МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Объектно-ориентированное программирование»
Тема: Шаблонные классы

Студент гр. 3384 _____ Поздеев В.Д

Преподаватель _____ Шестопалов Р.П.

Санкт-Петербург
2024

Цель работы.

Разработать архитектуру управления игрой, включающую шаблонные классы для обработки пользовательского ввода, взаимодействия с игровой логикой и визуализации состояния игры, обеспечивая гибкость и динамичность в управлении игровым процессом.

Задание.

Создать шаблонный класс управления игрой. Данный класс должен содержать ссылку на игру. В качестве параметра шаблона должен указываться класс, который определяет способ ввода команда, и переводящий введенную информацию в команду. Класс управления игрой, должен получать команду для выполнения, и вызывать соответствующий метод класса игры.

Создать шаблонный класс отображения игры. Данный класс реагирует на изменения в игре, и производит отрисовку игры. То, как происходит отрисовка игры определяется классом переданном в качестве параметра шаблона.

Реализовать класс считывающий ввод пользователя из терминала и преобразующий ввод в команду. Соответствие команды введенному символу должно задаваться из файла. Если невозможно считать из файла, то управление задается по умолчанию.

Реализовать класс, отвечающий за отрисовку поля.

Примечание:

Класс отслеживания и класс отрисовки рекомендуется делать отдельными сущностями. Таким образом, класс отслеживания инициализирует отрисовку, и при необходимости можно заменить отрисовку (например, на GUI) без изменения самого отслеживания

После считывания клавиши, считанный символ должен сразу обрабатываться, и далее работа должна проводить с сущностью, которая представляет команду.

Для представления команды можно разработать системы классов или использовать перечисление enum.

Хорошей практикой является создание “прослойки” между считыванием/обработкой команды и классом игры, которая сопоставляет команду и вызываемым методом игры. Существуют альтернативные решения без явной “прослойки”

При считывания управления необходимо делать проверку, что на все команды назначена клавиша, что на одну клавишу не назначено две команды, что на одну команду не назначено две клавиши.

Выполнение работы.

Класс *gameController*

Объявление класса *gameController* с параметром шаблона gameInput указывает, что класс может работать с различными типами входных данных игры. Класс наследуется от *messageHandler*, предполагая, что обрабатывает сообщения. В классе объявлены три члена: *input* типа *gameInput* — хранит входные данные игры; *game* типа *Game* (с ссылкой) — ссылка на объект игры, с которой взаимодействует контроллер; *handler* типа *messageHandler** — указатель на другой обработчик сообщений, возможно, для делегирования обработки некоторых сообщений. Таким образом, *gameController* управляет игрой, используя входные данные, взаимодействует с объектом игры и, при необходимости, использует другой обработчик сообщений.

Конструктор класса *gameController* принимает ссылку на объект *Game*. Инициализируется член game этой ссылкой. Затем вызывается метод *setNext* у текущего объекта, передавая в него указатель на объект game. Это предполагает, что *setNext* устанавливает цепочку обработки сообщений или действий, где game получает сообщение после *gameController*. Далее вызывается метод *setNext* у объекта input, передавая указатель на текущий объект gameController. Это устанавливает цепочку обработки, где gameController получает сообщение после input. Наконец, вызывается метод setControls у объекта input, настраивающий элементы управления игры. В итоге, устанавливается цепочка обработки: *input -> gameController -> game*.

Метод *update* класса *gameController* вызывает метод *update* объекта *input*. Это означает, что обновление состояния контроллера сводится к обновлению

состояния входных данных игры. Метод *input.update()* считывает новые данные с устройств ввода.

Метод *Handle* обрабатывает сообщения типа *Message*. Проверяет, является ли переданное сообщение экземпляром *keyMessage*. Если да, то выполняется приведение типа к *keyMessage* и обработка нажатых клавиш с помощью switch по перечислению *Key*. В зависимости от нажатой клавиши (*Key::pointer_up, Key::pointer_down* и т.д.) изменяются координаты game.pointer или вызываются функции игры (*game.main_action, game.extra_action_0, game.extra_action_1, game.save, game.load*). Обработка *Key::save_action* и *Key::load_action* включает генерацию и отправку сообщений в handler для отображения сообщений пользователю об успехе или ошибке. Обработка ошибок при загрузке (*std::runtime_error, nlohmann::json_abi_v3_11_3::detail::parse_error*, *nlohmann::json_abi_v3_11_3::detail::type_error*) также включает отправку сообщений об ошибках через *handler*. Если сообщение не является *keyMessage*, передается следующему обработчику в цепочке (*handler->Handle(std::move(message))*).

Метод *setNext* устанавливает следующий обработчик сообщений в цепочке. Принимает указатель на объект *messageHandler* и присваивает его члену *handler* класса *gameController*. Это позволяет передавать сообщения дальше по цепочке обработки, если текущий обработчик не может или не должен обрабатывать данное сообщение.

Класс *gameTracker*

Конструктор класса *gameTracker* принимает ссылку на объект *Game* и инициализирует член game этой ссылкой. Затем вызывается метод setupGame объекта game, передавая указатель на текущий объект *gameTracker*. Это предполагает, что метод *setupGame* в классе *Game* инициализирует игру,

используя переданный *gameTracker* для вывода информации или управления игровым процессом.

Метод *update* класса *gameTracker* вызывает метод *update* объекта *output*. Это предполагает, что объект *output* отвечает за отображение или обновление выходных данных игры, и этот метод обновляет состояние вывода.

Метод *Handle* класса *gameTracker* обрабатывает сообщения разных типов. Проверяет тип сообщения и, в зависимости от типа, вызывает соответствующий метод объекта *output*: *sendText* для сообщений типа *textMessage*, *sendField* для сообщений типа *playFieldMessage* и *sendPointer* для сообщений типа pointerMessage. В каждом случае происходит преобразование *std::unique_ptr<Message>* к соответствующему типу с помощью *static_cast* и передача его методу *output*. Важно отметить использование *message.release()*, передающего владение указателем методу *output*, а также создание нового *std::unique_ptr* для безопасного управления памятью. Таким образом, *gameTracker* рутинно обрабатывает и передает различные типы сообщений в подсистему вывода.

Метод *setNext* класса *gameTracker* устанавливает следующий обработчик сообщений в цепочке обработки. Принимает указатель на объект *messageHandler* и присваивает его члену *handler*. Если *gameTracker* не может обработать сообщение, передаст его следующему обработчику в цепочке.

Класс *controlReader* содержит карту *default_controls*, которая сопоставляет строки с элементами перечисления *Key*. Этот словарь хранит значения настроек управления по умолчанию для игры.

Конструктор класса *controlReader* инициализирует карту *default_controls*, заполняя ее значениями по умолчанию для различных клавиш. Каждое имя

клавиши (строка) сопоставляется с соответствующим значением из перечисления Key, определяющим действие, которое должна выполнить эта клавиша. Например, клавиша "*W*" связана с действием *Key::pointer_down*, "*Return*" — с *Key::main_action* и так далее.

