**РЕФЕРАТ**

БГУИР КП 1-39 02 01 029 ПЗ

**Чернышенко, М.С.** Игра-платформер «Afterglow» под операционную систему Android на языке программирования Java с использованием фреймворка LibGDX : пояснительная записка к курсовому проекту / М.С. Чернышенко. – Минск : БГУИР, 2021. – 50 с.

Пояснительная записка 50 с., 11 рис., 10 источников, 3 приложения

ИГРА-ПЛАТФОРМЕР, AFTERGLOW, ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA, ФРЕЙМВОРК LIBgdx, ОПЕРАЦИОННАЯ СИСТЕМА ANDROID, ANDROID STUDIO, РЕДАКТОР КАРТ TILED, НАСЛЕДОВАНИЕ КЛАССА GAME, реализация интерфейса SCREEN

*Цель* *проектирования*: разработка игры-платформера «Afterglow» под операционную систему Android на языке программирования Java с использованием фреймворка LibGDX.

*Методология проведения работы*: В процессе решения поставленных задач использованы принципы объектно-ориентированного программирования, учтены основы взаимодействия межды различными классами, применены аналитические и физико-математические методы программирования.

*Результаты работы*: выполнен анализ литературных источников; применена архитектура фреймворка libGDX; произведено моделирование игрового уровня; реализовано создание игрового цикла; проведено тестирование игрового процесса; выполнено обоснование устройства программы; разработана графическая часть проекта.

Программа обеспечивает запуск игрового цикла и прохождение уровней на операционных системах Android и Windows.

Структура подразумевает выбор игровой сцены, использование кнопок клавиатуры или сенсорного экрана для совершения действий в игровом процессе и прохождение уровней.

Результаты работы доложены на 1-ой научной конференции, опубликована 1 статья.

*Область применения результатов*: могут быть использованы при написании программного кода с использованием фреймворка libGDX на языке программирования Java

# Содержание

[Введение 5](#_Toc72682225)

[1 Описание проекта 6](#_Toc72682226)

[1.1 Общее описание курсовой работы 6](#_Toc72682227)

[1.2 Игровые классы 7](#_Toc72682228)

[2 Обоснование выбора технологий 18](#_Toc72682229)

[2.1 Использование языка программирования Java 18](#_Toc72682230)

[2.2 Использование технологий объектно-ориентированного программирования 19](#_Toc72682231)

[2.3 Использование фреймворка libGDX 20](#_Toc72682232)

[3 Инструментарий 23](#_Toc72682233)

[3.1 Среда разработки Android Studio 23](#_Toc72682234)

[3.2 Графический редактор Photoshop 25](#_Toc72682235)

[3.3 Система контроля версий GIT 26](#_Toc72682236)

[3.4 Редактор ортогональных карт Tiled 27](#_Toc72682237)

[4 Кроссплатформенный фреймворк LibGDX 28](#_Toc72682238)

[4.1 Архитектура фреймворка libGDX 28](#_Toc72682239)

[4.2 Отрисовка текстур 29](#_Toc72682240)

[4.3 Камера и область просмотра игрового пространства 31](#_Toc72682241)

[5 Шаблоны проектирования практических задач 33](#_Toc72682242)

[Заключение 39](#_Toc72682243)

[Список использованных источников 40](#_Toc72682244)

[Приложение А (обязательное) Листинг программного кода 41](#_Toc72682245)

[Приложение Б (обязательное) Отчет по анализу заимствования 49](#_Toc72682246)

[Приложение В (обязательное) Ведомость курсового проекта 50](#_Toc72682247)

# Введение

Игровая индустрия в современном мире приобретает всё большую актуальность в качестве средства проведения досуга в сфере развлечений, что делает её широко востребованной среди разработчиков. Написание оптимального программного кода подразумевает грамотный выбор языка программирования и архитектуры проектирования, которые обеспечат привлечение пользователей.

Одним из наиболее подходящих вариантов для написания игр под операционную систему Android является язык программирования Java, который использует объектно-ориентированную парадигму. Программа состоит из объектов классов, которые взаимодействуют друг с другом и реализуют методы на основе принципов абстракции, инкапсуляции и полиморфизма.

Фреймворк libGDX для создания игр основан на языке программирования Java с некоторыми компонентами, написанными на C и C++ для повышения производительности определённого кода. Представленный фреймворк является кроссплатформенным, то есть он позволяет разработчику писать и тестировать один код и переносить его на такие платформы как Windows, Linux, MacOS, Android, iOS и HTML5. При этом используются отдельные модули для сборки приложения под каждую платформу, а также независимый модуль, который содержит основной код приложения.

Причиной выбора фреймворка libGDX для написания данной курсовой работы является мощный набор API, который позволяет решать общие в разработке игр задачи, такие как визуализация спрайтов и создание анимации, построение пользовательских интерфейсов, прогрывание звуковых эффектов и музыки, работа с объектами и физикой реального мира, использование 2D объектов и так далее.

Для написания курсовой работы было выделено три основных задачи:

1. Отрисовка персонажей и карты игрового мира.
2. Использование созданных текстур для написания программного кода.
3. Тестирование программного кода.

Актуальность работы высока, так как она поднимает вопрос нахождения и внедрения наиболее оптимальных технологий и наиболее соответствующей архитектуры кода для получения максимальной производительности игрового процесса.

# 1 Описание проекта

## Общее описание курсовой работы

Целью курсовой работы было написание кроссплатформенной игры Afterglow на языке программирования Java с использованием фреймворка LibGDX. Написанное приложение может быть запущено, как в десктопной версии, так и в мобильной версии на операционной системе Android.

Игра Afterglow – это платформер с горизонтальной прокруткой экрана, место действия может быть выбрано из шести вариантов: Fairy, Forest, Field, Ocean, Glade, Flowery, – представленных на рисунке 2.



Рисунок 1 – Выбор места действия

С помощью нажатий на экран игрок управляет собакой, которая передвигается по выбранной теме, преодолевая препятствия в виде деревьев, ям и рвов, а также поражая либо обходя врагов – грибов и мышей. Каждый уровень считается пройденным и заканчивается, когда персонаж доходит до последней платформы [1].

*Техническая реализация курсовой работы.* Как показано на рисунке 2, проект состоит из нескольких модулей: android и desktop с классами AndroidLauncher и DesktopLauncher, отвечающими за запуск приложения на соответствующих платформах; а также core, в котором содержится основной управляющий класс игры Afterglow и четыре пакета для объединения классов схожей тематики. Пакет Screens предоставляет три игровых экрана игры, Scenes содержит индикатор состояний Hud, Sprites состоит из двух внутренних комплектов объектов Enemies и TileObjects, а пакет Tools представляет классы создания объектов мира и регулирования их столкновений.

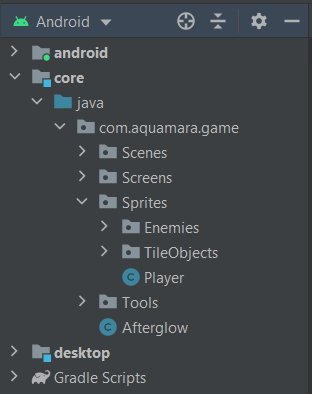


Рисунок 2 – Модули проекта

Основным классом игры является класс Afterglow, он наследует класс Game, реализующий интерфейс ApplicationListener из библиотеки badlogic.

Любой класс, расширяющий интерфейс ApplicationListener, выполняется на протяжении всего активного цикла игры и вызывает управляющие методы create(), render(), resume(), pause() и destroy(). Вышеперечисленные методы вызываются в потоке, содержащем текущий контекст OpenGL, что позволяет создавать графические ресурсы и управлять ими без трудностей с управлением памятью [2].

Отличительным свойством класса Game является возможность делегирования выполнения всех методов классам, реализующим интерфейс [Screen](https://libgdx.badlogicgames.com/ci/nightlies/docs/api/com/badlogic/gdx/Screen.html), что позволяет приложению переключаться между несколькими экранами. Так как интерфейс Screens не предусматривает автоматического освобождения ресурсов, разработчик сам решает, сохранять экраны или очищать при переключении между состояниями меню, уровней, завершения игры и так далее.

## 1.2 Игровые классы

*Игровой класс Afterglow*. Данный класс наследует класс Game и служит основным игровым циклом, который беспрерывно выполняет метод render(), однако, вместо того, чтобы обрабатывать то, что находится в методе самостоятельно, он передаёт это действие классам PlayScreen, GameOverScreen и LevelScreen, реализующим интерфейс Screen. В основном классе Afterglow определены виртуальные размеры ширины и высоты мира, а также константа, которая переводит все числа для шкалы мира Box2D.

Переменная batch типа SpriteBatch служит контейнером для хранения всех графических ресурсов, таких как изображения и текстуры. Так как этот тип данных требует выделения большого количества памяти, он создан один раз с модификатором доступа public и вызывается из остальных классов. Переменные класса SpriteBatch выполняют все шаги для вырисовки экрана и могут обрабатывать несколько элементов за один раз, что оптимизирует работу с GPU и ставит все инструкции в очередь, таким образом, разработчик должен только начать выполнение [1].

В методе create() вызывается метод setScreen(), который принимает указатель this, то есть использует саму игру для выбора экрана. Метод render() делегирует отрисовку графики активному экрану, а метод dispose() отвечает за освобождение всех использованных графических ресурсов.

*Игровой класс LevelScreen.* Данный класс реализует интерфейс Screen, он является первым экраном, который видит пользователь, продемонстрировано на рисунке 3.

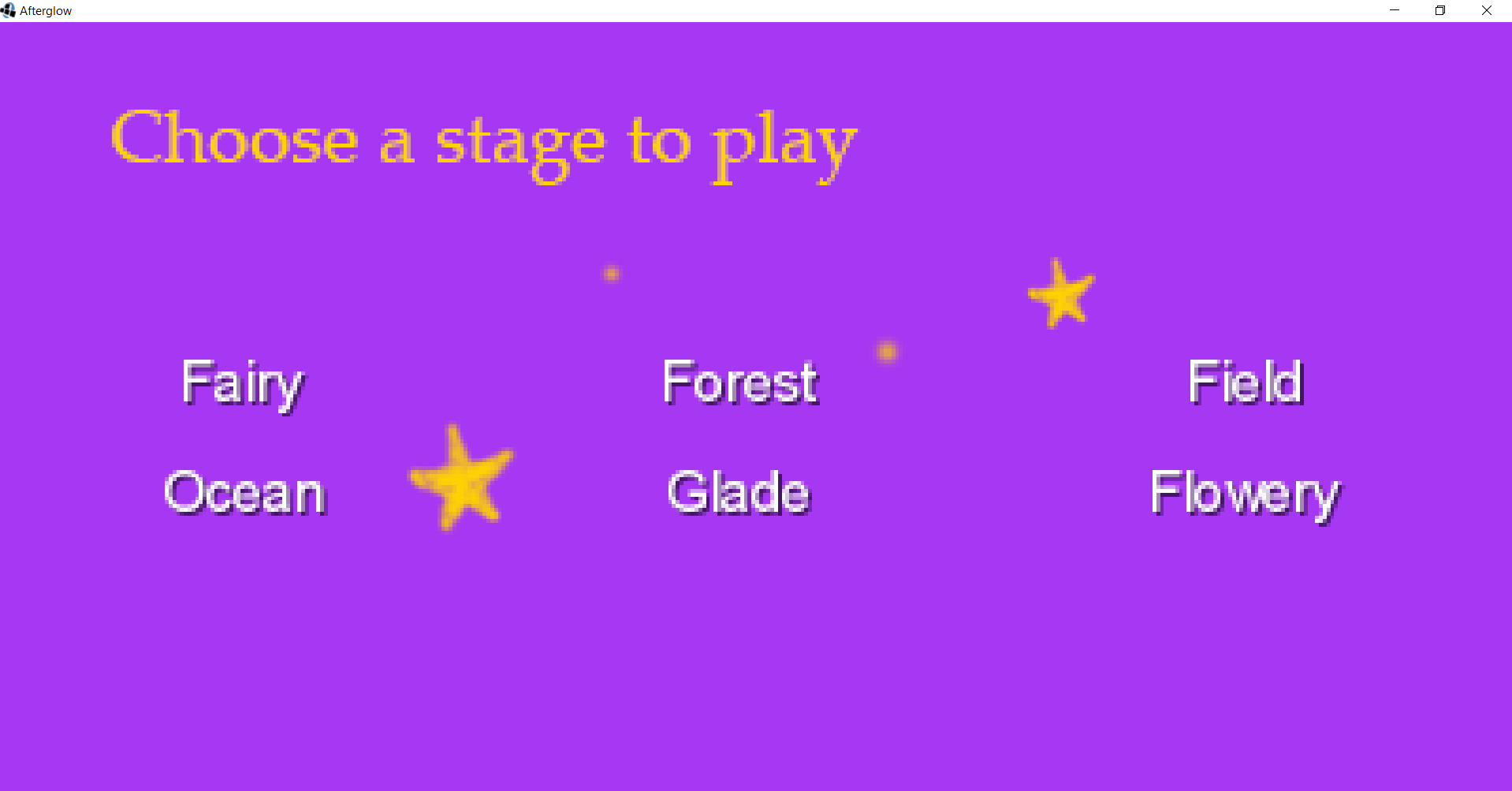


Рисунок 3 – Экран класса LevelScreen

Одним из решений для обработки каждого экрана в игре является использование класса Viewport, который предоставляет несколько вариантов окна просмотра, то есть того, как будет выглядеть приложение на разных устройствах. При выборе StretchViewport параметры программы не сохраняются, она растягивается на весь экран, что показано на рисунке 4.

