Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

Рязанский государственный радиотехнический университет имени В.Ф. Уткина

Кафедра САПР ВС

К защите

Руководитель работы:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

дата, подпись

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОЙ РАБОТЕ**

по дисциплине

**«Компьютерная графика»**

Тема:

«Изучение фреймворка создания игр libGDX на примере игры Star Battle»

|  |  |  |
| --- | --- | --- |
| Выполнил студент группы 045 |  |  |
| Вашкулатов Н.А. |  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  дата сдачи на проверку, подпись |
|  |  |  |
| Руководитель работы |  |  |
| доцент кафедры САПР ВС |  |  |
| Митрошин А.А. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  оценка | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  дата защиты, подпись |

ЗАДАНИЕ

на курсовую работу по дисциплине   
«Компьютерная графика»

Тема: «Изучение библиотеки создания игр libGDX на примере игры Морской Бой».

Задание: Изучить принцип работы и главные особенности фреймворка для создания игр libGDX на примере создания игры Star Battle.

**Содержание**

[Введение 4](#_Toc122631923)

[1 Теоретическая часть 5](#_Toc122631924)

[1.1 Фреймворк libGDX 5](#_Toc122631925)

[1.1.1 Архитектура игры на основе scene2D 6](#_Toc122631926)

[1.1.2 Работа с графическими компонентами 8](#_Toc122631927)

[2 Создание проекта с подключенным libGDX 10](#_Toc122631928)

[3 Разработка ПО 12](#_Toc122631929)

[3.1 Основные классы 12](#_Toc122631930)

[3.1.1 Структура проекта и описание классов 12](#_Toc122631931)

[3.1.2 Диаграмма классов 18](#_Toc122631932)

# Введение

С развитием цифровых технологий компьютеры все больше вливаются в жизнь человека. Если раньше ЭВМ использовались исключительно для сложных математических вычислений, то сегодня сфера их применения существенно расширилась. Компьютерные игры - одно из наиболее массовых применений электронных вычислительных машин.

Развитие игровой индустрии шло стремительным темпом, и особенно пользовалось популярностью у подростков. Первые игры отличались простотой интерфейса и логики, но со временем они становились все сложнее и сложнее, над их созданием работал уже не один человек, а целая команда разработчиков.

Современные игры требуют достаточно большой производительности от компьютера, и не каждая офисная машина в силах воспроизводить их. Однако для отдыха от монотонной работы зачастую достаточно простой, не требовательной к технике, игры. Именно такой разработке посвящен данный курсовой проект - игра «*Star Battle*».

*Star Battle* – аналог Морского боя со всеми его принципами и правилами, но в тематике вселенной Звездных Войн.

Для упрощения разработки игры мною был выбран LibGDX — фреймворк для создания игр и приложений, написанный на Java с использованием C и C++ и позволяющий писать кроссплатформенные игры и приложения, используя один код.

# 1 Теоретическая часть

## Фреймворк libGDX

**LibGDX**— [фреймворк](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA) для создания игр и приложений, написанный на [Java](https://ru.wikipedia.org/wiki/Java) с использованием [C](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) и [C++](https://ru.wikipedia.org/wiki/C%2B%2B) (для более быстрой работы) и позволяющий писать кроссплатформенные игры и приложения, используя один код.

Основной особенностью фреймворка является кроссплатформенность. Любое приложение написанное на этом фреймворке можно запустить на Windows, Linux, Mac, Android, WebGL. Это достигается за счет отделения логики приложения, от процедуры отрисовки при помощи прокси (в libGDX называются backends), которые оборачивают библиотеку для определенной платформы (lgjwl, Android API).

LibGDX позволяет перейти на такой низкий уровень, какой требуется в той или иной ситуации, давая примой доступ к файловой системе, устройствам ввода, аудио устройствам и OpenGL через единый  
OpenGL ES 2.0 и 3.0 интерфейс.

