HSB
Hochschule Bremen
City University of Applied Sciences

Hochschule Bremen
- University of Applied Sciences -

Fakultät 4

Medieninformatik 2

Interaktive Multimedia Anwendung in einer (2D) Grafik - Mit Phaser.io

Internationaler Studiengang Medieninformatik B. Sc.

Thematik:	Entwicklung einer 2D Multimedia Anwendung mit Phaser 3
Autor:	Niclas Jeremy Martin Rieckers E-Mail: nrieckers@stud.hs-bremen.de Matrikelnummer: 5198771 Fachsemester: 2
Version vom:	29. Juli 2024
Dozent:	Prof. Dr. Volker Paelke

Zusammenfassung

Im Rahmen der MI 2-Veranstaltung ist es erforderlich, in einem Einzelprojekten die technische Umsetzung zu demonstrieren, um die Fähigkeit zur Programmierung interaktiver Multimedia-Projekte nachzuweisen. Projektideen und Gestaltung können sich dabei an bekannten Spielen wie den Klassikern der 80er Jahre, etwa Zelda, orientieren.

Die Kriterien für ein Einzelprojekt beinhalten:

- Die Entwicklung eines fehlerfrei lauffähigen, spielbaren Projekts.
- Die Implementierung von mindestens drei verschiedenen Interaktionsmöglichkeiten.
- Die Nutzung von Keyframe-Animationen.
- Die Einbindung von Kinematik oder Dynamik (Physik).
- Die sinnvolle Verwendung von Audio und Sound als Feedback.
- Die Dokumentation der implementierten Features sowie deren Demonstration in einem kurzen Video (maximal 3 Minuten).

Da es sich bei meiner Arbeit um ein Einzelprojekt handelt ist die Verwendung existierender Grafik- und Sound-Assets, unter der Voraussetzung, dass diese angegeben werden, erlaubt.

Inhaltsverzeichnis

Listingverzeichnis

1 Konzeption

1.1 Konzeptentwicklung

Das Spiel soll ein actiongeladenes Spiel sein, das in einem futuristischen Hotel stattfindet, das von feindlichen Eindringlingen überrannt wird. Der Spieler übernimmt die Rolle eines Elitekämpfers, der die Etagen des Gebäudes durchkämpfen muss, um die Eindringlinge zu eliminieren und das Gebäude zu sichern.

- Rezeption als Zentrale: Die Rezeption soll als Ausgangspunkt für den Spieler dienen. Hier kann er verschiedene Aktionen durchführen, wie z.B. Waffen kaufen und aufrüsten, seine Statistiken überprüfen und den nächsten Angriff planen.(nicht erfüllt)
- Waffenshop: In der Zentrale befindet sich ein Waffenshop, in dem der Spieler verschiedene Waffen kaufen kann, um sich auf die bevorstehenden Kämpfe vorzubereiten. Jede Waffe hat ihre eigenen einzigartigen Animationen und Eigenschaften.(nicht erfüllt)
- Mehrere Etagen: Das Gebäude besteht aus insgesamt sieben Etagen, die jeweils mit einem unterschiedlichen Schwierigkeitsgrad gestaltet sind. Jede Etage hat 12 Zimmer, die durchnummeriert sind (z.B. 101 bis 712).(teilweise erfüllt)
- Gegner-Management: Es gibt verschiedene Gegner-Templates, aus denen zufällig Feinde für jede Etage generiert werden. Das Level der Gegner steigt proportional zum Fortschritt des Spielers, was für eine zunehmende Herausforderung sorgt.(teilweise erfüllt)
- Gravitation: Die Gravitation im Spiel ist um das 1,4-fache erhöht, was zu einer interessanten Spielmechanik führt und dem Spieler erlaubt, auf neue Weise zu navigieren und zu kämpfen.(teilweise erfüllt)
- Punktesystem: Der Spieler verdient Punkte, indem er Gegner eliminiert. Der Score wird nach der Formel Z = X + (1,4 * (Y/2)) berechnet, wobei X der übliche Score, Y das Level des Spielers und Z der endgültige Score ist.(teilweise erfüllt)
- Spielabbruch-Button: In der Zentrale gibt es einen speziellen Button, mit dem der Spieler das Spiel abbrechen kann. Wenn der Spieler das Spiel abbricht, wird sein aktueller Score gespeichert. Wenn der Spieler stirbt, wird sein Score auf 0 gesetzt.(anders erfüllt)
- Bosskämpfe: In der siebten Etage des Gebäudes sollen mächtige Bosse darauf warten, vom Spieler herausgefordert zu werden. Diese Bosse sollen besonders stark sein und erfordern strategisches Denken sowie Geschicklichkeit, um diese zu besiegen. Jeder Boss soll einzigartige Angriffsmuster und Fähigkeiten erhalten(Zukünftige Entwicklung), die es zu überwinden gilt.(nicht erfüllt)
- Begehbare Räume (zukünftige Entwicklung): In späteren Entwicklungsphasen könnten die Räume begehbar gemacht werden, was dem Spieler ermöglicht, das Gebäude zu erkunden und geheime Bereiche zu entdecken.(nicht erfüllt)

Das Spiel soll ein packendes Spielerlebnis sein, das Geschicklichkeit, Strategie und Schnelligkeit erfordert, um zu überleben und das Gebäude zu verteidigen.