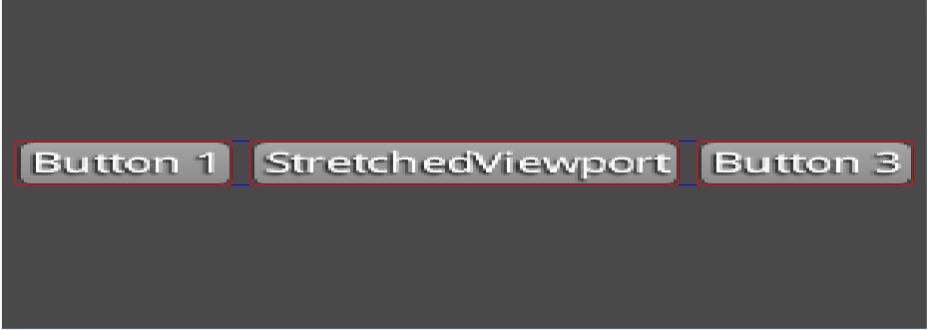


Рисунок 4 – Окно просмотра при выборе StretchViewport в параметрах программы

На рисунке 5 продемонстрирован ScreenViewport, он регулирует размер игрового мира на основании размера экрана, по умолчанию одна единица измерения мира эквивалентна одному пикселу экрана, что может предоставлять преимущества пользователями с широкоэкранными устройствами и менять ход игрового процесса.

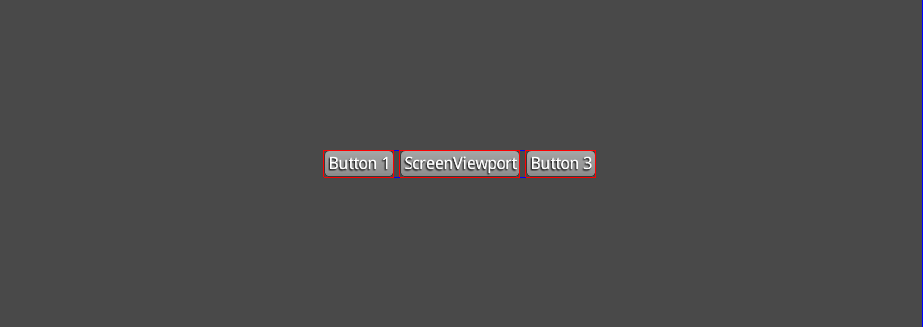


Рисунок 5 – Окно просмотра при выборе ScreenViewport

FitViewport сохраняет параметры мира на всех устройствах, растягивая мир в длину и ширину и добавляя чёрные полосы на оставшемся месте в случае необходимости, как показано на рисунке 6.

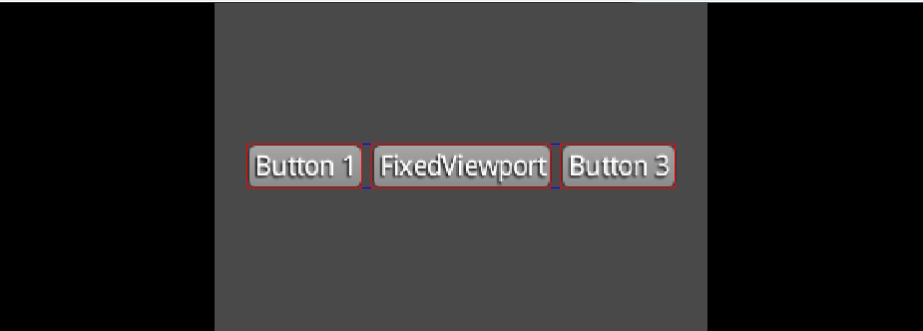


Рисунок 6 – Окно просмотра при выборе FitViewport

При создании данного приложения была выбрана стратегия создания окна просмотра игрового мира с помощью окна просмотра FitViewport.

Менеджер компановки, то есть Scene2d.ui содержит класс Stage, являющийся контейнером для актёров, в роли которых выступают таблицы, виджеты, кнопки и многие другие. Этот класс обрабатывает окно просмотра и управляет событиями ввода. Чтобы класс Stage мог распространять реакцию на входные события среди актёров, все события должны быть обработаны. Этап обработки передаётся методу Gdx.input.setInputProcessor, который входит в состав методов интерфейса InputProcessor и нужен для получения входных событий от мыши (или нажатия на экран) и клавиатуры.

Класс Stage и ее составляющие не являются потокобезопасными и должны обновляться и запрашиваться только из одного потока (предположительно, основного потока рендеринга). Методы должны быть достаточно функциональными для обновления актёров и этапов из обратных вызовов и обработчиков.

В программном коде была создана сцена класса Stage с определённым окном просмотра viewport и указанием созданного в классе Afterglow упаковщика спрайтов batch [3].

Класс Texture нужен для загрузки изображений в GPU и вывода их на экран. Для отображения текстур нужно указать полный путь к ней, при этом принято хранить все изображения в директории assets. Далее изображения передаются на отрисовку переменной batch класса Spritebatch, указываются координаты левого нижнего угла и размера текстуры.

Класс Button представляет пустую кнопку, он имеет несколько видов фона: вверх, отображаемый в обычном состоянии, и вниз, отображаемый при нажатии. Класс Button имеет несколько наследников, которые расширяют его функционал и позволяют создавать кнопки нескольких видов. Класс TextButton расширяет кнопку и позволяет наносить на неё текст. Класс ImageButton расширяет кнопку и содержит изображение, которое может меняться при изменении состояния. Класс ImageButton комбинирует два вышеперечисленных класса и используется для отображения текста и изображений.

При написании программного кода были созданы кнопки класса TextButton, которые представляют разные сцены игры: Fairy, Forest, Field, Ocean, Glade, Flowery.

Класс Table используется для группировки всех актёров, он состоит из невидимых строк и столбцов, в которые помещаются все актёры для упорядоченного отображения на экране.

В данном программном коде создана переменная table класса Table, которая служит контейнером для упорядоченного расположения кнопок на экране, кнопки выводятся в две строки с помощью использования разделяющего метода row().

Метод addListener( new ClickListener() ) обнаруживает наведение курсора мыши, нажатие мыши или пальца и щелчок по актёру, при этом роль играет только первое прикосновение, которое служит триггером для выполнения реакции, описанной в методе.

В методе render() происходит очищение экрана, отрисовка текстуры и сцены.

Все используемые ресурсы освобождаются в методе dispose().

*Игровой класс PlayScreen.* Данный класс выполняет основную логику игры (сцены один: Fairy) и отрисовывает карту игрового мира на экран. Для начала работы с классом задаются переменные gameCam и gamePort типов OrthographicCamera и FitViewport соответственно.

Камера класса Camera определяет точку обзора и то, какая часть мира видна на экране. Всё в игре видно через камеру, которая определяет взгляд на мир. В случае 2D-игр используется камера типа OrthographicCamera, которая определяет прямоугольник мира, который в данный момент виден. OpenGL работает с матрицами для вычисления того, что видно, поэтому, чтобы использовать камеру, нужно обновлять числовые значения и брать объединённую матрицу combined для передачи в SpriteBatch.

Окно просмотра класса Viewport это прямоугольная область экрана, на которую настраивается OpenGL для рисования в пикселях. Независимо от того, какие размеры имеет ортографическая камера, видимый прямоугольник мира, который она определяет, растягивается, чтобы соответствовать окну просмотра OpenGL, что означает, что он может выглядеть искажённым или раздавленным. Переменные viewportWidth и viewportHeight у OrthographicCamera были придуманы ранее, когда предполагалось, что окно камеры всегда будет равно окну просмотра. На самом деле они описывают размер прямоугольника игрового мира, который будет виден на экране, этот прямоугольник может иметь любые размеры и будет изменяться, в зависимости от выбора окна просмотра OpenGL.

Класс Viewport одновременно управляет камерой и окном просмотра OpenGL для достижения определенной стратегии работы с бесконечными возможностями плотности экрана и соотношения сторон. Переменные класса Viewport настраиваются путём предоставления ширины мира и высоты мира, которые определяют размеры прямоугольника игрового мира, который виден на экране, в зависимости от того, какой потомок класса выбран (FitViewport, StretchViewport и так далее).

Таким образом, камера используется для отображения заданных размеров игрового мира, а окно просмотра регулирует, как этот игровой мир будет отображаться на разных устройствах.

При загрузке текстур (изображений) в игру, они загружаются в GPU по отдельности, что занимает много памяти и тормозит игровой процесс. В целях экономии ресурсов принято хранить большое число маленьких изображений на одном большом изображении, а затем многократная визуализация отдельных частей. Утилита TexturePacker служит для упаковки множества маленьких изображений в одну большую текстуру и хранит их местоположение так, что в коде приложения на текстуру можно сослаться по имени, используя класс TextureAtlas.

Из одной большой загруженной текстуры Класс TextureRegion вырезает нужные области, называемые субрегионами TextureRegions, из текстуры в соответствии с предоставленными координатами и размерами, передаёт её классу SpriteBatch, который рисует только область, соответствующую элементу.

Для создания игры Afterglow все изображения, персонажи и карта мира были отрисованы с помощью программы Photoshop. Изображения персонажа и врагов были сохранены в формате .png и собраны в текстуру Player\_and\_Enemies.pack с использованием утилиты TexturePacker.

Карта мира имеет параметры 240х13, при этом каждая единица мира состоит из 16 пикселей. Через программу Tiled, которая является редактором ортогональных карт, были созданы уровни и сохранены с расширением .tmx, которое использует класс TiledMap, распознающий слои. При работе с редактором был создан прямоугольный канвас, представляющий карту шириной 240 и длиной (высотой) 13 ячеек, каждая ячейка в свою очередь делилась ещё на 16 частей, все подразделения сопоставлялись с пикселями, что показано на рисунке 7.

