МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ

КАЛИНИНГРАДСКОЙ ОБЛАСТИ

ПРОФЕССИОНАЛЬНАЯ ОБРАЗОВАТЕЛЬНАЯ ОРГАНИЗАЦИЯ

«КОЛЛЕДЖ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И СТРОИТЕЛЬСТВА»

(ГБУ КО ПОО «КИТиС»)

|  |  |  |
| --- | --- | --- |
| Курсовой проект  допущен к защите  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (дата)  Зам.директора по УМР  \_\_\_\_\_\_\_\_\_\_\_\_\_Павленко Г.Я.  (подпись) |  | Курсовой проект  защищен с оценкой  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (оценка)  Руководитель работы  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись, дата) |

Пояснительная записка к курсовому проекту

по дисциплине: МДК 01.01 Разработка программных модулей

Тема Разработка и программная реализация модели персонажа игры

Специальность 09.02.07 «Информационные системы и программирование»

|  |  |
| --- | --- |
| Выполнил: | студент 3 курса,  группы ИСп 21-2К  Ильин Тимур Евгеньевич  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) |
| Руководитель: | Большакова-Стрекалова Анна Викторовна, преподаватель ГБУ КО ПОО КИТиС  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) |

Калининград

2024

# Аннотация

Данный курсовой проект посвящен разработке Java-приложения под названием "Dragon Tale". Это 2D платформер, который предлагает захватывающее приключение пользователям, позволяя управлять персонажем, сражаться с врагами и исследовать локацию. Проект разработан в соответствии с техническим заданием, включающим в себя требования к платформам, графике, управлению и производительности. Создание этого приложения позволило расширить свои навыки программирования и погрузиться в процесс разработки игр, обеспечивая пользователям интересный и увлекательный игровой опыт. Исходный код проекта и другие материалы доступны по ссылку [master/dragon-full-8 at main · dXdyZ/master (github.com)](https://github.com/dXdyZ/master/tree/main/dragon-full-8).

Оглавление

[Аннотация 2](#_Toc165524116)

[Введение 4](#_Toc165524117)

[1 Описание предметной области 4](#_Toc165524118)

[1.1 Аналогия разрабатываемого приложения 4](#_Toc165524119)

[1.2 Техническое задание 7](#_Toc165524120)

[Описание разработки приложения 8](#_Toc165524121)

[2.1 Обоснование сред разработки 8](#_Toc165524122)

[2.2 Разработка интерфейса 9](#_Toc165524123)

[2.3 Разработка логики работы приложения. 10](#_Toc165524124)

[2.4 Описание переменных, компонентов, классов и подпрограмм: 11](#_Toc165524125)

[Тестирование и установка приложения 42](#_Toc165524126)

[3.1 Пользовательское тестирование: 43](#_Toc165524127)

[3.2 Установка приложения: 43](#_Toc165524128)

[Заключение 45](#_Toc165524129)

[Список используемой литературы 46](#_Toc165524130)

[Приложение A: Техническое задание 47](#_Toc165524131)

[Приложение B. Руководство пользователя 51](#_Toc165524132)

[Приложение С. Диаграмма классов 53](#_Toc165524133)

[Приложение D. Код программы 54](#_Toc165524134)

# Введение

## 1 Описание предметной области

Игра "Dragon Tale" представляет собой захватывающее пиксельное приключение, которое тщательно создано с учетом интересов широкой аудитории, включая как детей, так и взрослых.. Игроку предстоит управлять милым зеленым дракончиком, который отправляется в увлекательное путешествие по разнообразным мирам, наполненным опасностями и волшебством. Атмосфера игры создана с особым вниманием к деталям, чтобы погрузить игрока в захватывающий мир фэнтези. От красочных локаций до детализированных персонажей и впечатляющих спецэффектов, "Dragon Tale" стремится предложить игрокам незабываемый опыт, который они захотят пережить снова и снова.

### 1.1 Аналогия разрабатываемого приложения

Shovel Knight - это пиксельный платформер, выпущенный в 2014 году студией Yacht Club Games. Игрок управляет рыцарем по имени Шаблон, который использует лопату в качестве оружия и инструмента.



Рисунок 1 – Логотип Shovel Knight

Теперь предлагаю рассмотреть функционал игры:

1. Геймплей:

Игрок управляет рыцарем по имени Шаблон, который использует лопату в качестве оружия и инструмента.

Цель игры - пройти через разнообразные уровни, сражаться с врагами и решать загадки, чтобы спасти потерянную возлюбленную и победить зловещих рыцарей ордена Призраков.

Геймплей включает в себя элементы платформера, экшена и головоломок, создавая разнообразные ситуации и вызовы для игрока.



Рисунок 2 – Геймплей игры Shovel Knight

2. Управление:

Управление осуществляется с помощью контроллера или клавиатуры, обеспечивая комфортное и интуитивно понятное управление персонажем.

3. Оружие и инструменты:

Основное оружие Шаблона - лопата, которая может использоваться для атаки врагов, копания земли и преодоления препятствий.

В течение игры игрок может получать различные улучшения и новые инструменты, расширяя тактические возможности в бою и обогащая геймплей.



Рисунок 3 – Оружие персонажа

4. Уровни и миры:

Игра состоит из разнообразных уровней и миров, каждый из которых имеет уникальный дизайн, врагов и головоломки.



Рисунок 4 – Мир игры Shovel Knight

5. Враги и боссы:

На протяжении игры игрок сталкивается с различными врагами, каждый из которых имеет свои уникальные атаки и тактики.



Рисунок 5 – Враг из игры Shovel Knight

6. Графика и аудио:

Графический стиль игры выполнен в пиксельном формате, отражая дух классических игр 8-битной эпохи.

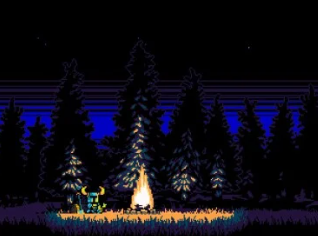


Рисунок 6 – Графика игры Shovel Knight

Как работает Dragon Tale:

Dragon Tale – предлагает веселый геймплей, в котором вы будете проходит веселый путь за зеленного дракончика, исследовать новый и уникальный мир. Вот основные принцы работы:

1. Учение через контекст:

Вместо того, чтобы просто объяснять правила игры или предоставлять инструкции, "Dragon Tale" использует контекстуальные события, анимации и визуальные элементы, чтобы помочь игрокам понять игровые механики и цели.

Обучение игрока:

При запуске уровня игроку предоставляются инструкции на заднем фоне экрана с изображением соответствующих клавиш клавиатуры. Это позволяет игроку понять, как управлять дракончиком, прежде чем он столкнется с первыми препятствиями или врагами.

Появление первого моба:

Когда игрок приближается к первому мобу, на экране появляется инструкция "R", указывая, что игроку следует использовать клавишу "R" для атаки ближнего боя, чтобы справиться с мобом.

Первый прыжок и парение:

Когда игроку впервые нужно прыгнуть через пропасть или парить над опасностью, на экране появляются инструкции "W+E", указывая, что нужно одновременно нажать клавиши "W" для прыжка и "E" для парения.

Первый раз стрельба:

Когда игрок впервые сталкивается с врагом на расстоянии, на экране появляется инструкция "F", указывая, что игроку следует использовать клавишу "F" для стрельбы огненными шарами и нанесения урона врагу.

### 1.2 Техническое задание

Приложение "Dragon Tale" было создано в соответствии с техническим заданием (ТЗ), представленным в рамках курсовой работы. Разработка приложения началась с анализа требований ТЗ, включая определение основных функциональных возможностей, платформы, на которой оно будет работать, и требований к графике, звуку и управлению. Затем была разработана архитектура приложения, определены основные классы и их взаимосвязи. Далее последовал этап создания игрового движка, объектов и интерфейса, согласно требованиям ТЗ. В процессе разработки осуществлялся контроль соответствия результатов промежуточных этапов поставленным задачам и корректировка проекта при необходимости. Полную версию См. в приложение A.

# Описание разработки приложения

## 2.1 Обоснование сред разработки

1. Intellij IDEA – это официальная IDEA для разработки на языке Java, разработанная компанией JetBrains. Intellij IDEA предоставляет широкий спектр инструментов и возможностей для комфортной работы.

Плюсы:

* Официальная среда разработки для работы с языком Java.
* Широкий выбор инструментов и функций.
* Активное сообщество разработчиков.

Минусы:

* Не столь легковесная и быстрая по сравнению с некоторыми другими IDE.
* Может потребовать значительных ресурсов системы, особенно для крупных проектов.
* Иногда могут возникать проблемы со стабильностью и производительностью.

2. Eclips – был один их наиболее популярных инструментов для разработки до появление Intellij IDEA. Но после перехода на Intellij IDEA он потерял в своей популярности.

Плюсы:

* Имеет долгую историю использования для разработки на языке Java.
* Поддерживает различные языки.
* Гибкая конфигурация и настройка.
* Большое кол-во плагинов и расширение, позволяющих расширить функциональность IDEA.

Минусы:

* Устаревшая и менее активно поддерживаемая IDEA.
* Не всегда стабильна и производительная работа с большими проектами.
* Интерфейс и опыт использования могут менее интуитивны.

Java:

Обоснование:

Java - это высокоуровневый, объектно-ориентированный язык программирования, который широко используется для разработки различных типов приложений.

Надежность:

* Java известна своей высокой стабильностью и надежностью.

Широкое применение:

* Java широко применяется в различных областях разработки программного обеспечения, включая корпоративные приложения, мобильные приложения, и т.д.

## 2.2 Разработка интерфейса

Интерфейс разработан с учет пользовательских потребностей, обеспечивая удобство использования и интуитивно понятное взаимодействие с приложение. Рассмотрим основные аспекты интерфейса:

1. Главное меню:

* Используется метод keyPressed(int k) для переключения между кнопками меню.
* Копки главного меню:
  + “Start” – начать игру.
  + “Help” – помощь.
  + “Quit” – выход на главный экран.
* Анимированные кнопки.
* Анимированный задний фон.

1. Экран игры:

* Статичный задний фон.
* Уровень жизни.
* Кол-во зарядов.
* Отображение обучающих иконок кнопок.
* Отображение платформы.

1. Анимации:

* Использование анимаций при наведение на кнопку на главное экране для более приятного визуально опыта и ориентировки в главном меню.
* Анимация заднего фона на главном экране.
* Анимация передвижения персонажа игры.
* Анимация прыжка персонажа игры.
* Анимация полета персонажа игры.
* Анимация атаки персонажа игры.
* Анимация стрельбы персонажа игры.
* Анимация смерти враждебного существа.
* Анимация получения урона.
* Анимация летящего снаряда.

## 2.3 Разработка логики работы приложения.

Процесс работы приложения основан на взаимодействии различных компонентов, включая работу с файлами, работу с клавиатурой.

1. Вход в приложение:

* Пользователь входит в приложение и при выборе пункта “Start” переходит на главный экран игры.

1. Главный экран игры:

* Отображает игровой мир, персонажа игры, задний фон, иконки обучений, статистку персонажа и площадку, по которой он движется.
* Пользователь может взаимодействовать с персонажам игры а так же с существами на карте.

## 2.4 Описание переменных, компонентов, классов и подпрограмм:

Диаграмму классов См. приложение С.

Класс Game:

Переменные:

* window:
* Тип: JFrame
* Описание: Окно приложения, созданное с названием "Dragon Tale".

Методы:

* main(String[] args):
* Описание: Точка входа в приложение. Создает новое окно JFrame, устанавливает его содержимое на экземпляр GamePanel.

Объекты:

* GamePanel:
* Тип: GamePanel
* Описание: Экземпляр класса GamePanel, который используется для содержимого окна JFrame.

Класс GamePanel:

Константы:

* WIDTH:
  + Тип: int
  + Значение: 320
  + Описание: Ширина окна игры в пикселях.
* HEIGHT:
  + Тип: int
  + Значение: 240
  + Описание: Высота окна игры в пикселях.
* SCALE:
  + Тип: int
  + Значение: 2
  + Описание: Масштаб окна игры.

Переменные:

* thread:
  + Тип: Thread
  + Описание: Поток игры.
* running:
  + Тип: boolean
  + Описание: Флаг, указывающий на состояние выполнения игры.
* FPS:
  + Тип: int
  + Значение: 60
  + Описание: Желаемое количество кадров в секунду.
* targetTime:
  + Тип: long
  + Значение: 1000 / FPS
  + Описание: Время на каждый кадр.
* image:
  + Тип: BufferedImage
  + Описание: Изображение, на котором рисуется игра.
* g:
  + Тип: Graphics2D
  + Описание: Графический контекст для рисования на изображении.
* gsm:
  + Тип: GameStateManager
  + Описание: Менеджер состояний игры.