Наверху таких низкоуровневых возможностей создан мощный набор API, который позволяет решать общие в разработке игр задачи, такие как визуализация спрайтов, теста, построение пользовательских интерфейсов, проигрывание звуковых эффектов и музыки, линейная алгебра и тригонометрические вычисления, разбор JSON и XML, и так далее.

По своей сути libGDX состоит из шести модулей в виде интерфейсов, которые предоставляют средства для взаимодействия с операционной системой. Каждый прокси реализует эти интерфейсы:

***Application***: запускает приложение и информирует клиента API о событиях уровня приложения, таких как изменение размера окна. Предоставляет средства ведения журнала и методы запроса, например, использование памяти.

***Files***: предоставляет базовую файловую систему (ы) платформы. Обеспечивает абстракцию над различными типами расположения файлов поверх пользовательской системы обработки файлов (которая не взаимодействует с классом файлов Java).

***Input***: информирует клиента API о вводимых пользователем данных, таких как события мыши, клавиатуры, касания или акселерометра. Поддерживаются как опрос, так и обработка, управляемая событиями.

***Net***: предоставляет средства для доступа к ресурсам через HTTP/HTTPS, а также для создания TCP-серверных и клиентских сокетов.

***Audio***: предоставляет средства для воспроизведения звуковых эффектов и потоковой передачи музыки, а также прямого доступа к аудиоустройствам для ввода / вывода звука PCM.

***Graphics***: предоставляет OpenGL ES 2.0 (там, где это доступно) и позволяет запрашивать / устанавливать видеорежимы и подобные вещи.

Приложение libGDX имеет четко определенный жизненный цикл, управляющий состояниями приложения, такими как создание, приостановка и возобновление, рендеринг и завершение работы приложения.

Разработчик приложения подключается к этим событиям жизненного цикла, реализуя интерфейс ApplicationListener и передавая экземпляр этой реализации прикладной реализации конкретного серверного модуля (прокси). С этого момента приложение будет вызывать ApplicationListener каждый раз, когда происходит событие уровня приложения.

При разработке игры были использованы классы пакета scene2D

### Архитектура игры на основе scene2D

Пакет **scene2D** в libGDX представляет собой классы для реализации двухмерной сцены, которые могут быть полезны для управления группой иерархически связанных актеров (сущностей). Ниже будут рассмотрены преимущества данного пакета.

Во-первых, с помощью scene2D осуществляется вращение и масштабирование группы объектов. Дочерние объекты работают в своей системе координат, так что родительские преобразования для них прозрачны.

Во-вторых, данный пакет упрощает 2D-рендеринг изображения. Каждый объект существует в своей не вращаемой и не скалируемой системе координат, где (0,0) - нижний левый угол объекта.

В-третьих, scene2D имеет очень гибкую систему событий, которая позволяет обрабатывать события родителя перед или после обработки событий дочерних объектов.

Но у данного пакета также есть и недостаток, который заключается в том, что объекты совмещают и модель, и представление. Такое сцепление логики не позволяет полноценно использовать MVC (Model-View-Controller). Но в некоторых случаях это может быть полезно и удобно.

Главной особенностью scene2D является то, что у него есть три центровых класса: Stage, Group и Actor.

Класс **Stage** (сцена) имеет «корневую» группу (т.е. класс Group), куда приложение может добавлять своих актеров. У класса Stage есть своя камера и упаковщик спрайтов (SpriteBatch). Stage отсылает события дочерним элементам и реализует интерфейс InputProcessor, который предназначен для считывания событий ввода.

Класс **Group** является актером, который может содержать в себе других актеров. Вращение и масштабирование группы соответствующим образом отражается на его дочерних актерах. Класс Group делегирует рисование и события ввода соответствующим дочерним актерам. Он превращает координаты событий ввода так, что дочерние элементы получают эти координаты в своей собственной системе координат.

