поля.

### Проверка уничтожения корабля

Каждый выстрел обрабатывается для определения, уничтожен ли корабль полностью. Координаты попадания (x, y) помещаются в проверочный алгоритм, где, помимо текущего состояния клетки, анализируются соседние клетки на наличие целых частей корабля. Результат проверки сохраняется в логике игры. При создании используется основная процедура проверки направлений. Блок-схема алгоритма данной функции приведена на рисунке 3.4.

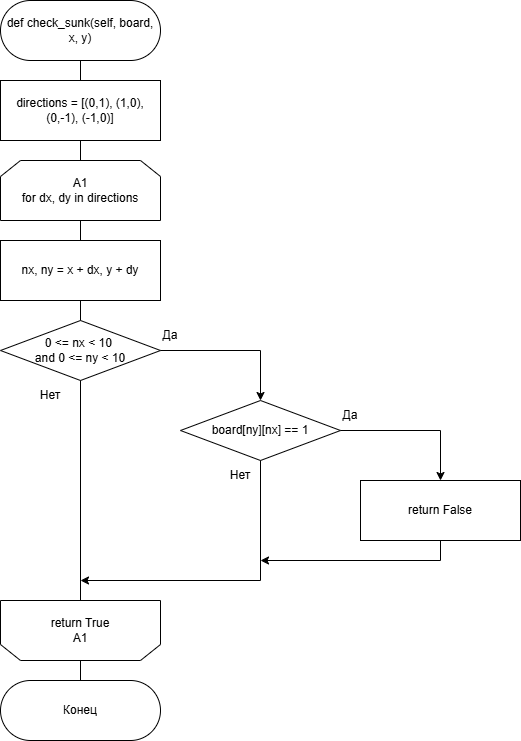


Рисунок 3.4 – Проверка уничтожения корабля

### Размещение кораблей на поле

Каждая выбранная игроком позиция обрабатывается для корректного размещения корабля. Координаты (x, y) и направление помещаются в алгоритм проверки, где, помимо текущего размера корабля, анализируется его расположение на доске. Результат валидации сохраняется в логике интерфейса. При создании используется основная процедура обновления статуса. Блок-схема алгоритма данной функции приведена на рисунке 3.5.

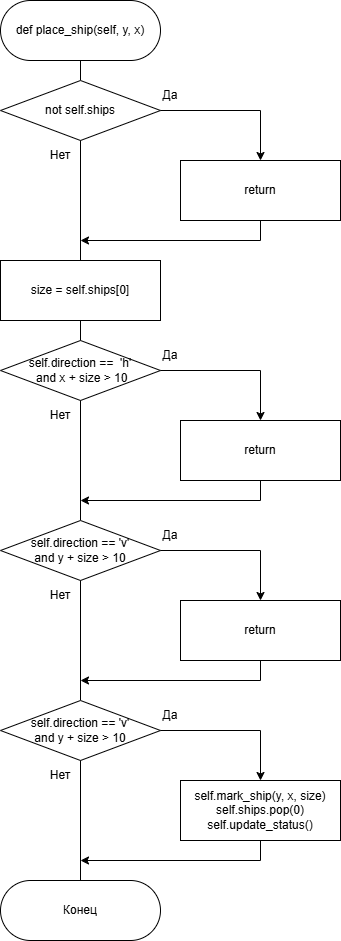


Рисунок 3.5 – Размещение кораблей на поле

### Проверка возможности размещения корабля

Каждая выбранная игроком позиция анализируется на соответствие правилам расстановки. Координаты (y, x), направление и размер корабля проверяются на отсутствие пересечений с другими кораблями и соблюдение безопасного расстояния. Результат валидации возвращается как логическое значение. Блок-схема алгоритма данной функции приведена на рисунке 3.6.

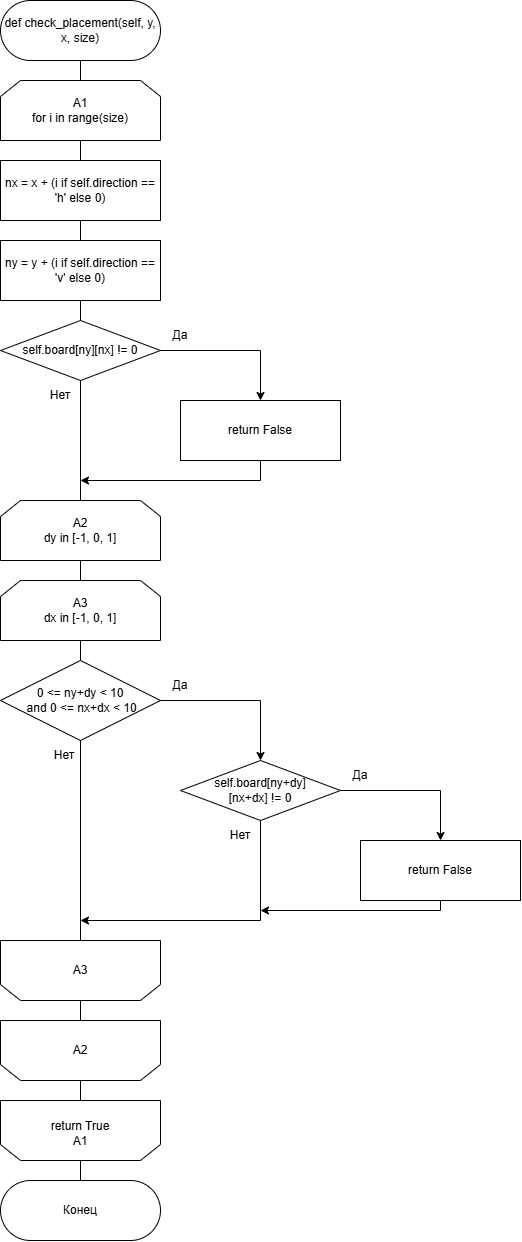


Рисунок 3.6 – Проверка возможности размещения корабля

### Создание игрового поля

Отвечает за инициализацию игровых досок (своей и противника) в графическом интерфейсе клиента. Создает матрицу кнопок, представляющих клетки поля, и настраивает их поведение в зависимости от типа доски. Блок-схема алгоритма данной функции приведена на рисунке 3.7.