Методы:

* GamePanel():
  + Описание: Конструктор класса GamePanel.
* addNotify():
  + Описание: Метод для запуска потока игры при добавлении панели на экран.
* init():
  + Описание: Метод инициализации игры.
* run():
  + Описание: Основной игровой цикл.
* update():
  + Описание: Метод для обновления состояния игры.
* draw():
  + Описание: Метод для отрисовки игры на изображении.
* drawToScreen():
  + Описание: Метод для отображения изображения на экране.

Методы обработки событий клавиатуры:

* keyTyped(KeyEvent key):
  + Описание: Обработчик события нажатия клавиши.
* keyPressed(KeyEvent key):
  + Описание: Обработчик события нажатия клавиши. Передает нажатие клавиши в менеджер состояний игры.
* keyReleased(KeyEvent key):
  + Описание: Обработчик события отпускания клавиши.

Класс Background:

Переменные:

* image:
  + Тип: BufferedImage
  + Описание: Изображение фона.
* x:
  + Тип: double
  + Описание: Координата x фона.
* y:
  + Тип: double
  + Описание: Координата y фона.
* dx:
  + Тип: double
  + Описание: Вектор движения по оси x.
* dy:
  + Тип: double
  + Описание: Вектор движения по оси y.
* moveScale:
  + Тип: double
  + Описание: Масштабирование скорости движения.

Методы:

* Background(String s, double ms):
  + Описание: Конструктор класса Background.
* setPosition(double x, double y):
  + Описание: Метод установки позиции фона.
* setVector(double dx, double dy):
  + Описание: Метод установки вектора движения фона.
* update():
  + Описание: Метод обновления состояния фона.
* draw(Graphics2D g):
  + Описание: Метод отрисовки фона.

Класс Tile:

Переменные:

* image:
  + Тип: BufferedImage
  + Описание: Изображение тайла.
* type:
  + Тип: int
  + Описание: Тип тайла.

Методы:

* Tile(BufferedImage image, int type):
  + Описание: Конструктор класса Tile.
* getImage():
  + Тип возвращаемого значения: BufferedImage
  + Описание: Метод получения изображения тайла.
* getType():
  + Тип возвращаемого значения: int
  + Описание: Метод получения типа тайла.

Класс TileMap:

Переменные:

* x:
  + Тип: double
  + Описание: Позиция по оси X.
* y:
  + Тип: double
  + Описание: Позиция по оси Y.
* xmin:
  + Тип: int
  + Описание: Минимальное значение координаты x.
* ymin:
  + Тип: int
  + Описание: Минимальное значение координаты y.
* xmax:
  + Тип: int
  + Описание: Максимальное значение координаты x.
* ymax:
  + Тип: int
  + Описание: Максимальное значение координаты y.
* tween:
  + Тип: double
  + Описание: Коэффициент для плавного перемещения.
* map:
  + Тип: int[][]
  + Описание: Массив, представляющий карту.
* tileSize:
  + Тип: int
  + Описание: Размер тайла.
* numRows:
  + Тип: int
  + Описание: Количество строк на карте.
* numCols:
  + Тип: int
  + Описание: Количество столбцов на карте.
* width:
  + Тип: int
  + Описание: Ширина карты.
* height:
  + Тип: int
  + Описание: Высота карты.
* tileset:
  + Тип: BufferedImage
  + Описание: Изображение тайлсета.
* numTilesAcross:
  + Тип: int
  + Описание: Количество тайлов в тайлсете по горизонтали.
* tiles:
  + Тип: Tile[][]
  + Описание: Двумерный массив тайлов.
* rowOffset:
  + Тип: int
  + Описание: Смещение строк для отрисовки.
* colOffset:
  + Тип: int
  + Описание: Смещение столбцов для отрисовки.
* numRowsToDraw:
  + Тип: int
  + Описание: Количество строк для отрисовки.
* numColsToDraw:
  + Тип: int
  + Описание: Количество столбцов для отрисовки.

Методы:

* TileMap(int tileSize):
  + Описание: Конструктор класса.
* loadTiles(String s):
  + Описание: Загружает изображения тайлов из файла.
* loadMap(String s):
  + Описание: Загружает карту из файла.
* getTileSize():
  + Тип возвращаемого значения: int
  + Описание: Возвращает размер тайла.
* getx():
  + Тип возвращаемого значения: double
  + Описание: Возвращает текущую позицию по оси X.
* gety():
  + Тип возвращаемого значения: double
  + Описание: Возвращает текущую позицию по оси Y.
* getWidth():
  + Тип возвращаемого значения: int
  + Описание: Возвращает ширину карты.
* getHeight():
  + Тип возвращаемого значения: int
  + Описание: Возвращает высоту карты.
* getType(int row, int col):
  + Тип возвращаемого значения: int
  + Описание: Возвращает тип тайла по указанным координатам.
* setTween(double d):
  + Описание: Устанавливает коэффициент плавного перемещения.
* setPosition(double x, double y):
  + Описание: Устанавливает позицию карты.
* fixBounds():
  + Описание: Корректирует границы отображения карты.
* draw(Graphics2D g):
  + Описание: Отрисовывает карту.

Класс GameState:

Переменные:

* gsm:
  + Тип: GameStateManager
  + Описание: Менеджер состояний.

Методы:

* init():
  + Описание: Абстрактный метод для инициализации состояния.
* update():
  + Описание: Абстрактный метод для обновления состояния.
* draw(java.awt.Graphics2D g):
* Параметры:
* g: java.awt.Graphics2D
  + Описание: Абстрактный метод для отрисовки состояния.
* keyPressed(int k):
* Параметры:
* k: int
  + Описание: Абстрактный метод для обработки нажатия клавиши.
* keyReleased(int k):
* Параметры:
* k: int
  + Описание: Абстрактный метод для обработки отпускания клавиши.

Класс GameStateManager:

Переменные:

* gameStates:
  + Тип: массив GameState
  + Описание: Массив состояний игры.
* currentState:
  + Тип: int
  + Описание: Текущее состояние игры.
* NUMGAMESTATES:
  + Тип: int (константа)
  + Описание: Количество состояний игры.
* MENUSTATE:
  + Тип: int (константа)
  + Описание: Индекс состояния меню.
* LEVEL1STATE:
  + Тип: int (константа)
  + Описание: Индекс состояния уровня.

Методы:

* GameStateManager():
  + Описание: Конструктор класса, инициализирует массив состояний и устанавливает начальное состояние.
* loadState(int state):
* Параметры:
* state: int
  + Описание: Загружает состояние игры в зависимости от переданного индекса.
* unloadState(int state):
* Параметры:
* state: int
  + Описание: Выгружает состояние игры, освобождая ресурсы.
* setState(int state):
* Параметры:
* state: int
  + Описание: Устанавливает текущее состояние игры.
* update():
  + Описание: Обновляет текущее состояние игры.
* draw(java.awt.Graphics2D g):
* Параметры:
* g: java.awt.Graphics2D
  + Описание: Отрисовывает текущее состояние игры.
* keyPressed(int k):
* Параметры:
* k: int
  + Описание: Обрабатывает нажатие клавиши.
* keyReleased(int k):
* Параметры:
* k: int
  + Описание: Обрабатывает отпускание клавиши.

Класс Level1State:

Переменные:

* tileMap:
  + Тип: TileMap
  + Описание: Карта уровня.
* bg:
  + Тип: Background
  + Описание: Фон.
* player:
  + Тип: Player
  + Описание: Игрок.
* enemies:
  + Тип: ArrayList<Enemy>
  + Описание: Список врагов.
* explosions:
  + Тип: ArrayList<Explosion>
  + Описание: Список взрывов.
* hud:
  + Тип: HUD
  + Описание: Интерфейс игрока.
* bgMusic:
  + Тип: AudioPlayer
  + Описание: Фоновая музыка.

Методы:

* Level1State(GameStateManager gsm):
* Параметры: gsm - GameStateManager
  + Описание: Конструктор, инициализирует состояние игры.
* init():
  + Описание: Инициализирует состояние, создает карту, фон, игрока, врагов, интерфейс игрока и воспроизводит фоновую музыку.
* populateEnemies():
  + Описание: Создает врагов на карте.
* update():
  + Описание: Обновляет состояние игры, включая игрока, карту, фон, врагов и взрывы.
* draw(Graphics2D g):
* Параметры: g - java.awt.Graphics2D
  + Описание: Отрисовывает состояние игры, включая фон, карту, игрока, врагов, взрывы и интерфейс игрока.
* keyPressed(int k):
* Параметры: k - int
  + Описание: Обрабатывает нажатие клавиши игроком.
* keyReleased(int k):
* Параметры: k - int
  + Описание: Обрабатывает отпускание клавиши игроком.

Класс MenuState:

Переменные:

* bg:
  + Тип: Background
  + Описание: Фон меню.
* currentChoice:
  + Тип: int
  + Описание: Текущий выбор в меню.
* options:
  + Тип: String[]
  + Описание: Варианты меню.
* titleColor:
  + Тип: Color
  + Описание: Цвет заголовка меню.
* titleFont:
  + Тип: Font
  + Описание: Шрифт заголовка меню.
* font:
  + Тип: Font
  + Описание: Шрифт текста меню.

Методы:

* MenuState(GameStateManager gsm):
* Параметры: gsm - GameStateManager
  + Описание: Конструктор, инициализирует меню игры.
* init():
  + Описание: Инициализация состояния. В этом классе не используется.
* update():
  + Описание: Обновляет состояние меню, включая фон.
* draw(Graphics2D g):
* Параметры: g - java.awt.Graphics2D
  + Описание: Отрисовывает состояние меню, включая фон, заголовок и варианты меню.
* select():
  + Описание: Обрабатывает выбор варианта меню и выполняет соответствующие действия.
* keyPressed(int k):
* Параметры: k - int
  + Описание: Обрабатывает нажатие клавиш для перемещения по меню и выбора пункта.
* keyReleased(int k):
* Параметры: k - int
  + Описание: Обрабатывает отпускание клавиш. В данном состоянии не используется.

Класс Slugger:

Переменные:

* sprites:
  + Тип: BufferedImage[]
  + Описание: Массив спрайтов для анимации врага.

Методы:

* Slugger(TileMap tm):
* Параметры: tm - TileMap
  + Описание: Конструктор класса, инициализирует врага и загружает его спрайты.
* getNextPosition():
  + Описание: Определяет следующую позицию врага в зависимости от его движения.
* update():
  + Описание: Обновляет состояние врага, включая его позицию, коллизии и анимацию.
* draw(Graphics2D g):
* Параметры: g - java.awt.Graphics2D
  + Описание: Отрисовывает врага.

Класс Animation:

Переменные:

* frames:
  + Тип: BufferedImage[]
  + Описание: Массив кадров анимации.
* currentFrame:
  + Тип: int
  + Описание: Индекс текущего кадра анимации.
* startTime:
  + Тип: long
  + Описание: Время начала анимации.
* delay:
  + Тип: long
  + Описание: Задержка между кадрами анимации.
* playedOnce:
  + Тип: boolean
  + Описание: Флаг, указывающий, была ли анимация воспроизведена один раз.

Методы:

* setFrames(BufferedImage[] frames):
* Параметры: frames - массив BufferedImage, содержащий кадры анимации.
  + Описание: Устанавливает массив кадров анимации и сбрасывает состояние анимации.
* setDelay(long d):
* Параметры: d - задержка между кадрами анимации в миллисекундах.
  + Описание: Устанавливает задержку между кадрами анимации.
* setFrame(int i):
* Параметры: i - индекс кадра анимации.
  + Описание: Устанавливает текущий кадр анимации.
* update():
  + Описание: Обновляет состояние анимации, переходя к следующему кадру по прошествии задержки.
* getFrame():
  + Описание: Возвращает индекс текущего кадра анимации.
* getImage():
  + Описание: Возвращает изображение текущего кадра анимации.
* hasPlayedOnce():
  + Описание: Возвращает true, если анимация была воспроизведена один раз, иначе false.

Класс Enemy:

Переменные:

* health:
  + Тип: int
  + Описание: Здоровье врага.
* maxHealth:
  + Тип: int
  + Описание: Максимальное здоровье врага.
* dead:
  + Тип: boolean
  + Описание: Флаг, указывающий, мертв ли враг.
* damage:
  + Тип: int
  + Описание: Урон, который враг может нанести.
* flinching:
  + Тип: boolean
  + Описание: Флаг, указывающий, находится ли враг в состоянии "подпрыгивания" после получения урона.
* flinchTimer:
  + Тип: long
  + Описание: Время начала "подпрыгивания" после получения урона.

