МИНИСТЕРСТВО НАУКИ И ОБРАЗОВАНИЯ РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Уральский государственный горный университет»

Кафедра Информатики

Курсовой проект

По дисциплине «Программная инженерия»

На тему «Q\*bert»

Выполнил студент

2 курса, группы АУБП-23-1

Порошин А. Р.

Проверила:

ст. преп. каф. информатики

Волкова Е.А.

Екатеринбург

2024

|  |  |
| --- | --- |
| 1.Постановка задачи........................................................................................ | 3 |
| 1.1. Формулировка задания............................................................................ | 3 |
| 1.2. Алгоритмическое разрешение задачи.................................................... | 4 |
| 1.3. Контрольные примеры............................................................................. | 6 |
| 2. Решение задачи............................................................................................ | 8 |
| 2.1. Выбор средств реализации...................................................................... | 8 |
| 2.2. Описание основных классов................................................................... | 9 |
| 2.3. Интерфейс приложения........................................................................... | 49 |
| 3.Тестирование приложения.......................................................................... | 55 |
| 3.1. Контрольный робот.................................................................................. | 55 |
| 3.2. Работа приложения на контрольных тестах.......................................... | 59 |
| 3.3. Результаты работы .................................................................................. | 59 |

1.Постановка задачи

1.1. Формулировка задания

«Q\*bert»

Реализовать логическую игру «Q\*bert». В игре Q\*bert игрок гоняет одноимённого персонажа по изометрической пирамидальной структуре из трёхцветных кубиков. Цель Q\*bert – прыгая по верху этих кубиков, перекрасить их в заданный цвет, прыгая по ним.

Пример игрового поля:

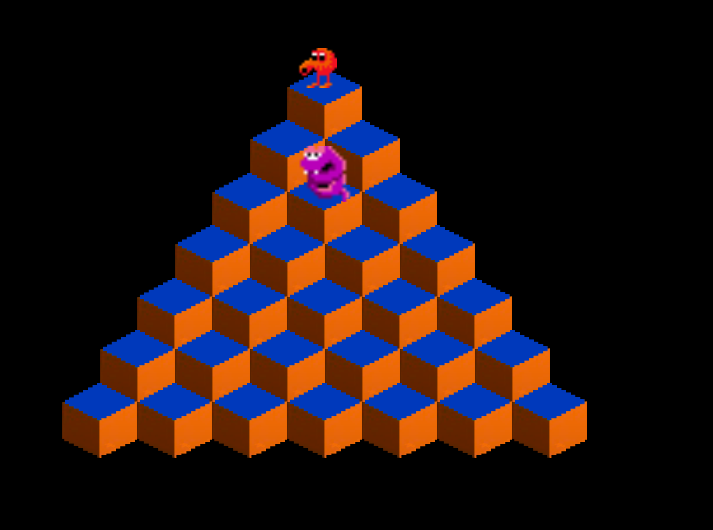


Рисунок 1. Пример карты

Постановка задачи:

Реализовать алгоритм игры «Q\*bert».

1.2. Алгоритмическое решение задачи

Алгоритмическое решение игры Q\*bert

Алгоритмическое решение для игры " Q\*bert ":

Запуск игры начинается в классе Main.java:

Создается главное окно игры (JFrame)

Настраиваются параметры окна (размер, заголовок, возможность изменения размера)

Создается игровая панель (GamePanel)

Запускается игровой поток

Инициализация GamePanel:

Устанавливаются основные параметры экрана:

Размер тайла (15 пикселей)

Масштаб (3x)

Размеры экрана (768x576 пикселей)

FPS (60 кадров в секунду)

Инициализируются менеджеры:

SoundManager (для звуков)

AnimationManager (для анимаций)

Загружаются изображения для разных экранов:

Стартовый экран

Экран выбора режима

Экран уровня

Экран смерти

Создается обработчик клавиш (KeyHandler)

Создается игрок (Player)

Создается карта (Map)

Создается бот (GameBot) для режима игры с ботом

Инициализация игрока (Player):

Устанавливаются начальные значения:

Позиция на карте (0,0)

Скорость движения

Направление

Состояние прыжка

Загружаются спрайты для разных направлений движения

Инициализация карты (Map):

Создается двумерный массив для хранения тайлов

Инициализируется массив для отслеживания окрашенных тайлов

Устанавливаются параметры отображения:

Размер куба

Горизонтальные и вертикальные интервалы

Смещения по X и Y

Инициализация бота (GameBot):

Устанавливаются начальные координаты

Создается массив для отслеживания посещенных тайлов

Инициализируется путь к следующему тайлу

Загружаются спрайты для разных направлений движения

Инициализация монстров:

Для первого уровня создается один монстр

Для последующих уровней количество монстров увеличивается

Каждый монстр получает начальную позицию и параметры движения

Загрузка сохраненного прогресса:

Проверяется наличие файла сохранения

Если файл существует, загружается текущий уровень

После инициализации игра переходит в состояние START\_SCREEN, где игрок может:

Выбрать режим игры (за игрока или за бота)

Начать игру

Просмотреть текущий уровень

Управлять персонажем (в режиме игры за игрока) или наблюдать за ботом (в режиме игры за бота)

1.3. Контрольные примеры

В качестве контрольного примера возьмем матрицу:

{0,},

            {0, 0},

            {0, 0, 0},

            {0, 0, 0, 0},

            {0, 0, 0, 0, 0},

            {0, 0, 0, 0, 0, 0},

            {0, 0, 0, 0, 1, 0, 0}

Когда игрок начинает движение из 1,1 в 1,2 то плитка закрашивается,

{0,},

            {1, 0},

            {0, 0, 0},

            {0, 0, 0, 0},

            {0, 0, 0, 0, 0},

            {0, 0, 0, 0, 0, 0},

            {0, 0, 0, 0, 0, 0, 0}

{1,},

            {1, 1},

            {1, 1, 1},

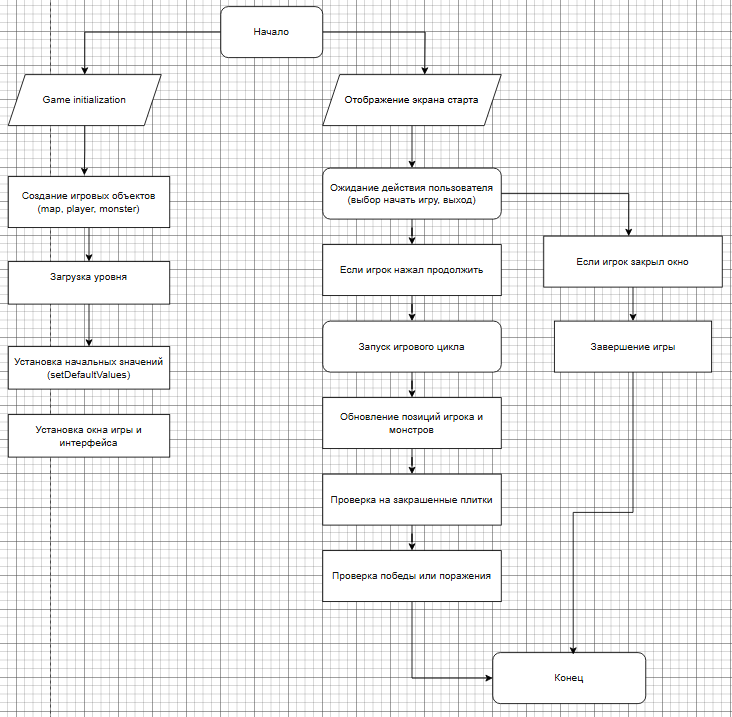
            {1, 1, 1, 1},

            {1, 1, 1, 1, 1},

            {1, 1, 1, 1, 1, 1},

            {1, 1, 1, 1, 1, 1, 1}

Таким образом поле матрицы заполняется и если на ней полностью единицы, то игрок победил