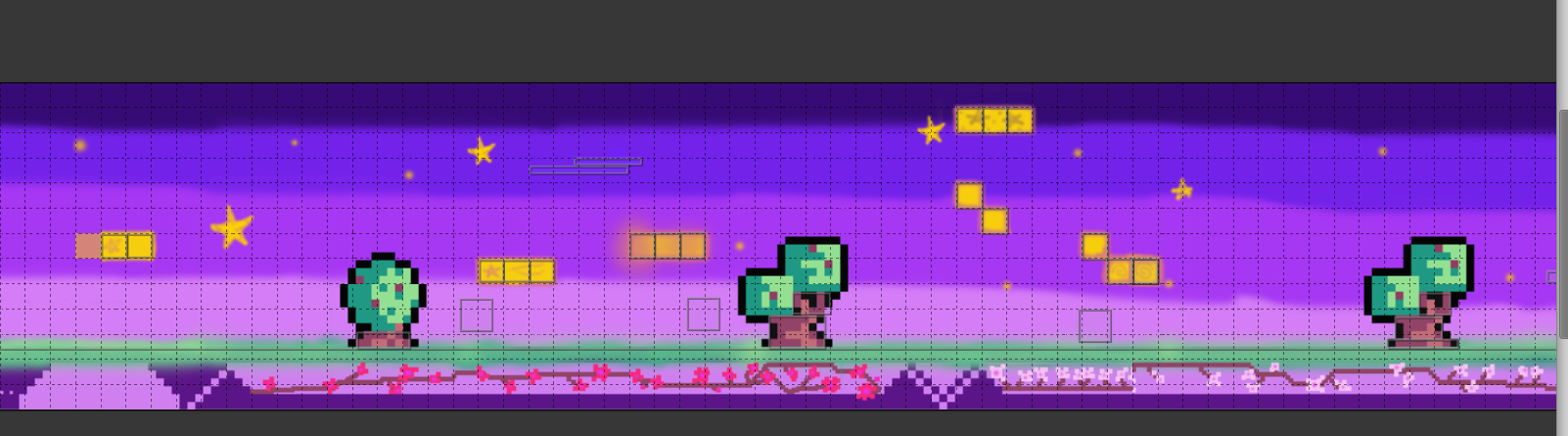


Рисунок 7 – Карта игрового мира

Программа предоставляет возможность работы с тремя типами слоёв: слой картинок (image layer) позволяет наносить на канвас изображения; плиток (tile layer) – используется для отрисовки различных текстур; объектов (object layer) позволяет создавать вокруг нарисованных на предыдущем слое частей карты невидимые многоугольники, на которые можно ссылаться в коде, что облегчает описание столкновений между персонажами. На рисунке 8 можно увидеть, что для создания карты мира было создано девять слоёв: два слоя плиток background и graphics с заливкой заднего фона и нарисованными объектами соответственно и семь слоёв объектов, на которых были выделены земля ground, платформы platforms, призы prizes, облака clouds, деревья trees, грибы mushrooms, крысы rats. Чтобы работать со слоями в коде достаточно вызвать специальный метод и указать индекс нужного слоя, учитывая то, что индексы начинаются с нулевого (ground) и доходят вверх до восьмого (rats).

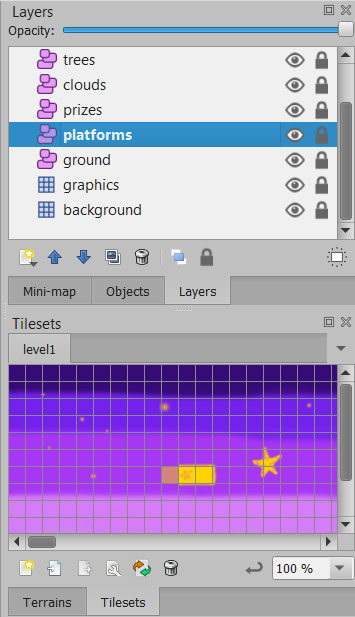


Рисунок 8 – Редактор ортогональных карт Tiled

Таким образом, игровой мир был отрисован в программе Photoshop и собран с помощью редактора карт Tiled. В коде все объекты были перенесены в физический мир Box2D в классе .

Box2D может быть представлен как контейнер, который включает все объекты мира и управляет ими при помощи физики. Объекты называются телами (bodies) и имеют такие атрибуты как масса (mass), скорость (velocity), местоположение (location), углы (angles) – то есть координаты, в которых находится тело в данный момент времени, фиксаторы (fixtures) – физические атрибуты тел для хранения всех описанных выше характеристик, которые нужны так как тела сами по себе являются нематериальными контейнерами. Чтобы представить тело материальным объектом вводится понятие fixtures, которое имеет характеристики имеют формы (shape), плотности или удельного веса (density), friction (трение), возвращение, реституция (restitution) – мера отскока при соприкосновении с другими объектами [4].

Внутри класса были созданы переменные Box2D типов World и Box2DDebugRenderer для графического представления fixtures и bodies внутри мира. Все тела и фиксаторы создаются в дополнительном классе B2WorldCreator, при этом сначала создаются переменные для описания тела и фиксаторов. Тела в Box2D бывают трёх типов: динамические тела (DynamicBody) для обозначения персонажей, на которые влияют силы и скорость и которые имеют такие характеристики как скорость и масса; статические тела (StaticBody) для обозначения объектов без характеристик, на которые не влияют физические силы мира; кинематические тела (KinematicBody), которые имеют характеристики, но не поддаются никаким силам мира, такие тела обычно представлены дыижущимися платформами.

Каждому телу присваивается тип и позиция, каждому фиксатору – форма, после чего телу присваивается соответствующий фиксатор и слой, сопоставимый с прорисовкой объекта в Tiled. Для отдельных фиксаторов станается значение бита коллизии, которое будет рассмотрено позже, а создание кирпичей платформ и призов, грибов и крыс отдаётся специальным классам.

В игровом классе PlayScreen реализована основная логика перемещения игрока с помощью нажатий на стрелки клавиатуры (на экран в случае сенсорных устройств), а также отслеживается состояние нахождения персонажа в игровом процессе, чтобы при принятии состояния DEAD закончить игру.

*Игровой класс GameOverScreen.* Данный класс создаёт текстуру для заднего фона и окно просмотра, а также осуществляет метод по переключению игрового процесса LevelScreen в случае нажатия на экран. Изображение экрана данного класса можно увидеть на рисунке 9.



Рисунок 9 – Экран класса GameOverScreen

*Игровой класс Hud (Heads-up display).* Данный класс содержит строку текущих состояний персонажа, а именно: уровень, количество заработанных очков и количество времени в секундах до конца игры. Все вышеперечисленные характеристики выводятся через использование переменных класса Label, который представляет текстового актёра для занесения в таблицу [5].

На момент начала игры устанавливается начальное значение времени в триста секунд, которое постоянно декрементируется, а также нулевое значение очков, которое увеличивается на двести и сто единиц в случае разрушения персонажем кирпичей платформ игровых классов Prize и Brick соответственно.

Так как игровой класс Hud должен постоянно находиться на экране, для него создаётся отдельное окно просмотра, которое следует за классом на протяжении всего игрового цикла, что можно увидеть на рисунке 10.

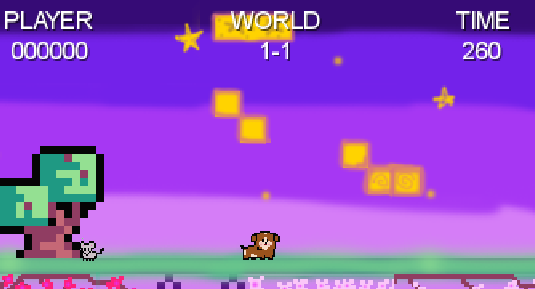
**

Рисунок 10 – Сцена строки состояний Hud

Игровой класс *WorldContactListener.* Данный класс реализует интерфейс класса ContactListener из библиотеки badlogic, который предлагает четыре основных действия по реализации взаимодействия между объектами: метод preSolve для реализации реакции объектов до взаимодействия, postSolve – для реализации реакции объектов после взаимодействия, endContact – для реализации поведения объектов во время последней единицы взаимодействия объектов и метод beginContact для описания поведения объектов при взаимодействии.

В программном коде используется только метод beginContact, который задаёт поведение объектов основных классов Player, Enemies и объектов trees, clouds, ground – то есть тех, для кого были определены биты коллизии в игровом классе B2WorldCreator.

*Игровой класс Player.* Данный класс описывает основную логику поведения игрока.

В начале класса создаётся перечисление основных состояний персонажа FALLING, JUMPING, STANDING, RUNNING, DEAD, определяются текущее и предыдущее состояния игрока в положении State.STANDING, определяется stateTimer для отслеживания за временем нахождения игрока в каждом состоянии для отрисовки анимации и устанавливается значение переменной runningRight в true, чтобы по умолчанию игрок всегда бежал направо. Главного персонажа можно увидеть на рисунке 10.



Рисунок 10 – Персонаж игрового класса Player

Для создания анимации состояния передвижения игрока с ускорением RUNNING создаётся массив, в который с помощью определённого метода записываются кадры созданного ранее изображения. Так как другие состояния не подразумевают анимацию, в них записываются отдельные кадры. Устанавливаются границы тела, радиус и начальное местоположение [6].

Метод getState() заведует выбором состояний игрока, в зависимости от значений скорости по осям X и Y. Следующий метод getFrame() принимает значение игрового времени и устанавливает текущее избражение игрока в зависимости от игрового времени, а также регулирует зеркальный возварот, если персонаж меняет направление и бежит в обратную сторону (влево).

В Box2D существует два варианта управления движением игрока: с помощью скорости и импульса в данном приложении используется импульс как мгновенная реакция персонажа на столкновение.

*Игровой класс Enemy.* Данный класс является абстрактным и определяет базовые функции поведения всех потомков из игровых классов Rat и Mushroom, обозначающих врагов.

*Игровые классы Rat и Mushroom.* Данные классы определяют врагов главного персонажа, то есть крыс и грибов, которых можно увидеть на рисунке 11. Задаются границы и радиусы персонажей, устанавливаются значения переменных setToDestroy и destroyed; которые отвечают за уничтожение врагами главного персонажа и уничтожение главным персонажем врагов соответственно. Для того, чтобы игрок мог разрушить врага, к врагам добавляется дополнительный многоугольник над основной фигурой круга, на который устанавливается дополнительный бит коллизии. Задаётся переменная restitution, которая задаёт меру отскока персонажа собаки от врагов во время коллизии.

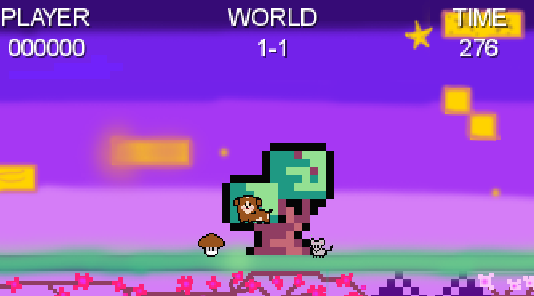


Рисунок 11 – Персонажи игровых классов Mushroom и Rat

*Игровой класс InteractiveTileObject.* Данный класс является абстрактным и определяет базовые функции поведения всех потомков из игровых классов Prize и Brick, обозначающих блоки. Основным методом класса является метод getCell() по получению значения идентификационного номера ячейки в соответствии с её положением в текстовом редакторе Tiled.

*Игровые классы Brick и Prize.* Данные классы расширяют игровой класс InteractiveTileObject и определяют два вида кирпичей, из которых состоят платформы. Для всех кирпичей задаются значения очков, полученных при разрушении их игроком, для Brick – 100, а для Prize – 200. В случае разрушения объекта класса Prize кирпич заменяется кирпичём, заданным по идентификационному номеру; при разрушении объекта класса Brick, ячейка принимает значение null и исчезает. Каждому объекту присваивается бит коллизии, который регулирует столкновение персонажей с кирпичами.

# 2 Обоснование выбора технологий

## Использование языка программирования Java

Язык программирования Java разработала компания Sun Microsystems в начале 90-х годов XX века. Ведущую роль в создании языка сыграл канадский инженер Джеймс Гослинг (James Gosling). На ранних этапах разработки язык назывался Oak. Затем его переименовали в честь сорта кофе Java. Связь языка с напитком отражается в логотипе. Джеймс Гослинг и его единомышленники хотели создать язык с си-подобным синтаксисом, в то же время он должен был быть более простым по сравнению с C/C++. Создатели планировали использовать Java для программирования бытовой электроники. Однако практически сразу после выпуска версии 1.0 в 1995 язык стали использовать разработчики серверного и клиентского ПО.

Java – язык программирования общего назначения. Относится к объектно-ориентированным языкам программирования, к языкам с сильной типизацией. Java является объектно-ориентированным языком, относится к языкам программирования с сильной типизацией. Создатели реализовали принцип WORA: write once, run anywhere или «пиши один раз, запускай везде». Это значит, что написанное на Java приложение можно запустить на любой платформе, если на ней установлена среда исполнения Java (JRE, Java Runtime Environment). Эта задача решается благодаря компиляции написанного на Java кода в байт-код. Этот формат исполняет JVM или виртуальная машина Java. JVM – часть среды исполнения Java (JRE). Виртуальная машина не зависит от платформы [7].

В Java реализован механизм управления памятью, который называется сборщиком мусора или garbage collector. Разработчик создаёт объекты, а JRE с помощью сборщика мусора очищает память, когда объекты перестают использоваться.

Синтаксис языка Java похож на синтаксис других си-подобных языков. К его особенностям относятся:

1. Чувствительность к регистру – идентификаторы User и user в Java представляют собой разные сущности.
2. Для именования методов используется lowerCamelCase. Если название метода состоит из одного слова, оно должно начинаться со строчной буквы.
3. Для именования классов используется UpperCamelCase. Если название состоит из одного слова, оно должно начинаться с прописной буквы.
4. Название файлов программы должно точно совпадать с названием класса с учётом чувствительности к регистру.
5. Идентификаторы всегда начинаются с буквы (A – Z, a – z), знака $ или нижнего подчёркивания.