Методы:

* isDead():
  + Описание: Возвращает true, если враг мертв, иначе false.
* getDamage():
  + Описание: Возвращает урон, который враг может нанести.
* hit(int damage):
* Параметры: damage - урон, полученный врагом.
  + Описание: Вызывается при получении урона врагом. Уменьшает здоровье врага на значение урона
* update():
  + Описание: Пустой метод для обновления состояния врага.

Класс Explosion:

Переменные:

* x:
  + Тип: int
  + Описание: Позиция взрыва по оси X.
* y:
  + Тип: int
  + Описание: Позиция взрыва по оси Y.
* xmap:
  + Тип: int
  + Описание: Позиция взрыва на карте по оси X.
* ymap:
  + Тип: int
  + Описание: Позиция взрыва на карте по оси Y.
* width:
  + Тип: int
  + Описание: Ширина взрыва.
* height:
  + Тип: int
  + Описание: Высота взрыва.
* animation:
  + Тип: Animation
  + Описание: Анимация взрыва.
* sprites:
  + Тип: BufferedImage[]
  + Описание: Спрайты для анимации взрыва.
* remove:
  + Тип: boolean
  + Описание: Флаг, указывающий на необходимость удаления взрыва.

Методы:

* Explosion(int x, int y):
* Параметры: x - начальная позиция по оси X, y - начальная позиция по оси Y.
  + Описание: Конструктор класса.
* update():
  + Описание: Обновляет анимацию взрыва.
* shouldRemove():
* Возвращает: boolean
  + Описание: Возвращает флаг, указывающий на необходимость удаления взрыва.
* setMapPosition(int x, int y):
* Параметры: x - позиция на карте по оси X, y - позиция на карте по оси Y.
  + Описание: Устанавливает позицию взрыва на карте.
* draw(Graphics2D g):
* Параметры: g - объект Graphics2D для рисования.
  + Описание: Рисует текущий кадр анимации взрыва на экране.

Класс FireBall:

Переменные:

* hit:
  + Тип: boolean
  + Описание: Флаг, указывающий, попала ли огненная шаров во что-то.
* remove:
  + Тип: boolean
  + Описание: Флаг, указывающий, что необходимо удалить огненный шар.
* sprites:
  + Тип: BufferedImage[]
  + Описание: Спрайты для обычного состояния огненного шара.
* hitSprites:
  + Тип: BufferedImage[]
  + Описание: Спрайты для состояния огненного шара при попадании.

Методы:

* FireBall(TileMap tm, boolean right):
* Параметры: tm - карта тайлов, right - направление движения огненного шара.
  + Описание: Конструктор класса.
* setHit():
  + Описание: Устанавливает флаг попадания и обновляет анимацию на спрайты при попадании.
* shouldRemove():
* Возвращает: boolean
  + Описание: Возвращает флаг, указывающий на необходимость удаления огненного шара.
* update():
  + Описание: Обновляет состояние огненного шара.
* draw(Graphics2D g):
* Параметры: g - объект Graphics2D для рисования.
  + Описание: Устанавливает позицию на карте и вызывает метод отрисовки из родительского класса.

Класс HUD:

Переменные:

* player:
  + Тип: Player
  + Описание: Ссылка на игрока.
* image:
  + Тип: BufferedImage
  + Описание: Изображение для отображения HUD.
* font:
  + Тип: Font
  + Описание: Шрифт для текста на HUD.

Методы:

* HUD(Player p):
* Параметры: p - объект игрока.
  + Описание: Конструктор класса.
* draw(Graphics2D g):
* Параметры: g - объект Graphics2D для рисования.
  + Описание: Отрисовывает изображение HUD.

Класс MapObject:

Переменные:

* tileMap:
  + Тип: TileMap
  + Описание: Ссылка на карту тайлов.
* tileSize:
  + Тип: int
  + Описание: Размер тайла.
* xmap:
  + Тип: double
  + Описание: Координата x на карте.
* ymap:
  + Тип: double
  + Описание: Координата y на карте.
* x:
  + Тип: double
  + Описание: Координата x.
* y:
  + Тип: double
  + Описание: Координата y.
* dx:
  + Тип: double
  + Описание: Вектор скорости по x.
* dy:
  + Тип: double
  + Описание: Вектор скорости по y.
* width:
  + Тип: int
  + Описание: Ширина объекта.
* height:
  + Тип: int
  + Описание: Высота объекта.
* cwidth:
  + Тип: int
  + Описание: Ширина коллизионной коробки.
* cheight:
  + Тип: int
  + Описание: Высота коллизионной коробки.
* currRow:
  + Тип: int
  + Описание: Текущий ряд.
* currCol:
  + Тип: int
  + Описание: Текущий столбец.
* xdest:
  + Тип: double
  + Описание: Целевая координата x.
* ydest:
  + Тип: double
  + Описание: Целевая координата y.
* xtemp:
  + Тип: double
  + Описание: Временная координата x.
* ytemp:
  + Тип: double
  + Описание: Временная координата y.
* topLeft:
  + Тип: boolean
  + Описание: Верхний левый угол коллизионной коробки.
* topRight:
  + Тип: boolean
  + Описание: Верхний правый угол коллизионной коробки.
* bottomLeft:
  + Тип: boolean
  + Описание: Нижний левый угол коллизионной коробки.
* bottomRight:
  + Тип: boolean
  + Описание: Нижний правый угол коллизионной коробки.
* animation:
  + Тип: Animation
  + Описание: Анимация.
* currentAction:
  + Тип: int
  + Описание: Текущее действие.
* previousAction:
  + Тип: int
  + Описание: Предыдущее действие.
* facingRight:
  + Тип: boolean
  + Описание: Направление вправо.
* left:
  + Тип: boolean
  + Описание: Движение влево.
* right:
  + Тип: boolean
  + Описание: Движение вправо.
* up:
  + Тип: boolean
  + Описание: Движение вверх.
* down:
  + Тип: boolean
  + Описание: Движение вниз.
* jumping:
  + Тип: boolean
  + Описание: Прыжок.
* falling:
  + Тип: boolean
  + Описание: Падение.
* moveSpeed:
  + Тип: double
  + Описание: Скорость движения.
* maxSpeed:
  + Тип: double
  + Описание: Максимальная скорость.
* stopSpeed:
  + Тип: double
  + Описание: Скорость остановки.
* fallSpeed:
  + Тип: double
  + Описание: Скорость падения.
* maxFallSpeed:
  + Тип: double
  + Описание: Максимальная скорость падения.
* jumpStart:
  + Тип: double
  + Описание: Начальная скорость прыжка.
* stopJumpSpeed:
  + Тип: double
  + Описание: Скорость остановки прыжка.

Методы:

* MapObject(TileMap tm):
* Параметры: tm - объект карты тайлов.
  + Описание: Конструктор класса.
* intersects(MapObject o):
* Параметры: o - объект для проверки пересечения.
  + Описание: Проверяет пересекается ли данный объект с другим объектом.
* getRectangle():
* Возвращает: Прямоугольник коллизии объекта.
  + Описание: Получает прямоугольник, представляющий коллизию объекта.
* calculateCorners(double x, double y):
* Параметры: x - координата x, y - координата y.
  + Описание: Вычисляет углы коллизионной коробки.
* checkTileMapCollision():
  + Описание: Проверяет коллизию с картой тайлов и корректирует позицию объекта.
* getx():
* Возвращает: Координата x.
  + Описание: Получает текущую координату x объекта.
* gety():
* Возвращает: Координата y.
  + Описание: Получает текущую координату y объекта.
* getWidth():
* Возвращает: Ширина объекта.
  + Описание: Получает ширину объекта.
* getHeight():
* Возвращает: Высота объекта.
  + Описание: Получает высоту объекта.
* getCWidth():
* Возвращает: Ширина коллизионной коробки.
  + Описание: Получает ширину коллизионной коробки.
* getCHeight():
* Возвращает: Высота коллизионной коробки.
  + Описание: Получает высоту коллизионной коробки.
* setPosition(double x, double y):
* Параметры: x - координата x, y - координата y.
  + Описание: Устанавливает позицию объекта.
* setVector(double dx, double dy):
* Параметры: dx - вектор скорости по x, dy - вектор скорости по y.
  + Описание: Устанавливает вектор скорости объекта.
* setMapPosition():
  + Описание: Устанавливает позицию объекта на карте тайлов.
* setLeft(boolean b):
* Параметры: b - true, если движение влево, иначе false.
  + Описание: Устанавливает движение влево.
* setRight(boolean b):
* Параметры: b - true, если движение вправо, иначе false.
  + Описание: Устанавливает движение вправо.
* setUp(boolean b):
* Параметры: b - true, если движение вверх, иначе false.
  + Описание: Устанавливает движение вверх.
* setDown(boolean b):
* Параметры: b - true, если движение вниз, иначе false.
  + Описание: Устанавливает движение вниз.
* setJumping(boolean b):
* Параметры: b - true, если прыжок, иначе false.
  + Описание: Устанавливает состояние прыжка.
* notOnScreen():
* Возвращает: true, если объект находится за пределами экрана, иначе false.
  + Описание: Проверяет, находится ли объект за пределами экрана.
* draw(java.awt.Graphics2D g):
* Параметры: g - объект Graphics2D для рисования.
  + Описание: Отрисовывает объект.

Класс Player:

Переменные:

* health:
  + Тип: int
  + Описание: Здоровье игрока.
* maxHealth:
  + Тип: int
  + Описание: Максимальное здоровье игрока.
* fire:
  + Тип: int
  + Описание: Огонь (энергия) игрока.
* maxFire:
  + Тип: int
  + Описание: Максимальное значение огня игрока.
* dead:
  + Тип: boolean
  + Описание: Флаг, указывающий, мертв ли игрок.
* flinching:
  + Тип: boolean
  + Описание: Флаг, указывающий, подвержен ли игрок "флинчу".
* flinchTimer:
  + Тип: long
  + Описание: Таймер для флинча.
* firing:
  + Тип: boolean
  + Описание: Флаг, указывающий, происходит ли стрельба.
* fireCost:
  + Тип: int
  + Описание: Стоимость стрельбы.
* fireBallDamage:
  + Тип: int
  + Описание: Урон от фаерболла.
* fireBalls:
  + Тип: ArrayList<FireBall>
  + Описание: Массив с фаерболлами.
* scratching:
  + Тип: boolean
  + Описание: Флаг, указывающий, происходит ли скретч.
* scratchDamage:
  + Тип: int
  + Описание: Урон от скретча.
* scratchRange:
  + Тип: int
  + Описание: Радиус скретча.
* gliding:
  + Тип: boolean
  + Описание: Флаг, указывающий, парит ли игрок.
* sprites:
  + Тип: ArrayList<BufferedImage[]>
  + Описание: Спрайты для анимаций.
* numFrames:
  + Тип: int[]
  + Описание: Массив с количеством кадров для каждого действия.
* IDLE, WALKING, JUMPING, FALLING, GLIDING, FIREBALL, SCRATCHING:
  + Тип: int
  + Описание: Константы для действий анимации.
* sfx:
  + Тип: HashMap<String, AudioPlayer>
  + Описание: Мап для звуковых эффектов.

Методы:

* Player(TileMap tm):
* Параметры: tm - объект карты тайлов.
  + Описание: Конструктор класса Player. Инициализирует параметры игрока и загружает спрайты и звуковые эффекты.
* getHealth():
* Возвращает: Здоровье игрока.
  + Описание: Получает текущее значение здоровья игрока.
* getMaxHealth():
* Возвращает: Максимальное здоровье игрока.
  + Описание: Получает максимальное значение здоровья игрока.
* getFire():
* Возвращает: Уровень огня игрока.
  + Описание: Получает текущий уровень огня игрока.
* getMaxFire():
* Возвращает: Максимальный уровень огня игрока.
  + Описание: Получает максимальное значение огня игрока.
* setFiring():
  + Описание: Устанавливает состояние стрельбы.
* setScratching():
  + Описание: Устанавливает состояние скретча.
* setGliding(boolean b):
* Параметры: b - true, если игрок парит, иначе false.
  + Описание: Устанавливает состояние парения игрока.
* checkAttack(ArrayList<Enemy> enemies):
* Параметры: enemies - список врагов.
  + Описание: Проверяет атаки игрока и их столкновения с врагами.
* hit(int damage):
* Параметры: damage - количество урона.
  + Описание: Обрабатывает получение урона игроком.
* getNextPosition():
  + Описание: Вычисляет следующую позицию игрока.
* update():
  + Описание: Обновляет состояние игрока.
* draw(Graphics2D g):
* Параметры: g - объект Graphics2D для отрисовки.
  + Описание: Отрисовывает игрока.

# Тестирование и установка приложения

3.1 Интеграционное тестирование:

1. Вход в приложение:

* Откройте приложение и убедитесь, что экран главного меню отображается.
* Выберете пункт “Start” и нажмите на него. Убедитесь, что переход на начальный уровень игры был успешен.
* Выберете пункт “Quit” и нажмите на него. Убедитесь, что выход из игры был успешен.