Рисунок 2. ER-диаграмма работы игры

2.Решение задачи

2.1. Для решения данной задачи был выбран высокоуровневый язык программирования Java, так как он обладает множеством встроенных полезных функций и коллекций, упрощающих работу с программой. Альтернативными вариантами могли бы быть C# или C++, которые также способны справиться с поставленной задачей.

Java — это объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems, позднее приобретённой Oracle. Java-приложения обычно компилируются в байт-код, который может выполняться на любой виртуальной Java-машине (JVM), независимо от архитектуры компьютера.

Java-программы транслируются в байт-код, который исполняется JVM — программой, обрабатывающей байт-код и передающей инструкции оборудованию, действуя как интерпретатор. Преимуществом такого подхода является полная независимость байт-кода от операционной системы и оборудования, что позволяет запускать Java-приложения на любом устройстве с поддержкой JVM. Кроме того, Java обладает гибкой системой безопасности, так как выполнение программы полностью контролируется виртуальной машиной. Любые действия, выходящие за пределы установленных полномочий программы (например, несанкционированный доступ к данным или соединение с другим компьютером), немедленно прерываются.

.

2.2. Описание основных классов

При решении задачи было разработано 10 классов показанные на

диаграмме (рисунок 3).

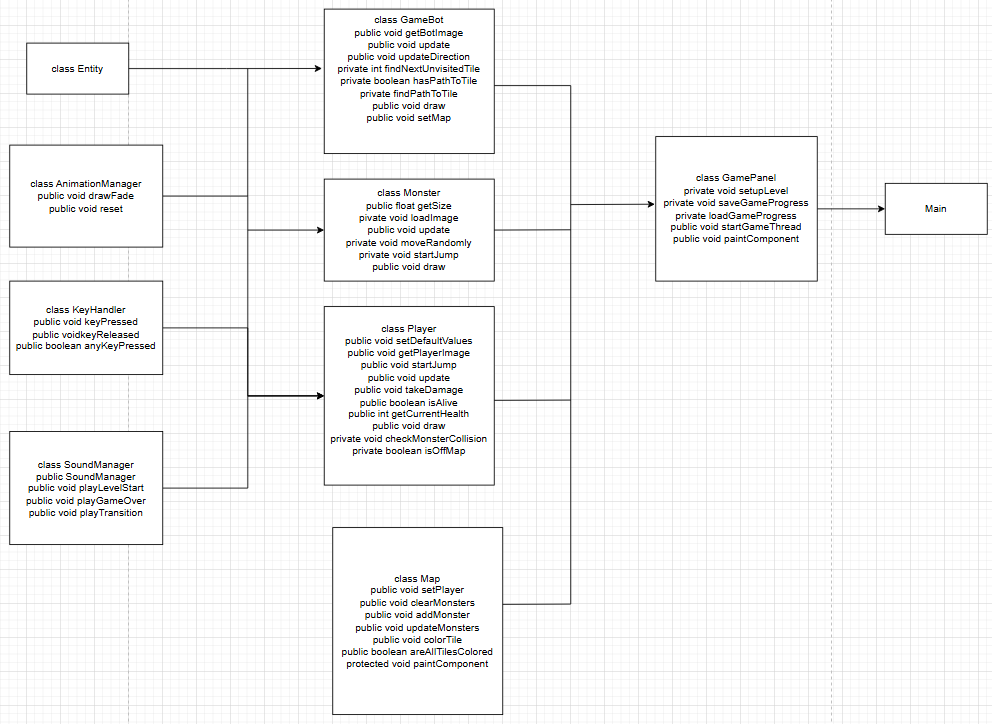


Рисунок 3. Диаграмма классов

Класс Main является точкой входа в игру и отвечает за создание и настройку основного окна приложения. Содержит статическое поле window типа JFrame для хранения основного окна приложения, автоматически подбирает размер окна под содержимоецентрирует окно на экране, делает окно видимым, запускает игровой цикл.

Листинг 1.

package Main;

import javax.swing.\*;

public class Main {

    private static JFrame window;

    public static void main(String[] *args*) {

        JFrame window = new JFrame();

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

        window.setResizable(false);

        window.setTitle("Qbert");

        GamePanel gamePanel = new GamePanel();

        window.add(gamePanel);

        window.pack();

        window.setLocationRelativeTo(null);

        window.setVisible(true);

        gamePanel.startGameThread();

    }

}

Класс Main является точкой входа в игру.

Описание работы:

Конструктор Main:

Cоздаётся новое окно (JFrame)

Настраиваются параметры окна:

Устанавливается операция закрытия окна (EXIT\_ON\_CLOSE)

Запрещается изменение размера окна (setResizable(false))

Устанавливается заголовок окна ("Qbert")

Создаётся экземпляр GamePanel

GamePanel добавляется в окно

Окно упаковывается (pack())

Окно центрируется на экране (setLocationRelativeTo(null))

Окно делается видимым (setVisible(true))

Запускается игровой поток (startGameThread())

Листинг 2

public class GamePanel extends JPanel implements Runnable {

    final int originalTileSize = 15;

    final int scale = 3;

    public final int tileSize = originalTileSize \* scale;

    final int maxScreenCol = 16;

    final int maxScreenRow = 12;

    final int screenWidth = tileSize \*maxScreenCol;

    final int screenHeight = tileSize \*maxScreenRow;

    int FPS = 60;

    private enum GameState {

        START\_SCREEN,

        MODE\_SELECTION,

        LEVEL\_SCREEN,

        PLAYING,

        DEATH\_SCREEN,

        RESTART\_SCREEN

    }

    private GameState gameState = GameState.START\_SCREEN;

    private BufferedImage startScreenImage;

    private BufferedImage modeSelectionImage;

    private BufferedImage levelScreenImage;

    private BufferedImage level2Image;

    private BufferedImage level3Image;

    private BufferedImage level4Image;

    private BufferedImage deathScreenImage;

    private float levelScreenTimer = 0;

    private float levelScreenDuration = 2.0f;

    private String deathMessage = "";

    private boolean isBotMode = false;

    private GameBot gameBot;

    private int currentLevel = 1;

    private static final String SAVE\_FILE = "game\_save.dat";

    private SoundManager soundManager;

    private AnimationManager animationManager;

