

# Documentation: Java Game Project

Wintersemester 2025/26 – Hochschule Bielefeld – Programmieren 1 – Author: Manuel Borghardt, Darnell Borghardt, Joel Jantschik, Leonid Nikkel

---

## 1. Projektidee & Konzept

- Spielfigur sammelt Coins und weicht Hindernissen aus. Wobei die Coins und Hindernisse von oben fallen.
  - Ziel: Spieler muss im ersten Level 20 Coins und im zweiten Level 40 Coins einsammeln. Darf dabei aber nicht einmal vom Hinderniss getroffen werden.
  - Steuerung mit Pfeiltasten und 'A' und 'D', es gibt zwei Level wie schon im Ziel erwähnt.
- 

## 2. Entwicklungsumgebung

- Java-Version: SE 21
    - Version SE 21, weil dann die Arbeit mit libGDX richtig funktioniert
  - Framework/Library: libGDX
  - Tools: Visual Studio Code mit folgenden Extensions:
    - Java
    - Extension Pack for Java
    - Gradle for Java
    - Jupyter
- 

## 3. LLM

- Copilot
  - Gemini
- 

## 4. Spielfeld

- Beschreibung:
  - Fenstergröße: 1000 × 1000 Pixel.
  - Hintergrundfarbe: dunkelblau/grau (`glClearColor(0.1f, 0.1f, 0.2f, 1)`).
  - Spielfigur: grünes Quadrat (60 × 70 Pixel).
  - Punktestand: Textanzeige oben links.

Prompt für Copilot(aufgabe.pdf angehängt): kannst du ein fenster erstellen in dem eine Figur ist, mit der man sich nach links und rechts bewegen kann. (Copilot hat erst nur eine Main erstellt und da alles definiert) Aber in der aufgabe ist ja angegeben das ich Objekt orientiert programmieren soll.

Code der von Copilot für Main.java genutzt wurde:

```

public class Main extends ApplicationAdapter {
    private OrthographicCamera camera;
    private SpriteBatch batch;
    private ShapeRenderer shapeRenderer;
    private BitmapFont font;

    private Player player;

    @Override
    public void create() {
        // Kamera einrichten
        camera = new OrthographicCamera();
        camera.setToOrtho(false, 1000, 1000);

        // Renderer und Schriftart initialisieren
        batch = new SpriteBatch();
        shapeRenderer = new ShapeRenderer();
        font = new BitmapFont();

        // Spieler initialisieren
        player = new Player(480);
    }
    // Render-Schleife
    @Override
    public void render() {
        handleInput();

        // Hintergrundfarbe setzen
        Gdx.gl.glClearColor(0.1f, 0.1f, 0.2f, 1);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

        camera.update();

        // Spieler zeichnen
        shapeRenderer.setProjectionMatrix(camera.combined);
        shapeRenderer.begin(ShapeRenderer.ShapeType.Filled);
        player.render(shapeRenderer);
        shapeRenderer.end();

        // Punktestand anzeigen
        batch.setProjectionMatrix(camera.combined);
        batch.begin();
        font.draw(batch, "Punkte: " + player.getPoints(), 10, 980);
        batch.end();
    }
    // Eingaben verarbeiten
    private void handleInput() {
        if (Gdx.input.isKeyPressed(Input.Keys.LEFT)) ||

```

```

Gdx.input.isKeyPressed(Input.Keys.A)) player.moveLeft();
    if (Gdx.input.isKeyPressed(Input.Keys.RIGHT) ||
Gdx.input.isKeyPressed(Input.Keys.D)) player.moveRight();
}
// Ressourcen freigeben
@Override
public void dispose() {
    batch.dispose();
    shapeRenderer.dispose();
    font.dispose();
}
}

```

## 5. Spielfigur(Player)

- Objektorientierte Umsetzung: eigene Klasse Player.
- Attribute: x, y (Position), points (Punktestand).
- Methoden: moveLeft(), moveRight(), moveUp(), moveDown(), render().
- Darstellung: grünes Quadrat, das sich über Tastatureingaben bewegt.