2. Отображение главного экрана игры:

* Перейдите на экран игры и убедитесь, что он корректно отображается.
* Проверьте, что все клавиши работают и правильно выполняют свои команды.
* Проверьте корректно ли отображается персонаж игры, а также его анимации.
* Проверьте корректно ли отображаются существа на карте, а также их анимации.
* Проверьте корректность отображения статус – бара.
* Проверьте корректность анимации и отображения снаряда игрока.

## 3.1 Пользовательское тестирование:

1. Навигация:

* Переключитесь между различными экранами приложения, используя главное меню, и убедитесь, что переходы происходят без задержек.
* Проверьте анимации кнопок, меняются ли цвета корректно, корректно ли отображается задний фон и его анимация.

2. Взаимодействие с игрой:

* Попробуй нажать на различные кнопки, такие как перемещение персонажа, атака и огонь, и убедитесь, что они реагируют на ваши действия корректно.
* Проверьте правильно ли происходит атака, по существу, в игре и его анимацию.

1. Отображение данных:

* Посмотрите различные элементы главного меню, такие как кнопки навигации, и убедитесь, что данные отображаются корректно.
* Проверьте корректность отображения статус – бара игрока, площадки на которой он находится, задний фон, а также самого персонажа.

Предполагаемые ошибки:

1. Ошибка запуска:

* При запуске приложение оно может не запустить, по причине того, что у вас не установлена java.

1. Отображение данных:

* При переходе на экран игры может перестать перемещаться карта по мере движения игрока.
* При смерти игрока, модель может зависнуть.

## 3.2 Установка приложения:

Для установки приложения на компьютер, загрузите установочный файл с официального сайта или другого надежного источника. После загрузки запустите установщик и следуйте инструкциям, указанным в приложении, чтобы завершить установку. По завершении процесса приложение будет готово к использованию на вашем компьютере. Полную инструкцию смотреть в Приложении B.

# Заключение

В результате выполнения курсовой работы было успешно разработано Java приложение под названием "Dragon Tale", представляющее собой захватывающий 2D платформер с использованием тайловой графики. Разработка этого проекта прошла через несколько ключевых этапов, начиная с проектирования архитектуры, где определялись основные классы и их взаимосвязи, и выбора инструментов разработки, обеспечивающих эффективную работу.

Далее был реализован игровой движок, включающий в себя классы для работы с тайловыми картами, анимациями и обработки коллизий. Это был один из ключевых этапов, так как от него зависела основа функциональности всего приложения.

После этого началась разработка игровых объектов, таких как игрок, враги и другие элементы игрового мира. Каждый объект был тщательно проработан с учетом его уникальных характеристик и функционала.

Важным этапом стало внедрение графики и анимаций, где были добавлены графические ресурсы для игровых объектов и созданы анимации для различных действий игрока и врагов. Это придало игре эстетическое оформление и улучшило ее визуальное восприятие.

В итоге, созданное приложение "Dragon Tale" предлагает пользователям увлекательный игровой опыт и становится важным шагом в моем профессиональном развитии в области разработки программного обеспечения. Полученный опыт открывает новые возможности для дальнейших проектов и углубления знаний в области разработки игр и программного обеспечения в целом.

# Список используемой литературы

Java Swing: Эффектные пользовательские интерфейсы - Издание второе (2011) - Joshua Marinacci.

Книга «Программируем на Java. 5-е межд. изд.» - Кети Сьерра и Берт Бейтс.

Превращение java проекта в exe файл. - <https://javarush.com/groups/posts/3910-kofe-breyk-148-kak-prevratitjh-ljubuju-java-programmu-v-avtonomnihy-exe-fayl>

# Приложение A: Техническое задание

1. Общие положения

Наименование продукта

Игра "Dragon Tale"

1.2 Назначение

Игра "Dragon Tale" представляет собой захватывающее пиксельное приключение, где игрок управляет милым зеленым дракончиком через различные локации, сражается с врагами и собирает ценные предметы. Эта игра предназначена для ПК и разработана с учетом интересов как детей, так и взрослых.

1.3 Область применения

Игра "Dragon Tale" создана для того, чтобы подарить игрокам волнующие приключения, вдохновленные классическими платформерами, и заставить их пережить увлекательные моменты, полные опасностей и загадок.

2. Технические характеристики

2.1 Платформа

Игра доступна на персональных компьютерах под управлением операционных систем Windows, MacOS и Linux.

2.2 Разрешение экрана

Графика игры должна быть адаптирована для различных разрешений экрана согласно ГОСТ 2.114-95, включая 720x480, 1080x1920, и 1440x2560.

2.3 Производительность

Игра должна быть оптимизирована для плавной работы на различных конфигурациях компьютеров, соответствуя требованиям ГОСТ 26474-85. Это включает работу на старых и менее мощных устройствах.

2.4 Управление

Игра будет поддерживать управление с помощью клавиатуры компьютера с возможностью настройки клавиш для комфортной игры.

2.5 Графика

Графический контент игры будет выполнен в соответствии с пиксельным стилем, представляя красочные локации, детализированных персонажей и впечатляющие спецэффекты.

2.6 Звук

Фоновая музыка и звуковые эффекты будут реализованы с использованием высококачественных аудиофайлов, чтобы подчеркнуть атмосферу игры и увлечь игроков.

3. Игровой процесс

* 3.1 Управление

Игрок будет управлять дракончиком с помощью клавиатуры, используя стрелки для передвижения, клавишу "W" для прыжка, и клавиши "E", "R" и "F" для дополнительных действий.

* 3.2 Характеристики героя

Герой будет иметь показатели здоровья и огня, отображаемые на экране, чтобы игрок мог следить за их текущим состоянием.

* 3.3 Препятствия и враги

Игра будет содержать разнообразные препятствия, такие как пропасти, шипы и движущиеся платформы, а также врагов, которые будут препятствовать движению дракончика.

* 3.4 Игровой мир

Игровой мир будет представлять собой различные локации, включая заснеженные горы, зеленые леса и древние подземелья, создавая разнообразие в визуальном оформлении и геймплейных ситуациях.

4. Интерфейс

* 4.1 Главное меню

Главное меню игры будет содержать кнопки для начала игры, настройки и выхода, выполненные в стиле пиксель-арта и адаптированные для различных разрешений экрана.

* 4.2 Интерфейс в игре

Интерфейс в игре будет представлен в виде всплывающих окон с информацией о текущем состоянии героя, собранных предметах и достижениях.

5. Тестирование

* 5.1 Функциональное тестирование

Проведение тестирования на всех основных функциях игры для обеспечения их правильной работы и отсутствия ошибок.

* 5.2 Совместимость

Тестирование игры на различных конфигурациях ПК и операционных системах для проверки совместимости и корректного отображения.

* 5.3 Производительность

Проверка производительности игры на различных устройствах для обеспечения плавного игрового процесса.

6. Системные требования

* 6.1 Операционная система
  + Windows 7/8/10
  + MacOS X
  + Linux
* 6.2 Процессор
  + Intel Core i3 или эквивалентный.
* 6.3 Оперативная память
  + Минимум 4 ГБ.
* 6.4 Графика
  + Поддержка OpenGL 2.0 или выше.
* 6.5 Свободное место на диске
  + Минимум 500 МБ.

7. Требования к программной разработке и поддержке

* 7.1 Язык программирования

Игра будет разработана на языке программирования Java с использованием платформы LibGDX для работы с графикой и звуком.

* 7.2 Инструменты разработки

Для разработки игры рекомендуется использовать среду разработки IntelliJ IDEA или Eclipse.

* 7.3 Фреймворки и библиотеки

При разработке игры будут использованы фреймворк LibGDX для работы с графикой и звуком.

* 7.4 Управление версиями

Для контроля версий и совместной работы над проектом будет использован система контроля версий Git.

8. Макет приложение

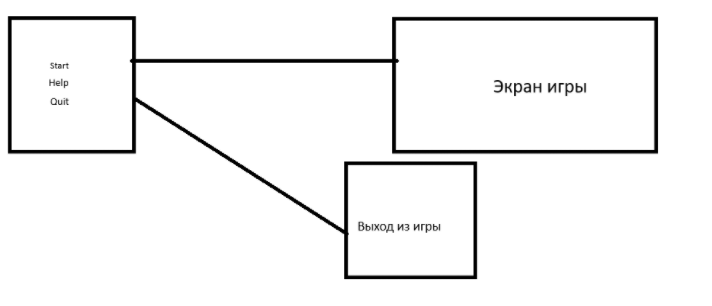


Рисунок 8 – Макет приложение

# Приложение B. Руководство пользователя

1. Установка игры

• Прежде чем начать играть в "Dragon Tale", убедитесь, что игра установлена на вашем компьютере. Для установки выполните следующие шаги:

• Загрузите установочный файл игры "Dragon Tale" с официального веб-сайта или другого авторизованного источника.

• Запустите установочный файл и следуйте инструкциям мастера установки.

• Выберите папку для установки игры и дождитесь завершения процесса установки.

• После завершения установки вы сможете запустить игру и начать свое приключение в мире "Dragon Tale".

2. Начало игры

После установки игры просто выполните следующие шаги:

• Найдите ярлык игры на рабочем столе вашего компьютера или откройте папку, в которой она установлена.

• Дважды щелкните по ярлыку "Dragon Tale", чтобы запустить игру.

• После запуска игры вы попадете в главное меню, где вы сможете выбрать одну из следующих опций:

• Start (Начать игру): Нажмите эту кнопку, чтобы начать новое приключение.

• Settings (Настройки): Здесь вы можете изменить параметры игры, такие как управление, громкость звука и разрешение экрана.

• Exit (Выход): Нажмите эту кнопку, чтобы закрыть игру.

3. Управление

Управление вашим дракончиком в "Dragon Tale" осуществляется с помощью клавиатуры. Вот основные команды:

• Стрелка влево/вправо: Перемещение дракончика влево и вправо по экрану.

• Клавиша "W": Прыжок. Нажмите эту клавишу, чтобы ваш дракончик подпрыгнул.

• Клавиши "E", "R", "F": Выполнение дополнительных действий. Каждая из этих клавиш может быть назначена на различные действия в игре.

4. Цель игры

• Главная цель игры "Dragon Tale" – преодолеть все уровни, собрать ценные предметы и одолеть всех врагов, встречающихся на вашем пути. Исследуйте различные локации и завершите игру, чтобы стать настоящим героем!

5. Здоровье и огонь

• Ваш дракончик обладает показателями здоровья и огня. Будьте осторожны и следите за этими показателями, чтобы не попасть в ловушки и не быть побежденным врагами.

6. Настройки

• В меню настроек вы можете настроить игру согласно вашим предпочтениям. Здесь вы можете изменить управление, настроить графику, громкость звука и многое другое.

7. Завершение игры

• После того как вы завершите все уровни и достигнете конца игры, вы получите заслуженную награду за ваше отвагу и мастерство.

8. Требования к системе

Перед тем как начать играть, убедитесь, что ваш компьютер соответствует следующим требованиям:

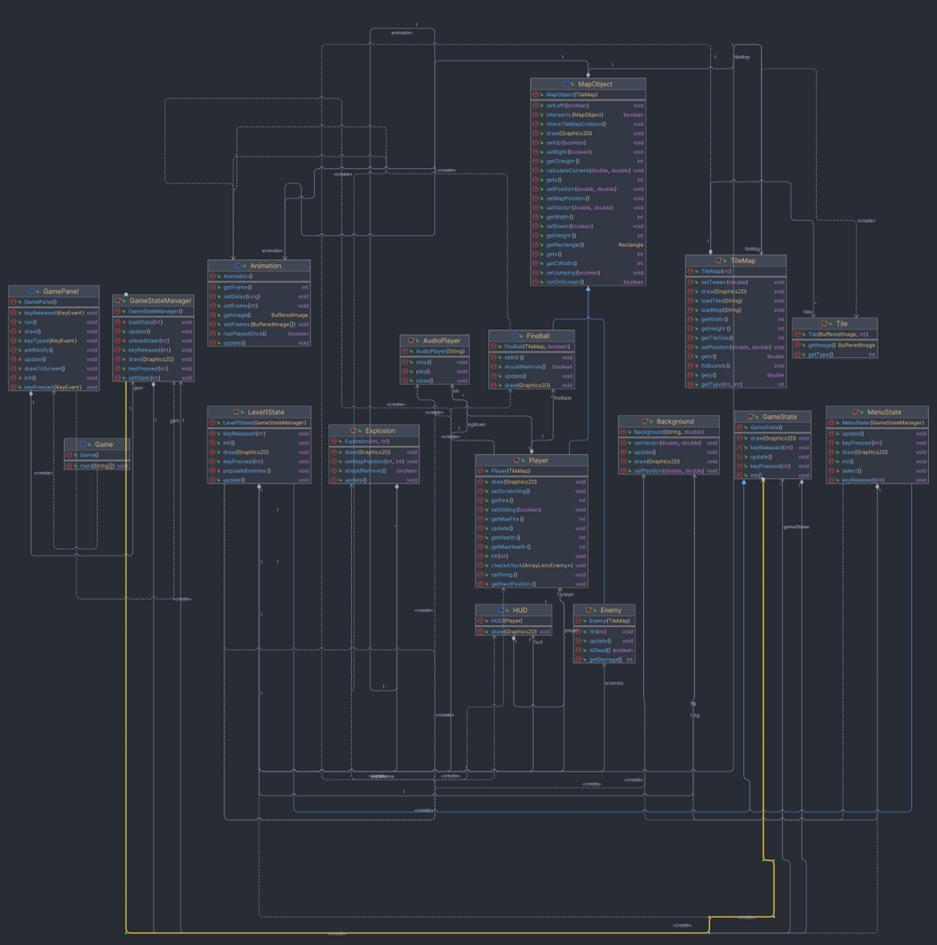
• Операционная система: Windows 7/8/10, MacOS X, Linux.