    KeyHandler keyH = new KeyHandler();

    Thread gameThread;

    Player player = new Player(this,keyH);

    public Map map = new Map();

    int playerX = 100;

    int playerY = 100;

    int playerSpeed = 4;

    public GamePanel() {

        this.setPreferredSize(new Dimension(screenWidth, screenHeight));

        this.setBackground(Color.BLACK);

        this.setDoubleBuffered(true);

        this.addKeyListener(keyH);

        this.setFocusable(true);

        this.addMouseListener(new java.awt.event.MouseAdapter() {

            public void mouseClicked(java.awt.event.MouseEvent *evt*) {

                if (gameState == GameState.DEATH\_SCREEN) {

                    currentLevel = 1;

                    setupLevel();

                    player.setAlive(true);

                    gameState = GameState.LEVEL\_SCREEN;

                    levelScreenTimer = 0;

                    soundManager.playTransition();

                    animationManager.reset();

                }

            }

        });

        soundManager = new SoundManager();

        animationManager = new AnimationManager();

        gameBot = new GameBot(this, map);

        try {

            startScreenImage = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/startscreenqbert.png"));

            modeSelectionImage = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/ChooseMode.png"));

            if (modeSelectionImage == null) {

                System.out.println("Ошибка: изображение ChooseMode.png не загружено");

            }

            levelScreenImage = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/level1.png"));

            level2Image = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/level2.png"));

            level3Image = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/level3.png"));

            level4Image = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/level4.png"));

            deathScreenImage = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/deathscreen.png"));

        } catch (Exception *e*) {

            System.out.println("Ошибка при загрузке изображений: " + e.getMessage());

            e.printStackTrace();

        }

        loadGameProgress();

        currentLevel = 1;

        map.setPlayer(player);

        setupLevel();

    }

    private void setupLevel() {

        map = new Map();

        map.setPlayer(player);

        if (gameBot != null) {

            gameBot.setMap(map);

        }

        player.setDefaultValues();

        if (currentLevel == 1) {

            map.clearMonsters();

            map.addMonster(2, 1, 0.8f);

        } else {

            int monsterCount = currentLevel;

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

                map.addMonster(2 + i, 1, 0.8f);

            }

        }

    }

    private void saveGameProgress() {

        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(SAVE\_FILE))) {

            oos.writeInt(currentLevel);

        } catch (IOException *e*) {

            e.printStackTrace();

        }

    }

    private void loadGameProgress() {

        File saveFile = new File(SAVE\_FILE);

        if (saveFile.exists()) {

            try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(SAVE\_FILE))) {

                currentLevel = ois.readInt();

            } catch (IOException *e*) {

                e.printStackTrace();

            }

        }

    }

    public void startGameThread() {

        gameThread = new Thread(this);

        gameThread.start();

    }

    public void run() {

        double drawInterval = 1000000000/FPS; *// 0.01666 сек*

        double nextDrawTime = System.nanoTime() + drawInterval;

        double delta = 0;

        long lastTime = System.nanoTime();

        long currentTime;

        long timer = 0;

        int drawCount = 0;

        while(gameThread != null) {

            currentTime = System.nanoTime();

            delta += (currentTime - lastTime) / drawInterval;

            timer += (currentTime - lastTime);

            lastTime = currentTime;

            if(delta >= 1) {

                update();

                repaint();

                delta--;

                drawCount++;

            }

            if(timer >= 1000000000) {

                System.out.println("FPS: " + drawCount);

                drawCount = 0;

                timer = 0;

            }

        }

    }

    public void update() {

        animationManager.update();

        if (gameState == GameState.START\_SCREEN || gameState == GameState.RESTART\_SCREEN) {

            if (keyH.anyKeyPressed()) {

                System.out.println("Переход к экрану выбора режима");

                gameState = GameState.MODE\_SELECTION;

                soundManager.playTransition();

                animationManager.reset();

            }

            return;

        }

        if (gameState == GameState.MODE\_SELECTION) {

            if (keyH.qPressed) {

                System.out.println("Switching to bot mode");

                isBotMode = true;

                currentLevel = 1;

                setupLevel();

                player.setAlive(true);

                gameState = GameState.LEVEL\_SCREEN;

                levelScreenTimer = 0;

                soundManager.playTransition();

                animationManager.reset();

                gameBot.reset();

            }

            else if (keyH.ePressed) {

                System.out.println("Switching to player mode");

                isBotMode = false;

                currentLevel = 1;

                setupLevel();

                player.setAlive(true);

                gameState = GameState.LEVEL\_SCREEN;

                levelScreenTimer = 0;

                soundManager.playTransition();

                animationManager.reset();

            }

            return;

        }

        if (gameState == GameState.LEVEL\_SCREEN) {

            levelScreenTimer += 0.016f; *// Примерно 60 FPS*

            if (levelScreenTimer >= levelScreenDuration) {

                gameState = GameState.PLAYING;

                soundManager.playLevelStart();

            }

            return;

        }

        if (gameState == GameState.PLAYING) {

            if (isBotMode) {

                gameBot.update(0.016f);

                player.setMapPosition(gameBot.getCurrentRow(), gameBot.getCurrentCol());

            } else {

                player.update(0.016f);

            }

            map.updateMonsters(0.016f);

            if (!player.isAlive()) {

                gameState = GameState.DEATH\_SCREEN;

                if (player.getMapRow() < 0 || player.getMapRow() >= map.map.length ||

                    player.getMapCol() < 0 || player.getMapCol() >= map.map[player.getMapRow()].length) {

                    deathMessage = "Вы выпрыгнули за карту!";

                } else {

                    deathMessage = "Игра окончена!";

                }

                soundManager.playGameOver();

                if (isBotMode) {

                    gameBot.reset();

                    player.setMapPosition(0, 0);

                }

                return;

            }

            if (map.areAllTilesColored()) {

                currentLevel++;

                saveGameProgress();

                setupLevel();

                gameState = GameState.LEVEL\_SCREEN;

                levelScreenTimer = 0;

                soundManager.playTransition();

                animationManager.reset();

                if (isBotMode) {

                    gameBot.reset();

                }

            }

        }

    }

Класс GamePanel является основным классом игры, который управляет всем игровым процессом.

Описание работы:

Инициализация и настройка:

Устанавливает базовые размеры игрового поля, определяет размеры экрана, настраивает масштабирование.

Состояния игры:

GameState управляет различными состояниями игры, переключается между ними в зависимости от действий игрока.

Игровой цикл:

Run поддерживает постоянный цикл обновления и отрисовки, контролирует FPS (60 кадров в секунду)

Обработка ввода:

Обрабатывает нажатия клавиш, обрабатывает клики мыши, передает управление игроку или боту

Управление уровнями:

SetupLevel cоздает новую карту для каждого уровня, настраивает монстров в зависимости от уровня и сбрасывает позицию игрока.