1.2 Technische Konzeption

1.2.1 Technologien

- **HTML5:** Strukturierung der Webanwendung.
- **CSS3:** Gestaltung und Layout der Benutzeroberfläche.
- **JavaScript:** Implementierung der Logik und Interaktivität.
- **Phaser 3:** Game-Engine zur Erstellung der 2D-Grafiken und Spielmechaniken.

1.2.2 Entwicklungsumgebung

- **Editor:** Visual Studio Code
- **Versionskontrolle:** Git und GitHub für die Verwaltung des Quellcodes.

1.2.3 Spielmechaniken

- **Bewegung:** Der Spieler kann die Spielfigur nach links/rechts bewegen und springen zuschlagen sowie ein Messer werfen.
- **Gegner:** Einfache KI-gesteuerte Gegner, die der Spieler überwinden muss.

1.2.4 Animationen

- **Keyframe-Animationen:** Für die Spielfigur und Gegnerbewegungen.
- **Spritesheets:** Verwendung von Spritesheets für flüssige Animationen.

1.2.5 Sounddesign

- **Hintergrundmusik:** Atmosphärische Musik, die das Spielerlebnis unterstützt.
- **Soundeffekte:** Feedback für Aktionen wie Sprünge, Kollisionen und das Sammeln von Gegenständen.

1.2.6 Audioformate

- Verwendung von gängigen Audioformaten wie MP3 zur Unterstützung der meisten Browser.

1.2.7 Physik-Engine

Phaser 3 bietet integrierte Unterstützung für Physik, die genutzt wird, um realistische Bewegungen und Kollisionen zu simulieren.

2 Technische Umsetzung

2.1 Platform und Werkzeuge

In meinem Projekt habe ich folgende Plattformen und Werkzeuge verwendet:

- **Phaser 3:** Phaser 3 ist ein vielseitiges und leistungsstarkes Framework für die Entwicklung von 2D-Spielen mit JavaScript. Es wird häufig für die Erstellung von Spielen für Webbrowser eingesetzt und bietet Entwicklern eine breite Palette von Werkzeugen und Funktionen zur Gestaltung interaktiver und visuell ansprechender Spiele. Durch die Unterstützung von WebGL und Canvas kann Phaser 3 Grafiken effizient rendern und ermöglicht die Umsetzung komplexer Spielmechaniken, darunter Physik-Engines, Animationen, Eingabeverarbeitung und Soundunterstützung.
- **Leshy Labs Spritesheet Tool:** Das Leshy Labs Spritesheet Tool ist eine webbasierte Anwendung, die speziell für die Erstellung und Bearbeitung von Spritesheets konzipiert ist. Spritesheets sind ein wesentlicher Bestandteil der Spieleentwicklung, da sie alle Grafiken eines Charakters oder Objekts in einem einzigen Bild zusammenfassen, was die Leistung optimiert und die Verwaltung von Animationen vereinfacht. Das Tool bietet benutzerfreundliche Bearbeitungswerkzeuge und Exportoptionen, die die Integration in verschiedene Spiel-Engines erleichtern.

2.2 Implementierung der Spielmechanik

Zu beginn müssen folgende Mechaniken fest gesetzt werden:

- Was ist das Hauptziel?
- Welche Regeln und Einschränkungen gelten?
- Wie interagieren Spieler mit dem Spiel? (z.B. Bewegung, Angriff, Sammeln von Items)
- Wie reagiert das Spiel auf die Aktionen des Spielers?

2.2.1 Was ist das Hauptziel?

Das Hauptziel des Spiels besteht darin, so viele Zombies wie möglich zu besiegen und anschließend zum Aufzug zurückzukehren. Die Spielmechanik ist in zwei wesentliche Phasen unterteilt:

- **Zombies Besiegen:**
  1. **Herausforderung:** Die Spieler müssen sich durch verschiedene Level kämpfen, in denen ihnen eine zunehmende Anzahl von Zombies begegnet.
  2. **Strategie:** Spieler müssen ihre Ressourcen und Fähigkeiten geschickt einsetzen, um die Zombies effektiv zu bekämpfen. Dies kann den Einsatz von Waffen, das Ausweichen von Angriffen und das strategische Positionieren im Raum umfassen.
  3. **Punkte:** Jeder besiegte Zombie zählt zu einem Punktestand, der die Leistung des Spielers reflektiert. Höhere Punktzahlen können mit Belohnungen oder Upgrades im Spiel verbunden sein.
- **Die Rückkehr zum Aufzug:**
  1. **Mission:** Die Spieler müssen zum Aufzug zurückkehren, um das Level zu beenden und ihren Score zu speichern.