Класс **Actor** представляет собой некоторая сущность, которая может быть нарисована и которая может обрабатывать события ввода. Актер имеет позицию, размер, информацию о масштабе, вращение и родительский компонент (origin). Также актер может иметь имя, что позволяет найти его по этому имени - это может быть полезно при отладке.

Чтобы использовать абстрактные классы Stage, Group, Actor нужно их расширить и переопределить методы для рисования, определение нажатий и т.д.

### Работа с графическими компонентами

При написании игр больше всего времени тратится именно на работу с графикой. В данном параграфе будет рассмотрено, как правильно работать с графикой (текстурами, регионами, атласами и т.д.) в libGDX.

**Текстура** – двумерное изображение, хранящееся в памяти компьютера или графического акселератора в одном из пиксельных форматов. В случае хранения в сжатом виде на дисках компьютера текстура может представлять собой битмап (от англ. binary map – двоичная карта), который можно встретить в форматах bmp, jpg, gif и т.д..

**Спрайт** – растровое изображение, свободно перемещающееся по экрану. По своей сути, спрайт – это проекция объекта на плоскости. В 2D-графике это реализуется без проблем, но при работе с 3D-графикой при изменении угла обзора происходит разрушение иллюзии. Если представить себе стену в игре, то когда камера смотрит на неё прямо – всё в порядке, но если посмотреть на неё сбоку – камера увидит лишь линию.

**Текстурный атлас** – большое изображение, которое содержит много изображений меньшего размера, каждое из которых является текстурой для какой-то части объекта. Такие «подтекстуры» можно накладывать путём указания текстурных координат на атласе, то есть, выбирая только определённую часть изображения. В приложениях, в которых часто используются мелкие текстуры, более эффективно хранить текстуры в текстурном атласе, так как это позволяет сократить количество смен состояний до одного для всего атласа, уменьшает количество занятых текстурных слотов, а также минимизируется фрагментация видеопамяти. Чаще всего изображения, которые содержатся в атласе, имеют одинаковый размер, поэтому рассчитать их координаты не составит труда.

С помощью класса **SpriteBatch** в libGDX можно легко работать с графическими компонентами. Также данный класс помогает оптимизировать работу с графикой, поскольку привязка текстуры является довольно дорогостоящей операцией, он хранит много маленьких изображений в атласе, а затем рисует регионы из атласа для предотвращения частой смены текстур.