Prompt für Copilot: (Siehe oben bei 4.Spielfeld)

Code der von Copilot für Player.java genutzt wurde:

```

public class Player {
    // Spielfigur-Position und Punktestand
    private float x, y;
    private int points;

    // Konstruktor
    public Player(float startX, float startY) {
        this.x = startX;
        this.points = 0;
    }

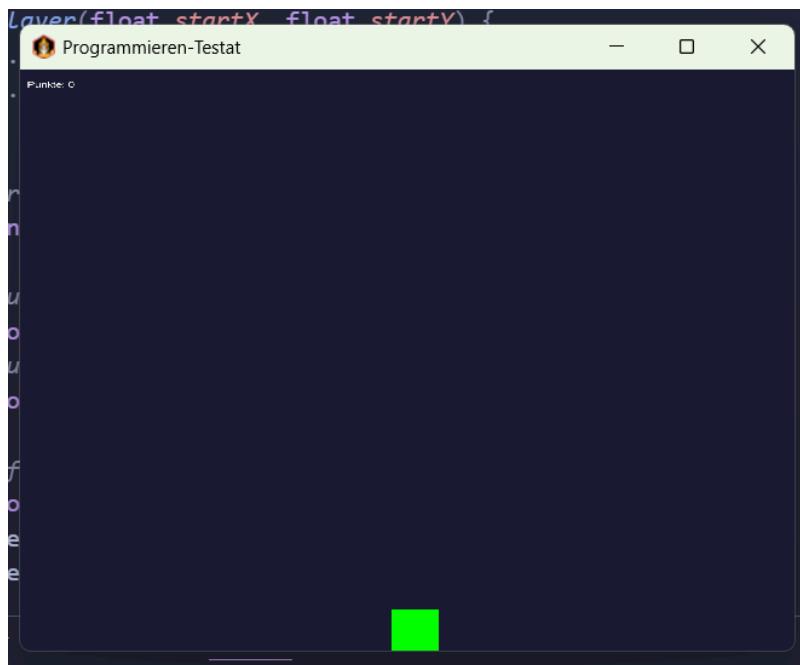
    // getter für Punktestand
    public int getPoints() { return points; }

    // Bewegung der Spielfigur nach Links
    public void moveLeft() { x -= 3; points++; }
    // Bewegung der Spielfigur nach rechts
    public void moveRight() { x += 3; points++; }

    // Spielfigur zeichnen
    public void render(ShapeRenderer renderer) {
        renderer.setColor(0, 1, 0, 1);
        renderer.rect(x, y, 60, 70);
    }
}

```

## Erstes Fenster mit Spielfigur



## 6. Bug-fix

- Spielfigur konnte links und rechts aus dem Window verschwinden
- Lösung (selber gemacht):

```
// Bewegung der Spielfigur nach Links
public void moveLeft() {
    x -= 3;
    if(x == 0) {
        x += 3;
    }
}

// Bewegung der Spielfigur nach rechts
public void moveRight() {
    x += 3;
    if(x == 939) {
        x -= 3;
    }
}
```

## 6. Coins und Enemies

### 6.1 Coins

- Verhalten: Coins spawnen periodisch oben an einer zufälligen X-Position und fallen nach unten.
- Standard-Parameter (in der Implementierung):
  - Spawn-Intervall: 0.8s
  - Größe: 20 x 20 Pixel
  - Fallgeschwindigkeit: 200 px/s
- Lebenszyklus:

- `spawnCoin()` erzeugt neue Coins
- `update(delta)` bewegt jeden Coin (`y -= speed * delta`)
- Coins werden entfernt, sobald sie vollständig unterhalb des Bildschirms sind (`y + height < 0`)
- Bei Kollision mit dem Player (`collectCollisions(player)`) wird der Coin entfernt und der Punktestand erhöht (`player.addPoints(1)`).

Prompt für Copilot: coin objekte sollen in regelmäßigen Abständen von oben nach unten fallen. Dabei sollen die coins von einer zufälligen position, der horizontalen, aus fallen. wenn die coins am unteren rand angekommen sind sollen diese coins entfernt werden. wenn der player ein coin berührt wird der coin entfernt und der punktestand wird um ein erhöht.