• Процессор: Intel Core i3 или эквивалентный.

• Оперативная память: Минимум 4 ГБ.

• Графика: Поддержка OpenGL 2.0 или выше.

# Приложение С. Диаграмма классов



# Приложение D. Код программы

package Main;

import javax.swing.JFrame;

/\*\*

\* @author Ильин

\*/

public class Game {

public static void main(String[] args) {

// Создаем новое окно JFrame с названием "Dragon Tale"

JFrame window = new JFrame("Dragon Tale");

// Устанавливаем содержимое окна на экземпляр GamePanel

window.setContentPane(new GamePanel());

// Устанавливаем операцию закрытия окна по умолчанию, чтобы при закрытии приложение завершалось

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

// Запрещаем изменение размеров окна

window.setResizable(false);

// Автоматически подстраиваем размеры окна под его содержимое

window.pack();

// Делаем окно видимым

window.setVisible(true);

}

}

public class GamePanel extends JPanel implements Runnable, KeyListener {

// Размеры окна

public static final int WIDTH = 320;

public static final int HEIGHT = 240;

public static final int SCALE = 2;

// Поток игры

private Thread thread;

private boolean running;

private int FPS = 60; // Желаемое количество кадров в секунду

private long targetTime = 1000 / FPS; // Время на каждый кадр

// Изображение

private BufferedImage image;

private Graphics2D g;

// Менеджер состояний игры

private GameStateManager gsm;

// Конструктор

public GamePanel() {

super();

// Устанавливаем предпочтительные размеры панели

setPreferredSize(new Dimension(WIDTH \* SCALE, HEIGHT \* SCALE));

setFocusable(true); // Позволяет панели получать фокус ввода

requestFocus(); // Запрашиваем фокус у панели

}

// Метод для запуска потока игры при добавлении панели на экран

public void addNotify() {

super.addNotify();

if (thread == null) {

// Создаем новый поток и запускаем его

thread = new Thread(this);

addKeyListener(this); // Добавляем слушателя клавиатуры

thread.start();

}

}

// Метод инициализации игры

private void init() {

// Создаем изображение

image = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE\_INT\_RGB);

g = (Graphics2D) image.getGraphics();

running = true;

// Инициализируем менеджер состояний игры

gsm = new GameStateManager();

}

// Основной игровой цикл

public void run() {

init(); // Инициализируем игру

long start;

long elapsed;

long wait;

while (running) {

start = System.nanoTime();

update(); // Обновляем состояние игры

draw(); // Рисуем игру на изображении

drawToScreen(); // Отображаем изображение на экране

elapsed = System.nanoTime() - start;

// Вычисляем время ожидания до следующего кадра

wait = targetTime - elapsed / 1000000;

if (wait < 0) wait = 5; // Не даем отрицательное время ожидания

try {

Thread.sleep(wait); // Приостанавливаем поток на указанное время

} catch (Exception e) {

e.printStackTrace();

}

}

}

// Метод для обновления состояния игры

private void update() {

gsm.update();

}

// Метод для отрисовки игры на изображении

private void draw() {

gsm.draw(g);

}

// Метод для отображения изображения на экране

private void drawToScreen() {

Graphics g2 = getGraphics();

// Рисуем изображение на экране с учетом масштаба

g2.drawImage(image, 0, 0, WIDTH \* SCALE, HEIGHT \* SCALE, null);

g2.dispose(); // Освобождаем ресурсы графики

}

// Методы обработки событий клавиатуры

public void keyTyped(KeyEvent key) {

}

public void keyPressed(KeyEvent key) {

// Передаем нажатие клавиши в менеджер состояний игры

gsm.keyPressed(key.getKeyCode());

}

public void keyReleased(KeyEvent key) {

// Передаем отпускание клавиши в менеджер состояний игры

gsm.keyReleased(key.getKeyCode());

}

}

package TileMap;

import Main.GamePanel;

import java.awt.\*;

import java.awt.image.\*;

import javax.imageio.ImageIO;

/\*\*

\* @author Ильин Тимур

\*/

public class Background {

private BufferedImage image; // Изображение фона

private double x; // Координата x фона

private double y; // Координата y фона

private double dx; // Вектор движения по оси x

private double dy; // Вектор движения по оси y

private double moveScale; // Масштабирование скорости движения

// Конструктор

public Background(String s, double ms) {

try {

// Загружаем изображение фона из файла

image = ImageIO.read(getClass().getResourceAsStream(s));

moveScale = ms; // Устанавливаем масштаб скорости движения

} catch (Exception e) {

e.printStackTrace();

}

}

// Метод установки позиции фона

public void setPosition(double x, double y) {

// Зацикливаем фон, чтобы он перемещался бесконечно

this.x = (x \* moveScale) % GamePanel.WIDTH;

this.y = (y \* moveScale) % GamePanel.HEIGHT;

}

// Метод установки вектора движения фона

public void setVector(double dx, double dy) {

this.dx = dx;

this.dy = dy;

}

// Метод обновления состояния фона

public void update() {

x += dx;

y += dy;

}

// Метод отрисовки фона

public void draw(Graphics2D g) {

// Отрисовка фона на текущей позиции

g.drawImage(image, (int) x, (int) y, null);

// Если фон выходит за левый край экрана, рисуем его справа от экрана

if (x < 0) {

g.drawImage(image, (int) x + GamePanel.WIDTH, (int) y, null);

}

// Если фон выходит за правый край экрана, рисуем его слева от экрана

if (x > 0) {

g.drawImage(image, (int) x - GamePanel.WIDTH, (int) y, null);

}

}

}

package TileMap;

import java.awt.image.BufferedImage;

/\*\*

\* @author Ильин Тимур

\*/

public class Tile {

private BufferedImage image; // Изображение тайла

private int type; // Тип тайла

// Виды тайлов

public static final int NORMAL = 0; // Обычный тайл

public static final int BLOCKED = 1; // Заблокированный тайл

// Конструктор

public Tile(BufferedImage image, int type) {

this.image = image; // Устанавливаем изображение тайла

this.type = type; // Устанавливаем тип тайла

}

// Метод получения изображения тайла

public BufferedImage getImage() {

return image;

}

// Метод получения типа тайла

public int getType() {

return type;

}

}

package TileMap;

import java.awt.\*;

import java.awt.image.\*;

import java.io.\*;

import javax.imageio.ImageIO;

import Main.GamePanel;

/\*\*

\* @author Ильин Тимур

\*/

public class TileMap {

// Позиция

private double x;

private double y;

// Границы

private int xmin;

private int ymin;

private int xmax;

private int ymax;

private double tween; // Для плавного перемещения камеры

// Карта

private int[][] map;

private int tileSize;

private int numRows;

private int numCols;

private int width;

private int height;

// Тайлсет

private BufferedImage tileset;

private int numTilesAcross;

private Tile[][] tiles;

// Отрисовка

private int rowOffset;

private int colOffset;

private int numRowsToDraw;

private int numColsToDraw;

public TileMap(int tileSize) {

this.tileSize = tileSize;

// Определение количества строк и столбцов для отрисовки

numRowsToDraw = GamePanel.HEIGHT / tileSize + 2;

numColsToDraw = GamePanel.WIDTH / tileSize + 2;

tween = 0.07; // Коэффициент плавности перемещения

}

// Загрузка тайлов изображений из файла

public void loadTiles(String s) {

try {

tileset = ImageIO.read(getClass().getResourceAsStream(s));

numTilesAcross = tileset.getWidth() / tileSize;

tiles = new Tile[2][numTilesAcross]; // Создание массива тайлов

BufferedImage subimage;

// Загрузка изображений для каждого типа тайла

for (int col = 0; col < numTilesAcross; col++) {

subimage = tileset.getSubimage(col \* tileSize, 0, tileSize, tileSize);

tiles[0][col] = new Tile(subimage, Tile.NORMAL);

subimage = tileset.getSubimage(col \* tileSize, tileSize, tileSize, tileSize);

tiles[1][col] = new Tile(subimage, Tile.BLOCKED);

}

} catch (Exception e) {

e.printStackTrace();

}

}

// Загрузка карты из файла

public void loadMap(String s) {

try {

InputStream in = getClass().getResourceAsStream(s);

BufferedReader br = new BufferedReader(new InputStreamReader(in));

// Чтение размеров карты

numCols = Integer.parseInt(br.readLine());

numRows = Integer.parseInt(br.readLine());

map = new int[numRows][numCols];

width = numCols \* tileSize;

height = numRows \* tileSize;

// Установка границ карты

xmin = GamePanel.WIDTH - width;

xmax = 0;

ymin = GamePanel.HEIGHT - height;

ymax = 0;

String delims = "\\s+"; // Разделитель между значениями в строке

// Загрузка данных карты

for (int row = 0; row < numRows; row++) {

String line = br.readLine();

String[] tokens = line.split(delims);

for (int col = 0; col < numCols; col++) {

map[row][col] = Integer.parseInt(tokens[col]);

}

}

} catch (Exception e) {

e.printStackTrace();

}

}

// Получение размера тайла

public int getTileSize() {

return tileSize;

}

// Получение текущей позиции X

public double getx() {

return x;

}

// Получение текущей позиции Y

public double gety() {

return y;

}

// Получение ширины карты

public int getWidth() {

return width;

}

// Получение высоты карты

public int getHeight() {

return height;

}

// Получение типа тайла по указанным координатам

public int getType(int row, int col) {

int rc = map[row][col];

int r = rc / numTilesAcross;

int c = rc % numTilesAcross;

return tiles[r][c].getType();

}

// Установка коэффициента плавного перемещения

public void setTween(double d) {

tween = d;

}

// Установка позиции карты

public void setPosition(double x, double y) {

// Плавное перемещение камеры

this.x += (x - this.x) \* tween;

this.y += (y - this.y) \* tween;

fixBounds(); // Корректировка границ отображения карты

colOffset = (int) -this.x / tileSize;

rowOffset = (int) -this.y / tileSize;

}

// Корректировка границ отображения карты

private void fixBounds() {

if (x < xmin) x = xmin;

if (y < ymin) y = ymin;

if (x > xmax) x = xmax;

if (y > ymax) y = ymax;

}

// Отрисовка карты

public void draw(Graphics2D g) {

for (int row = rowOffset; row < rowOffset + numRowsToDraw; row++) {

if (row >= numRows) break;

for (int col = colOffset; col < colOffset + numColsToDraw; col++) {

if (col >= numCols) break;

if (map[row][col] == 0) continue;

int rc = map[row][col];

int r = rc / numTilesAcross;

int c = rc % numTilesAcross;

g.drawImage(tiles[r][c].getImage(), (int) x + col \* tileSize, (int) y + row \* tileSize, null);

}

}

}

}

package GameState;

public abstract class GameState {

protected GameStateManager gsm; // Менеджер состояний

// Метод инициализации состояния

public abstract void init();

// Метод обновления состояния

public abstract void update();

// Метод отрисовки состояния

public abstract void draw(java.awt.Graphics2D g);

// Метод обработки нажатия клавиши

public abstract void keyPressed(int k);

// Метод обработки отпускания клавиши

public abstract void keyReleased(int k);