Язык программирования Java относится к языкам общего назначения, на нём возможно написание серверных и клиентских приложений, веб-приложений с помощью фреймворков Spring, Stuts, JSP, игр с помощью фреймворка libGDX.

## Использование технологий объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Одним из ключевых методов объектно-ориентированного программирования является абстракция. Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой. В ООП рассматривают лишь абстракцию данных (нередко называя её просто «абстракцией»), подразумевая набор значимых характеристик объекта, доступный остальной программе.

Одним из ключевых методов объектно-ориентированного программирования является инкапсуляция. Свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе. Одни языки (например, С++, Java или Ruby) отождествляют инкапсуляцию с сокрытием, но другие (Smalltalk, Eiffel, OCaml) различают эти понятия. То есть это сокрытие поведения объекта внутри него. Объекту «водитель» не нужно знать, что происходит в объекте «машина», чтобы она ехала.

Одним из ключевых методов объектно-ориентированного программирования является наследование. Свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс – потомком, наследником, дочерним или производным классом.

Есть объекты «человек» и «водитель». У них есть явно что-то общее. Наследование позволяет выделить это общее в один объект (в данном случае более общим будет человек), а водителя – определить как человека, но с дополнительными свойствами и/или поведением. Например, у водителя есть водительские права, а у человека их может не быть.

Одним из ключевых методов объектно-ориентированного программирования является полиморфизм. Свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Это переопределение поведения. Можно снова рассмотреть «человека» и «водителя», но теперь добавить «пешехода». Человек умеет как-то передвигаться, но как именно, зависит от того, водитель он или пешеход. То есть у пешехода и водителя схожее поведение, но реализованное по-разному: один перемещается ногами, другой – на машине.

Класс – универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей). В частности, в классах широко используются специальные блоки из одного или чаще двух спаренных методов, отвечающих за элементарные операции с определенным полем (интерфейс присваивания и считывания значения), которые имитируют непосредственный доступ к полю.

Объект – сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

## Использование фреймворка libGDX

Фреймворк libGDX – это бесплатная платформа с открытым исходным кодом, написанная на языке программирования Java, которая позволяет разрабатывать настольные и мобильные игры, используя одну и ту же кодовую базу. Так как фреймворк кросс-платформенный, осуществляется беспроблемный перенос программы на такие аппаратные платформы как Windows, Linux, Mac OS X, Android, iOS, BlackBerry и веб-браузеры с поддержкой WebGL.

Написание кода ведётся на языке программирования Java, несомненным плюсом платформы является возможность большого количества управления со стороны разработчика, например, контроль циклов рисования и обработка событий ввода на базовом уровне. Для написания кода не выставлено жёстких требований, поэтому его стиль задаётся программистом, при этом полученная игра не занимает много места [6].

Box2D является физическим движком реального времени и предназначен для работы с двухмерными физическими объектами, он используется, если целью является разработка игры с видом сбоку/платформера. Разработка в libGDX ведётся через пять основных интерфейсов, которые взаимодействуют с операционной системой и реализуют бэкенд. Интерфейс Application запускает приложение и сообщает клиенту о событиях уровня приложения, например, изменение размера окна, обеспечивает логирование запросов. Интерфейс Files предоставляет лежащую в основе платформы файловую систему. Интерфейс Input информирует API клиента о пользовательском вводе и поддерживает обработку событий. Интерфейс Audio предоставляет средства для воспроизведения звуковых эффектов и потока музыки, обеспечивает прямой доступ к аудиоустройствам. Интерфейс Graphics предоставляет OpenGL и позволяет запрашивать и устанавливать видеорежим.

Для вызова конкретной платформы создаётся специальный класс, так, модуль Android запускает код на соответствующей мобильной платформе или эмуляторе, а модуль Core содержит пакет игры, в котором находится главный класс с логикой и другими пакетами в слцчае необходимости. Главный класс реализует интерфейс ApplicationListener (или его потомков) и поэтому имеет несколько обязательных методов: create() создаёт основные переменные игры, render() отрисовывает изображение на экран, resize() изменяет размер игрового окна, pause() вызывается при остановке игры, а resume() – при продолжении, dispose() освобождает ресурсы и седит за очищением памяти. Эти методы позволяют настроить игру и управлять её отрисовкой. Пока они пусты, игра ничего не отображает, только выводит пустой экран.

*Основные классы фреймворка libGDX.* Класс Stage может содержать в себе актёров (Actor) и управлять ими. Также, есть класс Group, который является и актёром, и контейнером для актёров. Все визуальные компоненты из пакета scene2d.ua унаследованы или от Actor, или от Group. Соответсвенно, наследники Group могут содержать в себе другие компоненты. Прямым наследником класса Actor является класс Widget, а от него унаследуются следующие шесть компонентов: Label, TextField, Image, List, SelectBox, Slider.

Все визуальные компоненты реализуют интерфейс Layout. Этот интерфейс нужен для правильного расположения компонентов в компонентах-контейнерах. Он предоставляет методы для перерисовки компонента, определения предпочтительного, минимального, максимального размера.

Внутри у почти каждого графического компонента есть статический класс с названием <ИмяКомпонента>Style. Например, LabelStyle. В этом классе хранятся нужные для корректной работы этого класса ресурсы – например, шрифт, цвет шрифта и так далее. Так как OpenGL не может напрямую отрисовывать шрифты, нужны дополнительные механизмы. Очевидным вариантом является создание картинки с изображениями букв и текстового файла, который описывает, где какая буква расположена. Потом из этого описания можно сделать класс, который сможет строить строчки текста из отдельных букв-картинок. Для хранения стилей используется класс Skin. Класс Skin читает стили из файла с настройками и сохраняет их внутри себя. Затем при создании нового графического компонента передаётся Skin ему в конструктор. Если есть подходящий стиль, он используется. Если нет – выдаётся исключение [8].

Для каждого компонента можно определить несколько стилей с разными именами. Получить нужный стиль можно с помощью метода getStyle(Class type, String name) класса Skin. По умолчанию, в конструктор передаётся компонента Skin и из него компонент берет стиль с названием default.

Для файла настроек нужно, чтобы в том же каталоге лежал одноименный файл-картинка, но с расширением png. Если, например, вызывается файл настроек SimpleSkin, то графический файл будет называться SimpleSkin.png. Он необходим, если используются ресурсы типа TextureRegion. Без этого файла программа выдаст исключение и не запустится.

Класс Label являет собой простую текстовую метку с возможностью переноса текста по словам. Имеет несколько конструкторов, например, по созданию метки с текстом и по созданию метки без текста.

Имеются методы для установки и возвращения стиля – setStyle(), getStyle(), метод для установки текста setText() и метод для установки переноса по словам setWrapt(). По умолчанию, перенос по словам отключен. Чтобы использовать его, предварительно задаётся предпочтительный размер метки (preferred size).

Класс TextField являет собой поле для ввода текста. Этот класс подразумевает возможность назначения слушателя (listener). Установка слушателя производится методом setTextFieldListener(), который принимает параметр типа TextFieldListener. У класса есть несколько конструкторов, например, по созданию текстового поля с текстом и по созданию пустого текстового поля. К основным методам класса относятся: метод установления текста setText(); метод вывода текста-подсказки setMessageText(), если ничего не было введено в текстовое поле; метод setPasswordMode() по отбражению символов вместо текста и метод setPasswordCharacter() для выбора символов.

Класс Image не требует стиля для своей работы. Создать обьект этого класса можно несколькими конструкторами, например, Image(TextureRegion region). Класс поддерживает слушателя типа ClickListener, то есть можно обрабатывать события нажатия на наше изображение. Дополнительным методом является setRegion(), который позволяет изменить изображение. Можно передать конструктору как текстуру, так и ее регион. Однако масштабирование, вращение будет работать лишь в случае региона текстуры.

Класс List являет собой список текстовых полей с возможностью выбора определенного поля. Он имеет несколько конструкторов, один из наиболее простых – List(Object[] items, Skin skin). Он создает обьект из списком из массива items. Узнать выделенный элемент вы можете методом getSelection(), а его индекс – getSelectedIndex(). Установить новый список можно методом setItems(), сделать какой-либо элемент выделенным можно или по имени или по индексу: setSelection(), setSelectionIndex(). List поддерживает слушателя SelectionListener, который вызывает метод selected() при выборе элемента [9].

Класс SelectBox являет собой выпадающий список из нескольких элементов. В неактивном состоянии это строка, которая показывает выбранный элемент. Когда он становится активным, показывается список элементов, которые можно выбрать нажатием. Имеет несколько конструкторов, один из них – SelectBox(Object[] items, Skin skin) – создает выпадающий список элементов. Первый элемент считается выбранным. В остальном класс очень похож на List, так как он поддерживает таких же слушателей как List – SelectionListener, имеет такие же методы для манипулирования элементами списка.

# 3 Инструментарий

## 3.1 Среда разработки Android Studio

Android Studio – это официальная интегрированная среда разработки (IDE) для разработки приложений под Android, основанная на идее IntelliJ. Кроме мощного редактора кода и инструментов разработчика IntelliJ, Android Studio предлагает еще больше функций, которые повышают производительность при создании приложений для Android, таких как:

* гибкая система сборки на основе Gradle;
* быстрый и многофункциональный эмулятор;
* единая среда, в которой можно разрабатывать программы для всех устройств Android;
* применение изменений к коду в запущенном приложении без перезапуска приложения;
* шаблоны кода и интеграция с GitHub для создания общих функций приложения и импортирования примеров кода;
* обширные инструменты тестирования и фреймворки;
* инструменты Lint для отслеживания производительности, удобства использования, совместимости версий и других проблем;
* поддержка C++ и NDK;
* встроенная поддержка облачной платформы Google, облегчающая интеграцию облачных сообщений Google и движка приложений.

*Структура проекта.* Каждый проект в Android Studio содержит один или несколько модулей с файлами исходного кода и файлами ресурсов. Типы модулей включают в себя:

* модули приложений для Android;
* библиотечные модули;
* модули Google App Engine.

По умолчанию Android Studio отображает файлы проекта в представлении проекта Android. Это представление организовано по модулям, чтобы обеспечить быстрый доступ к ключевым исходным файлам вашего проекта.

Все файлы сборки отображаются на верхнем уровне в разделе Скрипты Gradle, и каждый модуль приложения содержит следующие папки:

* манифесты: содержит AndroidManifest.xml файл;
* java: содержит файлы исходного кода Java, включая тестовый код Junit;
* res: содержит все некодовые ресурсы, такие как XML-макеты, строки пользовательского интерфейса и растровые изображения.

*Пользовательский интерфейс.* Главное окно Android Studio состоит из нескольких логических областей.

Панель инструментов позволяет выполнять широкий спектр действий, включая запуск приложения и запуск инструментов Android.

Панель навигации помогает перемещаться по проекту и открывать файлы для редактирования. Это обеспечивает более компактное представление структуры, видимой в окне проекта.

В окне редактора создаётся и изменяется код. В зависимости от текущего типа файла редактор может изменяться. Например, при просмотре файла макета редактор отображает редактор макета.

Панель окна инструментов содержит кнопки, позволяющие разворачивать или сворачивать отдельные окна инструментов.

Окна инструментов предоставляют доступ к конкретным задачам, таким как управление проектами, поиск, контроль версий и многое другое.

В строке состояния отображается состояние проекта и самой среды разработки, а также любые предупреждения или сообщения.

*Система сборки Gradle.* Android Studio использует Gradle в качестве основы системы сборки, с более специфичными для Android возможностями, предоставляемыми плагином Android для Gradle. Эта система сборки работает как интегрированный инструмент из меню Android Studio и независимо от командной строки. Можно использовать функции системы сборки для выполнения следующих действий:

* настройка и расширение процесса сборки;
* создание нескольких APK для своего приложения с различными функциями, используя один и тот же проект и модули;
* повторное использование кода и ресурсов в разных наборах источников.

*Файлы сборки Android Studio*. Данный файлы называются build.gradle. Это простые текстовые файлы, которые используют синтаксис Groovy для настройки сборки с элементами, предоставляемыми плагином Android для Gradle. Каждый проект имеет один файл сборки верхнего уровня для всего проекта и отдельные файлы сборки уровня модуля для каждого модуля. При импорте существующего проекта Android Studio автоматически генерирует необходимые файлы сборки.