Wichtige Methoden / Aufrufe im Spiel vom Copilot übernommen:

```
// Update-Methode: spawnt neue Coins, bewegt existierende nach unten
// und entfernt sie, wenn sie aus dem Bildschirm gefallen sind
public void update(float delta) {
    spawnTimer += delta;
    if (spawnTimer >= spawnInterval) {
        spawnTimer -= spawnInterval;
        spawnCoin();
    }

    // update coins and entfernt die, die das Fenster
    verlassen haben
    Iterator<Coin> it = coins.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        c.update(delta);
        if (c.y + c.height < 0) {
            it.remove();
        }
    }
}

// rendert alle Coins
public void render(ShapeRenderer renderer) {
    for (Coin c : coins) {
        c.render(renderer);
    }
}

// Prüft Kollisionen zwischen Spieler und Coins; entferne
getroffene Coins und erhöhe Punkte
public void collectCollisions(Player player) {
    Iterator<Coin> it = coins.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        if (overlaps(c.x, c.y, c.width, c.height,
player.getX(), player.getY(), player.getWidth(), player.getHeight())) {
            it.remove();
            player.addPoints(1);
        }
    }
}
```

```

        }

    }

    // einfache AABB-Kollisionsprüfung
    private boolean overlaps(float ax, float ay, float aw, float
ah, float bx, float by, float bw, float bh) {
        return ax < bx + bw && ax + aw > bx && ay < by + bh &&
ay + ah > by;
    }

    // erzeugt einen neuen Coin an einer zufälligen X-Position oben
    im Bildschirm
    private void spawnCoin() {
        float x = random.nextFloat() * (worldWidth -
coinWidth);
        float y = worldHeight; // start at top
        coins.add(new Coin(x, y, coinWidth, coinHeight,
fallSpeed));
    }

    // innere Klasse für einzelne Coin-Instanzen
    private static class Coin {
        float x, y, width, height, speed;

        // Konstruktor
        Coin(float x, float y, float width, float height, float
speed) {
            this.x = x;
            this.y = y;
            this.width = width;
            this.height = height;
            this.speed = speed;
        }

        // aktualisiert die Position des Coins
        void update(float delta) {
            y -= speed * delta;
        }

        // rendert den Coin als gelbes Rechteck
        void render(ShapeRenderer renderer) {
            renderer.setColor(1f, 0.9f, 0f, 1f);
            renderer.rect(x, y, width, height);
        }
    }
}

```

## 6.2 Enemy

- Verhalten: Enemys verhalten sich ähnlich wie Coins, haben aber andere Parameter und bewirken einen Punktabzug beim Kontakt.
- Standard-Parameter (in der Implementierung):
  - Spawn-Intervall: 2.6s
  - Größe: 30 x 25 Pixel

- Fallgeschwindigkeit: 180 px/s
- Lebenszyklus:
  - `spawnEnemy()` erzeugt neue Enemys
  - `update(delta)` bewegt den Enemy nach unten
  - Enemys werden entfernt, sobald sie unterhalb des Bildschirms sind
  - Bei Kollision mit dem Player (`badCollisions(player)`) wird der Enemy entfernt und der Player verliert einen Punkt (`player.addPoints(-1)`).

Prompt für Copilot: Enemy funktioniert wie coin, mit dem Unterschied dass Enemys eine andere spwanrate etc. haben und das Player einen punktabzug bei kontakt mit Enemy bekommt.

Wichtige Methoden / Aufrufe im Spiel von Copilot übernommen:

```
//Prüft Kollisionen zwischen Player und Gegnern.
//Entfernt getroffene Gegner und zieht dem Player einen Punkt ab.
public void update(float delta) {
    // Zeit sammeln und ggf. neuen Gegner erzeugen
    spawnTimer += delta;
    if (spawnTimer >= spawnInterval) {
        spawnTimer -= spawnInterval;
        spawnEnemy();
    }

    // Gegner aktualisieren und entfernen, wenn sie den Bildschirm
    // verlassen haben
    Iterator<EnemyUnit> it = enemies.iterator();
    while (it.hasNext()) {
        EnemyUnit e = it.next();
        e.update(delta);
        if (e.y + e.height < 0) {
            it.remove();
        }
    }
}

//Rendern aller aktiven Gegner (rotes Rechteck pro Gegner).
public void render(ShapeRenderer renderer) {
    for (EnemyUnit e : enemies) {
        e.render(renderer);
    }
}

//Prüft Kollisionen zwischen Player und Gegnern.
//Entfernt getroffene Gegner und zieht dem Player einen Punkt ab.
public void badCollisions(Player player) {
    Iterator<EnemyUnit> it = enemies.iterator();
    while (it.hasNext()) {
        EnemyUnit e = it.next();
        if (overlaps(e.x, e.y, e.width, e.height, player.getX(),
                     player.getY(), player.getWidth(), player.getHeight())) {
            it.remove();
            player.addPoints(-1); // Punktabzug bei Kontakt
        }
    }
}
```

```

        }

    // AABB-Kollisionstest (Axis-Aligned Bounding Box)
    private boolean overlaps(float ax, float ay, float aw, float ah,
    float bx, float by, float bw, float bh) {
        return ax < bx + bw && ax + aw > bx && ay < by + bh && ay + ah
        > by;
    }

    // Erzeugt einen neuen Gegner an einer zufälligen X-Position oben
    private void spawnEnemy() {
        float x = random.nextFloat() * (worldWidth - enemyWidth);
        float y = worldHeight; // oben am Bildschirmrand
        enemies.add(new EnemyUnit(x, y, enemyWidth, enemyHeight,
        fallSpeed));
    }

    // Innere Klasse für einzelne Gegner-Instanzen.
    private static class EnemyUnit {
        float x, y, width, height, speed;

        EnemyUnit(float x, float y, float width, float height, float
        speed) {
            this.x = x;
            this.y = y;
            this.width = width;
            this.height = height;
            this.speed = speed;
        }

        // Bewegung nach unten
        void update(float delta) {
            y -= speed * delta;
        }

        // Zeichnet den Gegner als rotes Rechteck
        void render(ShapeRenderer renderer) {
            renderer.setColor(1f, 0.2f, 0.2f, 1f);
            renderer.rect(x, y, width, height);
        }
    }
}