РАЗДЕЛИТЬ НА ПУНКТЫ И НАПИСАТЬ ПРО ГИБОКСТЬ

# Создание проекта с подключенным libGDX

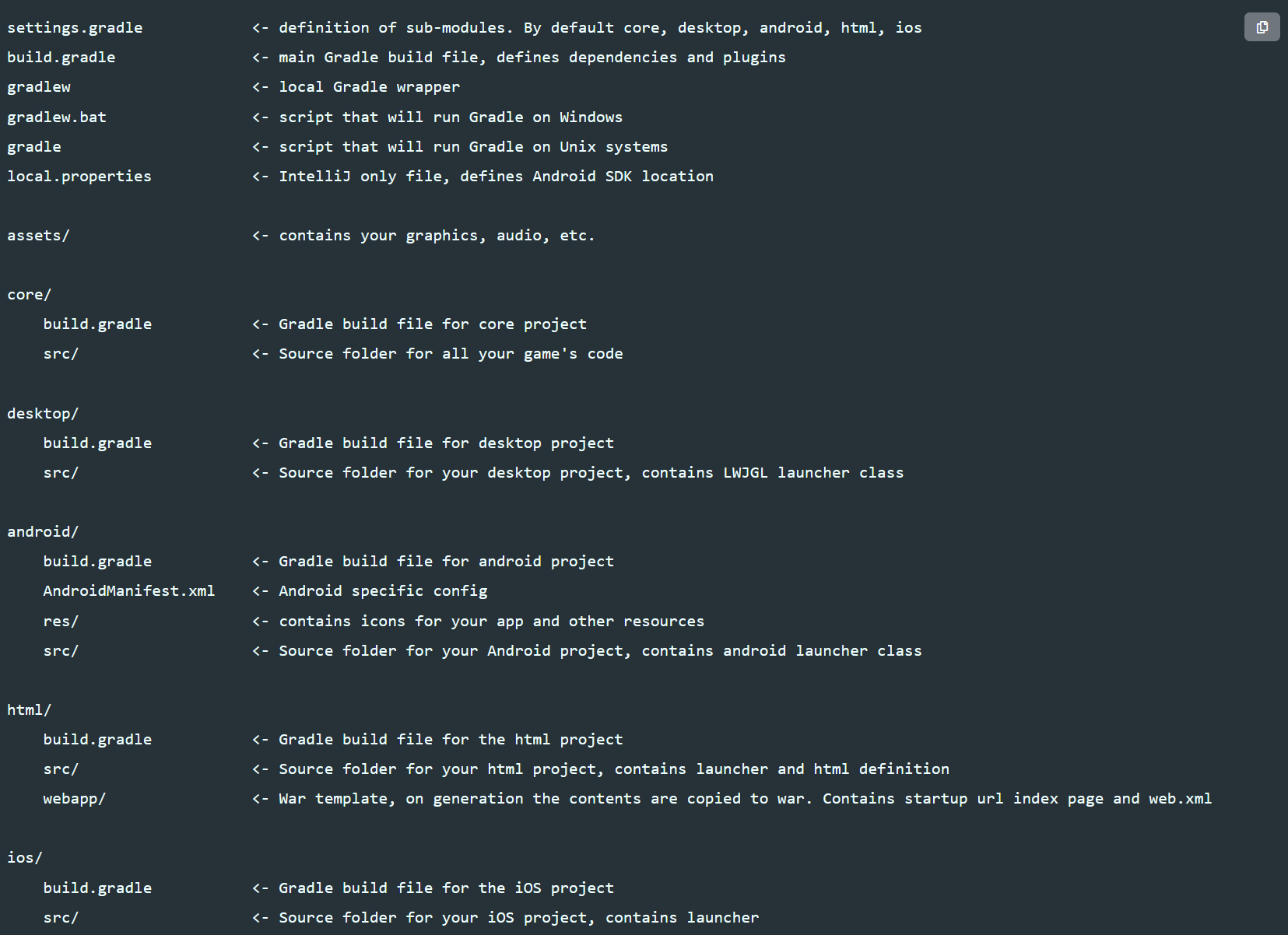
Для создания проекта с подключенными зависимостями libGDX в официальной документации предлагается утилита gdx-setup при помощи которой можно создать рабочий проект для выбранных платформ и с выбранными расширениями. Получим проект с сборщиком Gradle такой структуры (рисунок 1)

Рисунок 1 – Стандартная структура проекта

При разработке проект был настроен вручную с сборщиком Maven, который загрузил необходимые зависимости. Мною был выбран Maven поскольку его структура проще, сборка проекта в jar выполняется быстрее и все работает стабильнее.

Были подключены и загружены следующие зависимости:

com.badlogicgames.gdx – классы для разработки, одинаковые для обоих платформ.

com.badlogic.gdx.backends.lwjgl3 – прокси для Windows, Linux, Mac.

com.badlogicgames.gdx.gdx-freetype – классы для генерации BitMapFont из otf.

Все зависимости были загружены с репозитория Maven Central.

# Разработка ПО

## Основные классы

### Структура проекта и описание классов

Конечная структура пакетов проекта показана на рисунке 2.