}

package GameState;

import java.util.ArrayList;

public class GameStateManager {

private GameState[] gameStates; // Массив состояний игры

private int currentState; // Текущее состояние игры

public static final int NUMGAMESTATES = 2; // Количество состояний

public static final int MENUSTATE = 0; // Индекс состояния меню

public static final int LEVEL1STATE = 1; // Индекс состояния уровня

// Конструктор

public GameStateManager() {

gameStates = new GameState[NUMGAMESTATES]; // Инициализация массива состояний

currentState = MENUSTATE; // Установка начального состояния

loadState(currentState); // Загрузка начального состояния

}

// Метод для загрузки состояния

private void loadState(int state) {

// Создание нового экземпляра состояния в зависимости от переданного индекса

if(state == MENUSTATE)

gameStates[state] = new MenuState(this);

if(state == LEVEL1STATE)

gameStates[state] = new Level1State(this);

}

// Метод для выгрузки состояния

private void unloadState(int state) {

gameStates[state] = null; // Освобождение ресурсов состояния

}

// Метод для установки текущего состояния

public void setState(int state) {

unloadState(currentState); // Выгрузка текущего состояния

currentState = state; // Установка нового текущего состояния

loadState(currentState); // Загрузка нового текущего состояния

}

// Метод для обновления текущего состояния

public void update() {

try {

gameStates[currentState].update(); // Вызов метода обновления текущего состояния

} catch(Exception e) {} // Обработка исключений

}

// Метод для отрисовки текущего состояния

public void draw(java.awt.Graphics2D g) {

try {

gameStates[currentState].draw(g); // Вызов метода отрисовки текущего состояния

} catch(Exception e) {} // Обработка исключений

}

// Метод для обработки нажатия клавиши

public void keyPressed(int k) {

gameStates[currentState].keyPressed(k); // Передача нажатой клавиши текущему состоянию

}

// Метод для обработки отпускания клавиши

public void keyReleased(int k) {

gameStates[currentState].keyReleased(k); // Передача отпущенной клавиши текущему состоянию

}

}

package GameState;

import Main.GamePanel;

import TileMap.\*;

import Entity.\*;

import Entity.Enemies.\*;

import Audio.AudioPlayer;

import java.awt.\*;

import java.awt.event.KeyEvent;

import java.util.ArrayList;

public class Level1State extends GameState {

private TileMap tileMap; // Карта уровня

private Background bg; // Фон

private Player player; // Игрок

private ArrayList<Enemy> enemies; // Список врагов

private ArrayList<Explosion> explosions; // Список взрывов

private HUD hud; // Интерфейс игрока

private AudioPlayer bgMusic; // Фоновая музыка

// Конструктор, принимающий менеджер состояний

public Level1State(GameStateManager gsm) {

this.gsm = gsm;

init(); // Инициализация состояния

}

// Метод инициализации состояния

public void init() {

// Создание карты

tileMap = new TileMap(30);

tileMap.loadTiles("/Tilesets/grasstileset.gif");

tileMap.loadMap("/Maps/level1-1.map");

tileMap.setPosition(0, 0);

tileMap.setTween(1);

// Создание фона

bg = new Background("/Backgrounds/grassbg1.gif", 0.1);

// Создание игрока

player = new Player(tileMap);

player.setPosition(100, 100);

// Создание врагов

populateEnemies();

// Создание списка взрывов

explosions = new ArrayList<Explosion>();

// Создание интерфейса игрока

hud = new HUD(player);

// Воспроизведение фоновой музыки

bgMusic = new AudioPlayer("/Music/level1-1.mp3");

bgMusic.play();

}

// Метод для создания врагов на карте

private void populateEnemies() {

enemies = new ArrayList<Enemy>();

Slugger s;

Point[] points = new Point[] {

new Point(200, 100),

new Point(860, 200),

new Point(1525, 200),

new Point(1680, 200),

new Point(1800, 200)

};

for(int i = 0; i < points.length; i++) {

s = new Slugger(tileMap);

s.setPosition(points[i].x, points[i].y);

enemies.add(s);

}

}

// Метод обновления состояния

public void update() {

// Обновление игрока

player.update();

// Позиционирование карты относительно игрока

tileMap.setPosition(

GamePanel.WIDTH / 2 - player.getx(),

GamePanel.HEIGHT / 2 - player.gety()

);

// Обновление фона

bg.setPosition(tileMap.getx(), tileMap.gety());

// Проверка атаки игрока

player.checkAttack(enemies);

// Обновление всех врагов

for(int i = 0; i < enemies.size(); i++) {

Enemy e = enemies.get(i);

e.update();

if(e.isDead()) {

enemies.remove(i);

i--;

explosions.add(

new Explosion(e.getx(), e.gety()));

}

}

// Обновление взрывов

for(int i = 0; i < explosions.size(); i++) {

explosions.get(i).update();

if(explosions.get(i).shouldRemove()) {

explosions.remove(i);

i--;

}

}

}

// Метод отрисовки состояния

public void draw(Graphics2D g) {

// Отрисовка фона

bg.draw(g);

// Отрисовка карты

tileMap.draw(g);

// Отрисовка игрока

player.draw(g);

// Отрисовка врагов

for(int i = 0; i < enemies.size(); i++) {

enemies.get(i).draw(g);

}

// Отрисовка взрывов

for(int i = 0; i < explosions.size(); i++) {

explosions.get(i).setMapPosition(

(int)tileMap.getx(), (int)tileMap.gety());

explosions.get(i).draw(g);

}

// Отрисовка интерфейса игрока

hud.draw(g);

}

// Метод обработки нажатия клавиши

public void keyPressed(int k) {

if(k == KeyEvent.VK\_LEFT) player.setLeft(true);

if(k == KeyEvent.VK\_RIGHT) player.setRight(true);

if(k == KeyEvent.VK\_UP) player.setUp(true);

if(k == KeyEvent.VK\_DOWN) player.setDown(true);

if(k == KeyEvent.VK\_W) player.setJumping(true);

if(k == KeyEvent.VK\_E) player.setGliding(true);

if(k == KeyEvent.VK\_R) player.setScratching();

if(k == KeyEvent.VK\_F) player.setFiring();

}

// Метод обработки отпускания клавиши

public void keyReleased(int k) {

if(k == KeyEvent.VK\_LEFT) player.setLeft(false);

if(k == KeyEvent.VK\_RIGHT) player.setRight(false);

if(k == KeyEvent.VK\_UP) player.setUp(false);

if(k == KeyEvent.VK\_DOWN) player.setDown(false);

if(k == KeyEvent.VK\_W) player.setJumping(false);

if(k == KeyEvent.VK\_E) player.setGliding(false);

}

}

package GameState;

import TileMap.Background;

import java.awt.\*;

import java.awt.event.KeyEvent;

/\*\*

\* @author Ильин Тимур

\*/

public class MenuState extends GameState {

private Background bg; // Фон меню

private int currentChoice = 0; // Текущий выбор

private String[] options = { // Варианты меню

"Start",

"Help",

"Quit"

};

private Color titleColor; // Цвет заголовка

private Font titleFont; // Шрифт заголовка

private Font font; // Шрифт текста

// Конструктор

public MenuState(GameStateManager gsm) {

this.gsm = gsm; // Устанавливаем менеджер состояний

try {

// Загрузка фона

bg = new Background("/Backgrounds/menubg.gif", 1);

bg.setVector(-0.1, 0); // Установка скорости движения фона

// Установка цвета и шрифта заголовка

titleColor = new Color(128, 0, 0);

titleFont = new Font("Century Gothic", Font.PLAIN, 28);

// Установка шрифта текста

font = new Font("Arial", Font.PLAIN, 12);

} catch (Exception e) {

e.printStackTrace();

}

}

// Инициализация состояния

public void init() {}

// Обновление состояния

public void update() {

bg.update(); // Обновление фона

}

// Отрисовка состояния

public void draw(Graphics2D g) {

// Отрисовка фона

bg.draw(g);

// Отрисовка заголовка

g.setColor(titleColor);

g.setFont(titleFont);

g.drawString("Dragon Tale", 80, 70);

// Отрисовка вариантов меню

g.setFont(font);

for (int i = 0; i < options.length; i++) {

if (i == currentChoice) {

g.setColor(Color.BLACK);

} else {

g.setColor(Color.RED);

}

g.drawString(options[i], 145, 140 + i \* 15);

}

}

// Метод выбора варианта меню

private void select() {

// Обработка выбора

if (currentChoice == 0) {

gsm.setState(GameStateManager.LEVEL1STATE); // Переход к игровому уровню

}

if (currentChoice == 1) {

// Вывод помощи (возможно, реализуется позднее)

}

if (currentChoice == 2) {

System.exit(0); // Завершение приложения

}

}

// Обработка нажатия клавиш

public void keyPressed(int k) {

// Обработка нажатия клавиш для перемещения по меню и выбора пункта

if (k == KeyEvent.VK\_ENTER) {

select(); // Вызов метода выбора

}

if (k == KeyEvent.VK\_UP) {

currentChoice--; // Перемещение вверх

if (currentChoice == -1) {

currentChoice = options.length - 1; // Циклическое перемещение вверх

}

}

if (k == KeyEvent.VK\_DOWN) {

currentChoice++; // Перемещение вниз

if (currentChoice == options.length) {

currentChoice = 0; // Циклическое перемещение вниз

}

}

}

// Обработка отпускания клавиш (в данном состоянии не используется)

public void keyReleased(int k) {}

}

package Entity.Enemies;

import Entity.\*;

import TileMap.TileMap;

import java.awt.image.BufferedImage;

import java.awt.Graphics2D;

import javax.imageio.ImageIO;

public class Slugger extends Enemy {

private BufferedImage[] sprites; // Массив спрайтов

public Slugger(TileMap tm) {

super(tm); // Вызов конструктора суперкласса

moveSpeed = 0.3; // Скорость перемещения

maxSpeed = 0.3; // Максимальная скорость

fallSpeed = 0.2; // Скорость падения

maxFallSpeed = 10.0; // Максимальная скорость падения

width = 30; // Ширина врага

height = 30; // Высота врага

cwidth = 20; // Ширина коллизии врага

cheight = 20; // Высота коллизии врага

health = maxHealth = 2; // Здоровье врага

damage = 1; // Урон врага

// Загрузка спрайтов

try {

BufferedImage spritesheet = ImageIO.read(

getClass().getResourceAsStream(

"/Sprites/Enemies/slugger.gif"

)

);

sprites = new BufferedImage[3]; // Массив для спрайтов

for(int i = 0; i < sprites.length; i++) {

sprites[i] = spritesheet.getSubimage(

i \* width,

0,

width,

height

);

}

}

catch(Exception e) {

e.printStackTrace(); // Обработка исключений

}

animation = new Animation(); // Создание анимации

animation.setFrames(sprites); // Установка спрайтов для анимации

animation.setDelay(300); // Установка задержки между кадрами анимации

right = true; // Начальное направление вправо

facingRight = true; // Начальное направление вправо

}

private void getNextPosition() {

// Движение влево

if(left) {

dx -= moveSpeed;

if(dx < -maxSpeed) {

dx = -maxSpeed;

}

}

// Движение вправо

else if(right) {

dx += moveSpeed;

if(dx > maxSpeed) {

dx = maxSpeed;

}

}

// Падение

if(falling) {

dy += fallSpeed;

}

}

public void update() {

// Обновление позиции

getNextPosition();

checkTileMapCollision();

setPosition(xtemp, ytemp);

// Проверка на нахождение в состоянии тряски

if(flinching) {

long elapsed = (System.nanoTime() - flinchTimer) / 1000000;

if(elapsed > 400) {

flinching = false;

}

}

// Если враг ударился о стену, меняем направление

if(right && dx == 0) {

right = false;

left = true;

facingRight = false;

}

else if(left && dx == 0) {

right = true;

left = false;

facingRight = true;

}

// Обновление анимации

animation.update();

}

public void draw(Graphics2D g) {

// Установка позиции на карте

setMapPosition();

// Вызов метода отрисовки суперкласса

super.draw(g);

}

}

package Entity;

import java.awt.image.BufferedImage;

public class Animation {

private BufferedImage[] frames; // Массив кадров анимации

private int currentFrame; // Текущий кадр

private long startTime; // Время начала анимации

private long delay; // Задержка между кадрами

private boolean playedOnce; // Флаг, указывающий на то, была ли анимация воспроизведена один раз

public Animation() {

playedOnce = false; // Инициализация флага

}

public void setFrames(BufferedImage[] frames) {

this.frames = frames; // Установка массива кадров

currentFrame = 0; // Установка начального кадра

startTime = System.nanoTime(); // Получение текущего времени

playedOnce = false; // Сброс флага

}

public void setDelay(long d) { delay = d; } // Установка задержки между кадрами

public void setFrame(int i) { currentFrame = i; } // Установка текущего кадра

public void update() {

if(delay == -1) return; // Если задержка равна -1, выход

long elapsed = (System.nanoTime() - startTime) / 1000000; // Просчитываем прошедшее время

if(elapsed > delay) { // Если прошедшее время превышает задержку

currentFrame++; // Переходим к следующему кадру

startTime = System.nanoTime(); // Обновляем время начала анимации

}

if(currentFrame == frames.length) { // Если достигнут конец массива кадров

currentFrame = 0; // Возвращаемся к начальному кадру

playedOnce = true; // Устанавливаем флаг, что анимация воспроизведена один раз

}

}

public int getFrame() { return currentFrame; } // Получение текущего кадра

public BufferedImage getImage() { return frames[currentFrame]; } // Получение изображения текущего кадра

public boolean hasPlayedOnce() { return playedOnce; } // Проверка, была ли анимация воспроизведена один раз