*Сокращение ресурсов.* Сокращение ресурсов в Android Studio автоматически удаляет неиспользуемые ресурсы из ваших упакованных зависимостей приложений и библиотек.

*Профилировщик памяти.* Профилировщик памяти можно использовать для отслеживания выделения памяти и отслеживания того, где находятся объекты при выполнении определенных действий. Владение данной информацией позволяет оптимизировать производительность приложения и использование памяти, регулируя вызовы методов, связанные с этими действиями.

*Доступ к файлам данных.* Инструменты Android SDK, такие как Systrace и logcat, генерируют данные о производительности и отладке для детального анализа приложений.

## 3.2 Графический редактор Photoshop

Adobe Photoshop – это программное приложение для редактирования изображений и ретуши фотографий на компьютерах с Windows или macOS. Photoshop предлагает пользователям возможность создавать, улучшать или иным образом редактировать изображения и иллюстрации. Это наиболее широко используемый программный инструмент для редактирования фотографий, обработки изображений и ретуши многочисленных форматов изображений и видеофайлов. Инструменты в Photoshop позволяют редактировать как отдельные изображения, так и большие партии фотографий.

Adobe Photoshop – это важный инструмент для дизайнеров, веб-разработчиков, художников-графиков, фотографов. Он широко используется для редактирования изображений, ретуши, создания композиций изображений, макетов веб-сайтов и добавления эффектов. Цифровые или отсканированные изображения можно редактировать для использования в Интернете или в печати. Макеты веб-сайтов могут быть созданы в Photoshop; их дизайн может быть доработан до того, как разработчики перейдут к этапу кодирования. Автономная графика может быть создана и экспортирована для использования в других программах.

Adobe Photoshop Elements – это потребительская версия семейства продуктов Photoshop. Photoshop Elements содержит множество профессиональных возможностей, которые можно найти в Adobe Photoshop CC, но они снабжены более упрощенными опциями, разработанными с учетом пользователя начального уровня. Более конкретно, он предназначен для фотографов-любителей и любителей цифровой фотографии. Photoshop Elements построен с использованием той же базовой технологии цифрового изображения, что и Photoshop CC. Обычно используемые возможности Photoshop Elements включают в себя:

* манипулирование цветом изображения;
* обрезка изображений;
* устранение дефектов, таких как пыль на линзах или красные глаза;
* рисование на изображении ручкой или карандашом;
* добавление текста к изображениям;
* удаление людей или объектов внутри изображения;
* организация фотографий для быстрого доступа;
* публикация изображений в Интернете или отправка по электронной почте;

Данные функции по работе с изображениями делают графиический редактор Adobe Photoshop оптимальным вариантом для работы с обработкой изображений.

## 3.3 Система контроля версий GIT

Git – это бесплатная распределенная система контроля версий с открытым исходным кодом, предназначенная для быстрой и эффективной работы как с небольшими, так и очень крупными проектами. Git легок в освоении и имеет молниеносную производительность. Он превосходит SCM-инструменты, такие как Subversion, CVS, Perforce и ClearCase, с такими функциями, как локальное ветвление, удобные промежуточные области и несколько рабочих процессов.

Git предназначен для координации работы программистов, но его можно использовать для отслеживания изменений в любом наборе файлов.

Контроль версий – это система, которая записывает изменения в файле или наборе файлов с течением времени, чтобы разработчики могли впоследствии вызывать определенные версии.

Многие выбирают метод контроля версий для того, чтобы копировать файлы в другой каталог (возможно, в каталог с меткой времени). Этот подход очень распространен, потому что очень прост, но он также невероятно подвержен ошибкам.

Легко забыть о текущем каталоге и случайно записать в неправильный файл или скопировать файлы, которые нет необходимости использовать. Чтобы решить эту проблему, программисты давно разработали локальные VCS, которые имели простую базу данных, в которой все изменения в файлах находились под контролем версий.

Git поддерживает быстрое ветвление и слияние и включает в себя специальные инструменты для визуализации и навигации по нелинейной истории разработки. В Git основное предположение состоит в том, что изменение будет объединяться чаще, чем оно написано, поскольку оно передается различным рецензентам. В Git ветви очень легкие: ветка – это ссылка только на один коммит. С его родительскими коммитами может быть построена полная структура ветви.

Git предоставляет каждому разработчику локальную копию полной истории разработки, а изменения копируются из одного такого хранилища в другое. Эти изменения импортируются как добавленные ветки разработки и могут быть объединены так же, как и локально разработанная ветка.

Репозитории могут публиковаться через протокол передачи гипертекста (HTTP), протокол передачи файлов (FTP) или протокол Git через простой сокет или Secure Shell (ssh). Git также имеет эмуляцию сервера CVS, которая позволяет использовать существующие клиенты CVS и плагины IDE для доступа к репозиториям Git.

## 3.4 Редактор ортогональных карт Tiled

Tiled – это редактор 2D уровней, который помогает разработать контент игры. Его основная функция заключается в редактировании плиточных карт различных форм, он также поддерживает размещение изображений, и мощные способы снабжения уровня дополнительной информацией, используемой в игре.

Основным инструментом для редактирования слоев плитки является кисть-штамп, которая позволяет рисовать и копировать области плитки, линии и круги [10].

Карты tiled maps создаются в форме прямоугольного канваса и состоят из наборов плиток, размещённых на разных слоях.

Существует три основных слоя представления объектов:

* слой картинок (image layer) позволяет наносить на канвас изображения;
* слой плиток (tile layer) – используется для отрисовки различных текстур;
* слой объектов (object layer) позволяет создавать вокруг нарисованных на предыдущем слое частей карты невидимые многоугольники, на которые можно ссылаться в коде, что облегчает описание столкновений между персонажами.

Использование редактора карт уровней Tiled необходимо при работе с фреймворком libGDX в случае написания игр с ручной отрисовкой текстур с использованием пиксельной графики.

# 4 Кроссплатформенный фреймворк LibGDX

## 4.1 Архитектура фреймворка libGDX

Использование Scene2D при создании игры с помощью фреймворка libGDX является необязательным, однако структурирует архитектуру приложения. Scene2D это 2D граф сцены, то есть это структура данных для хранения всех дополнительных материалов мира. Таким образом, если игровой мир состоит из большого количества спрайтов, все они хранятся в графе сцены scene2D. Кроме хранения контекста мира, Scene2D предоставляет определённое количество методов, которые выполняются над данными. Это такие операции как отслеживание столкновений, создание иерархий между объектами, обработка ввода и так далее.

Scene2D может интерпретироваться как дополнительная надстройка над фреймворком LibGDX для создания игрового процесса, кроме этого данный граф сцены предоставляет библиотеку с виджетами.

Ниже представлены особенности пакета Scene2D:

Вращение и масштабирование группы объектов. Дочерние объекты работают в своей системе координат, так что родительские преобразования для них прозрачны.

Упрощённый 2D рендеринг с использованием SpriteBatch. Каждый объект существует в своей невращаемой нескалируемой системе координат, где координата (0,0) представляет нижний левый угол объекта.

1. Очень гибкая система событий, позволяющая обрабатывать события родителя перед/после обработки событий дочерних объектов.
2. Система воздействий для лёгкой манипуляции над объектами в любой момент. Воздействия можно объединять для более сложных эффектов.
3. Объекты совмещают и модель, и представление, поэтому всё, что касается объекта, находится в классе объекта.

Объектный дизайн Scene2D построен вокруг игрового процесса, который можно отождествить с пьесой. На вершине иерархии находится сцена класса Stage, на которой осуществляется игровой процесс. Сцена содержит окно просмотра класса Viewport. Класс Actor служит для создания актёров, то есть сущностей, которые заполняют класс Stage. Объектами класса Actor могут являться таблицы Table, виджеты Widget, кнопки класса Button и многие другие. Таким образом, игровой процесс разделяется на логические сцены (экраны, урони и так далее), состоящие из актёров.

Пакет Scene2d состоит из 3 основных классов:

1. Actor: Класс Actor (актёр) – это центр графа сцены, он имеет положение, размер с параметрами прямоугольника, начало координат, масштаб, поворот и цвет.
2. Group: Класс Group (группа) – это актёр, у которого могут быть дочерние актеры.
3. Stage: Класс Stage имеет камеру, SpriteBatch и обрабатывает отрисовку актёров и распределение входных событий.

Класс Stage – это контейнер, который содержит и обрабатывает актёров. Кроме того, у него есть метод, который принимает дельта-время в качестве аргумента, и вызывает метод act() каждого актёра, позволяя актёрам выполнять некоторые действия, основанные на времени работы игрового цикла. Конструктор класса Stage принимает два аргумента, это Viewport и Batch. Так как Viewport подразумевает отслеживание камеры класса Camera, она не передаётся в качестве аргумента. Аргумент Batch также является необязательным, так как класс Stage сам руководит его отслеживанием и очищением. Если же аргумент класса Batch передаётся в конструктор, нужно освобождать ресурсы вручную. Класс Stage являет собой «корневую» группу(Group) куда приложение может добавлять своих актеров. У класса есть своя камера и упаковщик (SpriteBatch). Stage реализует интерфейс InputProcessor и отсылает события дочерним элементам.

Класс Group является актёром который может содержать других актеров. Вращение и масштабирование Group 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.

Базовая архитектура Scene2D создаёт приложение с одним актёром класса Stage. Актёр загружает текстуру из файла и отрисовывает с помощью метода draw(), который вызывается каждый кадр игры классом Stage. Переменная класса Stage создаётся с помощью метода create() при передаче параметров устройства, метод render() очищает экран и вызывает метод draw(), метод dispose() очищает все русурсы сцены, чтобы предотвратить утечку памяти.

## 4.2 Отрисовка текстур

Изображение, полученное из его оригинального формата (например, PNG) и загруженное в GPU называется текстурой. Текстуры рисуются по некоторой спецификации, которая представляет собой описание геометрической фигуры и каким образом текстура накладывается на вершины этой фигуры. Например, геометрическая фигура может быть прямоугольником и каждый угол прямоугольника относится к соответствующему углу текстуры.

Для рисования необходимо, чтобы текстура была сделана текущей (привязана) и чтобы была задана геометрия. Размер и расположение места, куда будет выведена текстура, определяются геометрией и настройкой окна просмотра OpenGL. 2D игры настраивают окно просмотра так, чтобы оно совпадало с разрешением экрана. Это значит, что геометрия определяется в пикселях, что не затрудняет отрисовку текстур соответсвующего размера и в нужном месте экрана.

Рисование текстур происходит в прямоугольной геометрии, также часто одна текстура или её разные части рисуются многократно. Неэффективно отсылать один прямоугольник для отрисовки в GPU. Вместо этого, много прямоугольников для для одной текстуры могут быть описаны и отправлены в GPU все вместе. Для этого нужен класс SpriteBatch, являющийся упаковщиком спрайтов.

SpriteBatch получает текстуру и координаты каждого прямоугольника, куда будет выведена эта текстура. Он накапливает эту информация без отсылки в GPU. Когда он получает текстуру, которая отличается от последней загруженной текстуры, он делает активной последнюю загруженную текстуру, отсылает накопленную информация по рисованию в GPU и начинает накапливать данные по отрисовке для следующей текстуры.

Изменение текстуры каждые несколько прямоугольников, которые должны отрисовываться, мешает SpriteBatch группировать достаточно много прямоугольников. Также, привязка текстуры затратная для памяти операция. Учитывая эти причины, часто хранят много мелких изображений в одном большом изображении и потом рисуют части большого изображения, максимизируя этим количество накопленных прямоугольников для отрисовки и избегая смены текстуры. Все вызовы SpriteBatch для отрисовки должны быть заключены между методами begin() и end(). Вызовы методов для рисования другими средствами (не классом SpriteBatch) не должно встречаться между методами begin() и end().

*Texture (текстура).* Класс Texture получает изображение из файла, и загружает его в GPU. Файл изображения должен быть размещен в директории assets, размеры изображения должны быть степенью двойки (16х16, 64х256 и так далее). Созданная текстура отправляется классу SpriteBatch для отрисовки. Текстура рисуется в прямогольнике, с указанием левого нижнего угла и ширины и высоты в зависимости от размеров текстуры.

*TextureRegion (регион текстуры или часть текстуры).* Класс TextureRegion описывает прямоугольник внутри текстуры и используется для рисования только части текстуры. Это же действие может быть сделано передачей текстуры и дополнительных параметров классу SpriteBatch, но класс TextureRegion делает это удобнее, поскольку более оптимизированным решением является выделение отдельного обьекта и работа с ним, а не с большим количеством дополнительных параметров.