```

## 7. Integration in Main (Coins + Enemy)

- Initialisierung erfolgt in `create()` mit der Welt-/Kamera-Größe, damit Spawn-Positionen korrekt berechnet werden.
- Im Render-Loop werden zuerst `update(delta)` für Coins und Enemys aufgerufen, dann die Kollisionsprüfungen (`collectCollisions`, `badCollisions`), und zuletzt werden alle Objekte mit demselben `ShapeRenderer` gezeichnet.

Copilot hat von sich aus die Main verändert

Ausschnitt aus `Main.render()`:

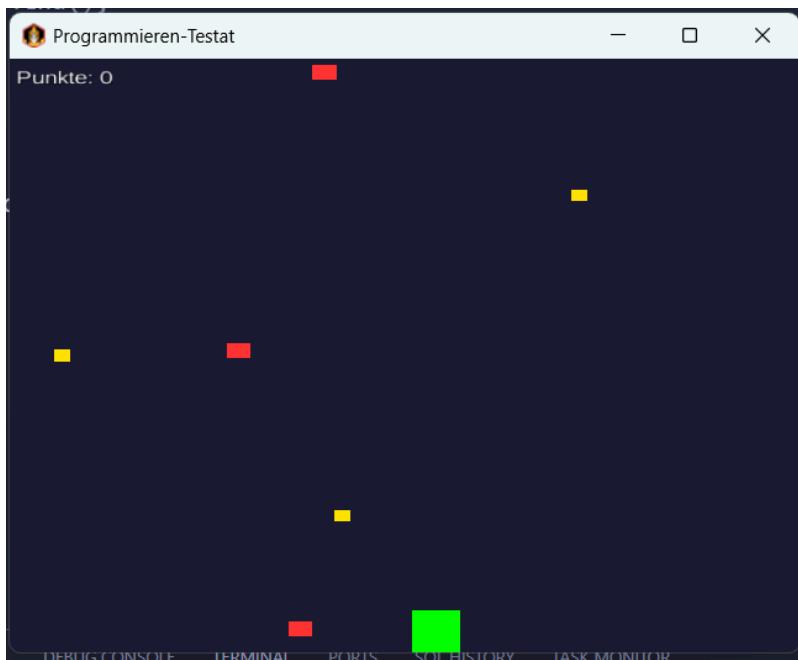
```
float delta = Gdx.graphics.getDeltaTime();

// Coins
coins.update(delta);
coins.collectCollisions(player);

// Enemys
enemy.update(delta);
enemy.badCollisions(player);

// Rendern (ShapeRenderer-Durchlauf)
shapeRenderer.begin(ShapeRenderer.ShapeType.Filled);
coins.render(shapeRenderer);
enemy.render(shapeRenderer);
player.render(shapeRenderer);
shapeRenderer.end();
```

## Zwischenstand



## 8. Spieleränderung

- Spieler-Geschwindigkeit wurde erhöht, weil es dann mehr spaß macht.

Code:

```
// Bewegung der Spielfigur nach Links
public void moveLeft() {
    x -= 5;
    if(x == 0) {
        x += 5;
    }
}
```

```
// Bewegung der Spielfigur nach rechts
public void moveRight() {
    x += 5;
    if(x == 940) {
        x -= 5;
    }
}
```

## 9. Level-Design (Level 1 & Level 2)

### 9.1 Geändertes Ziel

- Level 1 Ziel: 15 coins erreichen -Levelwechsel: Erfolgt sofort, wenn 15 Punkte erreicht sind, mit einer kurzen Unterbrechung zur Anzeige des Levelwechsels.