Сохранение прогресса:

saveGameProgress осуществляет сохранение прогресса игрока, loadGameProgress загрузку загружает сохраненный прогресс при старте

Отрисовка:

paintComponent орисовывает текущее состояние игры, управляет анимациями и переходами

Обработка столкновений и игровой логики:

Update проверяет столкновения между игроком и монстрами, обновляет позиции всех объектов, проверяет условия победы и поражения

Листинг 3.

public class Map extends JPanel {

    public int[][] map = {

            {1,},

            {1, 1},

            {1, 1, 1},

            {1, 1, 1, 1},

            {1, 1, 1, 1, 1},

            {1, 1, 1, 1, 1, 1},

            {1, 1, 1, 1, 1, 1, 1}

    };

    private boolean[][] coloredTiles;

    public int cubeSize = 75;

    public float horizontalSpacing = 0.5f;

    public float verticalSpacing = 0.7f;

    public Image player;

    public int playerRow = 0;

    public int playerCol = 0;

    public int xOffset = 290;

    public int yOffset = 70;

    private BufferedImage cubeTexture;

    private BufferedImage coloredCubeTexture;

    private List<Monster> monsters;

    private Player playerRef;

    public Map() {

        try {

            cubeTexture = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/Kub.png"));

            coloredCubeTexture = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/KubGreen.png"));

            if (cubeTexture == null || coloredCubeTexture == null) {

                System.out.println("Error: Failed to load cube textures");

            } else {

                System.out.println("Successfully loaded cube textures");

            }

        } catch (Exception *e*) {

            System.out.println("Error loading textures: " + e.getMessage());

            e.printStackTrace();

        }

        monsters = new ArrayList<>();

        coloredTiles = new boolean[map.length][];

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

            coloredTiles[i] = new boolean[map[i].length];

            System.out.println("Initialized row " + i + " with " + map[i].length + " tiles");

        }

    }

    public void setPlayer(Player *player*) {

        this.playerRef = *player*;

    }

    public void clearMonsters() {

        monsters.clear();

    }

    public void addMonster(int *row*, int *col*) {

        monsters.add(new Monster(null, this, *row*, *col*));

    }

    public void addMonster(int *row*, int *col*, float *size*) {

        Monster monster = new Monster(null, this, *row*, *col*);

        monster.setSize(*size*);

        monsters.add(monster);

    }

    public void setAllMonstersSize(float *size*) {

        for (Monster monster : monsters) {

            monster.setSize(*size*);

        }

    }

    public void setMonsterSize(int *index*, float *size*) {

        if (*index* >= 0 && *index* < monsters.size()) {

            monsters.get(*index*).setSize(*size*);

        }

    }

    public void updateMonsters(float *deltaTime*) {

        for (Monster monster : monsters) {

            monster.update(*deltaTime*);

        }

    }

    public void colorTile(int *row*, int *col*) {

        if (*row* >= 0 && *row* < map.length && *col* >= 0 && *col* < map[*row*].length && map[*row*][*col*] == 1) {

            coloredTiles[*row*][*col*] = true;

            System.out.println("Coloring tile at " + *row* + "," + *col*);

            System.out.println("Current state of coloredTiles[" + *row* + "][" + *col* + "] = " + coloredTiles[*row*][*col*]);

            repaint();

            revalidate();

        } else {

            System.out.println("Failed to color tile at " + *row* + "," + *col* +

                             " (row valid: " + (*row* >= 0 && *row* < map.length) +

                             ", col valid: " + (*col* >= 0 && *col* < map[*row*].length) +

                             ", is active: " + (map[*row*][*col*] == 1) + ")");

        }

    }

    public boolean areAllTilesColored() {

        for (int row = 0; row < map.length; row++) {

            for (int col = 0; col < map[row].length; col++) {

                if (map[row][col] == 1 && !coloredTiles[row][col]) {

                    return false;

                }

            }

        }

        return true;

    }

    public List<Monster> getMonsters() {

        return monsters;

    }

    public Player getPlayer() {

        return playerRef;

    }

Класс Map является ключевым компонентом игры, отвечающим за управление игровым полем.

Описание работы:

Структура карты:

Представляет собой пирамидальную структуру, каждая строка содержит на один элемент больше, чем предыдущая, значение 1 означает активную плитку.

Отслеживание окрашенных плиток:

Хранит информацию о том, какие плитки уже окрашены, инициализируется в конструкторе coloredTiles.

Настройки отображения:

Определяет размеры и расположение элементов на экране, настраивает изометрический вид.

Управление монстрами:

List<Monster> monsters Хранит список монстров на карте

Методы для работы с монстрами: clearMonsters, addMonster.

Управление игроком:

Хранит ссылку на игрока, Предоставляет методы для доступа к игроку.

Управление окрашиванием плиток:

colorTile отмечает плитку как окрашенную, обновляет отображение.

Проверка завершения уровня:

areAllTilesColored проверяет, все ли плитки окрашены, используется для определения победы на уровне.

Обновление состояния:

updateMonsters обновляет состояние всех монстров, вызывается в игровом цикле.

Листинг 4

public class AnimationManager {

    private float alpha = 0.0f;

    private float fadeSpeed = 0.05f;

    private boolean isFadingIn = true;

    public void update() {

        if (isFadingIn) {

            alpha += fadeSpeed;

            if (alpha >= 1.0f) {

                alpha = 1.0f;

                isFadingIn = false;

            }

        } else {

            alpha -= fadeSpeed;

            if (alpha <= 0.0f) {

                alpha = 0.0f;

                isFadingIn = true;

            }

        }

    }

    public void drawFade(Graphics2D *g2*, BufferedImage *image*, int *x*, int *y*, int *width*, int *height*) {

        Composite originalComposite = *g2*.getComposite();

*g2*.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC\_OVER, alpha));

*g2*.drawImage(*image*, *x*, *y*, *width*, *height*, null);

*g2*.setComposite(originalComposite);

    }

    public void reset() {

        alpha = 0.0f;

        isFadingIn = true;

    }

}

Класс AnimationManager отвечает за управление анимациями в игре, в частности за эффект плавного появления/исчезновения (fade) изображений.