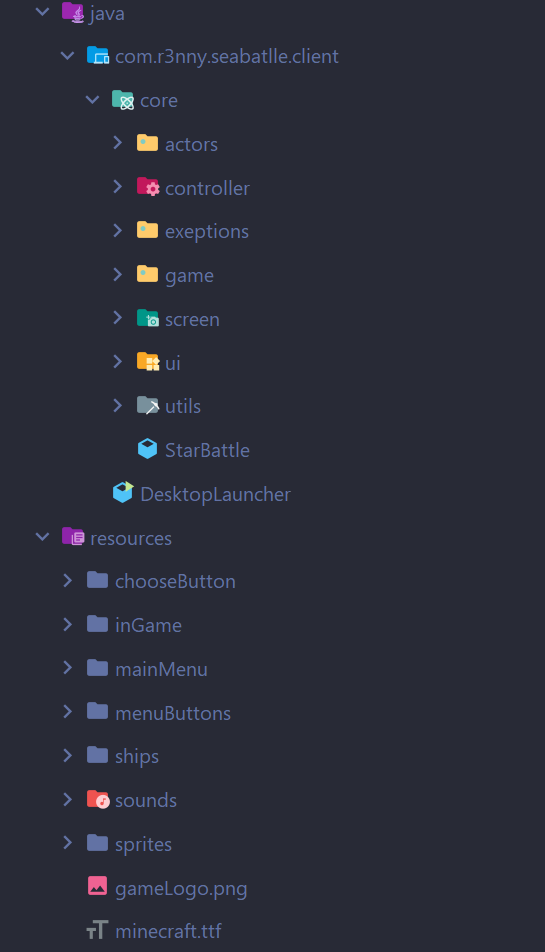


Рисунок 2 – Конечная структура пакетов проекта

1. Класс ***com.r3nny.seabattle.client.DesktopLauncher.java***

Этот класс является стартовым классом. В методе main создается конфигурация приложения и само приложение на основе прокси *com.badlogic.gdx.backends.lwjgl3.Lwjgl3Application.* Конструктор приложения принимает в себя объект, реализующий интерфейс *ApplicationListener* и конфигурацию. Все события жизненного цикла будут обрабатываться этим объектом.

1. Класс ***com.r3nny.seabattle.client.core.StarBattle.java***

Этот класс является основным. Он расширяет абстрактный класс com.badlogic.gdx.*Game.java,* который реализует интерфейс *ApplicationListener.*

В методе create (вызываемый при создании приложения) сразу меняется экран на SplashScreen.

В этом классе содержатся статические поля менеджера звука, анимации и ресурсов для удобного доступа к ним.

Метод setUpStage() возвращает настроенную сцену для использования на экранах.

1. Пакет m***com.r3nny.seabattle.client.core.actors*** содержит классы, расширяющие классы com.badlogic.gdx.scenes.scene2d.Actor или com.badlogic.gdx.scenes.scene2d.Group. Экземпляры этих классов отображаются на сцене Stage.
   1. Класс ***com.r3nny.seabattle.client.core.actors.Cell.java***

Данный класс является клеткой поля. Содержит информацию о своем положении (row, column), cсылку на корабль в этой клетке, а так же статус клетки (море, промах, целый корабль, ранен, убит).

* 1. Класс ***com.r3nny.seabattle.client.core.actors.Ship.java***содержит информацию о своем типе (количестве палуб), ссылку на клетки, в которых он расположен, ориентацию, расположение.
  2. Класс ***com.r3nny.seabattle.client.core.actors.Ship.ShipInputListener.java***является внутренним классом класса Ship и обрабатывает взаимодействие с кораблем. При помощи мыши можно перетащить корабль на игровое поле, повернуть его, обвести при наведении.
  3. Класс ***com.r3nny.seabattle.client.core.actors.ShipCountDTO.java*** является вспомогательным и отражает количество кораблей определенного типа. Содержит тип корабля и количество.
  4. Класс ***com.r3nny.seabattle.client.core.actors.ShipManager.java***отображает количество оставшихся кораблей у определенного поля. Содержит список из *ShipCountDTO.*
  5. Класс ***com.r3nny.seabattle.client.core.actors.ShipCreatingArea.java***отображает подсказки по созданию кораблей и поле, на котором изначально расположены корабли.
  6. Класс ***com.r3nny.seabattle.client.core.actors.GameField.java***абстрактный класс игрового поля. Содержит массив клеток, отражающих поле, список кораблей на поле, ShipManager. При создании поля создаются клетки игрового поля и корабли на начальных позициях. Через объект поля можно автоматически заполнить его кораблями, убить корабль и отобразить это в ShipManager.
  7. Класс ***com.r3nny.seabattle.client.core.actors.EnemyGameField.java*** расширяет класс *GameField*.*java.* При создании экземпляра сразу автоматически расставляются корабли.
  8. Класс ***com.r3nny.seabattle.client.core.actors.PlayerGameField.java***расширяет класс *GameField*.*java.* При создании экземпляра создается ShipsCreatingArea и на основе ее расставляются корабли на начальных позициях готовые для ручной расстановки.