Prompt für Gemini: Level wechsel soll nach 15 gesammelten Punkten passieren.

code in Main.java:

```
private int currentLevel = 1; // Startlevel
private final int LEVEL_THRESHOLD = 15; // Punkte, die für Level 2
erreicht werden müssen
private float levelChangeTimer = 0f; // Timer für eine kurze
Pause/Anzeige
private final float LEVEL_CHANGE_DURATION = 2.0f; // 2 Sekunden
Levelwechsel-Pause
```

### 9.2 Anpassungen in Enemy.java (Schwierigkeitsgrad-Steuerung)

- Um die Parameter der Gegner zur Laufzeit anpassen zu können, mussten die Variablen fallSpeed und spawnInterval in Enemy.java änderbar gemacht werden. Variable fallSpeed war als final deklariert und deswegen wurde das Schlüsselwort final entfernt. Die Main-Klasse brauchte Zugriff, um Parameter zu ändern und deswegen wurden die Setter-Methoden (setSpawnInterval, setFallSpeed) hinzugefügt.

Ohne LLM erkannt.

code:

```
// Enemy.java (Korrekturen und Ergänzungen)

// Eigenschaften der Gegner
private final float enemyWidth = 30f;
private final float enemyHeight = 25f;
private float fallSpeed = 180f; // <-- 'final' entfernt!

// ...

// HIER EINFÜGEN: Setter-Methoden
public void setSpawnInterval(float newInterval) {
    this.spawnInterval = newInterval;
```

```

}

public void setFallSpeed(float newSpeed) {
    this.fallSpeed = newSpeed;
}

```

## 10. Level-Management in Main.java

### 10.1 Neue Member-Variablen

-Zustandsvariablen wurden der Klasse Main hinzugefügt

Prompt für Gemini: Level wechsel soll nach 15 gesammelten Punkten passieren.

code:

```

// Level-Management-Variablen
private int currentLevel = 1;
private final int LEVEL_THRESHOLD = 15; // Punkte, die für Level 2
erreicht werden müssen
private float levelChangeTimer = 0f;
private final float LEVEL_CHANGE_DURATION = 2.0f; // 2 Sekunden
Pause/Anzeige

```

### 10.2 Aktualisierte render()-Methode (Logik und Optik)

-Levelwechsel-Logik: Prüft, ob currentLevel == 1 und ob player.getPoints() >= LEVEL\_THRESHOLD (15 Punkte). Bei Erfüllung wird currentLevel auf 2 gesetzt und die Gegnerparameter über die neuen Setter-Methoden angepasst. -Schwierigkeitssteigerung (Level 2): -Spawn-Intervall: von 2.6s auf 0.5s -Fallgeschwindigkeit: von 180 px/s auf 250 px/s Optik-Wechsel: Die Hintergrundfarbe wird basierend auf currentLevel gesetzt, und während des Levelwechsels wird eine große "LEVEL 2"-Anzeige eingeblendet. Level 1: Dunkelblau Level 2: Dunkelrot

Prompt für Gemini: Ich brauche ein Level 2, welches man durch eine andere Optik unterscheiden kann und schwieriger durch erhöhte Spawnrate und Fallgeschwindigkeit ist.

code in Main.java:

```

if (currentLevel == 1 && player.getPoints() >= LEVEL_THRESHOLD) {
    currentLevel = 2;
    // Gegner-Parameter für Level 2 setzen (mehr Spawns)
    enemy.setSpawnInterval(0.5f); // Verkürze das Spawn-
Intervall (mehr Gegner)
    enemy.setFallSpeed(250f); // Schnellere Gegner
    levelChangeTimer = LEVEL_CHANGE_DURATION;
}

if (levelChangeTimer > 0) {
    // Levelwechsel-Pause
    levelChangeTimer -= delta;
}

```

```

    ...
    ...
    if (currentLevel == 1) {
        Gdx.gl.glClearColor(0.1f, 0.1f, 0.2f, 1); // Dunkelblau
    } else {
        Gdx.gl.glClearColor(0.2f, 0.1f, 0.1f, 1); // Dunkelrot
    }
}

```

## 11. Level Font hinzugefügt

- Damit man weiß in welchem Level man sich befindet, wird oben rechts das Aktuelle Level angezeigt.

Promt für Copilot: änderer den code so, dass oben rechts im bild das Level angezeigt wird, in dem man sich befindet.