*Sprite (спрайт).* Класс Sprite описывает регион текстуры, положение, где будет рисоваться этот регион и цвет, которым будет для региона (цвет для tinting – затенения цветом). Это же может быть сделано передачей текстуры или её части в SpriteBatch и передачей других параметров, но класс Sprite делает это оптимальнее, поскольку хранит все параметры в одном месте. Также, благодаря тому, что класс Sprite хранит геометрию внутри себя и пересчитывет ее лишь в случае необходимости, это улучшает производительность в операциях масштабирования, вращения или других свойствах, которые не меняются между кадрами.

Класс Sprite смешивает информацию про модель (расположение, информацию про вращение и другие) с информацией про представление (текстура была нарисована этим же классом). Это делает неподходящим использование Sprite в архитектуре, где нужно строго отделить модель от представления. В этом случае использование класса Texture или TextureRegion может иметь больше смысла.

SpriteBatch имеет собственную проекцию и матрицу преобразований. Когда создается объект класса SpriteBatch, он использует размеры текущего приложения для настройки ортогональной проекции используя систему координат с осью Y, направленной вверх (то есть, 0, 0 – это левый нижний угол экрана). Когда вызывается метод begin(), SpriteBatch устанавливает свою область просмотра.

Улучшение производительности. SpriteBatch имеет конструктор, который принимает максимальное количество спрайтов, которое будет накоплено перед отправкой до GPU. Если это число слишком мало, будет много лишних вызовов видеоускорителя. Если число большое, SpriteBatch будет использовать слишком много памяти. SpriteBatch имеет public поле, которое называется maxSpritesInBatch. Оно указывает на максимальное число спрайтов, которое может быть отослано на отрисовку видеоускорителю за один раз на протяжении жизненного цикла SpriteBatch. Можно выставить это число очень большим и проверять его. Это поможет подобрать оптимальный размер SpriteBatch. Размер SpriteBatch (число, которое вы передаете в конструктор) должен слегка превышать число maxSpritesInBatch.

SpriteBatch имеет публичное поле renderCalls. После очередного цикла рендеринга здесь хранится количество раз, которое SpriteBatch отсылал разные данные про геометрию между вызовами begin() и end(). Это случается, если привязываются разные текстуры, или если SpriteBatch был переполнен (слишком маленький кэш). Если размер SpriteBatch подобран правильно и renderCalls слишком велико (около 15 – 20) это свидетельсвует, об использование слишком большого количества текстур.

SpriteBatch имеет дополнительный конструктор который принимает число и размеры буферов. Эта технологиятприказывает работать с VBO (vertex buffer objects) вместо обычного VA (vertex arrays). SpriteBatch хранит список буферов и при каждом очередном цикле рендеринга использует следующий буфер.

## 4.3 Камера и область просмотра игрового пространства

*Класс Camera.* Камера фреймворка libGDX работает как камера реального времени, она устанавливает взгляд на мир, то есть изображение объектов и точку обзора. OpenGL работает с матрицами для вычисления того, что видно, поэтому, чтобы использовать камеру, нужно обновлять её и высчитывать объединенную матрицу для передачи в SpriteBatch.

Класс Camera является базовым классом для классов [OrthographicCamera](https://libgdx.badlogicgames.com/ci/nightlies/docs/api/com/badlogic/gdx/graphics/OrthographicCamera.html) и [PerspectiveCamera](https://libgdx.badlogicgames.com/ci/nightlies/docs/api/com/badlogic/gdx/graphics/PerspectiveCamera.html).

*Класс* [*PerspectiveCamera*](https://libgdx.badlogicgames.com/ci/nightlies/docs/api/com/badlogic/gdx/graphics/PerspectiveCamera.html)*.* Перспективная камера используется при проектировании 3D мира и создаёт более естественный перспективный вид, так как при смене угла наклона камеры меняются размеры объектов. Так, чем дальше находятся предметы от камеры, тем меньше они выглядят.

*Класс* [*OrthographicCamera*](https://libgdx.badlogicgames.com/ci/nightlies/docs/api/com/badlogic/gdx/graphics/OrthographicCamera.html)*.* Ортографическая камера используется в 2D средах там, где нужно реализовать параллельную (ортографическую) проекцию и где нет масштабирования (изменения размеров) объектов в зависимости от их местонахождения в мире. Таким образом, расстояние между объектом и камерой не имеет значения, и предметы не меняют размеров в зависимости от угла наклона камеры.

*Окно просмотра Viewport.* Класс Camera регулирует восприятие мира, а класс Viewport регулирует прямоугольную область, которая будет показана на экране. Независимо от того, какие размеры имеет ортографическая камера, видимый прямоугольник мира, который она определяет, растягивается, чтобы соответствовать видовому окну OpenGL, что означает, что он может выглядеть искаженным/раздавленным. В зависимости от того, какой тип окна просмотра используется, этот прямоугольник будет выглядеть по-разному для разных экранов. Некоторые примеры видовых экранов Viewport:

StretchViewport растягивает ширину/высоту мира, чтобы соответствовать экрану, даже если это вызывает искажения.

FitViewport увеличивает ширину/высоту мира, чтобы соответствовать экрану, и добавляет чёрные полосы для сохранения пропорций мира таким образом, чтобы изображение не искажалось.

ExtendViewport увеличивает ширину/высоту мира, чтобы соответствовать экрану, а затем расширяет высоту/ширину демонстрации мира, чтобы заполнить оставшееся место на экране и не изменить размеры. Такой метод помогает избежать вставки чёрных прямоугольников, но делает необходимым дорисовку дополнительной области за пределами высоты и ширины игры.

ScreenViewport: не подразумевает входных данных для ширины/высоты мира, так как меняет эти параметры для соответствовали размеру экрана в пикселях, поэтому такой тип окна просмотра лучше использовать при создании дизайна UI.

# 5 Шаблоны проектирования практических задач

Как правило, шаблоны проектирования внедряются в уже существующую архитектуру проекта.

В отличие от готовых функций или библиотек, шаблон нельзя просто взять и скопировать в программу. Шаблон проектирования представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды программы.

Такой формализм в описании позволил создать обширный каталог шаблон, проверив каждый из них на состоятельность. Шаблоны проектирования практических решений отличаются по уровню сложности, детализации и охвата проектируемой системы. Практические решения подразделяются на порождающие шаблоны, структурные и шаблоны поведения.

1. Структурные шаблоны (structural patterns) показывают, как объекты и классы объединяются для образования сложных структур.
2. Порождающие шаблоны (creational patterns) контролируют процесс создания и жизненный цикл объектов.
3. Шаблоны поведения (behavioral patterns) используются для организации, управления и объединения различных вариантов поведения объектов.

Каждый шаблон проектирования описывает задачи, с которыми программисту часто приходится сталкиваться. И затем описывает основу решения этой задачи таким образом, что позволяет воплотить это решение при разработке других программ, ни разу не повторившись.

Порождающие шаблоны проектирования отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов. Известны следующие шаблоны этой группы:

1. Factory Method или фабричный метод. Определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.
2. Abstract Factory или абстрактная фабрика. Позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.
3. Builder или строитель. Позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.
4. Prototype или прототип. Позволяет копировать объекты, не вдаваясь в подробности их реализации.
5. Singleton или одиночка. Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Фабрика – это общая концепция проектирования функций, методов и классов, когда какая-то одна часть программы отвечает за создание других частей программы.

В программировании «фабрика» означает:

* функцию или метод, создающие все объекты программы;
* класс, создающий пользователей системы;
* статический метод, оборачивающий конструктор класса;
* один из фабричных шаблонов проектирования: простая фабрика, фабричный метод или абстрактная фабрика.

Простая фабрика – это класс, в котором есть один метод с большим условным оператором, выбирающим создаваемый продукт. Этот метод вызывают с неким параметром, по которому определяется какой из продуктов нужно создать.

Фабричный метод (Factory method) – порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, Фабрика делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне. Также известен под названием виртуальный конструктор (Virtual Constructor).

Фабричный метод используется для создания общего интерфейса. Например, приложению требуется объект с определенным интерфейсом для выполнения задач. Реализация интерфейса определяется некоторым параметром.

Вместо использования сложной структуры из условий if/else if/else для определения реализации, приложение делегирует это решение отдельному компоненту, который создает конкретный объект. При таком подходе код приложения упрощается, становится более удобным для повторного использования и поддержки.

Фабричный метод позволяет классу делегировать создание подклассов. Используется, когда:

1. Классу заранее неизвестно, объекты каких подклассов ему нужно создавать.
2. Класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами.
3. Класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируетсялокализовать знание о том, какой класс принимает эти обязанности на себя.

Абстрактная фабрика – это порождающий шаблон проектирования, который решает проблему создания целых семейств связанных продуктов, без указания конкретных классов продуктов. Шаблон полезен, когда есть несколько классов, зависящих друг от друга.

Абстрактную фабрику можно определить по методам, возвращающим фабрику, которая, в свою очередь, используется для создания конкретных продуктов, возвращая их через абстрактные типы или интерфейсы.

В разных языках программирования существуют различные подходы в реализации решений абстрактной фабрики, однако большинство решений базируется на взаимодействии следующих классов участников:

1. Класс абстрактная фабрика. Объявляет интерфейс для операций, создающих абстрактные объекты-продукты.
2. Класс конкретная фабрика. Реализует операции абстрактной фабрики.
3. Класс абстрактный продукт. Объявляет интерфейс для объекта продукта.
4. Класс конкретный продукт. Реализует операции абстрактного продукта.

**Строитель** (Builder) – это порождающий шаблон проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

Строитель предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым строителями*.* При этом, промежуточный результат всегда остаётся защищён.

Шаблон предлагает разбить процесс конструирования объекта на отдельные шаги. Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации.

Прототип (Prototype) относится к классу порождающих шаблонов. Прототип позволяет копировать объекты, не вдаваясь в подробности их реализации. Он используется для задания вида создаваемых объектов на основе объекта прототипа, от которого происходит передача внутреннего состояния. Он сродни фабричному методу, позволяет избавиться от жесткой привязки к классам, задаёт виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путём копирования этого прототипа.

Этот шаблон проектирования используется, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты.

Одиночка – это порождающий паттерн проектирования, который позволяет создавать только один экземпляр объекта в приложении, который будет обрабатывать все обращения, запрещая создавать новый экземпляр, и предоставляет к нему глобальную точку доступа.

Одиночка имеет такие же преимущества и недостатки, что и глобальные переменные. Его невероятно удобно использовать, но он нарушает модульность кода.

Отсюда вытекает то, что нельзя использовать класс, зависящий от одиночки в другой программе. Для этого придётся эмулировать присутствие одиночки и там. Чаще всего эта проблема проявляется при написании юнит-тестов.

Структурные шаблоны проектирования отвечают за построение удобных в поддержке иерархий классов. Ниже будут приведены примеры шаблонов этой группы.

Adapter или Адаптер. Позволяет объектам с несовместимыми интерфейсами работать вместе. Типичным примером использования шаблона Адаптер можно назвать создание классов, приводящих к единому интерфейсу функции языка PHP обеспечивающие доступ к различным СУБД.

Bridge или Мост. Разделяет один или несколько классов на две отдельные иерархии – абстракцию и реализацию, позволяя изменять их независимо друг от друга. При частом изменении класса преимущества объектно-ориентированного подхода становятся очень полезными, позволяя делать изменения в программе, обладая минимальными сведениями о реализации программы. Шаблон мост является полезным там, где часто меняется не только сам класс, но и то, что он делает.

Composite или Компоновщик. Позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект. Шаблон «Компоновщик» предполагает, что при разработке «контейнерных» объектов, которые собирают и упорядочивают «объекты содержимого», вы упрощаете операции, если предоставляете контейнерам и объектам содержимого общий набор методов. И тем самым поддерживаете максимум возможных методов при том, что вызывающему неважно, переданы отдельный объект контента или целый контейнер.

Decorator или Декоратор. Позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Facade или Фасад. Предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

Flyweight или Легковес. Позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте. Flyweight используется для уменьшения затрат при работе с большим количеством мелких объектов. При проектировании приспособленца необходимо разделить его свойства на внешние и внутренние. Внутренние свойства всегда неизменны, тогда как внешние могут отличаться в зависимости от места и контекста применения и должны быть вынесены за пределы приспособленца. Flyweight дополняет шаблон Factory Method таким образом, что при обращении клиента к Factory Method для создания нового объекта ищет уже созданный объект с такими же параметрами, что и у требуемого, и возвращает его клиенту. Если такого объекта нет, то фабрика создаст новый.