Функция *getKeyFromString* преобразует строковое представление клавиши в значение перечисления *Key*. Принимает строку *keyStr* в качестве аргумента и возвращает соответствующее значение *Key*. Если строка не соответствует ни одному из известных значений, функция бросает исключение *std::invalid_argument* с сообщением об ошибке, указывающим на неизвестную клавишу. Функция реализована с помощью последовательности условий *if*, проверяющих соответствие входной строки каждому элементу перечисления *Key*.

Оператор перегрузки *operator()* класса *controlReader* возвращает карту controls, содержащую пользовательские настройки управления, загруженные из файла. Сначала создаётся пустая карта controls. Затем функция *fileRead* считывает данные из файла *seabattle::CONTROL_DIR* и сохраняет в переменной *controls_data* типа *json*. Далее, происходит итерация по элементам *controls_data*. Каждая пара ключ-значение из *controls_data* преобразуется: ключ (*valueStr*) становится ключом в *controls*, а значение (*keyStr*) преобразуется в элемент перечисления *Key* с помощью функции *getKeyFromString* и присваивается в качестве значения в controls. Проверка на наличие дубликатов значений (*controls.contains(valueStr)*) предотвращает конфликты в настройках. Наконец, проверяется соответствие размера загруженных настроек (*controls*) и настроек по умолчанию (*default_controls*). Несовпадение размеров вызовет исключение. В результате возвращается карта пользовательских настроек управления.

Метод *getDefaultControls* класса *controlReader* возвращает карту *default_controls*, содержащую настройки управления по умолчанию. Это

позволяет получить доступ к предустановленным значениям клавиш, если чтение из файла настроек не удалось или не требуется.

Класс *GUIDrawText* предназначен для отрисовки текста на графическом интерфейсе с использованием библиотеки *SDL* и *SDL_ttf*. Содержит указатели на рендерер *SDL* (*renderer*) и три шрифта разных размеров (*big_font*, *medium_font*, *small_font*). Также содержит переменные *title* и *log* для хранения текста заголовка и лога сообщений соответственно, использующие класс *textMessage*. Класс предоставляет методы для отрисовки текста на экране.

Конструктор класса *GUIDrawText* инициализирует библиотеку *SDL_ttf* с помощью *TTF_Init()*. При неудаче выводится сообщение об ошибке. Затем создаются три шрифта с разными размерами, используя *TTF_OpenFont()*, с указанием пути к файлу шрифта (*seabattle::FONT_DIR*) и размеров шрифта (*seabattle::BIG_FONT_SIZE, seabattle::MEDIUM_FONT_SIZE, seabattle::SMALL_FONT_SIZE*). Если создание любого из шрифтов завершится неудачей, выводится сообщение об ошибке и выбрасывается исключение *std::runtime_error*.

Функция *GUIDrawText::enumToColor* принимает значение перечисления *textColor* в качестве аргумента и возвращает структуру *SDL_Color*, представляющую соответствующий цвет. Внутри функции используется оператор switch для обработки различных значений перечисления. Каждый case соответствует определенному цвету из перечисления (*green, red, yellow, white, purple, blue, black*), и возвращает соответствующую структуру *SDL_Color*, инициализированную значениями *RGB*. Значения *RGB* представляют интенсивность красного, зеленого и синего цветов соответственно, от *0* до *255*. В блоке *default* возвращается белый цвет (*{255, 255, 255}*) в случае, если переданное значение перечисления не соответствует ни одному из указанных случаев. Функция преобразует символическое представление цвета из

перечисления в его численное представление, необходимое для использования в *SDL (Simple DirectMedia Layer)* для отображения текста.

Функция *GUIDrawText::drawText* отвечает за отрисовку текста на экране с помощью *SDL*. Принимает строку текста, координаты, цвет, размер шрифта и флаг центрирования в качестве входных параметров. Вначале, в зависимости от переданного размера шрифта (*fontSize*), выбирается соответствующий шрифт (*big_font, medium_font или small_font*). Если строка не пуста, функция создает поверхность текста (*SDL_Surface*) с помощью *TTF_RenderText_Solid*, используя выбранный шрифт, текст и заданный цвет. При возникновении ошибки, выводится сообщение об ошибке и выбрасывается исключение *std::runtime_error*. Затем, из этой поверхности создается текстура (*SDL_Texture*) с помощью *SDL_CreateTextureFromSurface*. Опять же, при ошибке выбрасывается исключение. Создается прямоугольник *SDL_Rect* (*renderQuad*), определяющий область отрисовки, с координатами, полученными из coordinates, и шириной и высотой, взятыми из созданной поверхности текста. Если флаг *is_centered* установлен, координаты x прямоугольника корректируются для центрирования текста. После этого, текст отрисовывается на экране с помощью *SDL_RenderCopy*. Наконец, текстура и поверхность освобождаются из памяти, а функция возвращает *renderQuad*, содержащий информацию об области отрисованного текста. Если строка пуста, выбрасывается исключение *std::runtime_error*.

Функция *GUIDrawText::redirectText* принимает объект *textMessage* в качестве аргумента, который содержит текст и информацию о его позиции (*textPosition*). В зависимости от позиции (*title* или *log*), функция обновляет соответствующую переменную. Если позиция — *title*, то переменная *title* присваивается значение объекта text. Если позиция — *log*, то функция сдвигает элементы массива *log* на одну позицию назад, начиная с последнего элемента, и помещает новый объект *text* в начало массива. Таким образом, массив *log*

работает как *LIFO*-буфер (*Last-In, First-Out*), хранящий заданное количество (*seabattle::LOG_LENGTH*) последних сообщений.

Функция *GUIDrawText::drawTitle* отрисовывает заголовок на экране. Вычисляет координаты для заголовка, добавляя к центральной точке экрана по горизонтали (*seabattle::WIDTH/2*) и нулевой точке по вертикали, смещение *top_indent (0, 10)*. Если сообщение заголовка (*title.msg*) не пустое, функция вызывает функцию *drawText* для отрисовки текста заголовка с вычисленными координатами, используя цвет из *title.color*, большой размер шрифта (*big*) и центрирование по горизонтали (*true*).

Функция *GUIDrawText::drawLog* отрисовывает лог сообщений на экране. Инициализирует начальные координаты (*coordinates*), смещенные от точки (*seabattle::WIDTH/6, seabattle::HEIGHT*11/12*). Цикл перебирает массив *log* сообщений. Для каждого непустого сообщения (*log[i].msg*) функция сначала рисует прямоугольник-подложку черного цвета с полупрозрачностью (*alpha = 150*) с помощью *SDL_RenderFillRect*, затем отрисовывает само сообщение с помощью *drawText*, используя маленький размер шрифта (*small*) и цвет из *log[i].color*. После отрисовки каждого сообщения, координаты y уменьшаются на высоту отрисованного текста (*renderQuad.h*), чтобы следующее сообщение отображалось ниже предыдущего. Таким образом, сообщения лога выстраиваются вертикально, снизу вверх.