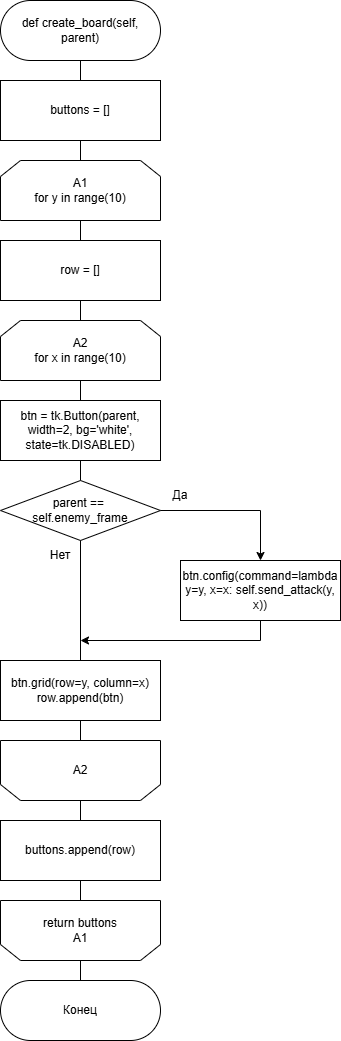


Рисунок 3.7 – Создание игрового поля

# Разработка программного средства

## Серверная логика: обработка игровых состояний

Центральным компонентом приложения является класс GameServer, который управляет сетевым взаимодействием и игровой логикой. Метод process\_move обрабатывает ходы игроков, определяя попадания в корабли и проверяя условия победы. При получении координат атаки сервер:

* Проверяет, содержит ли клетка корабль противника.
* Обновляет статус клетки.
* Рекурсивно помечает потопленные корабли через алгоритм обхода в глубину.

def check\_placement(self, y, x, size):

for i in range(size):

nx = x + (i if self.direction == 'h' else 0)

ny = y + (i if self.direction == 'v' else 0)

if self.board[ny][nx] != 0:

return False

for dy in [-1, 0, 1]:

for dx in [-1, 0, 1]:

if 0 <= ny+dy < 10 and 0 <= nx+dx < 10:

if self.board[ny+dy][nx+dx] != 0:

return False

return True

def mark\_ship(self, y, x, size):

for i in range(size):

nx = x + (i if self.direction == 'h' else 0)

ny = y + (i if self.direction == 'v' else 0)

self.board[ny][nx] = 1

self.buttons[ny][nx].config(bg='gray')

Сервер синхронизирует состояние досок между клиентами через метод update\_players, отправляя данные с текущим статусом клеток и информацией о ходе. При завершении игры (уничтожение всех кораблей) метод send\_game\_over рассылает финальные уведомления.

## Клиентская часть: расстановка кораблей

Класс ShipPlacementWindow реализует интерактивный интерфейс для размещения кораблей. Логика включает:

* Визуализацию сетки 10x10 с использованием кнопок Tkinter.
* Валидацию расстановки: предотвращение пересечений и выхода за границы.
* Динамическое изменение направления кораблей (горизонтальное/вертикальное).

def check\_placement(self, y, x, size):

for i in range(size):

nx = x + (i if self.direction == 'h' else 0)

ny = y + (i if self.direction == 'v' else 0)

if self.board[ny][nx] != 0:

return False

for dy in [-1, 0, 1]:

for dx in [-1, 0, 1]:

if 0 <= ny+dy < 10 and 0 <= nx+dx < 10:

if self.board[ny+dy][nx+dx] != 0:

return False

return True

def mark\_ship(self, y, x, size):

for i in range(size):

nx = x + (i if self.direction == 'h' else 0)

ny = y + (i if self.direction == 'v' else 0)

self.board[ny][nx] = 1

self.buttons[ny][nx].config(bg='gray')

После завершения расстановки данные отправляются на сервер через send\_board, что инициирует начало игры. Интерфейс динамически обновляет статус:

def update\_status(self):

if not self.ships:

self.status\_label.config(text="Все корабли   
 расставлены!")

self.done\_btn.config(state=tk.NORMAL)

return

self.status\_label.config(text=f"Разместите   
 {self.ships[0]}-палубный   
 корабль")

## Игровой интерфейс и сетевое воздействие

Класс BattleshipClient реализует основное игровое окно. Метод receive\_updates в отдельном потоке получает данные от сервера и обновляет доски:

def receive\_updates(self):

while not self.game\_over:

try:

data = self.client\_socket.recv(4096)

if not data:

break

self.handle\_message(json.loads(data.decode()))

except:

break

def handle\_message(self, message):

if message.get('game\_over'):

self.game\_over = True

text = "Вы победили!" if message['winner'] else "Вы   
 проиграли!"

self.status\_label.config(text=text)

messagebox.showinfo("Игра окончена", text)

self.on\_closing()

return

for y in range(10):

for x in range(10):

own\_cell = message['your\_board'][y][x]

color = 'gray'

if own\_cell == 1 else 'yellow'

if own\_cell == 2 else 'red'

if own\_cell == 3 else 'blue'

if own\_cell == -1 else 'white'

self.own\_buttons[y][x].config(bg=color)

enemy\_cell = message['enemy\_board'][y][x]

color = 'yellow'

if enemy\_cell == 2 else 'red'

if enemy\_cell == 3 else 'blue'

if enemy\_cell == -1 else 'white'

self.enemy\_buttons[y][x].config(bg=color)}

Клиент обрабатывает атаки через метод send\_attack, блокируя повторные ходы в одни и те же клетки:

def send\_attack(self, y, x):

if self.enemy\_buttons[y][x].cget('bg') != 'white':

return

try:

self.client\_socket.send(json.dumps({'x': x, 'y':   
 y}).encode())

except:

pass

## Сетевая инфраструктура

Класс MainMenu обеспечивает выбор режима (сервер/клиент) и параметров подключения. Запуск сервера выполняется в отдельном потоке для предотвращения блокировки интерфейса:

def start\_server(self):

port = self.get\_port()

if port is None: return

threading.Thread(target=lambda: GameServer(port).run(), dae  
 mon=True).start()

messagebox.showinfo("Сервер", f"Сервер запущен на порту   
 {port}")

def start\_client(self):

host = self.ip\_entry.get()

port = self.get\_port()

try:

client\_socket = socket.socket()

client\_socket.connect((host, port))

self.master.destroy()