Proxy или Заместитель. Позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

**Фасад** (Facade)– структурный шаблон проектирования, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы. экземпляр в разных местах программы, по факту не является таковым.

Поведенческие шаблоны проектирования решают задачи эффективного и безопасного взаимодействия между объектами программы. Ниже будут приведены примеры шаблонов этой группы.

**Chain of Responsibility** или Цепочка обязанностей. Позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

**Command** или Команда. Превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций. В данном контексте запрос – это любая задача, которая должна быть обработана несколькими объектами (или одним если есть только один обработчик). Например, сообщение в лог – это конкретная задача, т.е. запрос. Получатель в таком случае – это тот класс, который будет обрабатывать запрос (задачу). Команда является поведенческим шаблоном, в котором объект используется для инкапсуляции всей информации, необходимой для выполнения действия или вызова события в более позднее время. Эта информация включает в себя имя метода, объект, который является владельцем метода и значения параметров метода. Использование командных объектов упрощает построение общих компонентов, которые необходимо делегировать или выполнять вызовы методов в любое время без необходимости знать методы класса или параметров метода. Использование вызывающего объекта (invoker) позволяет ввести учёт выполненных команд без необходимости знать клиенту об этой модели учёта (такой учёт может пригодиться, например, для реализации отмены и повтора команд).

**Iterator** или Итератор. Даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

**Mediator** или Посредник. Позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

**Memento** или Хранитель. Позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

**Observer** или Наблюдатель. Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах. Объект, который сообщает наблюдателям о наступлении интересующих их событий, называется издателем. Издатель содержит хеш таблицу, в которой находятся ссылки на наблюдателей и типы событий, на которые наблюдатели подписаны. У издателя есть методы позволяющие наблюдателям подписываться и отписываться от событий.

Так как издатель содержит хеш-таблицу с типами событий и наблюдателями, то будет правильным шагом реализовать соответствующий тип данных для работы с такой таблицей. Нужен тип данных, хранящий структуру в виде объекта, ключами которого являются названия события, а значениями массивы ссылок на объекты наблюдателей. Еще тип данных должен иметь методы для добавления и удаления подписок из своей хеш структуры. Реализация подобной структуры данных избавит класс издателя от реализации логики, которая к нему напрямую не относится.

State или Состояние. Позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

Strategy или Стратегия. Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

Template Methodили Шаблонный метод. Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

Visitor или Посетитель. Позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

В сравнении с полностью самостоятельным проектированием, шаблоны обладают рядом преимуществ. Основная польза от использования шаблонов состоит в снижении сложности разработки за счёт готовых абстракций для решения целого класса проблем.

Шаблон даёт решению своё имя, что облегчает коммуникацию между разработчиками, позволяя ссылаться на известные шаблоны. Таким образом, за счёт шаблонов производится унификация деталей решений: модулей, элементов проекта, – снижается количество ошибок. Применение шаблонов концептуально сродни использованию готовых библиотек кода. Правильно сформулированный шаблон проектирования позволяет, отыскав удачное решение, пользоваться им снова и снова. Набор шаблонов помогает разработчику выбрать возможный, наиболее подходящий вариант проектирования.

С применением шаблонов могут быть связаны различные сложности.

# Заключение

Для написания курсовой работы было выделено три основных задачи:

1. Отрисовка персонажей и карты игрового мира.
2. Использование созданных текстур для написания программного кода.
3. Тестирование программного кода.

В разделе 1 подробно описано выполнение вышеперечисленных задач, что способствовало достижению цели проектирования, представлявшей разработку игры-платформера «Afterglow» под операционную систему Android на языке программирования Java с использованием фреймворка libGDX.

В процессе решения поставленных задач в графическом редакторе Photoshop были созданы наборы тайлов, которые были использованы для создания игрового 2D мира в редакторе карт Tiled; на объектно-ориентированном языке программирования Java был написан программный код, который использовал набор таких функций фреймворка libGDX как построение пользовательского интерфейса, работа с объектами и физикой реального мира, применение линейной алгебры для вычислений, использование 2D-объектов, визуализация спрайтов и создание анимации.

Показателями достижения поставленной цели являются: получение графической модели игрового уровня, реализация игрового цикла и тестирование логики игрового процесса на операционной системе Android.

Установлено, что в ходе данной курсовой работы была написана игра «Afterglow» с уровнями в стиле пиксель-арт, функционирующей логикой для совершения действий в игровом цикле и прохождения игры на операционной системе Android.

Данная работа является актуальной, так как она решает вопрос использования наиболее подходящих технологий для получения максимальной производительности программного кода, а также создаёт продукт для проведения досуга в сфере развлечений.

# Список использованных источников

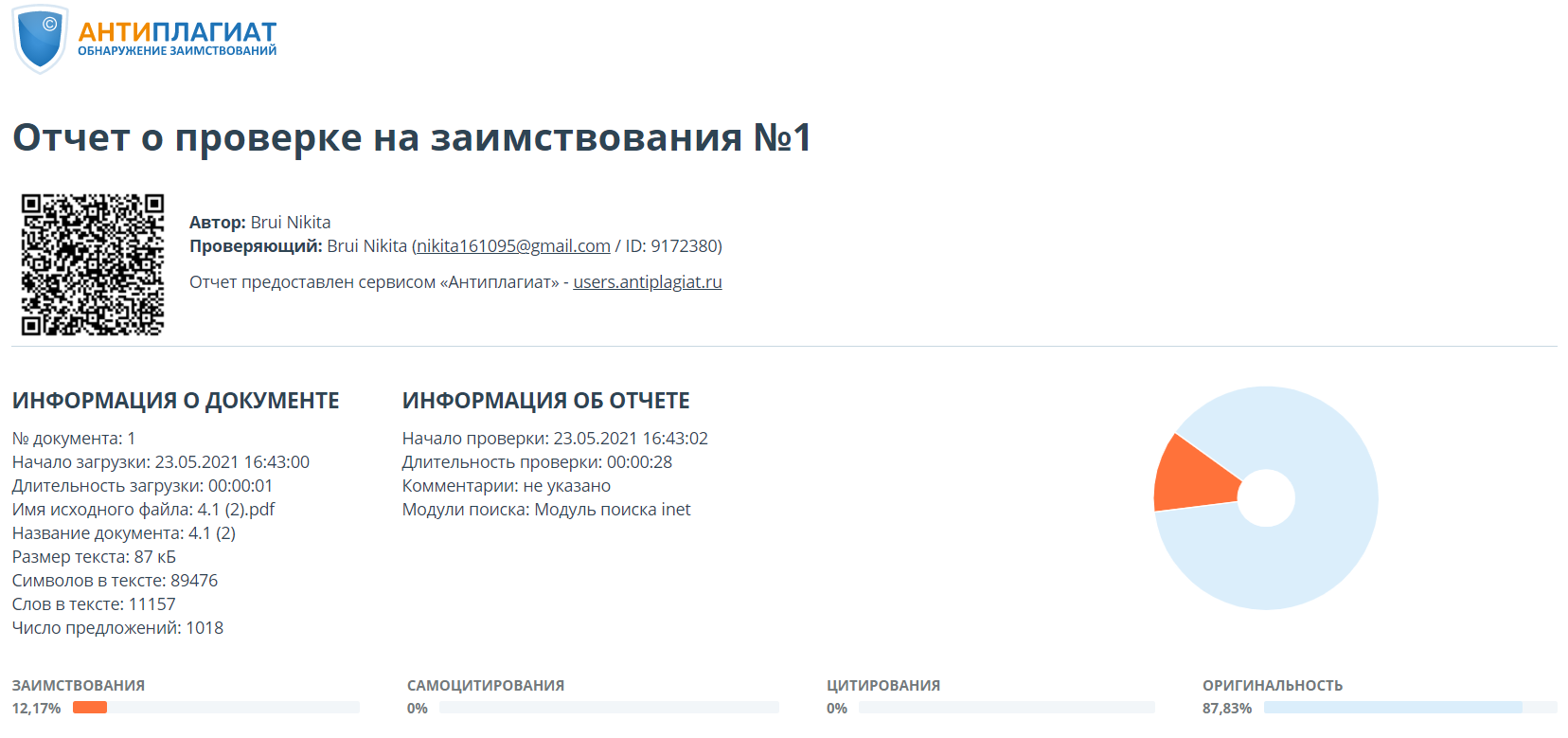
1. LibGDX Tutorial 9: Scene2D Part 1 [Электронный ресурс]. – Режим доступа: https://gamefromscratch.com/libgdx-tutorial-9-scene2d-part-1/
2. [Game Dev](https://xgm.guru/p/gamedev): [Изучаем Scene2d и делаем полноценное меню](https://xgm.guru/p/gamedev/132196) [Электронный ресурс]. – Режим доступа: https://xgm.guru/p/gamedev/132196
3. Перевод туториалов по libGDX часть 1 (настройка проекта в Eclipse) [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/142976/>
4. Перевод туториалов по libGDX часть 2 (рисование изображений) [Электронный ресурс]. – Режим доступа: https://habr.com/en/post/143405/
5. 2D scene graph containing hierarchies of [actors](https://libgdx.badlogicgames.com/ci/nightlies/docs/api/com/badlogic/gdx/scenes/scene2d/Actor.html) [Электронный ресурс]. – Режим доступа: https://libgdx.badlogicgames.com/ci/nightlies/docs/api/com/ badlogic/gdx/scenes/scene2d/Stage.html
6. Перевод туториалов по libGDX часть 3 (пакет scene2d) [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/143479/
7. LibGDX: Scene2d – Actors Part 1 [Электронный ресурс]. – Режим доступа: https://www.codinginsights.blog/libgdx-scene2d-actors-part-1/
8. libGDX – фреймворк для разработки игр [Электронный ресурс]. – Режим доступа: http://www.libgdx.ru/2013/08/introduction.html
9. Архитектура игры [Электронный ресурс]. – Режим доступа: https://suvitruf.ru/2012/10/20/2307/
10. libGDX is a cross-platform Java game development framework based on OpenGL (ES) that works on Windows, Linux, macOS, Android, your browser and iOS. [Электронный ресурс]. – Режим доступа: https://libgdx.com/

# Приложение А (обязательное) Листинг программного кода

package com.aquamara.game.Screens;  
  
import com.aquamara.game.Afterglow;  
import com.aquamara.game.Scenes.Hud;  
import com.aquamara.game.Sprites.Enemies.Enemy;  
import com.aquamara.game.Sprites.Enemies.Rat;  
import com.aquamara.game.Sprites.Player;  
import com.aquamara.game.Tools.B2WorldCreator;  
import com.aquamara.game.Tools.WorldContactListener;  
import com.badlogic.gdx.Gdx;  
import com.badlogic.gdx.Input;  
import com.badlogic.gdx.Screen;  
import com.badlogic.gdx.graphics.GL20;  
import com.badlogic.gdx.graphics.OrthographicCamera;  
import com.badlogic.gdx.graphics.g2d.TextureAtlas;  
import com.badlogic.gdx.maps.tiled.TiledMap;  
import com.badlogic.gdx.maps.tiled.TmxMapLoader;  
import com.badlogic.gdx.maps.tiled.renderers.OrthogonalTiledMapRenderer;  
import com.badlogic.gdx.math.Vector2;  
import com.badlogic.gdx.physics.box2d.Box2DDebugRenderer;  
import com.badlogic.gdx.physics.box2d.World;  
import com.badlogic.gdx.utils.viewport.FitViewport;  
import com.badlogic.gdx.utils.viewport.Viewport;  
  
public class PlayScreen implements Screen {  
  
 private Afterglow game;  
 private TextureAtlas atlas;  
  
 private OrthographicCamera gameCam;  
 private Viewport gamePort;  
 private Hud hud;  
  
 private TmxMapLoader mapLoader;  
 private TiledMap map;  
 private OrthogonalTiledMapRenderer renderer;  
  
 private World world;  
 private Box2DDebugRenderer b2dr;  
 private B2WorldCreator creator;  
  
 private Player player;  
 //private Rat mush;  
  
  
 public PlayScreen (Afterglow game) {  
 this.game = game;  
 gameCam = new OrthographicCamera();  
 gamePort = new FitViewport(Afterglow.V\_WIDTH / Afterglow.PPM, Afterglow.V\_HEIGHT / Afterglow.PPM, gameCam);  
 hud = new Hud(game.batch);  
  