Описание работы:

Метод update:

Обновляет значение прозрачности, переключает направление анимации при достижении пределов

Метод drawFade:

Отрисовывает изображение с эффектом прозрачности, использует AlphaComposite для управления прозрачностью, сохраняет и восстанавливает оригинальные настройки графического контекста

Метод reset:

Сбрасывает состояние анимации, используется при переходе между экранами

Листинг 5

public class Monster extends Entity {

    private Map map;

    private Random random;

    private float moveTimer = 0;

    private float moveInterval = 0.7f;

    private int damage = 1;

    private int attackRange = 40;

    public int mapRow;

    public int mapCol;

    private int cubeSize = 70;

    private int displaySize = 30;

    private int xOffset = 300;

    private int yOffset = 50;

    private float horizontalSpacing = 0.5f;

    private float verticalSpacing = 0.7f;

    private float sizeMultiplier = 1.0f;

    private int baseSize = 70;

    private BufferedImage monsterImage;

    private BufferedImage jumpUpLeft;

    private BufferedImage jumpUpRight;

    private BufferedImage jumpDownLeft;

    private BufferedImage jumpDownRight;

    private BufferedImage upLeft;

    private BufferedImage upRight;

    private BufferedImage downLeft;

    private BufferedImage downRight;

    private String currentDirection = "down";

    @Override

    public float lerp(float *start*, float *end*, float *t*) {

        return *start* + (*end* - *start*) \* *t*;

    }

    public Monster(GamePanel *gamePanel*, Map *map*, int *startRow*, int *startCol*) {

        super(*gamePanel*);

        this.map = *map*;

        this.random = new Random();

        this.mapRow = *startRow*;

        this.mapCol = *startCol*;

        x = xOffset + mapCol \* cubeSize - (int)(mapRow \* cubeSize \* horizontalSpacing);

        y = yOffset + (int)(mapRow \* cubeSize \* verticalSpacing);

        loadImage();

    }

    public void setSize(float *multiplier*) {

        this.sizeMultiplier = *multiplier*;

        this.displaySize = (int)(baseSize \* *multiplier*);

    }

    public float getSize() {

        return sizeMultiplier;

    }

    private void loadImage() {

        try {

            monsterImage = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/snake.png"));

            jumpUpLeft = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/snakejumpupleft.png"));

            jumpUpRight = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/snakejumpupright.png"));

            jumpDownLeft = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/snakejumpdownleft.png"));

            jumpDownRight = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/snakejumpdownright.png"));

            upLeft = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/snakeupleft.png"));

            upRight = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/snakeupright.png"));

            downLeft = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/snakedownleft.png"));

            downRight = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/snakedownright.png"));

        } catch (Exception *e*) {

            e.printStackTrace();

        }

    }

    public void update(float *deltaTime*) {

        if (isJumping) {

            jumpProgress += *deltaTime* / jumpDuration;

            if (jumpProgress >= 1.0f) {

                jumpProgress = 1.0f;

                isJumping = false;

                x = endX;

                y = endY;

                float exactRow = (y - yOffset) / (cubeSize \* verticalSpacing);

                float exactCol = (x - xOffset + exactRow \* cubeSize \* horizontalSpacing) / cubeSize;

                mapRow = Math.round(exactRow);

                mapCol = Math.round(exactCol);

            } else {

                float t = jumpProgress;

                x = lerp(startX, endX, t);

                y = lerp(startY, endY, t) - (float)Math.sin(t \* Math.PI) \* jumpHeight;

            }

        } else {

            moveTimer += *deltaTime*;

            if (moveTimer >= moveInterval) {

                moveTimer = 0;

                moveRandomly();

            }

        }

        if (map != null && map.getPlayer() != null) {

            Player player = map.getPlayer();

            float dx = x - player.x;

            float dy = y - player.y;

            float distance = (float) Math.sqrt(dx \* dx + dy \* dy);

            if (distance < (displaySize + player.getDisplaySize()) / 2) {

                System.out.println("Монстр наносит урон игроку!");

                player.takeDamage(1);

            }

        }

    }

    private void moveRandomly() {

        int[] possibleMoves = {0, 1, 2, 3}; *// 0: up, 1: right, 2: down, 3: left*

        int move = possibleMoves[random.nextInt(4)];

        switch (move) {

            case 0

                if (mapRow - 1 >= 0 && mapCol < map.map[mapRow - 1].length) {

                    currentDirection = "up";

                    startJump(mapRow - 1, mapCol);

                }

                break;

            case 1:

                if (mapCol + 1 < map.map[mapRow].length) {

                    currentDirection = "right";

                    startJump(mapRow, mapCol + 1);

                }

                break;

            case 2:

                if (mapRow + 1 < map.map.length && mapCol < map.map[mapRow + 1].length) {

                    currentDirection = "down";

                    startJump(mapRow + 1, mapCol);

                }

                break;

            case 3:

                if (mapCol - 1 >= 0) {

                    currentDirection = "left";

                    startJump(mapRow, mapCol - 1);

                }

                break;

        }

    }

    public void startJump(int *targetRow*, int *targetCol*) {

        if (isJumping) return;

        isJumping = true;

        jumpProgress = 0;

        startX = x;

        startY = y;

        float targetX = xOffset + *targetCol* \* cubeSize - (*targetRow* \* cubeSize \* horizontalSpacing);

        float targetY = yOffset + (*targetRow* \* cubeSize \* verticalSpacing);

        endX = Math.round(targetX);

        endY = Math.round(targetY);

    }

    public void draw(Graphics2D *g2*) {

        BufferedImage imageToDraw = monsterImage;

        if (isJumping) {

            switch(currentDirection) {

                case "up":

                    if (endX < x) imageToDraw = jumpUpLeft;

                    else imageToDraw = jumpUpRight;

                    break;

                case "down":

                    if (endX < x) imageToDraw = jumpDownLeft;

                    else imageToDraw = jumpDownRight;

                    break;

                case "left":

                    imageToDraw = jumpUpLeft;

                    break;

                case "right":

                    imageToDraw = jumpUpRight;

                    break;

            }

        } else {

            switch(currentDirection) {

                case "up":

                    if (endX < x) imageToDraw = upLeft;

                    else imageToDraw = upRight;

                    break;

                case "down":

                    if (endX < x) imageToDraw = downLeft;

                    else imageToDraw = downRight;

                    break;

                case "left":

                    imageToDraw = upLeft;

                    break;

                case "right":

                    imageToDraw = upRight;

                    break;

            }

        }

*g2*.drawImage(imageToDraw, (int)x, (int)y, displaySize, displaySize, null);

    }

Описание работы:

Обновление состояния (update):

Обработка прыжка:

Проверяет, находится ли монстр в состоянии прыжка (isJumping), увеличивает прогресс прыжка на основе прошедшего времени. При завершении прыжка: устанавливает конечную позицию, обновляет позицию на карте. Во время прыжка: вычисляет промежуточную позицию, создает эффект дуги прыжка

Обработка обычного движения:

Увеличивает таймер движения. Когда таймер достигает интервала: сбрасывает таймер, вызывает случайное движение

Проверка столкновений:

Проверяет наличие карты и игрока, вычисляет расстояние до игрока, при столкновении наносит урон.

Движение монстра:

Выбор направления:

Создает массив возможных направлений, использует Random для выбора случайного направления.