}

package Entity;

import TileMap.TileMap;

public class Enemy extends MapObject {

protected int health; // Здоровье врага

protected int maxHealth; // Максимальное здоровье врага

protected boolean dead; // Флаг, указывающий на смерть врага

protected int damage; // Урон, который может нанести враг

protected boolean flinching; // Флаг, указывающий на "подпрыгивание" при получении урона

protected long flinchTimer; // Время начала "подпрыгивания" при получении урона

public Enemy(TileMap tm) {

super(tm); // Вызов конструктора родительского класса

}

public boolean isDead() { return dead; } // Метод для проверки, мертв ли враг

public int getDamage() { return damage; } // Метод для получения урона, наносимого врагом

public void hit(int damage) {

if(dead || flinching) return; // Если враг мертв или уже "подпрыгивает", выход

health -= damage; // Уменьшаем здоровье врага на значение урона

if(health < 0) health = 0; // Если здоровье стало меньше нуля, устанавливаем его в ноль

if(health == 0) dead = true; // Если здоровье равно нулю, враг мертв

flinching = true; // Устанавливаем флаг "подпрыгивания"

flinchTimer = System.nanoTime(); // Устанавливаем время начала "подпрыгивания"

}

public void update() {} // Обновление состояния врага

}

package Entity;

import java.awt.Graphics2D;

import java.awt.image.BufferedImage;

import javax.imageio.ImageIO;

public class Explosion {

private int x; // Позиция по оси X

private int y; // Позиция по оси Y

private int xmap; // Позиция на карте по оси X

private int ymap; // Позиция на карте по оси Y

private int width; // Ширина взрыва

private int height; // Высота взрыва

private Animation animation; // Анимация взрыва

private BufferedImage[] sprites; // Спрайты для анимации взрыва

private boolean remove; // Флаг, указывающий на необходимость удаления взрыва

public Explosion(int x, int y) {

this.x = x; // Установка начальной позиции по оси X

this.y = y; // Установка начальной позиции по оси Y

width = 30; // Установка ширины взрыва

height = 30; // Установка высоты взрыва

try {

BufferedImage spritesheet = ImageIO.read(

getClass().getResourceAsStream(

"/Sprites/Enemies/explosion.gif"

)

);

sprites = new BufferedImage[6]; // Создание массива для спрайтов взрыва

for(int i = 0; i < sprites.length; i++) {

sprites[i] = spritesheet.getSubimage(

i \* width,

0,

width,

height

); // Загрузка спрайтов взрыва из спрайтшита

}

} catch(Exception e) {

e.printStackTrace(); // Вывод информации об ошибке, если что-то пошло не так

}

animation = new Animation(); // Создание анимации взрыва

animation.setFrames(sprites); // Установка спрайтов для анимации

animation.setDelay(70); // Установка задержки между кадрами анимации

}

public void update() {

animation.update(); // Обновление анимации взрыва

if(animation.hasPlayedOnce()) {

remove = true; // Установка флага на удаление взрыва, если анимация проиграна один раз

}

}

public boolean shouldRemove() { return remove; } // Метод, возвращающий флаг удаления взрыва

public void setMapPosition(int x, int y) {

xmap = x; // Установка позиции на карте по оси X

ymap = y; // Установка позиции на карте по оси Y

}

public void draw(Graphics2D g) {

g.drawImage(

animation.getImage(),

x + xmap - width / 2,

y + ymap - height / 2,

null

); // Отрисовка текущего кадра анимации взрыва

}

}

package Entity;

import TileMap.TileMap;

import java.awt.\*;

import java.awt.image.BufferedImage;

import javax.imageio.ImageIO;

public class FireBall extends MapObject {

private boolean hit; // Флаг, указывающий, попала ли огненная шаров во что-то

private boolean remove; // Флаг, указывающий, что необходимо удалить огненный шар

private BufferedImage[] sprites; // Спрайты для обычного состояния огненного шара

private BufferedImage[] hitSprites; // Спрайты для состояния огненного шара при попадании

public FireBall(TileMap tm, boolean right) {

super(tm); // Вызов конструктора родительского класса

facingRight = right; // Установка направления движения огненного шара

moveSpeed = 3.8; // Установка скорости движения огненного шара

if(right) dx = moveSpeed; // Установка скорости по оси X в зависимости от направления

else dx = -moveSpeed;

width = 30; // Установка ширины огненного шара

height = 30; // Установка высоты огненного шара

cwidth = 14; // Установка ширины столкновения огненного шара

cheight = 14; // Установка высоты столкновения огненного шара

// Загрузка спрайтов огненного шара

try {

BufferedImage spritesheet = ImageIO.read(

getClass().getResourceAsStream(

"/Sprites/Player/fireball.gif"

)

);

sprites = new BufferedImage[4]; // Массив для спрайтов обычного состояния

// Загрузка спрайтов для обычного состояния

for(int i = 0; i < sprites.length; i++) {

sprites[i] = spritesheet.getSubimage(

i \* width,

0,

width,

height

);

}

hitSprites = new BufferedImage[3]; // Массив для спрайтов при попадании

// Загрузка спрайтов для состояния при попадании

for(int i = 0; i < hitSprites.length; i++) {

hitSprites[i] = spritesheet.getSubimage(

i \* width,

height,

width,

height

);

}

animation = new Animation(); // Создание анимации огненного шара

animation.setFrames(sprites); // Установка спрайтов для анимации

animation.setDelay(70); // Установка задержки между кадрами анимации

} catch(Exception e) {

e.printStackTrace(); // Вывод информации об ошибке, если что-то пошло не так

}

}

public void setHit() {

if(hit) return; // Если огненный шар уже попал, выходим из метода

hit = true; // Устанавливаем флаг попадания

animation.setFrames(hitSprites); // Устанавливаем спрайты для анимации при попадании

animation.setDelay(70); // Установка задержки между кадрами анимации

dx = 0; // Устанавливаем скорость по оси X равной 0

}

public boolean shouldRemove() { return remove; } // Метод, возвращающий флаг удаления огненного шара

public void update() {

checkTileMapCollision(); // Проверка столкновений с тайлами

setPosition(xtemp, ytemp); // Установка новой позиции огненного шара

if(dx == 0 && !hit) {

setHit(); // Если огненный шар стоит на месте и еще не попал, вызываем метод попадания

}

animation.update(); // Обновление анимации огненного шара

if(hit && animation.hasPlayedOnce()) {

remove = true; // Установка флага на удаление огненного шара, если анимация при попадании проиграна один раз

}

}

public void draw(Graphics2D g) {

setMapPosition(); // Установка позиции на карте

super.draw(g); // Вызов метода отрисовки из родительского класса

}

}

package Entity;

import java.awt.\*;

import java.awt.image.BufferedImage;

import javax.imageio.ImageIO;

public class HUD {

private Player player; // Ссылка на игрока

private BufferedImage image; // Изображение для отображения HUD

private Font font; // Шрифт для текста на HUD

public HUD(Player p) {

player = p; // Установка ссылки на игрока

try {

// Загрузка изображения HUD и создание шрифта

image = ImageIO.read(

getClass().getResourceAsStream(

"/HUD/hud.gif"

)

);

font = new Font("Arial", Font.PLAIN, 14);

} catch(Exception e) {

e.printStackTrace(); // Вывод информации об ошибке, если что-то пошло не так

}

}

public void draw(Graphics2D g) {

// Отрисовка изображения HUD

g.drawImage(image, 0, 10, null);

g.setFont(font); // Установка шрифта для текста

g.setColor(Color.WHITE); // Установка белого цвета для текста

// Отображение текущего здоровья игрока и его максимального здоровья

g.drawString(

player.getHealth() + "/" + player.getMaxHealth(),

30,

25

);

// Отображение текущего уровня огня игрока и его максимального уровня огня

g.drawString(

player.getFire() / 100 + "/" + player.getMaxFire() / 100,

30,

45

);

}

}

package Entity;

import Main.GamePanel;

import TileMap.TileMap;

import TileMap.Tile;

import java.awt.Rectangle;

public abstract class MapObject {

// Переменные для работы с тайлами

protected TileMap tileMap; // Карта тайлов

protected int tileSize; // Размер тайла

protected double xmap; // Координата x на карте

protected double ymap; // Координата y на карте

// Переменные позиции и вектора

protected double x; // Координата x

protected double y; // Координата y

protected double dx; // Вектор скорости по x

protected double dy; // Вектор скорости по y

// Размеры

protected int width; // Ширина объекта

protected int height; // Высота объекта

// Коллизионные коробки

protected int cwidth; // Ширина коллизионной коробки

protected int cheight; // Высота коллизионной коробки

// Коллизия

protected int currRow; // Текущий ряд

protected int currCol; // Текущий столбец

protected double xdest; // Целевая координата x

protected double ydest; // Целевая координата y

protected double xtemp; // Временная координата x

protected double ytemp; // Временная координата y

protected boolean topLeft; // Верхний левый угол коллизионной коробки

protected boolean topRight; // Верхний правый угол коллизионной коробки

protected boolean bottomLeft; // Нижний левый угол коллизионной коробки

protected boolean bottomRight; // Нижний правый угол коллизионной коробки

// Анимация

protected Animation animation; // Анимация

protected int currentAction; // Текущее действие

protected int previousAction; // Предыдущее действие

protected boolean facingRight; // Направление вправо

// Движение

protected boolean left; // Движение влево

protected boolean right; // Движение вправо

protected boolean up; // Движение вверх

protected boolean down; // Движение вниз

protected boolean jumping; // Прыжок

protected boolean falling; // Падение

// Атрибуты движения

protected double moveSpeed; // Скорость движения

protected double maxSpeed; // Максимальная скорость

protected double stopSpeed; // Скорость остановки

protected double fallSpeed; // Скорость падения

protected double maxFallSpeed; // Максимальная скорость падения

protected double jumpStart; // Начальная скорость прыжка

protected double stopJumpSpeed; // Скорость остановки прыжка

// Конструктор

public MapObject(TileMap tm) {

tileMap = tm; // Установка карты тайлов

tileSize = tm.getTileSize(); // Получение размера тайла

}

// Проверка пересечения с другим объектом

public boolean intersects(MapObject o) {

Rectangle r1 = getRectangle(); // Получение границ текущего объекта

Rectangle r2 = o.getRectangle(); // Получение границ объекта o

return r1.intersects(r2); // Возвращает true, если границы пересекаются

}

// Получение прямоугольника коллизии

public Rectangle getRectangle() {

return new Rectangle(

(int)x - cwidth,

(int)y - cheight,

cwidth,

cheight

);

}

// Вычисление углов коллизионной коробки

public void calculateCorners(double x, double y) {

int leftTile = (int)(x - cwidth / 2) / tileSize;

int rightTile = (int)(x + cwidth / 2 - 1) / tileSize;

int topTile = (int)(y - cheight / 2) / tileSize;

int bottomTile = (int)(y + cheight / 2 - 1) / tileSize;

int tl = tileMap.getType(topTile, leftTile);

int tr = tileMap.getType(topTile, rightTile);

int bl = tileMap.getType(bottomTile, leftTile);

int br = tileMap.getType(bottomTile, rightTile);

topLeft = tl == Tile.BLOCKED;

topRight = tr == Tile.BLOCKED;

bottomLeft = bl == Tile.BLOCKED;

bottomRight = br == Tile.BLOCKED;

}

// Проверка коллизии с картой тайлов

public void checkTileMapCollision() {

currCol = (int)x / tileSize;

currRow = (int)y / tileSize;

xdest = x + dx;

ydest = y + dy;

xtemp = x;

ytemp = y;

calculateCorners(x, ydest);

if(dy < 0) {

if(topLeft || topRight) {

dy = 0;

ytemp = currRow \* tileSize + cheight / 2;

}

else {

ytemp += dy;

}

}

if(dy > 0) {

if(bottomLeft || bottomRight) {

dy = 0;

falling = false;

ytemp = (currRow + 1) \* tileSize - cheight / 2;

}

else {

ytemp += dy;

}

}

calculateCorners(xdest, y);

if(dx < 0) {

if(topLeft || bottomLeft) {

dx = 0;

xtemp = currCol \* tileSize + cwidth / 2;