1. Пакет ***com.r3nny.seabattle.client.core.controller*** содержат вспомогательные для логики игры классы.

4.1 Класс ***com.r3nny.seabattle.client.core.controller.ShipsCreator.java***содержит массив с типами кораблей, необходимых для расстановки и два открытых статических метода: autoCreateShips, который автоматически расставляет корабли на необходимом поле и addShipToGameField, который добавляет на поле корабль, начиная с определенной клетки. Для этого в методе пытаемся получить клетки, которые должен занимать корабль. Если корабль вылезет за пределы карты или клетки уже заняты, то выбрасывается исключение CantCreateShipException. Если получается получить все клетки, то вокруг корабля помечаем клетки как занятые и связываем корабль и полученные клетки.

4.2 Класс ***com.r3nny.seabattle.client.core.controller.ShootController.java*** содержит открытый метод shoot, который изменяет состояние клетки на поле в зависимости от того, куда был сделан выстрел и кем он был сделан.

1. Класс ***com.r3nny.seabattle.client.core.exceptions***

***.CantCreateShipException.java*** – это исключение, которое выбрасывается при попытке создать корабль вне поля или на занятых клетках.

1. Пакет ***com.r3nny.seabattle.client.core.game*** содержит классы, связанные с логикой проведения игры.

6.1 Класс ***com.r3nny.seabattle.client.core.game.Game.java –*** абстрактный класс, который содержит два игровых поля и статус игры (Меню, Создание Корабля, Расстановка кораблей, Ход игрока, Ход противника, Конец).

6.2 Класс ***com.r3nny.seabattle.client.core.game.SingleGame.java –*** класс, расширяющий Game.java, содержит в себе ссылку на Bot.java, который вызывается в процессе игры. В конструкторе инициализируются игровые поля.

7. Пакет ***com.r3nny.seabattle.client.core.screen*** содержат классы, реализующие интерфейс *com.badlogic.gdx.Screen.* Они являются как бы заготовкой отображения. Содержат методы show(), render(), hide(), resize(), dispose(), hide().

7.1 Класс ***com.r3nny.seabattle.client.core.screen.Screen.java -*** абстрактный класс, реализующий интерфейс *com.badlogic.gdx.Screen.* Содержит ссылку на ApplicationListener, Stage и protected методы для доступа к менеджерам Assets, SoundManager и AnimationManager.

7.2 Класс ***com.r3nny.seabattle.client.core.screen.InGameScreen.java –*** абстрактный класс Screen.java. Содержит настроенную кнопку возвращения в меню, фон экрана и лого игры.

7.3 Класс ***com.r3nny.seabattle.client.core.screen.SplashScreen.java*** – загрузочный экран, содержащий лого игры. Параллельно отображению лого игры загружаются все ресурсы из класса Asset.java.

7.4 Класс ***com.r3nny.seabattle.client.core.screen.MenuScreen.java*** – содержит кнопку для смены экрана на ChooseScreen.java и кнопку выхода из игры.