Движение вверх (case 0):

Проверяет, что новая позиция не выходит за границы карты. Проверяет, что новая позиция существует в массиве карты, устанавливает направление "up", начинает прыжок в новую позицию

Движение вправо (case 1):

Проверяет, что новая позиция не выходит за правую границу, устанавливает направление "right", начинает прыжок в новую позицию

Движение вниз (case 2):

Проверяет, что новая позиция не выходит за нижнюю границу

Проверяет, что новая позиция существует в массиве карты

Устанавливает направление "down"

Начинает прыжок в новую позицию

Движение влево (case 3):

Проверяет, что новая позиция не выходит за левую границу

Устанавливает направление "left"

Начинает прыжок в новую позицию

Начало прыжка:

Проверка состояния:

Проверяет, не находится ли монстр уже в прыжке

Если да, то прерывает выполнение функции

Предотвращает наложение прыжков

Инициализация прыжка:

Устанавливает флаг, что монстр начал прыжок

Сбрасывает прогресс прыжка в начальное положение

Подготавливает параметры для анимации

Сохранение начальной позиции:

Запоминает текущую позицию монстра

Используется для расчета промежуточных позиций во время прыжка

Расчет конечной позиции:

Рассчитывает целевую позицию на основе:

xOffset - базовое смещение по X

yOffset - базовое смещение по Y

cubeSize - размер куба

horizontalSpacing - горизонтальное смещение

verticalSpacing - вертикальное смещение

targetRow - целевая строка

targetCol - целевой столбец

Округление координат:

Округляет конечные координаты до целых чисел

Обеспечивает точное попадание на целевую плитку

Листинг 6

public class Player extends Entity {

    GamePanel gp;

    KeyHandler keyH;

    Map map;

    public int mapRow = 0;

    public int mapCol = 0;

    private int cubeSize = 70;

    private int displaySize = 40;

    private int xOffset = 300;

    private int yOffset = 50;

    private float horizontalSpacing = 0.5f;

    private float verticalSpacing = 0.7f;

    private int maxHealth = 1;

    private int currentHealth;

    public BufferedImage upleft, upright, downleft, downright;

    private boolean alive = true;

    private KeyHandler keyHandler;

    private BufferedImage playerImage;

    private BufferedImage jumpUpLeft;

    private BufferedImage jumpUpRight;

    private BufferedImage jumpDownLeft;

    private BufferedImage jumpDownRight;

    private BufferedImage upLeft;

    private BufferedImage upRight;

    private BufferedImage downLeft;

    private BufferedImage downRight;

    private String currentDirection = "down";

    @Override

    public float lerp(float *start*, float *end*, float *t*) {

        return *start* + (*end* - *start*) \* *t*;

    }

    public Player(GamePanel *gp*, KeyHandler *keyH*) {

        super(*gp*);

        this.gp = *gp*;

        this.keyH = *keyH*;

        this.map = new Map();

        this.currentHealth = maxHealth;

        setDefaultValues();

        getPlayerImage();

    }

    public void setDefaultValues() {

        mapRow = 0;

        mapCol = 0;

        x = xOffset + mapCol \* cubeSize - (int)(mapRow \* cubeSize \* horizontalSpacing);

        y = yOffset + (int)(mapRow \* cubeSize \* verticalSpacing);

        speed = 4;

        direction = "down";

        isJumping = false;

        jumpProgress = 0;

    }

    public void setMapPosition(int *row*, int *col*) {

        mapRow = *row*;

        mapCol = *col*;

        x = xOffset + mapCol \* cubeSize - (int)(mapRow \* cubeSize \* horizontalSpacing);

        y = yOffset + (int)(mapRow \* cubeSize \* verticalSpacing);

    }

    public void getPlayerImage() {

        try {

            upleft = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/playerupleft.png"));      *// Вверх-влево*

            upright = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/playerupright.png"));    *// Вверх-вправо*

            downleft = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/playerleftdown.png"));  *// Вниз-влево*

            downright = ImageIO.read(getClass().getClassLoader().getResourceAsStream("player/playerdownright.png")); *// Вниз-вправо*

        } catch (Exception *e*) {

            e.printStackTrace();

        }

    }

    public void startJump(int *targetRow*, int *targetCol*) {

        if (isJumping) return;

        if (*targetRow* < 0 || *targetRow* >= map.map.length ||

*targetCol* < 0 || *targetCol* >= map.map[*targetRow*].length) {

            takeDamage(1);

            return;

        }

        isJumping = true;

        jumpProgress = 0;

        startX = x;

        startY = y;

        float targetX = xOffset + *targetCol* \* cubeSize - (*targetRow* \* cubeSize \* horizontalSpacing);

        float targetY = yOffset + (*targetRow* \* cubeSize \* verticalSpacing);

        endX = Math.round(targetX);

        endY = Math.round(targetY);

    }

    public void update(float *deltaTime*) {

        if (map != null && map.getMonsters() != null) {

            checkMonsterCollision();

        }

        if (isJumping) {

            jumpProgress += *deltaTime* / jumpDuration;

            if (jumpProgress >= 1.0f) {

                jumpProgress = 1.0f;

                isJumping = false;

                x = endX;

                y = endY;

                if (!isOffMap()) {

                    float exactRow = (y - yOffset) / (cubeSize \* verticalSpacing);

                    float exactCol = (x - xOffset + exactRow \* cubeSize \* horizontalSpacing) / cubeSize;

                    mapRow = Math.round(exactRow);

                    mapCol = Math.round(exactCol);

                    if (gp.map != null) {

                        gp.map.colorTile(mapRow, mapCol);

                    }

                } else {

                    takeDamage(1);

                }

            } else {

                float t = jumpProgress;

                x = lerp(startX, endX, t);

                y = lerp(startY, endY, t) - (float)Math.sin(t \* Math.PI) \* jumpHeight;

            }

        } else {

            if (keyH.downPressed) {

                direction = "down";

                startJump(mapRow + 1, mapCol);

            } else if (keyH.upPressed) {

                direction = "up";

                startJump(mapRow - 1, mapCol);

            } else if (keyH.leftPressed) {

                direction = "left";

                startJump(mapRow, mapCol - 1);

            } else if (keyH.rightPressed) {

                direction = "right";

                startJump(mapRow, mapCol + 1);

            }

        }

    }

    public void takeDamage(int *damage*) {

        currentHealth -= *damage*;

        if (currentHealth <= 0) {

            currentHealth = 0;

            alive = false;

        }

    }

    public boolean isAlive() {

        return alive;

    }

    public int getCurrentHealth() {

        return currentHealth;

    }

    public void draw(Graphics2D *g2*) {

        BufferedImage image = null;

        switch(direction) {

            case "up":

                image = upleft;

                break;

            case "down":

                image = downleft;

                break;

            case "left":

                image = upleft;

                break;

            case "right":

                image = upright;

                break;

        }

*g2*.drawImage(image, (int)x, (int)y, displaySize, displaySize, null);

    }

    private void checkMonsterCollision() {

        for (Monster monster : map.getMonsters()) {

            if (monster != null) {

                int monsterX = monster.getX();

                int monsterY = monster.getY();

                System.out.println("Player position: (" + x + ", " + y + ")");

                System.out.println("Monster position: (" + monsterX + ", " + monsterY + ")");

                System.out.println("Distance: " + Math.abs(x - monsterX) + ", " + Math.abs(y - monsterY));

                if (Math.abs(x - monsterX) < displaySize && Math.abs(y - monsterY) < displaySize) {

                    takeDamage(1);

                    System.out.println("Игра окончена! Игрок столкнулся с монстром!");

                    return;