Оператор *()* перегружен для объекта *GUIDrawText*, превращая его в функтор. Вызов объекта *GUIDrawText()* эквивалентен вызову *GUIDrawText::drawLog()*; *GUIDrawText::drawTitle()*;. То есть, этот оператор вызывает последовательно отрисовку лога сообщений и заголовка.

Деструктор *GUIDrawText::~GUIDrawText* освобождает ресурсы, занятые шрифтами SDLttf. Закрывает шрифты *big_font*, *medium_font* и *small_font* с

помощью *TTFCloseFont* и завершает работу библиотеки *SDLttf* с помощью *TTF_Quit()*. Это гарантирует корректное освобождение памяти и ресурсов, используемых объектом *GUIDrawText*, после его удаления.

Класс *GUIDrawField* хранит указатели на рендерер *SDL* (*SDL_Renderer**) и объект для отрисовки текста (*GUIDrawText**). Предназначен для управления отрисовкой игровых полей (через *renderer*) и названий полей (через *textDrawer*) на игровом поле или подобной области. Помимо указателей на рендерер (*renderer*) и объект для отрисовки текста (*textDrawer*), класс содержит два массива указателей *fields* и *pointer*. *fields* – это массив из двух элементов, каждый из которых является *std::unique_ptr<playFieldMessage>*, хранящих информацию о состоянии игрового поля для двух игроков. *pointer* – это *std::unique_ptr<pointerMessage>*, хранящий информацию о положении курсора или указателя на игровом поле. Использование *std::unique_ptr* гарантирует автоматическое управление памятью и предотвращает утечки памяти.

Функция *GUIDrawField::drawOutline* отрисовывает рамку вокруг игрового поля. Принимает координаты верхнего левого угла поля (*coordinates*), его размеры в клетках (*size*), размер одной клетки (*size_cell*) и имя поля (field_name). Сначала создается прямоугольник *field_outline*, определяющий размеры и положение рамки, с учетом отступов в 10 пикселей. Затем устанавливается серый цвет (*200, 200, 200*) для рамки, и отрисовывается с помощью *SDL_RenderFillRect*. После этого вычисляются координаты для отрисовки имени поля (*name_coordinates*), расположенного под рамкой, и имя поля отрисовывается с помощью объекта *textDrawer* в центре, используя средний размер шрифта (*medium*) и белый цвет.

Функция *GUIDrawField::drawPointer* отрисовывает указатель на игровом поле. Использует информацию из *pointer->area* (прямоугольник, определяющий размер указателя) и *pointer->coordinates* (координаты указателя). Вложенные

циклы перебирают все клетки внутри прямоугольника указателя. Для каждой клетки вычисляются координаты на экране (*cell_coordinates*), создается прямоугольник *cell*, который заполняется цветом указателя (*seabattle::POINTER_COLOR*) и обводится тонкой темной рамкой. Таким образом, рисуется прямоугольный указатель, состоящий из заполненных цветом и обведенных клеток.

Функция *GUIDrawField::drawField* отрисовывает игровое поле. Перебирает все клетки поля (*field*) с помощью вложенных циклов. Для каждой клетки (*cell*) определяется цвет в зависимости от её состояния (*cell.state*) и флага тумана войны (*fog*). Если туман войны включен, то нераскрытые клетки отображаются цветом *seabattle::CELL_UNKNOWN*, иначе цвет зависит от состояния клетки и сегмента корабля в ней. Затем, для каждой клетки создается прямоугольник *cell_rect*, который заполняется рассчитанным цветом с помощью *SDL_RenderFillRect* и обводится темной рамкой (*SDL_RenderDrawRect*). Наконец, вокруг каждой клетки рисуется еще одна, чуть большая рамка (*outline*), также темного цвета. Таким образом, каждая клетка поля отрисовывается как прямоугольник с заливкой, соответствующей её состоянию, и с двумя рамками.

Перегруженный оператор *()* для класса *GUIDrawField* отрисовывает игровые поля. Цикл перебирает массив *fields*. Для каждого непустого поля (*field*) вычисляются размеры и координаты отрисовки на основе его размера (size), позиции (*field->position*) и доступного пространства экрана. Размер клетки (*size_cell*) определяется как минимальное значение из ширины и высоты доступного пространства, деленного на количество клеток по соответствующей оси. Затем отрисовывается рамка поля (*drawOutline*), само поле (*drawField*), а если *field->draw_pointer* истинно, то и указатель (*drawPointer*). Позиционирование полей происходит относительно центра экрана или по краям в зависимости от значения *field->position.*

Функция *GUIDrawField::setField* устанавливает игровое поле. Принимает указатель на *playFieldMessage* и, если поле видимо (*field->visible*), перемещает его владение в массив *fields* в зависимости от позиции поля (*field->position*). Если поле невидимо, то оба элемента массива *fields* устанавливаются в nullptr. Использование *std::move* эффективно передает владение объектом, избегая лишнего копирования.

Функция *GUIDrawField::setRenderer* устанавливает указатель на рендерер *SDL* (*SDL_Renderer**) для объекта *GUIDrawField*. Этот рендерер используется функциями отрисовки (*drawOutline*, *drawField*, *drawPointer*) для вывода графических элементов на экран. Функция необходима для инициализации объекта *GUIDrawField* указателем на активный рендерер *SDL*.

Функция *GUIDrawField::setTextDrawer* устанавливает указатель на объект *GUIDrawText* для объекта *GUIDrawField*. Этот объект используется для отрисовки текста на игровом поле (например, для отображения имени поля). Функция необходима для инициализации *GUIDrawField* объектом, ответственным за отрисовку текста.

Функция *GUIDrawField::setPointer* устанавливает указатель на игровое поле. Принимает *std::unique_ptr<pointerMessage>* и перемещает его владение в член pointer класса *GUIDrawField*. Использование *std::move* предотвращает лишнее копирование данных и обеспечивает корректное управление памятью.

Класс *GUIOutput*

Класс *GUIOutput* инкапсулирует объекты, необходимые для отрисовки графического интерфейса. Содержит указатели на окно *SDL* (*SDL_Window**) и рендерер (*SDL_Renderer**), а также объекты *textDrawer* и *fieldDrawer* для отрисовки текста и игрового поля соответственно. Этот класс, по всей

видимости, является центральным для управления отрисовкой всего графического интерфейса игры.