7.5 Класс ***com.r3nny.seabattle.client.core.screen.ChooseScreen.java*** – содержит кнопки выбора режима игры (одиночный и сетевой). Активна только кнопка одиночного режима.

7.6 Класс ***com.r3nny.seabattle.client.core.screen.ShipCreatingScreen.java***  содержит ShipsCreatingArea, поле игрока с кораблями. На данном поле создается экземпляр SingleGame.java и происходит расстановка кораблей игрока.

7.7 Класс ***com.r3nny.seabattle.client.core.screen.SingleGameScreen.java***  содержит поля игрока и противника, два ShipManager, SingleGame. В методе render() этого класса вызывается логика игры бота.

8. Класс ***com.r3nny.seabattle.client.core.ui.ChangeScreenButton.java*** является оберткой класса com.badlogic.gdx.scenes.scene2d.ui.TextButton. Принимает в себя текст кнопки и два Runnuble. Один вызывается сразу после нажатия кнопки и вызывается анимация пропадания кнопки. Второй вызывается после исчезновения кнопки.

9. Пакет ***com.r3nny.seabattle.client.core.utils*** содержит вспомогательные классы не связанные с логикой.

9.1 Класс ***com.r3nny.seabattle.client.core.utils.Assets.java –*** обертка над com.badlogic.gdx.assets.AssetsManager. При создании в загрузочном экране Asset загружает все ресурсы и предоставляет методы доступа к ним.

9.2 Класс ***com.r3nny.seabattle.client.core.utils.SoundManager.java –*** загружает все необходимые аудиофайлы, настраивает их громкость и предоставляет методы для запуска определенных звуков и их остановки.

9.3 Класс ***com.r3nny.seabattle.client.core.utils.AnimationManager.java –*** загружает все спрайты, преобразует их в удобный класс Animation для отрисовки в Actor, так же содержит готовые Actions для появления, исчезновения и передвижения.