                }

            }

        }

    }

    public int getDisplaySize() {

        return displaySize;

    }

    public void setAlive(boolean *alive*) {

        this.alive = *alive*;

    }

    private boolean isOffMap() {

        return mapRow < 0 || mapRow >= map.map.length ||

               mapCol < 0 || mapCol >= map.map[mapRow].length;

    }

    public int getMapRow() {

        return mapRow;

    }

    public int getMapCol() {

        return mapCol;

    }

}

Класс Player наследуется от Entity и управляет игроком в игре

Описание работы:

Установка начальных значений:

Установка начальной позиции на карте:

Устанавливает начальную позицию игрока в верхней точке пирамиды

mapRow = 0 - первая строка карты

mapCol = 0 - первый столбец

Расчет начальной позиции на экране:

Рассчитывает точные координаты на экране

Использует параметры:

xOffset = 300 - базовое смещение по X

yOffset = 50 - базовое смещение по Y

cubeSize = 70 - размер куба

horizontalSpacing = 0.5f - горизонтальный интервал

verticalSpacing = 0.7f - вертикальный интервал

Установка базовых параметров:

Устанавливает скорость движения

Задает начальное направление

Сбрасывает состояние прыжка

Обнуляет прогресс прыжка

Особенности работы:

Учитывает изометрическую проекцию карты

Обеспечивает корректное начальное позиционирование

Сбрасывает все параметры движения

Подготавливает игрока к началу игры

Фунцкия update:

Проверка столкновений с монстрами:

Вызывает метод checkMonsterCollision()

Проверяет столкновения со всеми монстрами на карте

Если есть столкновение:

Наносит урон игроку

Обновляет состояние здоровья

Проверяет смерть игрока

Обработка прыжка:

Проверяет, находится ли игрок в прыжке

Увеличивает прогресс прыжка

Когда прыжок завершен:

Сбрасывает состояние прыжка

Обнуляет прогресс

Обработка движения:

Проверяет нажатые клавиши

Обновляет направление движения

Изменяет позицию на карте:

Обновление позиции на экране:

Рассчитывает новые координаты на экране

Учитывает:

Смещение по X и Y

Размер куба

Горизонтальный и вертикальный интервалы

Текущую позицию на карте

Проверка на столкновение с монстром:

Получение списка монстров:

Получает список всех монстров с карты

Использует метод getMonsters() из класса Map

Список содержит все активные монстры на текущем уровне

Цикл проверки столкновений:

Перебирает всех монстров на карте

Проверяет столкновение с каждым монстром

Обрабатывает только активные монстры

Расчет расстояния до монстра:

Использует формулу расстояния между точками

x, y - координаты игрока

monster.getX(), monster.getY() - координаты монстра

Math.pow() - возведение в степень

Math.sqrt() - извлечение корня

Проверка столкновения:

Проверяет, меньше ли расстояние 50 пикселей

50 пикселей - это радиус столкновения

Если расстояние меньше - происходит столкновение

Обработка столкновения:

Вызывает метод takeDamage(1) для нанесения урона

Передает 1 как количество урона

break прерывает цикл после первого столкновения

Это предотвращает множественные столкновения за один кадр

Метод takeDamage:

Уменьшает текущее здоровье

Проверяет, не умер ли игрок

Обновляет состояние isAlive

Листинг 7

public class KeyHandler implements KeyListener {

    public boolean upPressed, downPressed, leftPressed, rightPressed;

    public boolean enterPressed;

    public boolean qPressed, ePressed;

    public int selectedOption = 0;

    @Override

    public void keyTyped(KeyEvent *e*) {

    }

    @Override

    public void keyPressed(KeyEvent *e*) {

        int code = *e*.getKeyCode();

        if (code == KeyEvent.VK\_W) {

            upPressed = true;

        }

        if (code == KeyEvent.VK\_S) {

            downPressed = true;

        }

        if (code == KeyEvent.VK\_A) {

            leftPressed = true;

        }

        if (code == KeyEvent.VK\_D) {

            rightPressed = true;

        }

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

            enterPressed = true;

        }

        if (code == KeyEvent.VK\_Q) {

            qPressed = true;

        }

        if (code == KeyEvent.VK\_E) {

            ePressed = true;

        }

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

            selectedOption = Math.max(0, selectedOption - 1);

            System.out.println("Selected option: " + selectedOption);

        }

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

            selectedOption = Math.min(1, selectedOption + 1);

            System.out.println("Selected option: " + selectedOption);

        }

    }

        @Override

        public void keyReleased (KeyEvent *e*) {

            int code = *e*.getKeyCode();

            if (code == KeyEvent.VK\_W) {

                upPressed = false;

            }

            if (code == KeyEvent.VK\_S) {

                downPressed = false;

            }

            if (code == KeyEvent.VK\_A) {

                leftPressed = false;

            }

            if (code == KeyEvent.VK\_D) {

                rightPressed = false;

            }

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

                enterPressed = false;

            }

            if (code == KeyEvent.VK\_Q) {

                qPressed = false;

            }

            if (code == KeyEvent.VK\_E) {

                ePressed = false;

            }

        }

    public boolean anyKeyPressed() {

        return upPressed || downPressed || leftPressed || rightPressed || enterPressed || qPressed || ePressed;

    }

    }

Класс KeyHandler реализует интерфейс KeyListener и отвечает за обработку всех клавиатурных событий в игре.

Описание работы:

Обработка клавиш движения (W, A, S, D)

Обработка специальных клавиш (Enter, Q, E)

Управление выбором режима игры

Отслеживание состояния нажатия клавиш

Предоставление информации о нажатых клавишах другим классам

Особенности реализации:

В классе KeyHandler используются булевы переменные (флаги) для отслеживания состояния нажатия клавиш

В функции keyPressed происходит установка флагов при нажатии клавиш, а в классе keyReleased происходит сброс флагов при отпускании клавиш.

в зависимости от нажатых клавиш.

2.3. Интерфейс приложения

Интерфейс приложения прост в использовании. В качестве меню используется стартовый экран.

Далее пользователю дают выбор за кого играть. И начинается сама игра

A screenshot of a video game

AI-generated content may be incorrect.

Рисунок 3. Интерфейс игры

Процесс при нажатии кнопки

Когда игрок нажимает кнопку любую из кнопок движения происходит следующее:

Переход на экран выбора режима: :

Рисунок 4. Выбор режима игры

Игроку даётся выбор играть самому или прохождение ботом.

Далее показывается начальный уровень и запускается игра

A green text on a black background

AI-generated content may be incorrect.

Рисунок 5. Уровень игры

Запуск игры:

A screenshot of a video game

AI-generated content may be incorrect.