Конструктор класса *GUIOutput* инициализирует *SDL*, создает окно и рендерер. Сначала проверяется успешность инициализации *SDL*. Затем создается окно с заданными размерами (*seabattle::WIDTH, seabattle::HEIGHT*) и заголовком "*Sea Battle*". Создается рендерер с аппаратным ускорением и вертикальной синхронизацией. Устанавливается фоновый цвет и выполняется очистка рендерера. В завершении, рендерер передается в объекты *textDrawer* и *fieldDrawer* для инициализации.

Функция *GUIOutput::update* обновляет и перерисовывает графический интерфейс. Устанавливает фоновый цвет (*seabattle::BACKGROUND_COLOR*), очищает рендерер (*SDL_RenderClear*), вызывает отрисовку текста (*textDrawer()*) и игрового поля (*fieldDrawer()*), и наконец, отображает содержимое рендерера на экране (*SDL_RenderPresent*). Эта функция, вызывается в цикле игры для постоянного обновления изображения на экране.

Функция *GUIOutput::sendText* передает текстовое сообщение в объект *textDrawer* для отображения. Принимает *std::unique_ptr<textMessage>* и передает его содержимое в функцию *redirectText* объекта *textDrawer*. Обратите внимание, что владение *textMessage* не передается — функция работает с копией данных.

Функция *GUIOutput::sendField* передает информацию об игровом поле в объект *fieldDrawer* для отрисовки. Принимает *std::unique_ptr<playFieldMessage>* и передает владение этим объектом в функцию *setField* объекта *fieldDrawer* с помощью *std::move*. Это эффективно передает владение указателем без лишнего копирования данных.

Функция *GUIOutput::sendPointer* передает информацию об указателе в объект *fieldDrawer*. Принимает *std::unique_ptr<pointerMessage>* и передает владение им функции *setPointer* объекта *fieldDrawer* используя *std::move*. Это обеспечивает эффективную передачу данных и управление памятью.

Деструктор *GUIOutput::~GUIOutput* освобождает ресурсы *SDL*, занятые объектом *GUIOutput*. Уничтожает рендерер (*SDL_DestroyRenderer*) и окно (*SDL_DestroyWindow*), а затем завершает работу библиотеки *SDL* (*SDL_Quit*). Это гарантирует корректное освобождение памяти и предотвращает утечки ресурсов.

Класс *GUIInput*

Класс *GUIInput* наследуется от *messageHandler* и предназначен для обработки ввода пользователя. Содержит указатель на другой обработчик сообщений (*handler*) и *std::map controls*, для сопоставления имен действий (строки) с клавишами (*Key*). Это позволяет настраивать управление игрой, связывая определенные действия с нажатием конкретных клавиш.

Функция *GUIInput::setControls* пытается загрузить настройки управления из файла (*JSON*) с помощью *controlReader()()*. При возникновении ошибок разного типа (*std::invalid_argument, nlohmann::json_abi_v3_11_3::detail::parse_error, nlohmann::json_abi_v3_11_3::detail::type_error, std::runtime_error*), соответствующее сообщение об ошибке выводится в лог (*textPosition::log*) с помощью функции *Handle*. Если загрузка настроек не удалась, устанавливаются настройки по умолчанию с помощью *controlReader().getDefaultControls()*, и сообщение об этом выводится в лог.

Функция *GUIInput::update* обрабатывает события *SDL*. Использует *SDL_PollEvent* для получения событий из очереди. В зависимости от типа

события (*event.type*), выполняется соответствующее действие. Если событие — *SDL_QUIT*, отправляется сообщение *Key::quit*. Если событие — *SDL_KEYDOWN*, определяется имя нажатой клавиши (*key_name*) и, если это имя есть в *controls*, отправляется сообщение с соответствующим значением из *controls* — *controls[key_name]*. Таким образом, функция преобразует события *SDL* в сообщения, которые затем обрабатываются другими компонентами системы.

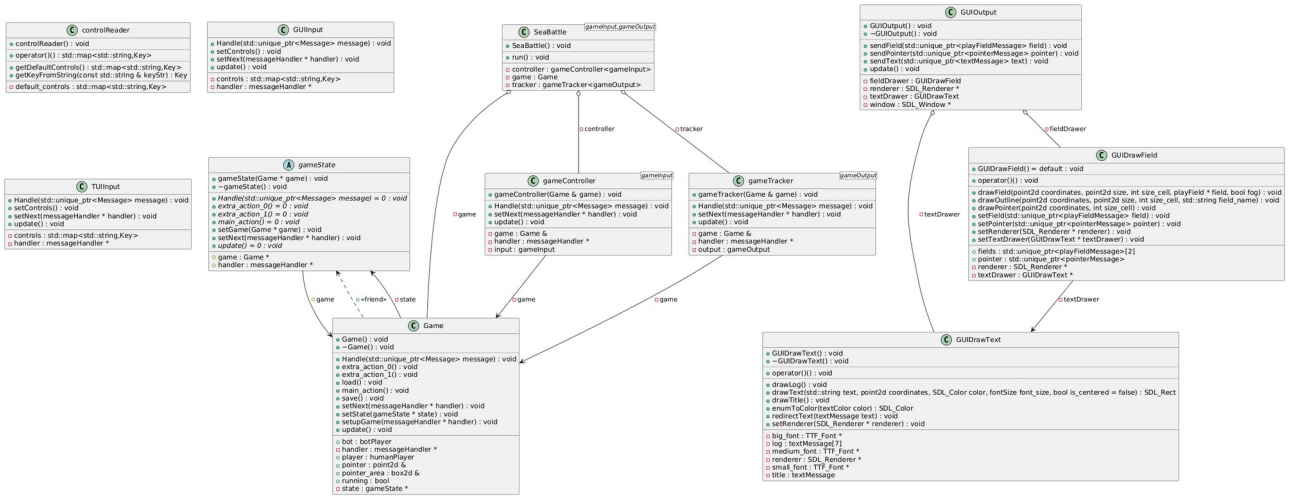
Функция *GUIInput::Handle* передает полученное сообщение (*Message*) следующему обработчику (*handler*) в цепочке обработки. Использует *std::move* для эффективной передачи владения сообщением. Функция *GUIInput::setNext* устанавливает следующий обработчик в цепочке (*handler*). Это реализует шаблон "Цепочка ответственности" (*Chain of Responsibility*), где каждый обработчик может передать сообщение дальше, если не может его обработать сам.

Класс *TUIInput*

Класс имеет одинаковую архитектуру с *GUIInput*, но отличается только в методе *update()*, внутри метода обрабатывает сообщения из терминала.