# Запуск окон расстановки и игры

except Exception as e:

messagebox.showerror("Ошибка", f"Ошибка подключения:   
 {e}")

В данном разделе были описаны основные подпрограммы, благодаря которым программа функционирует. Были реализованы алгоритмы взаимодействия между клиетом и сервером, а также главные игровые процедуры.

# Тестирование программного средства

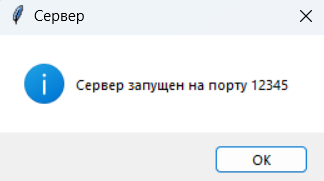
## Запуск, подключение, соединение

### Запуск на Windows (локально)

Исходные данные: запуск программы, нажатие на кнопки запуска сервера и клиентов.

Ожидаемый результат: открытие окон.

Тест пройден успешно.



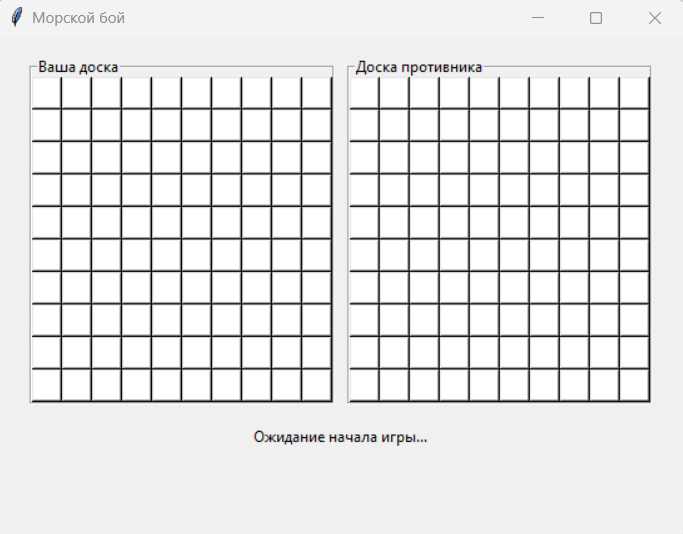
****

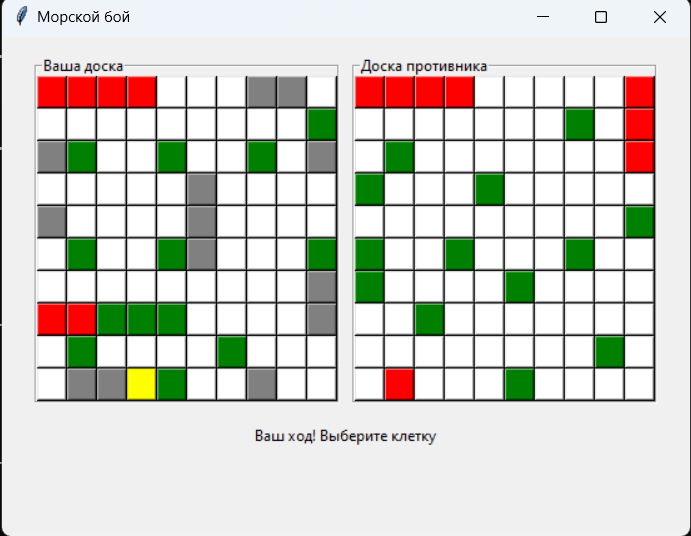
Рисунок 5.1 – Запуск на Windows (локально)

### Запуск на Windows (сетевое)

Исходные данные: запуск программы, нажатие на кнопки запуска сервера и клиентов.

Ожидаемый результат: открытие окон.

Тест пройден успешно.



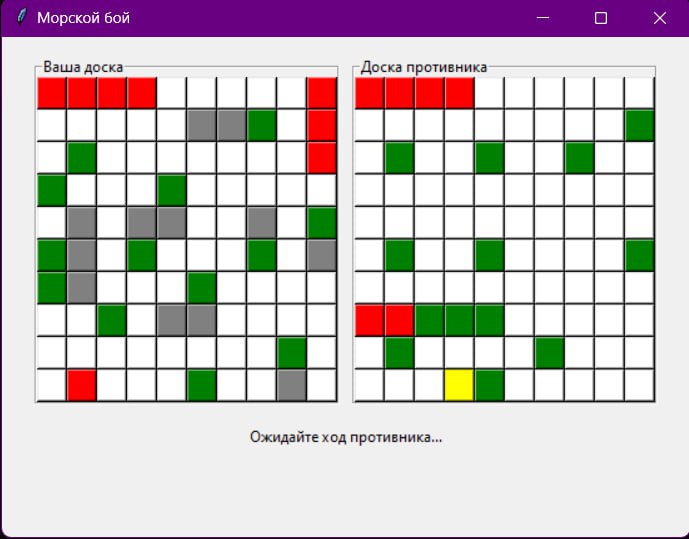


Рисунок 5.2 – Запуск на Windows (сетевой)

## Игровой процесс

Исходные данные: запрос на начало игры, связь клиентов.

Ожидаемый результат: два окна, определён начинающий игрок, корабли расставлены, изменения отображаются в обоих окнах.