}

else {

xtemp += dx;

}

}

if(dx > 0) {

if(topRight || bottomRight) {

dx = 0;

xtemp = (currCol + 1) \* tileSize - cwidth / 2;

}

else {

xtemp += dx;

}

}

if(!falling) {

calculateCorners(x, ydest + 1);

if(!bottomLeft && !bottomRight) {

falling = true;

}

}

}

// Получение координаты x

public int getx() { return (int)x; }

// Получение координаты y

public int gety() { return (int)y; }

// Получение ширины объекта

public int getWidth() { return width; }

// Получение высоты объекта

public int getHeight() { return height; }

// Получение ширины коллизионной коробки

public int getCWidth() { return cwidth; }

// Получение высоты коллизионной коробки

public int getCHeight() { return cheight; }

// Установка позиции

public void setPosition(double x, double y) {

this.x = x;

this.y = y;

}

// Установка вектора

public void setVector(double dx, double dy) {

this.dx = dx;

this.dy = dy;

}

// Установка позиции карты

public void setMapPosition() {

xmap = tileMap.getx();

ymap = tileMap.gety();

}

// Установка движения влево

public void setLeft(boolean b) { left = b; }

// Установка движения вправо

public void setRight(boolean b) { right = b; }

// Установка движения вверх

public void setUp(boolean b) { up = b; }

// Установка движения вниз

public void setDown(boolean b) { down = b; }

// Установка состояния прыжка

public void setJumping(boolean b) { jumping = b; }

// Проверка находится ли объект за пределами экрана

public boolean notOnScreen() {

return x + xmap + width < 0 ||

x + xmap - width > GamePanel.WIDTH ||

y + ymap + height < 0 ||

y + ymap - height > GamePanel.HEIGHT;

}

// Отрисовка объекта

public void draw(java.awt.Graphics2D g) {

if(facingRight) {

g.drawImage(

animation.getImage(),

(int)(x + xmap - width / 2),

(int)(y + ymap - height / 2),

null

);

} else {

g.drawImage(

animation.getImage(),

(int)(x + xmap - width / 2 + width),

(int)(y + ymap - height / 2),

-width,

height,

null

);

}

}

}

package Entity;

import TileMap.\*;

import Audio.AudioPlayer;

import java.util.ArrayList;

import javax.imageio.ImageIO;

import java.awt.\*;

import java.awt.image.BufferedImage;

import java.util.HashMap;

public class Player extends MapObject {

// Переменные игрока

private int health; // Здоровье

private int maxHealth; // Максимальное здоровье

private int fire; // Огонь (энергия)

private int maxFire; // Максимальное значение огня

private boolean dead; // Мертв ли игрок

private boolean flinching; // Подвержен ли игрок "флинчу"

private long flinchTimer; // Таймер для флинча

// Фаерболл

private boolean firing; // Стрельба

private int fireCost; // Стоимость стрельбы

private int fireBallDamage; // Урон от фаерболла

private ArrayList<FireBall> fireBalls; // Массив с фаерболлами

// Скретч

private boolean scratching; // Скретч

private int scratchDamage; // Урон от скретча

private int scratchRange; // Радиус скретча

// Парение

private boolean gliding; // Парение

// Анимации

private ArrayList<BufferedImage[]> sprites; // Спрайты

private final int[] numFrames = {

2, 8, 1, 2, 4, 2, 5

}; // Количество кадров для каждого действия

// Действия анимации

private static final int IDLE = 0;

private static final int WALKING = 1;

private static final int JUMPING = 2;

private static final int FALLING = 3;

private static final int GLIDING = 4;

private static final int FIREBALL = 5;

private static final int SCRATCHING = 6;

// Звуковые эффекты

private HashMap<String, AudioPlayer> sfx; // Мап для звуковых эффектов

// Конструктор

public Player(TileMap tm) {

super(tm);

width = 30;

height = 30;

cwidth = 20;

cheight = 20;

moveSpeed = 0.3;

maxSpeed = 1.6;

stopSpeed = 0.4;

fallSpeed = 0.15;

maxFallSpeed = 4.0;

jumpStart = -4.8;

stopJumpSpeed = 0.3;

facingRight = true;

health = maxHealth = 5;

fire = maxFire = 2500;

fireCost = 200;

fireBallDamage = 5;

fireBalls = new ArrayList<FireBall>();

scratchDamage = 8;

scratchRange = 40;

// Загрузка спрайтов

try {

BufferedImage spritesheet = ImageIO.read(

getClass().getResourceAsStream(

"/Sprites/Player/playersprites.gif"

)

);

sprites = new ArrayList<BufferedImage[]>();

for(int i = 0; i < 7; i++) {

BufferedImage[] bi =

new BufferedImage[numFrames[i]];

for(int j = 0; j < numFrames[i]; j++) {

if(i != SCRATCHING) {

bi[j] = spritesheet.getSubimage(

j \* width,

i \* height,

width,

height

);

} else {

bi[j] = spritesheet.getSubimage(

j \* width \* 2,

i \* height,

width \* 2,

height

);

}

}

sprites.add(bi);

}

} catch(Exception e) {

e.printStackTrace();

}

// Установка начальной анимации

animation = new Animation();

currentAction = IDLE;

animation.setFrames(sprites.get(IDLE));

animation.setDelay(400);

// Инициализация звуковых эффектов

sfx = new HashMap<String, AudioPlayer>();

sfx.put("jump", new AudioPlayer("/Resources/SFX/jump.mp3"));

sfx.put("scratch", new AudioPlayer("/Resources/SFX/scratch.mp3"));

}

// Получение здоровья

public int getHealth() { return health; }

// Получение максимального здоровья

public int getMaxHealth() { return maxHealth; }

// Получение уровня огня

public int getFire() { return fire; }

// Получение максимального уровня огня

public int getMaxFire() { return maxFire; }

// Установка состояния стрельбы

public void setFiring() { firing = true; }

// Установка состояния скретча

public void setScratching() { scratching = true; }

// Установка состояния парения

public void setGliding(boolean b) { gliding = b; }

// Проверка атаки

public void checkAttack(ArrayList<Enemy> enemies) {

// Перебор врагов

for(int i = 0; i < enemies.size(); i++) {

Enemy e = enemies.get(i);

// Скретч

if(scratching) {

if(facingRight) {

if(

e.getx() > x &&

e.getx() < x + scratchRange &&

e.gety() > y - height / 2 &&

e.gety() < y + height / 2

) {

e.hit(scratchDamage);

}

} else {

if(

e.getx() < x &&

e.getx() > x - scratchRange &&

e.gety() > y - height / 2 &&

e.gety() < y + height / 2

) {

e.hit(scratchDamage);

}

}

}

// Фаерболлы

for(int j = 0; j < fireBalls.size(); j++) {

if(fireBalls.get(j).intersects(e)) {

e.hit(fireBallDamage);

fireBalls.get(j).setHit();

break;

}

}

// Проверка столкновения с врагом

if(intersects(e)) {

hit(e.getDamage());

}

}

}

// Попадание

public void hit(int damage) {

if(flinching) return;

health -= damage;

if(health < 0) health = 0;

if(health == 0) dead = true;

flinching = true;

flinchTimer = System.nanoTime();

}

// Получение следующей позиции

private void getNextPosition() {

// Движение

if(left) {

dx -= moveSpeed;

if(dx < -maxSpeed) {

dx = -maxSpeed;

}

} else if(right) {

dx += moveSpeed;

if(dx > maxSpeed) {

dx = maxSpeed;

}

} else {

if(dx > 0) {

dx -= stopSpeed;

if(dx < 0) {

dx = 0;

}

} else if(dx < 0) {

dx += stopSpeed;

if(dx > 0) {

dx = 0;

}

}

}

// Невозможность движения во время атаки, кроме прыжка

if((currentAction == SCRATCHING || currentAction == FIREBALL) &&

!(jumping || falling)) {

dx = 0;

}

// Прыжок

if(jumping && !falling) {

sfx.get("jump").play();

dy = jumpStart;

falling = true;

}

// Падение

if(falling) {

if(dy > 0 && gliding) dy += fallSpeed \* 0.1;

else dy += fallSpeed;

if(dy > 0) jumping = false;

if(dy < 0 && !jumping) dy += stopJumpSpeed;

if(dy > maxFallSpeed) dy = maxFallSpeed;

}

}

// Обновление

public void update() {

// Обновление позиции

getNextPosition();

checkTileMapCollision();

setPosition(xtemp, ytemp);

// Проверка окончания атаки

if(currentAction == SCRATCHING) {

if(animation.hasPlayedOnce()) scratching = false;

}

if(currentAction == FIREBALL) {

if(animation.hasPlayedOnce()) firing = false;

}

// Атака фаерболлом

fire += 1;

if(fire > maxFire) fire = maxFire;

if(firing && currentAction != FIREBALL) {

if(fire > fireCost) {

fire -= fireCost;

FireBall fb = new FireBall(tileMap, facingRight);

fb.setPosition(x, y);

fireBalls.add(fb);

}

}

// Обновление фаерболлов

for(int i = 0; i < fireBalls.size(); i++) {

fireBalls.get(i).update();

if(fireBalls.get(i).shouldRemove()) {

fireBalls.remove(i);

i--;

}

}

// Проверка завершения флинча

if(flinching) {

long elapsed =

(System.nanoTime() - flinchTimer) / 1000000;

if(elapsed > 1000) {

flinching = false;

}

}

// Установка анимации

if(scratching) {

if(currentAction != SCRATCHING) {

sfx.get("scratch").play();

currentAction = SCRATCHING;

animation.setFrames(sprites.get(SCRATCHING));

animation.setDelay(50);

width = 60;

}

} else if(firing) {

if(currentAction != FIREBALL) {

currentAction = FIREBALL;

animation.setFrames(sprites.get(FIREBALL));

animation.setDelay(100);

width = 30;

}

} else if(dy > 0) {

if(gliding) {

if(currentAction != GLIDING) {

currentAction = GLIDING;

animation.setFrames(sprites.get(GLIDING));

animation.setDelay(100);

width = 30;

}

} else if(currentAction != FALLING) {

currentAction = FALLING;

animation.setFrames(sprites.get(FALLING));

animation.setDelay(100);

width = 30;

}

} else if(dy < 0) {

if(currentAction != JUMPING) {

currentAction = JUMPING;

animation.setFrames(sprites.get(JUMPING));

animation.setDelay(-1);

width = 30;

}

} else if(left || right) {

if(currentAction != WALKING) {

currentAction = WALKING;

animation.setFrames(sprites.get(WALKING));

animation.setDelay(40);

width = 30;

}

} else {

if(currentAction != IDLE) {

currentAction = IDLE;

animation.setFrames(sprites.get(IDLE));

animation.setDelay(400);

width = 30;

}

}

// Обновление анимации

animation.update();

// Установка направления

if(currentAction != SCRATCHING && currentAction != FIREBALL) {

if(right) facingRight = true;

if(left) facingRight = false;

}

}

// Отрисовка

public void draw(Graphics2D g) {

setMapPosition();

// Отрисовка фаерболлов

for(int i = 0; i < fireBalls.size(); i++) {

fireBalls.get(i).draw(g);

}

// Отрисовка игрока

if(flinching) {

long elapsed =

(System.nanoTime() - flinchTimer) / 1000000;

if(elapsed / 100 % 2 == 0) {

return;

}

}

super.draw(g);

}

}

package Audio;

import javax.sound.sampled.\*;

// Класс для воспроизведения аудиофайлов

public class AudioPlayer {

private Clip clip;

// Конструктор, принимающий путь к аудиофайлу

public AudioPlayer(String s) {

try {

// Загрузка аудиофайла

AudioInputStream ais =

AudioSystem.getAudioInputStream(

getClass().getResourceAsStream(s)

);

AudioFormat baseFormat = ais.getFormat();

AudioFormat decodeFormat = new AudioFormat(

AudioFormat.Encoding.PCM\_SIGNED,

baseFormat.getSampleRate(),

16,

baseFormat.getChannels(),

baseFormat.getChannels() \* 2,

baseFormat.getSampleRate(),

false

);

AudioInputStream dais =

AudioSystem.getAudioInputStream(

decodeFormat, ais);

// Открытие аудиоклипа

clip = AudioSystem.getClip();

clip.open(dais);

} catch (Exception e) {

e.printStackTrace();

}

}

// Метод для воспроизведения аудио

public void play() {

if (clip == null) return;

stop();

clip.setFramePosition(0);

clip.start();

}

// Метод для остановки воспроизведения аудио

public void stop() {

if (clip.isRunning()) clip.stop();

}

// Метод для закрытия аудиофайла

public void close() {

stop();

clip.close();

}

}