 atlas = new TextureAtlas("Player\_and\_Enemies.pack");  
  
 mapLoader = new TmxMapLoader();  
 map = mapLoader.load("level1.tmx");  
 renderer = new OrthogonalTiledMapRenderer(map, 1 / Afterglow.PPM);  
 gameCam.position.set(gamePort.getWorldWidth() / 2, gamePort.getWorldHeight() / 2, 0);  
 //gameCam.position.set(gamePort.getWorldWidth(), gamePort.getWorldHeight() / 2, 0);  
  
 world = new World(new Vector2(0,-10), true);  
 b2dr = new Box2DDebugRenderer();  
 b2dr.setDrawBodies(false);  
  
 creator = new B2WorldCreator(this);  
 player = new Player(this);  
  
 world.setContactListener(new WorldContactListener());  
 }  
  
 public TextureAtlas getAtlas() {  
 return atlas;  
 }  
  
 @Override  
 public void show() {  
  
 }  
  
 public void handleInput(float dt) {  
 if(player.currentState != Player.State.DEAD) {  
 if (Gdx.input.isKeyJustPressed(Input.Keys.UP))  
 player.b2body.applyLinearImpulse(new Vector2(0, 4f), player.b2body.getWorldCenter(), true);  
 if(Gdx.input.isKeyPressed(Input.Keys.RIGHT) && player.b2body.getLinearVelocity().x <= 2)  
 player.b2body.applyLinearImpulse(new Vector2(0.1f, 0), player.b2body.getWorldCenter(), true);  
 if(Gdx.input.isKeyPressed(Input.Keys.LEFT) && player.b2body.getLinearVelocity().x >= -2)  
 player.b2body.applyLinearImpulse(new Vector2(-0.1f, 0), player.b2body.getWorldCenter(), true);  
 }  
 }  
  
 public void checkGameOver() {  
 if(player.currentState == Player.State.DEAD && player.getStateTimer() > 3) {  
 dispose();  
 game.setScreen(new GameOverScreen(game));  
 }  
 }  
 public void checkWn() {  
 if(player.currentState == Player.State.WINNING && player.getStateTimer() > 1) {  
 dispose();  
 game.setScreen(new LevelScreen(game));  
 }  
 }  
  
 public void update (float dt) {  
  
 handleInput(dt);  
 world.step(1/60f, 6, 2);  
 player.update(dt);  
  
 for (Enemy enemy : creator.getMushrooms()) {  
 enemy.update(dt);  
 if(enemy.getX() < player.getX() + 224 / Afterglow.PPM) // (12 + 2) \* 16 блока, 12 - видны на экране справа, 2 в запасе  
 enemy.b2body.setActive(true);  
 }  
  
 for (Enemy enemy : creator.getRat()) {  
 enemy.update(dt);  
 if(enemy.getX() < player.getX() + 224 / Afterglow.PPM) // (12 + 2) \* 16 блока, 12 - видны на экране справа, 2 в запасе  
 enemy.b2body.setActive(true);  
 }  
  
 hud.update(dt);  
  
 if(player.currentState != Player.State.DEAD)  
 gameCam.position.x = player.b2body.getPosition().x;  
  
 gameCam.update();  
 renderer.setView(gameCam);  
 }  
  
 @Override  
 public void render(float delta) {  
 update(delta);  
  
 Gdx.gl.glClearColor(0, 0, 0 , 1);  
 Gdx.gl.glClear(GL20.GL\_COLOR\_BUFFER\_BIT);  
  
 renderer.render();  
 b2dr.render(world, gameCam.combined);  
  
 game.batch.setProjectionMatrix(gameCam.combined);  
 game.batch.begin();  
 player.draw(game.batch);  
 //mush.draw(game.batch);  
 for (Enemy enemy : creator.getMushrooms())  
 enemy.draw(game.batch);  
 for (Enemy enemy : creator.getRat())  
 enemy.draw(game.batch);  
 game.batch.end();  
  
 game.batch.setProjectionMatrix(hud.stage.getCamera().combined);  
 hud.stage.draw();  
  
 if(gameOver()) {  
 game.setScreen(new GameOverScreen(game));  
 dispose();  
 }  
 if(gameWin()) {  
 game.setScreen(new LevelScreen(game));  
 dispose();  
 }  
 }  
  
 public boolean gameOver() {  
 if(player.currentState == Player.State.DEAD && player.getStateTimer() > 3)  
 return true;  
 return false;  
 }  
 public boolean gameWin() {  
 if(player.currentState == Player.State.WINNING && player.getStateTimer() > 1)  
 return true;  
 return false;  
 }  
  
 @Override  
 public void resize(int width, int height) {  
 gamePort.update(width, height);  
 }  
  
 public TiledMap getMap() {  
 return map;  
 }  
  
 public World getWorld() {  
 return world;  
 }  
  
 @Override  
 public void pause() {  
 }  
  
 @Override  
 public void resume() {  
 }  
  
 @Override  
 public void hide() {  
 }  
  
 @Override  
 public void dispose() {  
 hud.dispose();  
 map.dispose();  
 renderer.dispose();  
 world.dispose();  
 b2dr.dispose();  
  
 }  
}

package com.aquamara.game.Screens;  
  
import com.aquamara.game.Afterglow;  
import com.aquamara.game.Scenes.Hud;  
import com.aquamara.game.Sprites.Enemies.Enemy;  
import com.aquamara.game.Sprites.Enemies.Rat;  
import com.aquamara.game.Sprites.Player;  
import com.aquamara.game.Tools.B2WorldCreator;  
import com.aquamara.game.Tools.WorldContactListener;  
import com.badlogic.gdx.Gdx;  
import com.badlogic.gdx.Input;  
import com.badlogic.gdx.Screen;  
import com.badlogic.gdx.graphics.GL20;  
import com.badlogic.gdx.graphics.OrthographicCamera;  
import com.badlogic.gdx.graphics.g2d.TextureAtlas;  
import com.badlogic.gdx.maps.tiled.TiledMap;  
import com.badlogic.gdx.maps.tiled.TmxMapLoader;  
import com.badlogic.gdx.maps.tiled.renderers.OrthogonalTiledMapRenderer;  
import com.badlogic.gdx.math.Vector2;  
import com.badlogic.gdx.physics.box2d.Box2DDebugRenderer;  
import com.badlogic.gdx.physics.box2d.World;  
import com.badlogic.gdx.utils.viewport.FitViewport;  
import com.badlogic.gdx.utils.viewport.Viewport;  
  
public class PlayScreen implements Screen {  
  
 private Afterglow game;  
 private TextureAtlas atlas;  
  
 private OrthographicCamera gameCam;  
 private Viewport gamePort;  
 private Hud hud;  
  
 private TmxMapLoader mapLoader;  
 private TiledMap map;  
 private OrthogonalTiledMapRenderer renderer;  
  
 private World world;  
 private Box2DDebugRenderer b2dr;  
 private B2WorldCreator creator;  
  
 private Player player;  
 //private Rat mush;  
  
  
 public PlayScreen (Afterglow game) {  
 this.game = game;  
 gameCam = new OrthographicCamera();  
 gamePort = new FitViewport(Afterglow.V\_WIDTH / Afterglow.PPM, Afterglow.V\_HEIGHT / Afterglow.PPM, gameCam);  
 hud = new Hud(game.batch);  
  
 atlas = new TextureAtlas("Player\_and\_Enemies.pack");  
  
 mapLoader = new TmxMapLoader();  
 map = mapLoader.load("level1.tmx");  
 renderer = new OrthogonalTiledMapRenderer(map, 1 / Afterglow.PPM);  
 gameCam.position.set(gamePort.getWorldWidth() / 2, gamePort.getWorldHeight() / 2, 0);  
 //gameCam.position.set(gamePort.getWorldWidth(), gamePort.getWorldHeight() / 2, 0);  
  
 world = new World(new Vector2(0,-10), true);  
 b2dr = new Box2DDebugRenderer();  
 b2dr.setDrawBodies(false);  
  
 creator = new B2WorldCreator(this);  
  
  
 player = new Player(this);  
  
 world.setContactListener(new WorldContactListener());  
 }  
  
 public TextureAtlas getAtlas() {  
 return atlas;  
 }  
  
 @Override  
 public void show() {  
  
 }  
  
 public void handleInput(float dt) {  
 if(player.currentState != Player.State.DEAD) {  
 if (Gdx.input.isKeyJustPressed(Input.Keys.UP))  
 player.b2body.applyLinearImpulse(new Vector2(0, 4f), player.b2body.getWorldCenter(), true);  
 if(Gdx.input.isKeyPressed(Input.Keys.RIGHT) && player.b2body.getLinearVelocity().x <= 2)  
 player.b2body.applyLinearImpulse(new Vector2(0.1f, 0), player.b2body.getWorldCenter(), true);  
 if(Gdx.input.isKeyPressed(Input.Keys.LEFT) && player.b2body.getLinearVelocity().x >= -2)  
 player.b2body.applyLinearImpulse(new Vector2(-0.1f, 0), player.b2body.getWorldCenter(), true);  
 }  
 }  
  
 public void checkGameOver() {  
 if(player.currentState == Player.State.DEAD && player.getStateTimer() > 3) {  
 dispose();  
 game.setScreen(new GameOverScreen(game));  
 }  
 }  
 public void checkWn() {  
 if(player.currentState == Player.State.WINNING && player.getStateTimer() > 1) {  
 dispose();  
 game.setScreen(new LevelScreen(game));  
 }  
 }  
  
 public void update (float dt) {  
  
 handleInput(dt);  
  
 world.step(1/60f, 6, 2);  
  
 player.update(dt);  
  
 for (Enemy enemy : creator.getMushrooms()) {  
 enemy.update(dt);  
 if(enemy.getX() < player.getX() + 224 / Afterglow.PPM) // (12 + 2) \* 16 блока, 12 - видны на экране справа, 2 в запасе  
 enemy.b2body.setActive(true);  
 }  
  
 for (Enemy enemy : creator.getRat()) {  
 enemy.update(dt);  
 if(enemy.getX() < player.getX() + 224 / Afterglow.PPM) // (12 + 2) \* 16 блока, 12 - видны на экране справа, 2 в запасе  
 enemy.b2body.setActive(true);  
 }  
  
 hud.update(dt);  
  
 if(player.currentState != Player.State.DEAD)  
 gameCam.position.x = player.b2body.getPosition().x;  
  
 gameCam.update();  
 renderer.setView(gameCam);  
 }  
  
 @Override  
 public void render(float delta) {  
 update(delta);  
  
 Gdx.gl.glClearColor(0, 0, 0 , 1);  
 Gdx.gl.glClear(GL20.GL\_COLOR\_BUFFER\_BIT);  
  
 renderer.render();  
 b2dr.render(world, gameCam.combined);  
  
  
 game.batch.setProjectionMatrix(gameCam.combined);  
 game.batch.begin();  
 player.draw(game.batch);  
 //mush.draw(game.batch);  
 for (Enemy enemy : creator.getMushrooms())  
 enemy.draw(game.batch);  
 for (Enemy enemy : creator.getRat())  
 enemy.draw(game.batch);  
 game.batch.end();  
  
 game.batch.setProjectionMatrix(hud.stage.getCamera().combined);  
 hud.stage.draw();  
  
 if(gameOver()) {  
 game.setScreen(new GameOverScreen(game));  
 dispose();  
 }  
 if(gameWin()) {  
 game.setScreen(new LevelScreen(game));  
 dispose();  
 }  
  
 }  
  
 public boolean gameOver() {  
 if(player.currentState == Player.State.DEAD && player.getStateTimer() > 3)  
 return true;  
 return false;  
 }  
 public boolean gameWin() {  
 if(player.currentState == Player.State.WINNING && player.getStateTimer() > 1)  
 return true;  
 return false;  
 }  
  
 @Override  
 public void resize(int width, int height) {  
 gamePort.update(width, height);  
 }  
  
 public TiledMap getMap() {  
 return map;  
 }  
  
 public World getWorld() {  
 return world;  
 }  
  
 @Override  
 public void pause() {  
 }  
  
 @Override  
 public void resume() {  
 }  
  
 @Override  
 public void hide() {  
 }  
  
 @Override  
 public void dispose() {  
 hud.dispose();  
 map.dispose();  
 renderer.dispose();  
 world.dispose();  
 b2dr.dispose();  
  
 }  
}

# Приложение Б (обязательное) Отчет по анализу заимствования



**Приложение В**  
(обязательное)  
Ведомость курсового проекта