Тест пройден успешно.

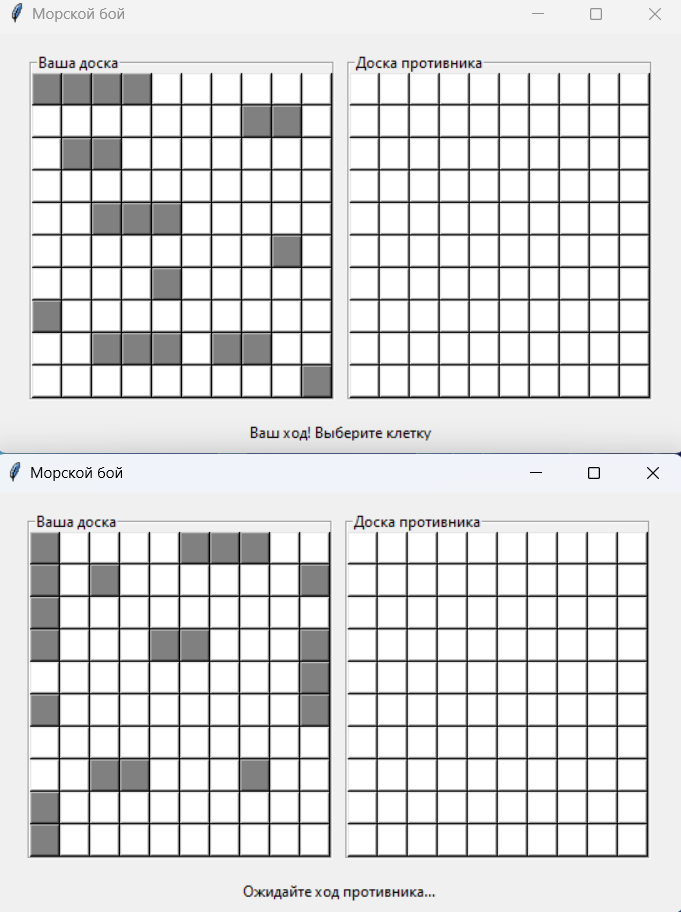


Рисунок 5.3 – Начало игрового процесса

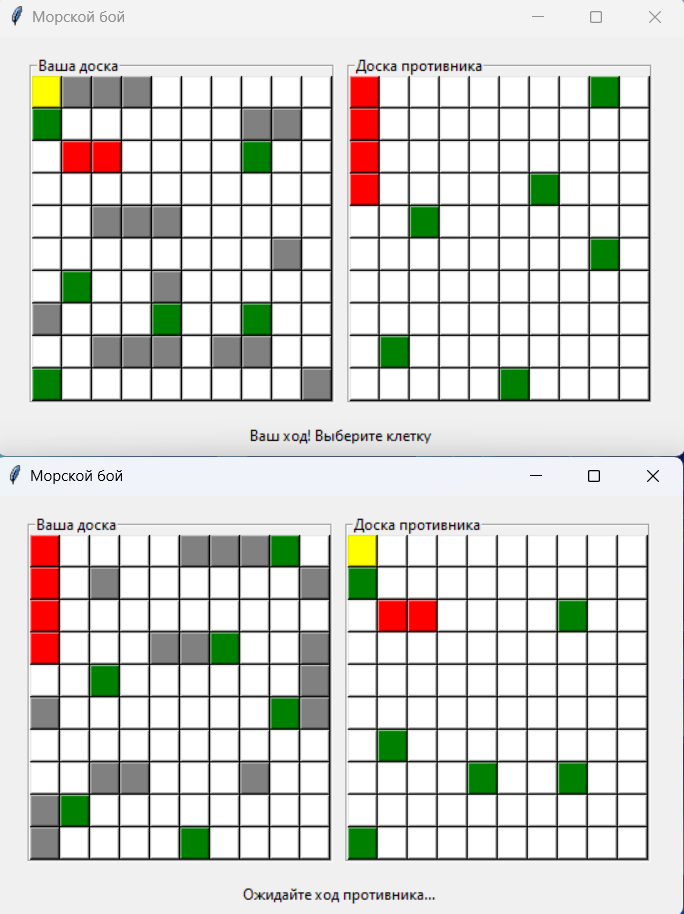


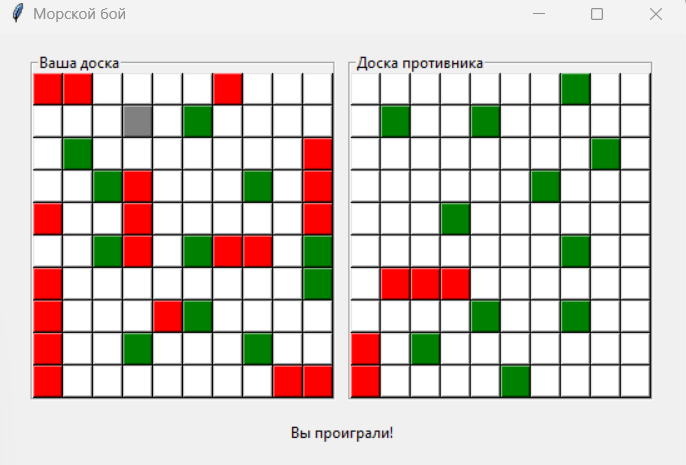
Рисунок 5.4 – Игровой процесс

## Конец игры

Исходные данные: игра выиграна.

Ожидаемый результат: сообщение о победе, играть больше нельзя.

Тест пройден успешно.