Рисунок 6. Начало игры

Переход к игровому состоянию:

if (gameState == GameState.LEVEL\_SCREEN) {

    levelScreenTimer += 0.016f;

    if (levelScreenTimer >= levelScreenDuration) {

        gameState = GameState.PLAYING;

        soundManager.playLevelStart();

    }

    return;

}

Настройка уровня (setupLevel):

private void setupLevel() {

    map = new Map();

    map.setPlayer(player);

    if (gameBot != null) {

        gameBot.setMap(map);

    }

    player.setDefaultValues();

    if (currentLevel == 1) {

        map.clearMonsters();

        map.addMonster(2, 1, 0.8f);

    } else {

        int monsterCount = currentLevel;

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

            map.addMonster(2 + i, 1, 0.8f);

        }

    }

}

Установка начальных значений игрока:

public void setDefaultValues() {

    mapRow = 0;

    mapCol = 0;

    x = xOffset + mapCol \* cubeSize - (int)(mapRow \* cubeSize \* horizontalSpacing);

    y = yOffset + (int)(mapRow \* cubeSize \* verticalSpacing);

    speed = 4;

    direction = "down";

    isJumping = false;

    jumpProgress = 0;

}

Начало игрового процесса:

if (gameState == GameState.PLAYING) {

    if (isBotMode) {

        gameBot.update(0.016f);

        player.setMapPosition(gameBot.getCurrentRow(), gameBot.getCurrentCol());

    } else {

        player.update(0.016f);

    }

    map.updateMonsters(0.016f);

*// ...*

}

Проверки во время игры:

Проверка смерти игрока

Проверка победы (все тайлы окрашены)

Обновление монстров

Обновление позиции игрока/бота

Анимации:

animationManager.reset();

Параметры уровня:

Количество монстров зависит от уровня

Скорость монстров: 0.8f

Позиции монстров: начиная с (2,1)

Инициализация карты:

Создание новой карты

Установка игрока на карту

Очистка старых монстров

Добавление новых монстров

Параметры игрока:

Начальная позиция: (0,0)

Скорость: 4

Направление: "down"

Состояние прыжка: false

Прогресс прыжка: 0

После этих действий игра переходит в активное состояние, где:

Игрок может управлять персонажем

Монстры начинают двигаться

Начинается отсчет времени

Активируются все игровые механики

A screenshot of a video game

AI-generated content may be incorrect.

Рисунок 7. Конец игры

При столкновении с монтром или выхода игрока за пределы карты игрок теряет здоровье и игра заканчивается.

A yellow text on a black background

AI-generated content may be incorrect.

Рисунок 8. Переход на новый уровень

При закрашивании всех плиток игрок переходит на уровень выше.

3.Тестирование приложения

3.1. Контрольный робот

Для того чтобы протестировать работу приложения воспользуемся распространённым подходом под названием юнит-тестирование и конкретным фреймворком - JUnit.

В проекте есть три основных тестовых класса, которые проверяют различные аспекты игры

PlayerTest - тестирование игрока:

public class PlayerTest {

    @Test

    void testInitialState() {

        assertEquals(0, player.getMapRow());

        assertEquals(0, player.getMapCol());

        assertEquals(1, player.getCurrentHealth());

        assertTrue(player.isAlive());

    }

    @Test

    void testTakeDamage() {

        player.takeDamage(1);

        assertEquals(0, player.getCurrentHealth());

        assertFalse(player.isAlive());

    }

    @Test

    void testSetMapPosition() {

        player.setMapPosition(2, 3);

        assertEquals(2, player.getMapRow());

        assertEquals(3, player.getMapCol());

    }

    @Test

    void testStartJump() {

        player.startJump(1, 1);

        assertTrue(player.isJumping);

        assertEquals(0, player.jumpProgress);

    }

    @Test

    void testDisplaySize() {

        assertEquals(40, player.getDisplaySize());

    }

}

EntityTest - тестирование базового класса сущности:

public class EntityTest {

    @Test

    void testInitialState() {

        assertFalse(entity.isJumping);

        assertEquals(0, entity.jumpProgress);

        assertEquals(0, entity.startX);

        assertEquals(0, entity.startY);

        assertEquals(0, entity.endX);

        assertEquals(0, entity.endY);

        assertEquals(0.3f, entity.jumpDuration);

        assertEquals(10f, entity.jumpHeight);

    }

    @Test

    void testStartJump() {

        entity.startJump(1, 1);

        assertTrue(entity.isJumping);

        assertEquals(0, entity.jumpProgress);

        assertEquals(0, entity.startX);

        assertEquals(0, entity.startY);

        assertEquals(gamePanel.tileSize, entity.endX);

        assertEquals(gamePanel.tileSize, entity.endY);

    }

    @Test

    void testLerp() {

        assertEquals(5, entity.lerp(0, 10, 0.5f));

    }

}

MonsterTest - тестирование монстров:

public class MonsterTest {

    @Test

    void testInitialState() {

        assertEquals(0, monster.getMapRow());

        assertEquals(0, monster.getMapCol());

        assertEquals(1, monster.getCurrentHealth());

        assertTrue(monster.isAlive());

    }

    @Test

    void testTakeDamage() {

        monster.takeDamage(1);

        assertEquals(0, monster.getCurrentHealth());

        assertFalse(monster.isAlive());

    }

    @Test

    void testSetMapPosition() {

        monster.setMapPosition(2, 3);

        assertEquals(2, monster.getMapRow());

        assertEquals(3, monster.getMapCol());

    }

    @Test

    void testStartJump() {

        monster.startJump(1, 1);

        assertTrue(monster.isJumping);

        assertEquals(0, monster.jumpProgress);

    }

    @Test

    void testDisplaySize() {

        assertEquals(40, monster.getDisplaySize());

    }

}

Основные аспекты, которые проверяются в тестах:

Начальное состояние:

Позиция на карте

Здоровье

Состояние жизни

Параметры прыжка

Механика урона:

Получение урона

Изменение здоровья

Смерть при нулевом здоровье

Позиционирование:

Установка позиции на карте

Корректность координат

Обновление позиции

Механика прыжка:

Начало прыжка

Прогресс прыжка

Состояние прыжка

Координаты начала и конца

Отображение:

Размер отображения

Параметры спрайтов

Визуальные характеристики

Математические функции:

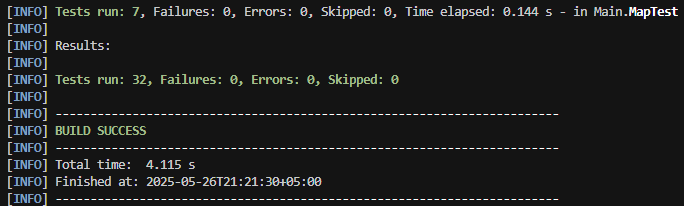
Линейная интерполяция

Расчеты позиций

Преобразования координат

3.2. Работа приложения на контрольных тестах

Все тесты проходят успешно и в указанные временные рамки и можно сделать выводы, что всё работает корректно.



3.3. Результаты работы

В результате выполнения курсовой работы было разработано приложение, которое реализует игру «Qbert». Игрок пердоставляется возможность играть за персонажа и закрасить всю изометрическую пирамидальную структуру из трёхцветных кубов, чтобы пройти игру. В данном приложении используется простой и понятный графическйи интерфейс, с которым сможет работать рядовой пользователь.