**ДОДЕЛАТЬ БОТА**

### 3.1.2 Диаграмма классов

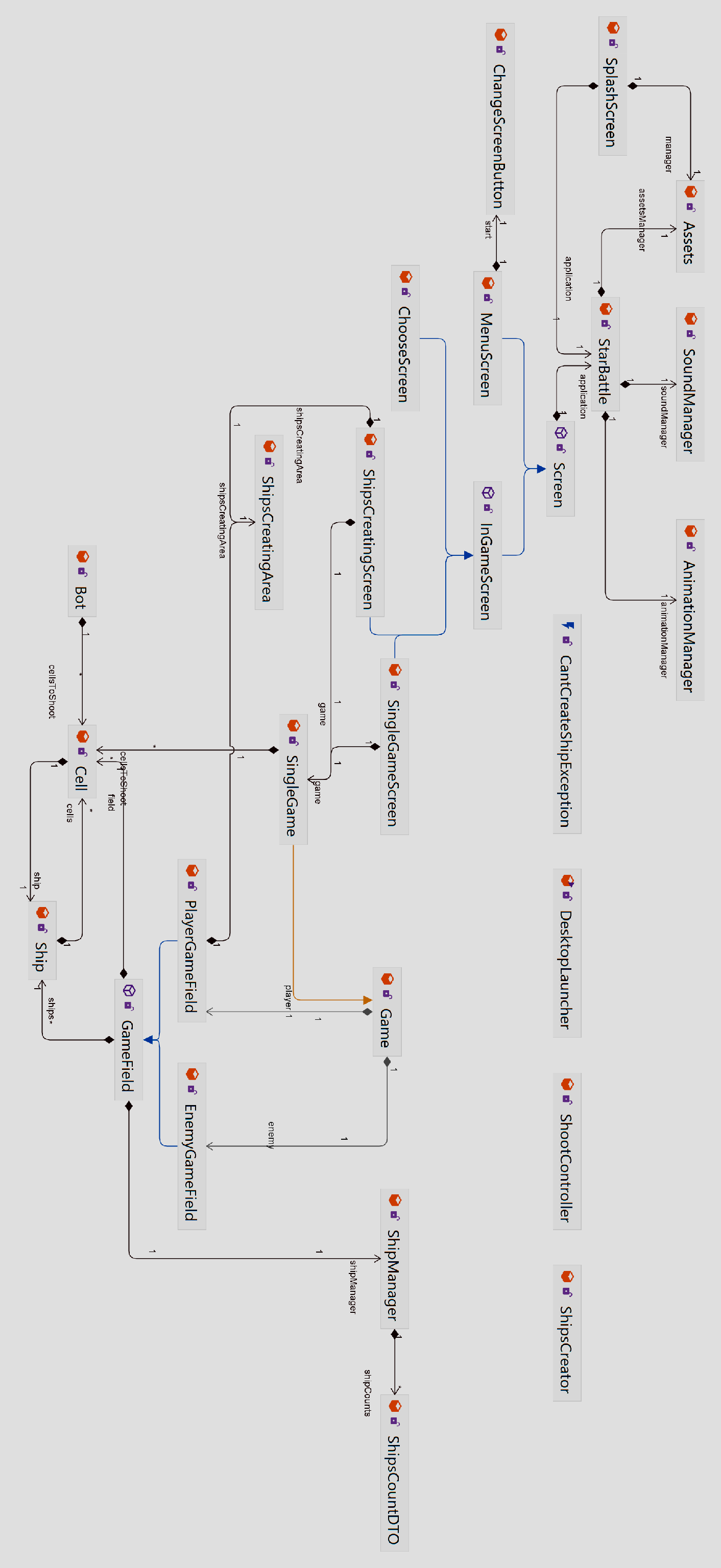


Рисунок 3 – Диаграмма классов

## Таблица основных методов и переменных

Таблица 1 – Основные методы и переменные

|  |  |  |
| --- | --- | --- |
| **Класс** | **Метод или переменная** | **Назначение** |
| Cell | int column, int row | Позиция клетки на поле |
| cellStatus status | Статус клетки |
| Ship ship | Ссылка на корабль |
| float animationTime | Время с начала анимации |
| Ship | ShipType type | Тип корабля (размер) |
| List<Cell> cells | Список клеток корабля |
| boolean isVertical | Ориентация корабля |
| boolean isSelected | Наведена ли мышка на корабль |
| updateBounds() | Обновляет границы объекта. Нужно при передвижении |
| makeCellsKilled() | Убивает все клетки |
| rotate() | Переворачивает текстуру корабля |
| canMoveShip() | Можно ли взаимодействовать мышкой с кораблем (корабль не расположен и не прибывает) |
|  |  |
| GameField | ShipManager shipManager | Учет количества кораблей и отображение этого параметра |
| boolean isShipsReady | Расставлены ли корабли |
| Cell[][] field | Поле игры |
| List<Ship> ships | Корабли поля |
| killShip(Ship ship) | Убивает корабль, вызывая у него метод makeCellsKilled, уменьшает количество кораблей этого типа у ShipManager, убирает корабль из группы |
| clearAllMissed() | Заменяет все промахи на море |
| ShipInputListener | exit() | Вызывает unSelect() у Ship |
| enter() | Вызывает select() у Ship |
| touchdown() | При нажатии ПКМ вызывает rotate() |
| touchDragged() | Передвигает корабль по экрану |
| touchUp() | Если отпускается, когда корабль на клетке, выполняется попытка создать корабль и анимация появления |
|  |  |  |
| Продолжение таблицы 1 |  |  |
| **Класс** | **Метод или переменная** | **Назначение** |
| ShipInputListener | Optional<Cell> getCellFrom(Event e) | Возвращает клетку, над которой был отпущен корабль |
|  | animateShip() | Запускает анимацию появления корабля |
|  |  |  |