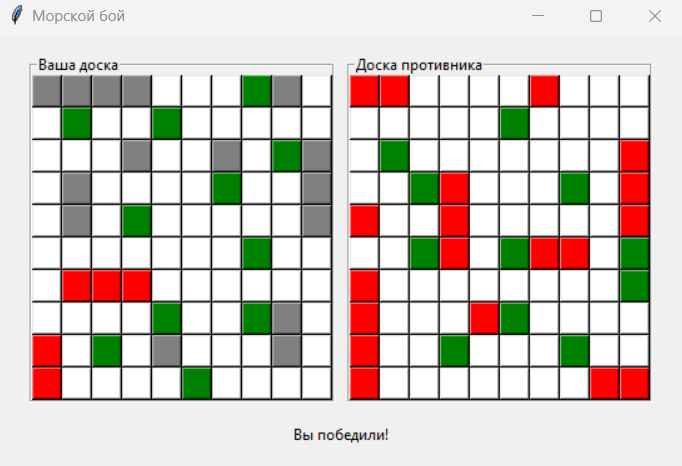


Рисунок 5.5 – Конец игры

# Руководство пользователя

## Минимальные системные требования

Для запуска игрового приложения «Морской бой» необходимо соответствие следующим требованиям:

* оперативная память 512 МБ или более;
* процессор с тактовой частотой 1 ГГц или выше
* свободное место на диске 50 мегабайт;
* операционная система Windows 7/10/11.

## Установка

Игровое приложение «Морской бой» распространяется в виде файла формата .py, требующего установки Python на компьютер.

Для запуска игры:

1. Скачайте файл Battleship.py  или скопируйте его с установочного носителя.
2. Для удобства рекомендуется поместить файл в отдельную папку на вашем компьютере.
3. Запустите приложение двойным щелчком левой кнопки мыши по файлу Battleship.py.

Для сетевой игры убедитесь, что на всех компьютерах используется одинаковая версия приложения.

## Лаунчер

Главное окно программы, изображённое на рисунке 6.1, служит центральным интерфейсом для настройки сетевого взаимодействия. Здесь пользователь может указать IP-адрес сервера (по умолчанию установлен localhost) и порт подключения (по умолчанию — 12345), а также запустить сервер или подключиться к нему через соответствующие кнопки. По умолчанию порт имеет значение 12345, но его можно изменить в пределах допустимого диапазона — от 2000 до 65535. Адрес localhost означает, что клиент будет искать сервер на текущем устройстве, что удобно для локальной игры на одном компьютере.

Для запуска локальной сессии достаточно нажать кнопку «Сервер» с настройками по умолчанию, после чего дважды открыть клиентское окно через «Клиент». Оба клиента автоматически подключатся к серверу через порт 12345, и игроки смогут сражаться друг с другом на одном экране. Если же требуется игра по сети, необходимо сначала запустить сервер на одном из устройств, предварительно узнав его локальный IP-адрес (например, 192.168.1.100). Затем на других компьютерах в поле IP вместо localhost нужно ввести этот адрес, сохранив порт 12345, и подключиться через кнопку «Клиент».

Важно помнить, что клиент не сможет установить соединение, если сервер не запущен или адрес введён некорректно. Для игры между устройствами в разных сетях (например, через интернет) потребуется настроить проброс портов на роутере серверной машины и указать её публичный IP-адрес. Программа поддерживает гибкие конфигурации: например, сервер может быть запущен на ПК №1, первый клиент — на этом же устройстве, а второй — на ПК №2. Возможен и вариант с тремя компьютерами, где сервер и клиенты работают на разных машинах.

Особое внимание стоит уделить работе в виртуальных средах: при использовании виртуальной машины рекомендуется выбрать режим сетевого моста, чтобы её IP-адрес был виден в основной сети. Если подключение не удаётся, проверьте, не блокирует ли брандмауэр порт 12345, и убедитесь, что все участники используют одинаковую версию приложения.

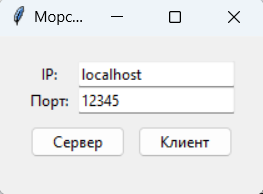


Рисунок 6.1 – Лаунчер

## Игровой процесс

После запуска игры перед пользователем открывается интерфейс, разделённый на две зоны (Рисунок 6.2). Слева расположена **ваша доска** с расставленными кораблями, где серым цветом отмечены их неповреждённые части, зеленым — промахи противника, а красным и жёлтым — попадания. Справа находится **доска противника**, на которой отображаются результаты ваших атак: белые клетки означают неизученные области, зеленые — промахи, жёлтые — попадания в корабль, а красные — полностью уничтоженные суда. В нижней части окна динамически обновляется статус хода, например: «Ваш ход! Выберите клетку» или «Ожидайте ход противника».

Цель игры — первым обнаружить и потопить все корабли противника. Каждый игрок перед началом битвы расставляет на своей доске флот, включающий корабли разного размера: один четырёхпалубный, два трёхпалубных, три двухпалубных и четыре однопалубных. Корабли не должны соприкасаться друг с другом, даже углами, а их ориентацию (горизонтальную или вертикальную) можно менять в процессе расстановки.

Игроки ходят по очереди, причём право первого хода определяется случайным образом. Когда наступает ваш черёд, вы выбираете клетку на доске противника, кликая по ней. Если в этой клетке находится часть корабля, она окрасится в жёлтый цвет, а при полном уничтожении судна все его клетки станут красными. Промахи отмечаются зеленым, и ход переходит к сопернику. Победа присуждается тому, кто первым превратит все корабли противника в обломки, отмеченные красными метками.

Предположим, вы атакуете клетку с координатами (X:5, Y:3). Если там скрывается часть трёхпалубного корабля, клетка станет жёлтой, и вы получите право на повторный ход. Допустим, следующим ударом вы попадаёте в соседнюю клетку (X:5, Y:4), которая также оказывается частью этого корабля. Теперь, когда все клетки судна поражены, они окрасятся в красный, а игра оповестит вас о его уничтожении. Если же вы выберете пустую клетку (X:2, Y:7), она станет зеленой, и ход перейдёт к противнику.