2.2.2 Wie interagieren Spieler mit dem Spiel? (z.B. Bewegung, Angriff, Sammeln von Items)

Die Spieler interagieren mit dem Spiel durch folgende Aktionen:

- **Bewegung:**
  - `A`: Bewege den Charakter nach links.
  - `D`: Bewege den Charakter nach rechts.
  - `W`: Lasse den Charakter springen.
- **Angriff:**
  - `Q`: Führe einen Schlag aus, der den Spieler 30 Lebenspunkte hinzufügt.
  - `E`: Wirf ein Messer, das den Spieler 10 Lebenspunkte hinzufügt.
- **Sammeln und Spielziel:**
  - `F`: Gewinnen des Spiels (nur verfügbar am Aufzug).

2.2.3 Wie reagiert das Spiel auf die Aktionen des Spielers?

Der bereitgestellte Code beschreibt, wie das Spiel auf die Aktionen des Spielers reagiert, insbesondere im Kontext der Interaktion mit Feinden. Hier ist eine detaillierte Erklärung, wie die Feinde auf die Präsenz und das Verhalten des Spielers reagieren:

- **Feindverhalten und Physik:**
  - Jeder Feind wird kontinuierlich aktualisiert, um sicherzustellen, dass seine physikalischen Eigenschaften korrekt verwaltet werden. Dazu gehört das Festlegen von Eigenschaften wie der Kollisionsgrenze mit der Welt (`setCollideWorldBounds(true)`) und der Schwerkraft (`setGravityY(600)`).
- **Angriffsverhalten:**
  - **Überprüfung der Reichweite:** Wenn ein Feind nicht angreift (`!enemy.isAttacking`), berechnet der Code die Entfernung zwischen dem Feind und dem Spieler.
  - **Angriffsauslösung:** Wenn der Feind innerhalb eines Angriffsbereichs von 50 Pixeln zum Spieler ist, wird der Angriff des Feindes ausgelöst:
    - Der Feind stoppt seine Bewegung (`setVelocityX(0)`) und spielt die Angriffsanimation (`enemy.anims.play('enemyAttack', true)`).
    - Es werden Ereignis-Listener an die Animation des Feindes angehängt. Sobald die Angriffsanimation abgeschlossen ist (`animationcomplete`), wird der Zustand `isAttacking` zurückgesetzt. Wenn der Spieler immer noch im Angriffsbereich ist, wird die Methode `hitByEnemy(player, enemy)` aufgerufen, um die Logik für den Angriff zu verarbeiten.
- **Verhalten beim Verfolgen des Spielers:**
  - **Verfolgungsreichweite:** Wenn der Feind nicht angreift und sich innerhalb einer Verfolgungsreichweite von 800 Pixeln befindet, wechselt der Feind in den Laufmodus (`enemy.anims.play('enemyRun', true)`) und bewegt sich mit einer Geschwindigkeit von 300 Pixeln pro Sekunde auf den Spieler zu (`this.physics.moveToObject(enemy, player, 300)`).
- **Idle-Verhalten:**
  - **Außerhalb der Reichweite:** Wenn der Feind sich nicht im Angriffs- oder Verfolgungsbereich befindet, wird der Feind in den Leerlaufmodus versetzt (`enemy.anims.play('enemyIdle', true)`), und seine Geschwindigkeit wird auf Null gesetzt (`setVelocityX(0)` und `setVelocityY(0)`).

Zusammenfassend reagiert das Spiel auf die Aktionen des Spielers, indem es das Verhalten der Feinde dynamisch anpasst. Feinde reagieren auf die Nähe des Spielers, indem sie entweder angreifen, verfolgen oder inaktiv bleiben. Dieses Verhalten sorgt für eine herausfordernde und abwechslungsreiche Spielerfahrung, indem die Feinde gezielt auf die Position und Aktionen des Spielers reagieren.

```
updateEnemies() {
    enemys.children.iterate((enemy) => {
        if (enemy) {
            // Maintain enemy physics properties
            enemy.setCollideWorldBounds(true);
            enemy.setGravityY(600);

            if (!enemy.isAttacking) {
                const distance = Phaser.Math.Distance.Between(player.x, player.y, e

                if (distance < 50) { // If within attack range
                    enemy.isAttacking = true;
                    enemy.setVelocityX(0); // Stop movement during attack
                    enemy.anims.play('enemyAttack', true);

                    // Attach event listeners to the animation on this enemy
                    enemy.once('animationcomplete', () => {
                        enemy.isAttacking = false; // Reset state after attack
                        if (Phaser.Math.Distance.Between(player.x, player.y, enemy.
                            this.hitByEnemy(player, enemy); // Attack logic
                        }
                    });

                } else if (distance < 800) { // If within chase range
                    enemy.anims.play('enemyRun', true);
                    this.physics.moveToObject(enemy, player, 300);
                } else { // Idle behavior
                    enemy.anims.play('enemyIdle', true);
                    enemy.setVelocityX(0);
                    enemy.setVelocityY(0);
                }
            }
        }
    });
}
```