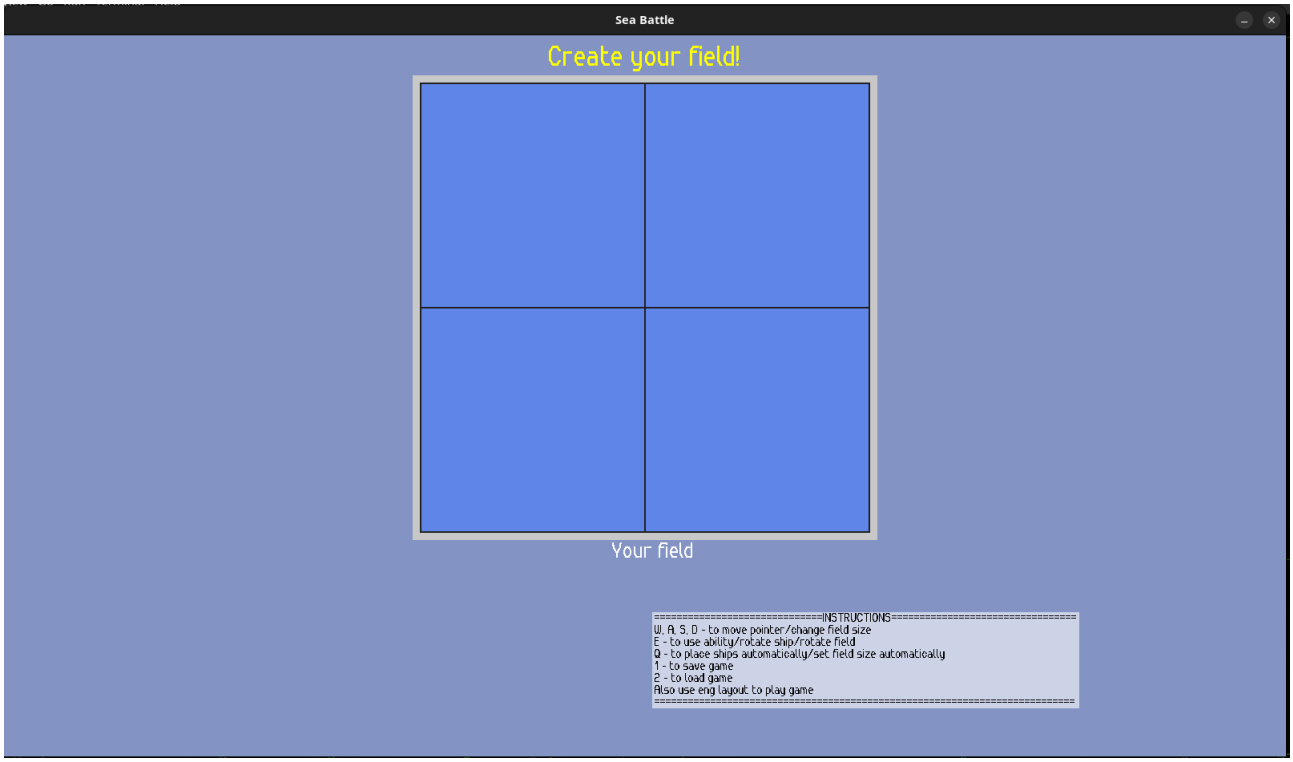
Функция *TUIInput::update* обрабатывает ввод с текстового интерфейса. Считывает строку из стандартного потока ввода (*std::cin*) и проверяет, содержит ли *controls* этот ввод в качестве ключа. Если содержит, то отправляется сообщение *keyMessage*, соответствующее значению из *controls*, — *controls[input]*. Эта функция предназначена для работы в текстовом режиме, где управление осуществляется через ввод команд с клавиатуры.

Диаграммы классов.

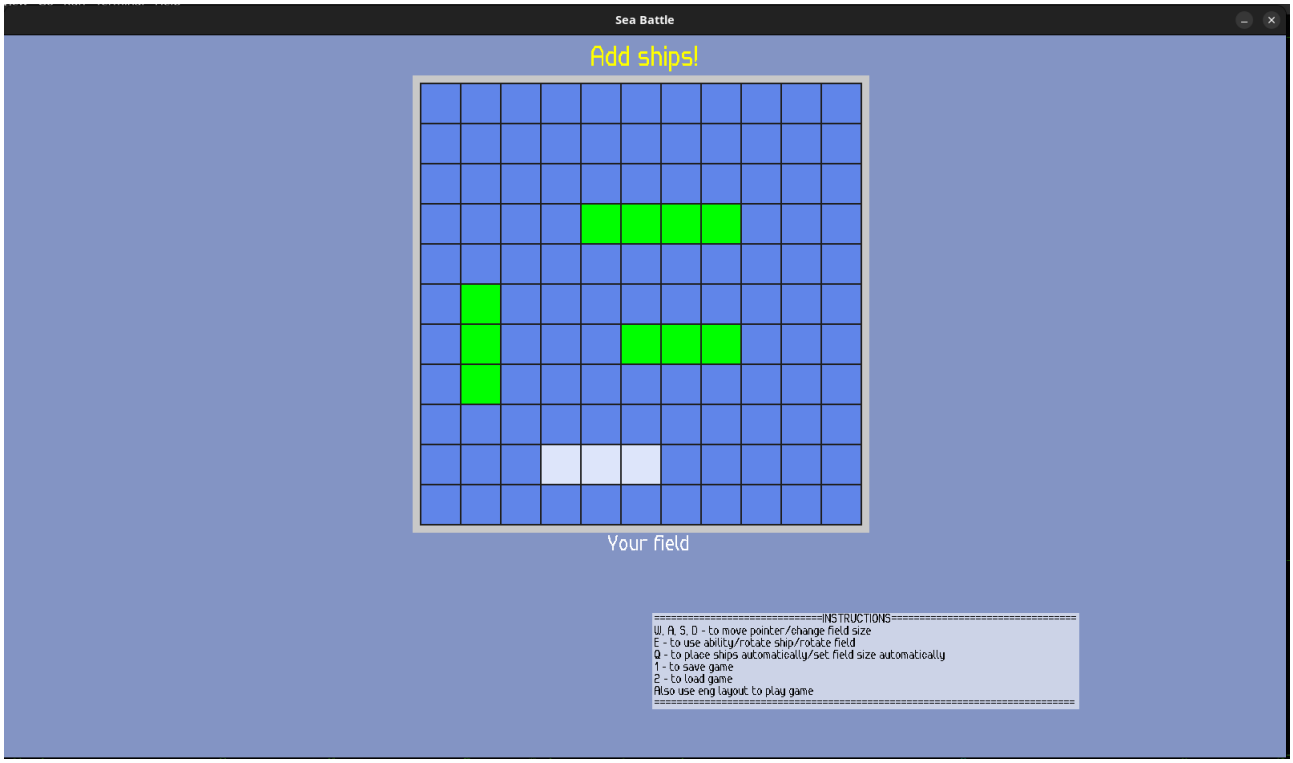


Тестирование.