В случае если игрок закрывает окно во время боя, это автоматически считается сдачей, и победа присуждается сопернику. Программа отслеживает корректность ходов и завершает сеанс при выполнении условий победы, выводя соответствующее уведомление: «Вы победили!» или «Вы проиграли». Для сетевых партий важно, чтобы все участники были подключены к одному серверу, а используемые порты не блокировались брандмауэром.

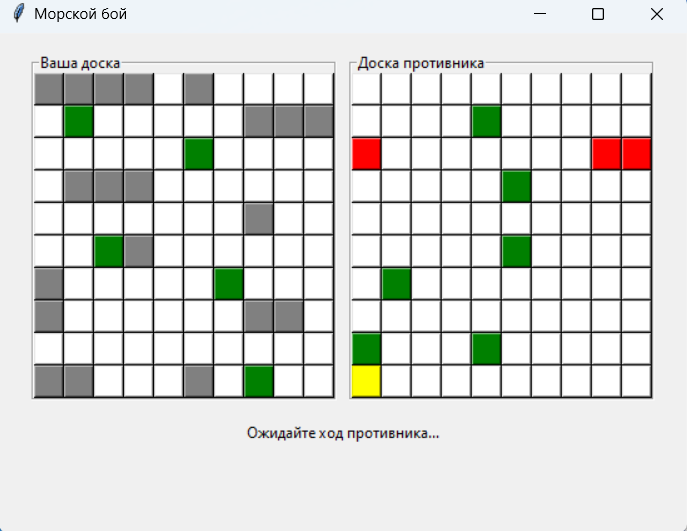


Рисунок 6.2 – Игровой процесс

# Заключение

В результате выполнения курсового проекта было разработано сетевое приложение «Морской бой», позволяющее пользователям играть друг с другом на разных устройствах через протокол TCP. Программа обладает графическим интерфейсом, созданным на базе библиотеки Tkinter, и поддерживает мультиплатформенность (Windows, Linux, macOS).

При разработке данного средства были успешно выполнены следующие поставленные задачи:

* приложение, которое позволяет сыграть с другом по сети;
* реализован сервер и клиенты;
* доступен выбор порта и логического адреса;
* мультиплатформенность: Linux, Windows, Mac. Два клиента и сервер разворачиваются на любой из этих систем, в том числе одновременно на трёх;
* игра прекращается после победы одной из сторон.

Преимущества приложения:

* Интуитивный интерфейс: разделение на «Вашу доску» и «Доску противника» с цветовой индикацией состояний клеток.
* Простота использования: запуск сервера и клиентов через лаунчер с настройкой порта и IP-адреса.
* Низкие системные требования: приложение работает даже на слабых компьютерах.

– Сетевая гибкость: поддержка локальных и удалённых сессий, включая игру через интернет.

Программа успешно реализует классические правила «Морского боя», предоставляя пользователям удобный инструмент для локальных и сетевых сражений. Проект подтвердил эффективность Python и Tkinter для создания кроссплатформенных приложений с сетевым функционалом. Разработанный опыт может быть использован для дальнейшего расширения функциональности или адаптации под другие настольные игры.

# Список использованной литературы

[1] Официальная документация Python [Электронный ресурс]. – Режим доступа: https://docs.python.org/3/ – Дата доступа: 10.03.2025.

[2] Python GUI Programming with Tkinter / D. Moore. – Издательство Packt, 2021 [Электронный ресурс]. – Режим доступа: https://www.packtpub.com/product/python-gui-programming-with-tkinter/9781801815925 – Дата доступа: 15.03.2025.

[3] Real Python: Socket Programming in Python [Электронный ресурс]. – Режим доступа: https://realpython.com/python-sockets/ – Дата доступа: 20.03.2025.

[4] GeeksforGeeks: Tkinter Tutorial [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/python-gui-tkinter/ – Дата доступа: 23.03.2025.

[5] "Python Crash Course" / Eric Matthes [Электронный ресурс]. – No Starch Press, 2023. – Режим доступа: https://nostarch.com/python-crash-course-3rd-edition – Дата доступа: 28.03.2025.

[6] Real Python: Working With JSON Data in Python [Электронный ресурс]. – Режим доступа: https://realpython.com/python-json/ – Дата доступа: 02.04.2025.

[7] Python Threading and Multithreading Guide [Электронный ресурс]. – Режим доступа: https://www.digitalocean.com/community/tutorials/how-to-use-threadpoolexecutor-in-python-3 – Дата доступа: 05.04.2025.

[8] FreeCodeCamp: Socket Programming in Python Tutorial [Электронный ресурс]. – Режим доступа: https://www.freecodecamp.org/news/socket-programming-in-python/ – Дата доступа: .04.2025.

Приложение А

Исходный код программы

import tkinter as tk

from tkinter import ttk, messagebox

import socket

import json

import threading

import random

class GameServer:

def \_\_init\_\_(self, port):

self.players = []

self.boards = {}

self.current\_turn = 0

self.ships = [4, 3, 3, 2, 2, 2, 1, 1, 1, 1]

self.server\_socket = socket.socket(socket.AF\_INET,   
 socket.SOCK\_STREAM)

self.server\_socket.bind(('', port))

self.server\_socket.listen(2)

self.running = True

def handle\_client(self, conn, player\_id):

try:

data = conn.recv(4096)

if not data:

return

self.boards[player\_id] = json.loads(data.decode())

while len(self.players) < 2 and self.running:

pass

conn.send(json.dumps({

'your\_turn': player\_id == self.current\_turn,

'your\_board': self.boards[player\_id],

'enemy\_board': [[0]\*10 for \_ in range(10)]