Code in Main.java:

```

// Level rechts oben
String levelText = "Level: " + currentLevel;
float textWidth = font.getRegion().getRegionWidth(); // grobe
Breite
font.draw(batch, levelText, camera.viewportWidth - 125,
camera.viewportHeight - 20);
batch.end();

```

## Level 2 Aussehen

Wenn man 15 Punkte erreicht hat, kommt folgendes Bild:

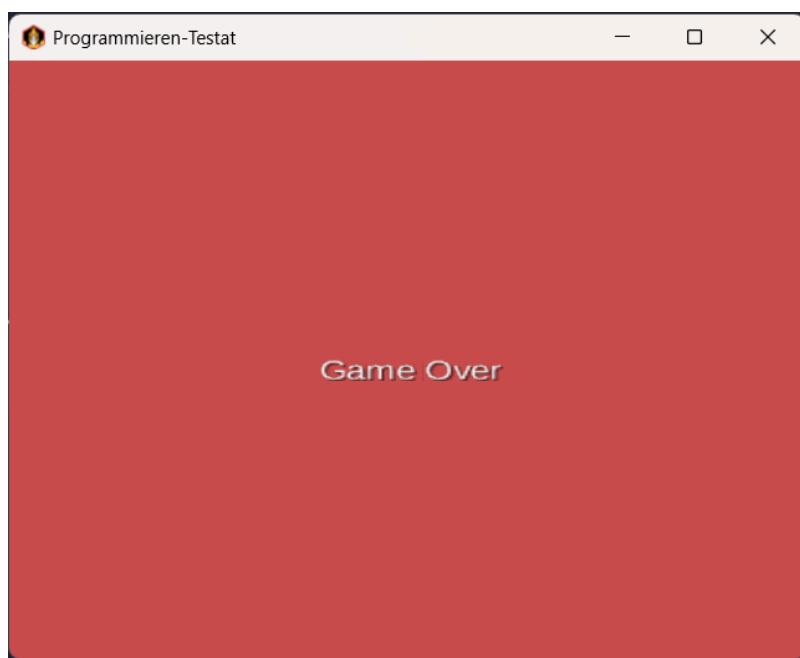


Und nach 2 Sekunden geht das Spiel dann mit einer anderen Hintergrundfarbe und schwierigeren Einstellungen weiter:



## 12. 'Game Over' integriert

- Statt dass der Spieler einen Punkt verliert, wenn er den Gegner berührt, haben wir uns dafür entschieden, dass das Spiel vorbei ist und ein 'Game Over' Bild erscheint.



- Dafür wurde erst der Rückgabewert der Funktion badCollisions(Player player) in Enemy.java auf boolean geändert.

Code (ohne LLM):

```
public boolean badCollisions(Player player) {
    Iterator<EnemyUnit> it = enemies.iterator();
    while (it.hasNext()) {
        EnemyUnit e = it.next();
        if (overlaps(e.x, e.y, e.width, e.height,
                    player.getX(), player.getY(),
                    player.getWidth(), player.getHeight())) {
            return true;
        }
    }
    return false;
}
```

```

        player.getWidth(), player.getHeight())) {
            it.remove();
            return true; // Kollision → Game Over
        }
    }
    return false; // keine Kollision
}

```

- Dann wurde in Main.java in der render-Methode eine if-Abfrage gemacht (`if (enemy.badCollisions(player) == true)`)
  - Und in der if-Abfrage wurde der Code, der für das 'Level 2' Bild verwendet wurde reinkopiert

Code (ohne LLM):

```

// Prüfe, ob der Spieler mit einem Enemy kollidiert
(Punktabzug)
if (enemy.badCollisions(player) == true) {
    gameState = false;

    // 'Game Over' Hintergrundfarbe
    Gdx.gl.glClearColor(0.8f, 0.3f, 0.3f, 1); // Hellrot für
    'Game Over'
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    camera.update();
    batch.setProjectionMatrix(camera.combined);
    batch.begin();
    // Große Anzeige des Levelwechsels
    font.getData().setScale(3.0f); // Größere Schrift
    font.draw(batch, "Game Over", camera.viewportWidth / 2 -
    110, camera.viewportHeight / 2);
    font.getData().setScale(2.0f); // Zurück zur normalen Größe
    batch.end();
    return; // Breche die normale Render- und Update-Logik ab
}

```

- Noch hat es aber nicht funktioniert, weil jedes mal wenn man einen Gegner berührt hat, das 'Game Over' Bild ganz kurz erschienen ist und das Spiel danach weiter ging.
- Das Spiel soll danach jedoch stoppen.
- Also wurde Copilot wieder benutzt.
  - Er meinte das man eine boolean Variable braucht die den Spielstatus festlegt und am Anfang der render-Methode eine if-Abfrage ob der Status false oder true ist.

Promp für Copilot: ich programmiere ein 2d spiel mit libGDX. ich möchte das wenn der spieler den enemy berührt ein bild kommt mit game over. das passiert auch, aber das bild ist nur ganz kurz da und dann ist es wieder weg und das spiel geht weiter obwohl es aufhören sollte.