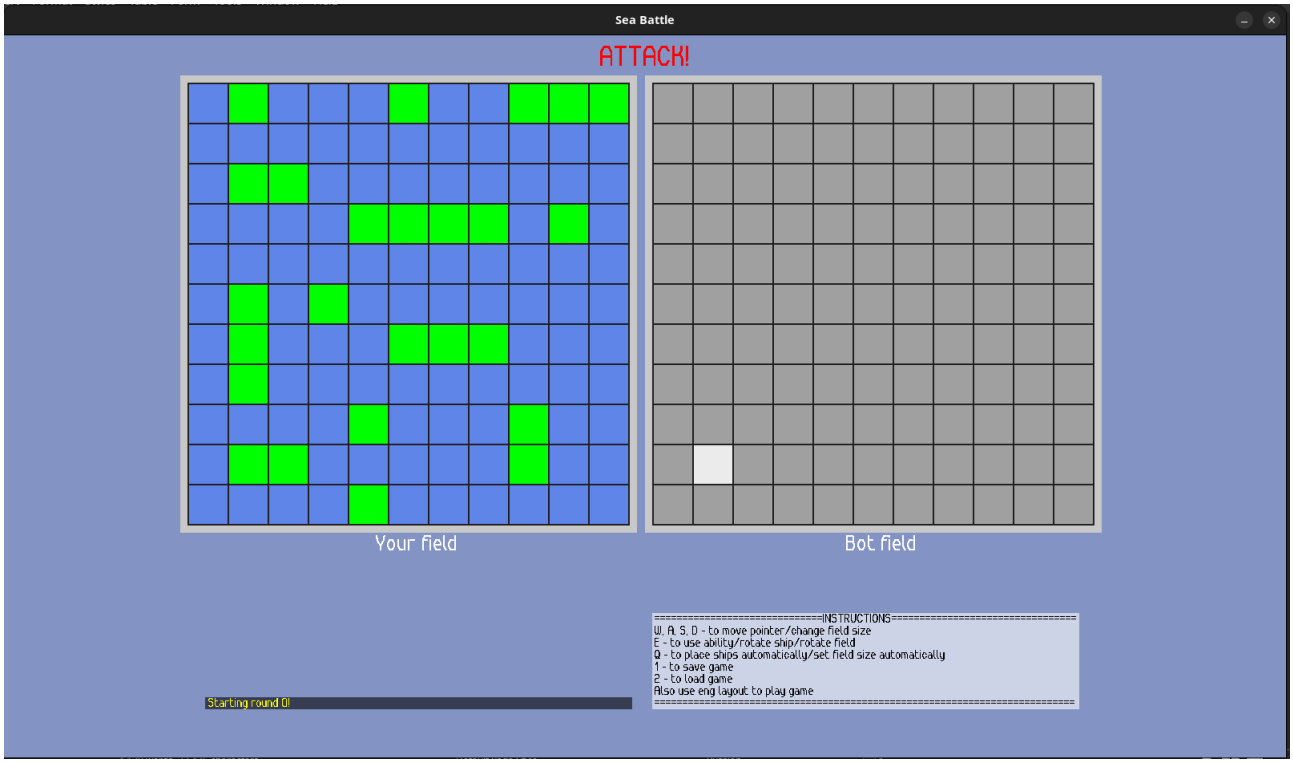
Запуск игры



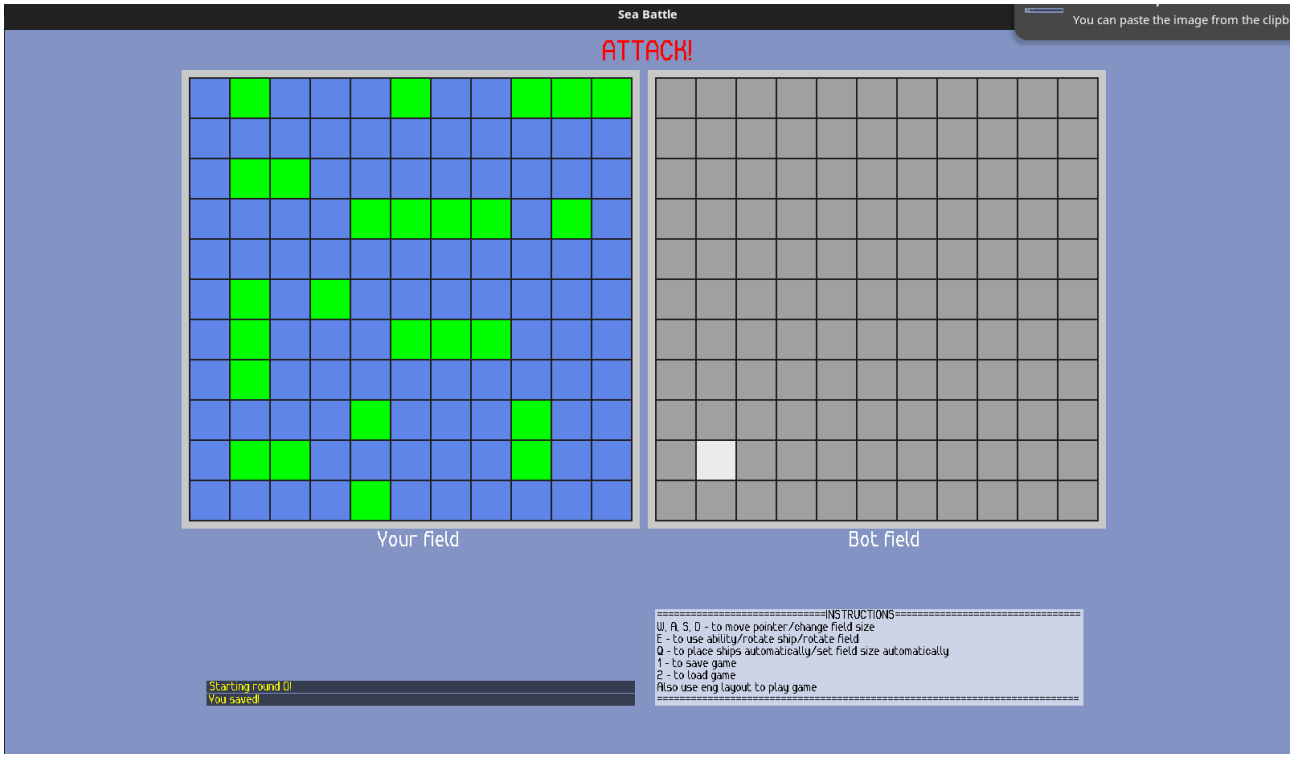
Расстановка кораблей



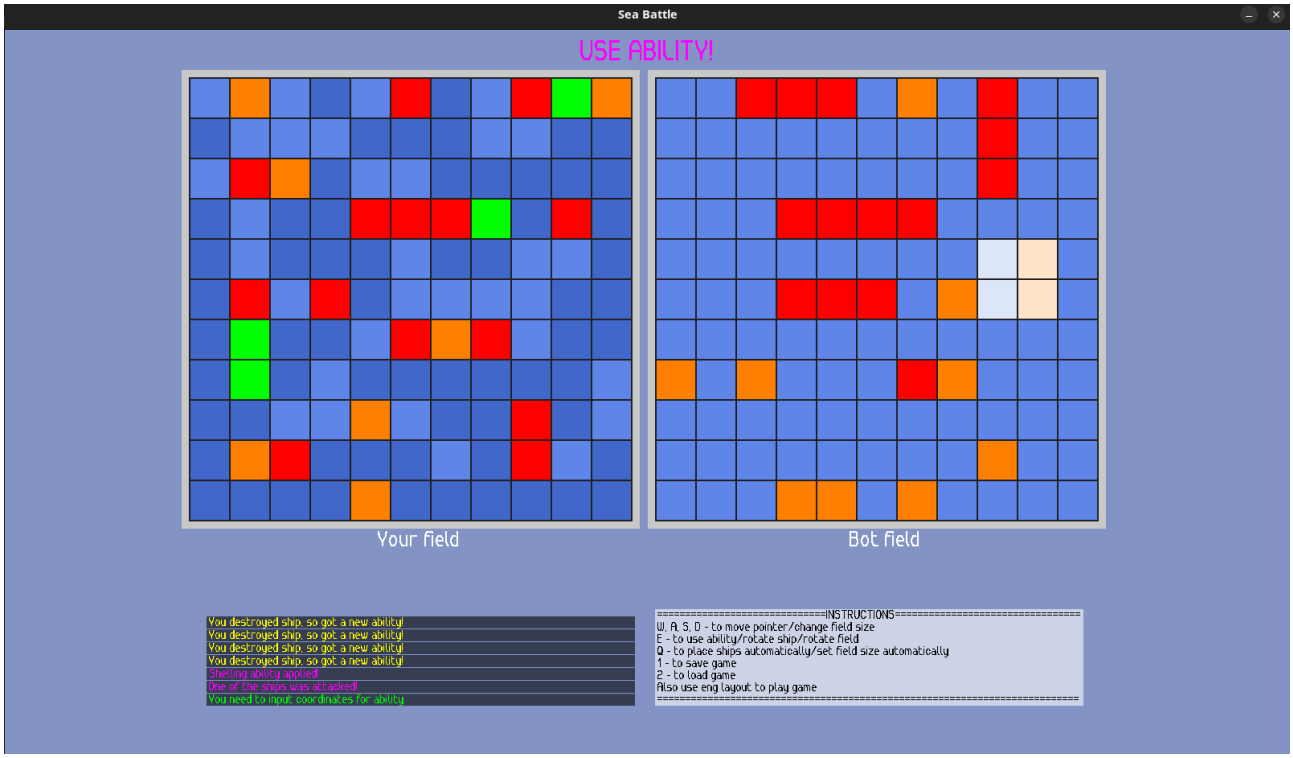
Игра



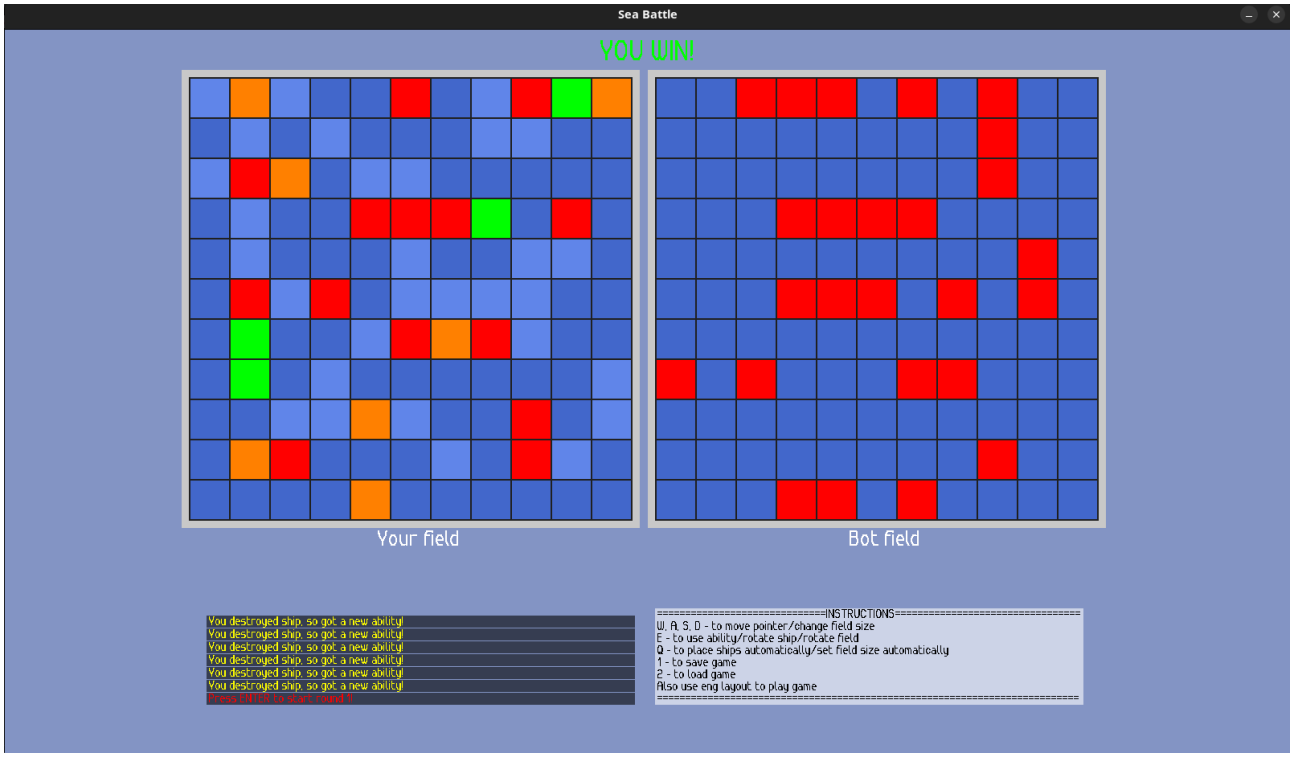
Сохранение



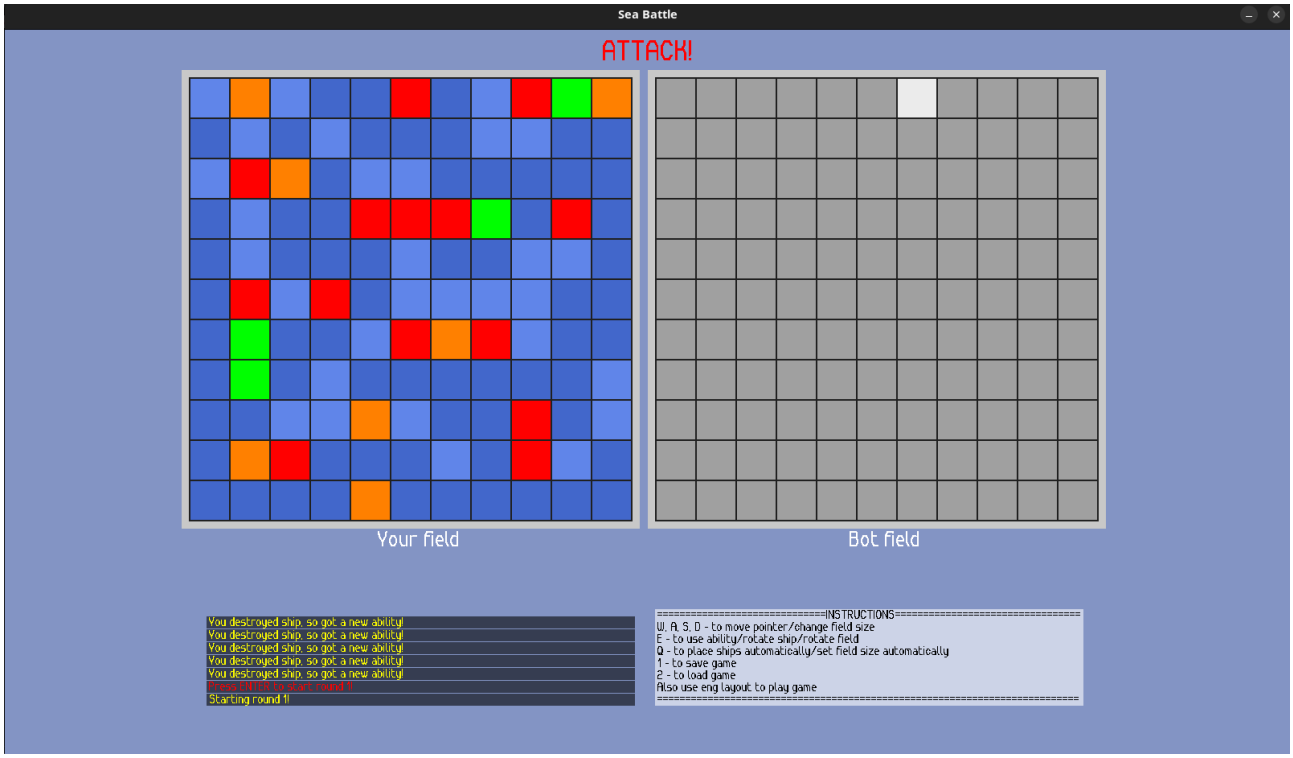
Игра 2



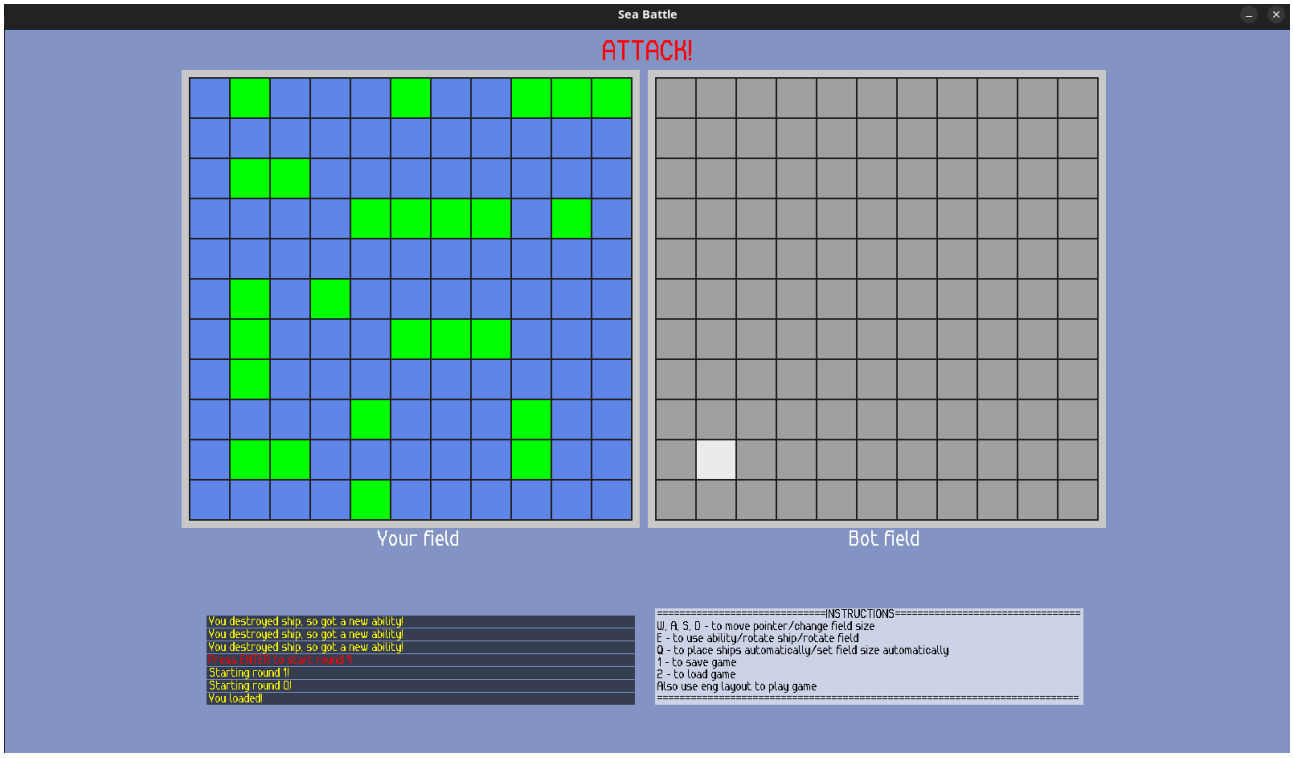
Выигрыш



Второй раунд



Загрузка



Выводы

В результате работы был создан программный модуль графического интерфейса пользователя (*GUI*) для игры "Морской бой", реализованный на языке *C++* с использованием библиотек *SDL* и *SDL_ttf*. Модуль включает обработку событий клавиатуры (*GUIInput*, *TUIInput*), отрисовку игрового поля, текста и указателя (*GUIDrawText*, *GUIDrawField*, *GUIOutput*), игровую логику (*gameController*, *gameTracker*), причём gameTracker отвечает за отслеживание состояния игры и вывод изменений на экран, а *gameController* - за обработку игровых действий. Загрузка и обработка настроек управления осуществляется классом *controlReader*, обеспечивающим чтение настроек из файла и предоставление значений по умолчанию при ошибках. Реализована поддержка как графического, так и текстового интерфейса ввода.