}).encode())

while self.running:

data = conn.recv(4096)

if not data:

break

move = json.loads(data.decode())

if 'surrender' in move:

self.send\_game\_over(1 - player\_id)

break

self.process\_move(player\_id, move)

except Exception as e:

print(f"Ошибка: {e}")

finally:

self.handle\_disconnect(player\_id)

conn.close()

def process\_move(self, player\_id, move):

x, y = move['x'], move['y']

enemy\_id = 1 - player\_id

enemy\_board = self.boards[enemy\_id]

if enemy\_board[y][x] == 1:

enemy\_board[y][x] = 2

if self.check\_sunk(enemy\_board, x, y):

self.mark\_sunk(enemy\_board, x, y)

if self.check\_win(enemy\_id):

self.send\_game\_over(player\_id)

return

else:

enemy\_board[y][x] = -1

self.current\_turn = 1 - self.current\_turn

self.update\_players()

def check\_sunk(self, board, x, y):

directions = [(0,1), (1,0), (0,-1), (-1,0)]

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < 10 and 0 <= ny < 10:

if board[ny][nx] == 1:

return False

return True

def mark\_sunk(self, board, x, y):

stack = [(x, y)]

while stack:

x, y = stack.pop()

board[y][x] = 3

for dx, dy in [(0,1), (1,0), (0,-1), (-1,0)]:

nx, ny = x + dx, y + dy

if 0 <= nx < 10 and 0 <= ny < 10 and board[ny][nx] in   
 (1, 2):

stack.append((nx, ny))

def check\_win(self, player\_id):

for row in self.boards[player\_id]:

if 1 in row:

return False

return True

def send\_game\_over(self, winner\_id):

self.running = False

for p in [0, 1]:

if p < len(self.players) and self.players[p]:

try:

self.players[p].send(json.dumps({

'game\_over': True,

'winner': winner\_id == p

}).encode())

except:

pass

def update\_players(self):

for p in [0, 1]:

if p < len(self.players) and self.players[p]:

try:

self.players[p].send(json.dumps({

'your\_turn': p == self.current\_turn,

'your\_board': self.boards[p],

'enemy\_board': [[cell if cell in (-1,2,3) else   
 0 for cell in row]

for row in self.boards[1 - p]]

}).encode())

except:

pass

def handle\_disconnect(self, player\_id):

print(f"Игрок {player\_id} отключился")

if self.running:

self.send\_game\_over(1 - player\_id)

for p in [0, 1]:

if p != player\_id and p < len(self.players) and   
 self.players[p]:

try:

self.players[p].close()

except:

pass

self.players = []

self.running = False

def run(self):

print("Сервер запущен")

while self.running and len(self.players) < 2:

try:

conn, addr = self.server\_socket.accept()

self.players.append(conn)

threading.Thread(target=self.handle\_client, args=(conn,   
 len(self.players)-1)).start()

except:

break

self.server\_socket.close()

class ShipPlacementWindow:

def \_\_init\_\_(self, master, client\_socket):

self.master = master

self.client\_socket = client\_socket

self.board = [[0]\*10 for \_ in range(10)]

self.ships = [4, 3, 3, 2, 2, 2, 1, 1, 1, 1]

self.current\_ship = None

self.direction = 'h'

self.is\_closed = False

master.title("Расстановка кораблей")

master.protocol("WM\_DELETE\_WINDOW", self.on\_closing)

self.setup\_ui()

def setup\_ui(self):

frame = tk.Frame(self.master)

frame.pack(padx=20, pady=20)

self.status\_label = tk.Label(frame, text="")

self.status\_label.grid(row=0, column=0, columnspan=11)

self.dir\_button = tk.Button(frame, text="Направление:   
 Горизонтальное",

command=self.toggle\_direction)

self.dir\_button.grid(row=1, column=0, columnspan=11, pady=5)

self.buttons = []

for y in range(10):

row = []

for x in range(10):

btn = tk.Button(frame, width=2, bg='white',

command=lambda y=y, x=x:   
 self.place\_ship(y, x))

btn.grid(row=y+2, column=x)

row.append(btn)

self.buttons.append(row)

self.done\_btn = tk.Button(frame, text="Готово",   
 command=self.send\_board,

state=tk.DISABLED)

self.done\_btn.grid(row=13, column=0, columnspan=11, pady=10)

self.update\_status()

def toggle\_direction(self):

self.direction = 'v' if self.direction == 'h' else 'h'

self.dir\_button.config(text=f"Направление: {'Вертикальное' if   
 self.direction == 'v' else 'Горизонтальное'}")

def place\_ship(self, y, x):

if not self.ships:

return

size = self.ships[0]

if self.direction == 'h' and x + size > 10:

return

if self.direction == 'v' and y + size > 10:

return

if self.check\_placement(y, x, size):

self.mark\_ship(y, x, size)

self.ships.pop(0)

self.update\_status()

def check\_placement(self, y, x, size):

for i in range(size):

nx = x + (i if self.direction == 'h' else 0)

ny = y + (i if self.direction == 'v' else 0)

if self.board[ny][nx] != 0:

return False

for dy in [-1, 0, 1]:

for dx in [-1, 0, 1]:

if 0 <= ny+dy < 10 and 0 <= nx+dx < 10:

if self.board[ny+dy][nx+dx] != 0:

return False

return True

def mark\_ship(self, y, x, size):

for i in range(size):

nx = x + (i if self.direction == 'h' else 0)

ny = y + (i if self.direction == 'v' else 0)

self.board[ny][nx] = 1

self.buttons[ny][nx].config(bg='gray')

def update\_status(self):

if not self.ships:

self.status\_label.config(text="Все корабли расставлены!")