Code der in Main.java in der render-Methode übernommen wurde:

```

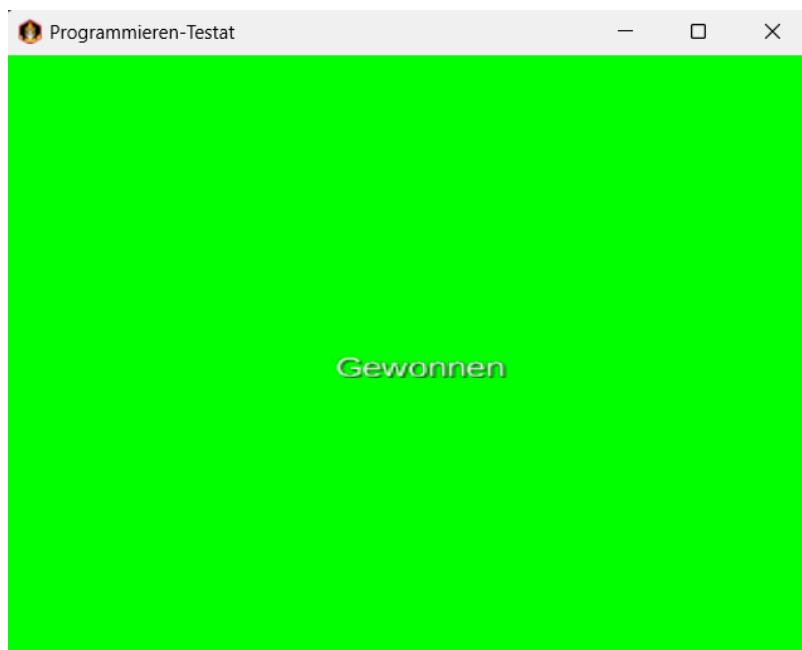
private boolean gameState = true;

// Render-Schleife
@Override
public void render() {
    if (gameState == false) {
        return; // Spiel ist vorbei, keine Updates mehr
    }
    ...
}

```

## 12. 'Gewonnen' integriert (Ohne LLM)

- Nach dem im zweiten Level 40 Coins gesammelt wurden, wird das Spiel beendet und ein grünes Bild mit dem Text 'Gewonnen' erscheint



- dafür eine neue Variable 'int LEVEL\_THRESHOLD2' hinzugefügt

```

private final int LEVEL_THRESHOLD2 = 40; // Punkte, die fürs
gewinnen erreicht werden müsse

```

- um das Spiel erfolgreich zu beenden sobald 40 Coins gesammelt wurden, wurde folgende if schleife geschrieben:

```

if (player.getPoints() >= LEVEL_THRESHOLD2) {
    gameState = false;

    // Gewonnen Hintergrundfarbe
    Gdx.gl.glClearColor(0f, 1f, 0f, 1); // Grün für Gewonnen
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    camera.update();
    batch.setProjectionMatrix(camera.combined);
    batch.begin();
    // Große Anzeige wenn Spielgewonnen
    font.getData().setScale(3.0f); // Größere Schrift
}

```

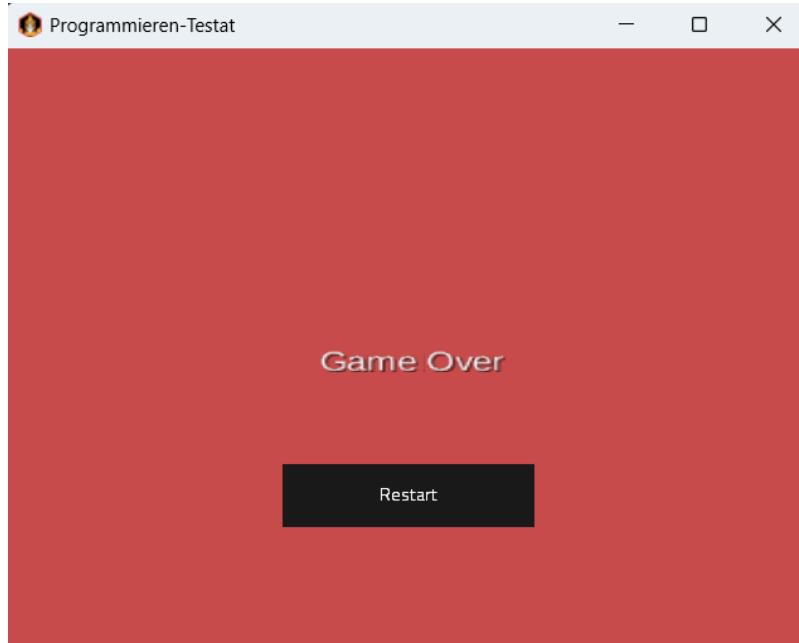
```

        font.draw(batch, "Gewonnen", camera.viewportWidth / 2 - 90,
camera.viewportHeight / 2);
        font.getData().setScale(2.0f); // Zurück zur normalen Größe
batch.end();
return; // Breche die normale Render- und Update-Logik ab
}