self.done\_btn.config(state=tk.NORMAL)

return

self.status\_label.config(text=f"Разместите {self.ships[0]}-  
 палубный корабль")

def send\_board(self):

try:

if not self.is\_closed and self.client\_socket:

self.client\_socket.send(json.dumps(self.board).encode())

self.master.destroy()

except Exception as e:

if not self.is\_closed:

messagebox.showerror("Ошибка", f"Ошибка отправки: {e}")

def on\_closing(self):

self.is\_closed = True

if self.client\_socket:

self.client\_socket.close()

self.master.destroy()

class BattleshipClient:

def \_\_init\_\_(self, master, client\_socket):

self.master = master

self.client\_socket = client\_socket

self.game\_over = False

self.window\_visible = True

self.setup\_ui()

self.master.protocol("WM\_DELETE\_WINDOW", self.on\_closing)

self.receive\_thread =   
 threading.Thread(target=self.receive\_updates, daemon=True)

self.receive\_thread.start()

def setup\_ui(self):

self.master.title("Морской бой")

frame = tk.Frame(self.master)

frame.pack(padx=10, pady=10)

self.own\_frame = tk.LabelFrame(frame, text="Ваша доска")

self.own\_frame.grid(row=0, column=0, padx=5, pady=5)

self.enemy\_frame = tk.LabelFrame(frame, text="Доска   
 противника")

self.enemy\_frame.grid(row=0, column=1, padx=5, pady=5)

self.own\_buttons = self.create\_board(self.own\_frame)

self.enemy\_buttons = self.create\_board(self.enemy\_frame)

self.status\_label = tk.Label(frame, text="Ожидание начала   
 игры...")

self.status\_label.grid(row=1, column=0, columnspan=2, pady=10)

def create\_board(self, parent):

buttons = []

for y in range(10):

row = []

for x in range(10):

btn = tk.Button(parent, width=2, bg='white',   
 state=tk.DISABLED)

if parent == self.enemy\_frame:

btn.config(command=lambda y=y, x=x:   
 self.send\_attack(y, x))

btn.grid(row=y, column=x)

row.append(btn)

buttons.append(row)

return buttons

def send\_attack(self, y, x):

if self.enemy\_buttons[y][x].cget('bg') != 'white':

return

try:

self.client\_socket.send(json.dumps({'x': x, 'y':   
 y}).encode())

except:

pass

def receive\_updates(self):

while not self.game\_over:

try:

data = self.client\_socket.recv(4096)

if not data:

break

self.handle\_message(json.loads(data.decode()))

except:

break

def handle\_message(self, message):

if message.get('game\_over'):

self.game\_over = True

text = "Вы победили!" if message['winner'] else "Вы   
 проиграли!"

self.status\_label.config(text=text)

for y in range(10):

for x in range(10):

self.enemy\_buttons[y][x].config(state=tk.DISABLED)

messagebox.showinfo("Игра окончена", text)

self.on\_closing()

return

if not self.window\_visible:

self.master.deiconify()

self.window\_visible = True

for y in range(10):

for x in range(10):

own\_cell = message['your\_board'][y][x]

color = 'white'

if own\_cell == 1: color = 'gray'

elif own\_cell == 2: color = 'yellow'

elif own\_cell == 3: color = 'red'

elif own\_cell == -1: color = 'blue'

self.own\_buttons[y][x].config(bg=color)

enemy\_cell = message['enemy\_board'][y][x]

color = 'white'

if enemy\_cell == 2: color = 'yellow'

elif enemy\_cell == 3: color = 'red'

elif enemy\_cell == -1: color = 'blue'

self.enemy\_buttons[y][x].config(bg=color)

if message.get('your\_turn'):

self.status\_label.config(text="Ваш ход! Выберите клетку")

for y in range(10):

for x in range(10):

if self.enemy\_buttons[y][x].cget('bg') == 'white':

self.enemy\_buttons[y][x].config(state=tk.NORMAL)

else:

self.status\_label.config(text="Ожидайте ход противника...")

for y in range(10):

for x in range(10):

self.enemy\_buttons[y][x].config(state=tk.DISABLED)

def on\_closing(self):

if not self.game\_over:

if messagebox.askokcancel("Выход", "Вы уверены, что хотите   
 выйти?"):

self.game\_over = True

if self.client\_socket:

try:

self.client\_socket.send(json.dumps({'surrender': True}).encode())

except:

pass

self.client\_socket.close()

self.master.destroy()

else:

if self.client\_socket:

self.client\_socket.close()

self.master.destroy()

class MainMenu:

def \_\_init\_\_(self, master):

self.master = master

master.title("Морской бой")

self.setup\_ui()

def setup\_ui(self):

frame = tk.Frame(self.master)

frame.pack(padx=20, pady=20)

tk.Label(frame, text="IP:").grid(row=0, column=0)

self.ip\_entry = ttk.Entry(frame)

self.ip\_entry.insert(0, "localhost")

self.ip\_entry.grid(row=0, column=1)

tk.Label(frame, text="Порт:").grid(row=1, column=0)

self.port\_entry = ttk.Entry(frame)

self.port\_entry.insert(0, "12345")

self.port\_entry.grid(row=1, column=1)

btn\_frame = tk.Frame(frame)

btn\_frame.grid(row=2, columnspan=2, pady=10)

ttk.Button(btn\_frame, text="Сервер",   
 command=self.start\_server).pack(side=tk.LEFT, padx=5)

ttk.Button(btn\_frame, text="Клиент",   
 command=self.start\_client).pack(side=tk.LEFT, padx=5)

def get\_port(self):

try:

return int(self.port\_entry.get())

except:

messagebox.showerror("Ошибка", "Неверный порт")

return None

def start\_server(self):

port = self.get\_port()

if port is None: return

threading.Thread(target=lambda: GameServer(port).run(),   
 daemon=True).start()

messagebox.showinfo("Сервер", f"Сервер запущен на порту   
 {port}")

def start\_client(self):

host = self.ip\_entry.get()

port = self.get\_port()

if port is None:

return

try:

client\_socket = socket.socket()

client\_socket.connect((host, port))

self.master.destroy()

placement\_root = tk.Tk()

placement\_root.geometry("800x600+300+150")

ShipPlacementWindow(placement\_root, client\_socket)

placement\_root.mainloop()

game\_root = tk.Tk()

game\_root.geometry("550x400+300+150")

BattleshipClient(game\_root, client\_socket)

game\_root.mainloop()

except Exception as e:

messagebox.showerror("Ошибка", f"Ошибка подключения: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

MainMenu(root)

root.mainloop()