```

## 14. Restart Button

- um das Spiel nach dem Game Over weiter zu spielen, haben wir einen Restart Button nach dem 'Game Over' und 'Gewonnen' eingebaut



- der selbe Button kommt auch beim Gewinnen des Spiels
- für das Erstellen des Button haben wir folgenden Prompt benutzt: "wie macht man mit libgdx ein button"

Code in Main.java wurde in create-Methode benutzt:

```

// Stage initialisieren mit einem ScreenViewport (passt sich an
Fenstergröße an)
stage = new Stage(new ScreenViewport());

// Stage als Input-Processor setzen, damit Maus/Touch-Events an
die Stage gehen
Gdx.input.setInputProcessor(stage);

// Skin aus den internen Assets Laden (uiskin.json muss im
assets-Ordner liegen)
skin = new Skin(Gdx.files.internal("libGDX-
Environment\\assets\\ui\\uiskin.json"));

// TextButton erstellen: sichtbarer Text "Klick mich" und das
geladene Skin verwenden
button = new TextButton("Restart", skin);

```

```

// Position des Buttons in Stage-Koordinaten setzen (x=100,
y=100)
button.setPosition(220, 100);

// Größe des Buttons setzen (Breite=200, Höhe=50)
button.setSize(200, 50);

// Listener hinzufügen: reagiert auf Zustandsänderungen wie
Klicks
button.addListener(new ChangeListener() {
    @Override
    public void changed(ChangeEvent event,
com.badlogic.gdx.scenes.scene2d.Actor actor) {
        // Diese Methode wird aufgerufen, wenn der Button
gedrückt/aktiviert wurde
        System.out.println("Button gedrückt!");
    }
});
```

*// Button der Stage hinzufügen, damit er gezeichnet und aktualisiert wird*

```
stage.addActor(button);
```

- nach dem testen des Spiels erschien aber nur ein schwarzer Screen anstelle des Game Over Fensters mit dem Restart Button. Stellte sich dan raus das wir diese Zeile vergessen hatten einzugeben:

```

// Button der Stage hinzufügen, damit er gezeichnet und aktualisiert wird
stage.addActor(button);
```

- nach dem der Restart Button gedrückwurden war, wurde nur der String "Button gedrückt" in Terminal ausgegeben. Also haben wir (ohne LLM) die Aktion des Button geändert mit folgendem Code:

```

// Listener hinzufügen: reagiert auf Zustandsänderungen wie
Klicks
button.addListener(new ChangeListener() {
    @Override
    public void changed(ChangeEvent event,
com.badlogic.gdx.scenes.scene2d.Actor actor) {
        gameState = true;
        currentLevel = 1;
        player.setPoints(0);
        enemy.setSpawnInterval(2.6f); // Verkürze das Spaw-
Intervall (mehr Gegner)
        enemy.setFallSpeed(180f);

        camera.update();

        // Zeichne Coins und Spieler im selben ShapeRenderer-
Durchlauf
shapeRenderer.setProjectionMatrix(camera.combined);
shapeRenderer.begin(shapeRenderer.ShapeType.Filled);
coins.render(shapeRenderer);
```

```

        enemy.render(shapeRenderer);
        player.render(shapeRenderer);
        shapeRenderer.end();

        // Punktstand anzeigen (oben Links, Y = viewportHeight
- margin)
        batch.setProjectionMatrix(camera.combined);
        batch.begin();
        float yPos = camera.viewportHeight - 20; // 20 Pixel
Abstand zur Oberkante
        font.draw(batch, "Punkte: " + player.getPoints(), 10,
yPos);

        // Level rechts oben
        String levelText = "Level: " + currentLevel;
        float textWidth = font.getRegion().getRegionWidth(); //
grobe Breite
        font.draw(batch, levelText, camera.viewportWidth - 125,
camera.viewportHeight - 20);

        batch.end();

    }
});

```

- damit nach dem Restart auch der Punktstand zurückgesetzt wird haben wir ein Setter für die Points gemacht

```

public void setPoints(int points) {
    this.points = points;
}

```