Listing 1: Code zur Aktualisierung von Feinden

2.3 Interaktive Elemente

Der folgende Code beschreibt, wie die Interaktionen des Spielers im Spiel verarbeitet werden. Es zeigt, wie die verschiedenen Tasteneingaben die Bewegungen und Aktionen des Spielers beeinflussen:

```
// Check player controls and animations
checkPlayer() {
    // Utility function to wait for an animation to complete
    function playAnimationAndWait(animation) {
        return new Promise(resolve => {
            player.play(animation);
            player.once('animationcomplete', resolve);
        });
    }

    // Check if any animation is currently playing
    if (!isAnimating) {
        if (aKey.isDown) { // Check if move left key is pressed
            player.setVelocityX(-360); // Set horizontal velocity for moving left
            player.flipX = true; // Flip player sprite horizontally
            if (player.body.touching.down) {
                player.anims.play('run', true); // Play run animation if player is
            }
        } else if (dKey.isDown) { // Check if move right key is pressed
            player.setVelocityX(360); // Set horizontal velocity for moving right
            player.flipX = false; // Ensure player sprite is not flipped
            if (player.body.touching.down) {
                player.anims.play('run', true); // Play run animation if player is
            }
        } else {
            player.setVelocityX(0); // Stop horizontal movement
            if (player.body.touching.down) {
                player.anims.play('idle', true); // Play idle animation if player
            }
        }

        if (wKey.isDown && player.body.touching.down) { // Check if jump key is pr
            player.setVelocityY(-360); // Set vertical velocity for jumping
            player.anims.play('jump', true); // Play jump animation
        }

        if (qKey.isDown) { // Check if punch key is pressed
            isAnimating = true; // Set animation flag
            playAnimationAndWait('punch').then(() => {
                isAnimating = false; // Reset animation flag
                this.hitEnemyWithPunch(); // Execute punch action
            });
        } else if (eKey.isDown) { // Check if throw key is pressed
            isAnimating = true; // Set animation flag
            playAnimationAndWait('throw').then(() => {
                isAnimating = false; // Reset animation flag
                this.hitEnemyWithThrow(); // Execute throw action
            });
        } else if (fKey.isDown) { // Check if end game key is pressed
            if ((elevator.x + 50 >= player.x) && (elevator.x - 50 <= player.x)) {
                isAnimating = true; // Set animation flag
                player.destroy(); // Remove player sprite
                elevator.play('close').once('animationcomplete', () => {
                    showScore(score);
```

```
55                          isAnimating = false; // Reset animation flag
56                      });
57                  }
58              }
59          }
60      }
```

Listing 2: Code zur Überprüfung der Spielersteuerung und Animationen

- **Bewegung nach links und rechts:**
  - Wenn die `A`-Taste gedrückt wird, bewegt sich der Spieler nach links mit einer Geschwindigkeit von 360 Pixeln pro Sekunde. Der Spieler-Sprite wird horizontal gespiegelt, um die Bewegungsrichtung anzuzeigen. Der Lauf-Animations-Clip wird abgespielt, wenn der Spieler den Boden berührt.
  - Wenn die `D`-Taste gedrückt wird, bewegt sich der Spieler nach rechts mit der gleichen Geschwindigkeit. Der Sprite wird nicht gespiegelt, und die Lauf-Animation wird ebenfalls abgespielt, wenn der Spieler den Boden berührt.
  - Wenn keine der Bewegungstasten gedrückt wird, wird die horizontale Bewegung gestoppt, und die Idle-Animation wird abgespielt, solange der Spieler den Boden berührt.
- **Springen:**
  - Wenn die `W`-Taste gedrückt wird und der Spieler den Boden berührt, wird eine vertikale Geschwindigkeit von 360 Pixeln pro Sekunde nach oben eingestellt, um den Sprung zu ermöglichen. Die Sprung-Animation wird abgespielt, um visuelles Feedback für den Sprung zu geben.
- **Angriffe:**
  - Wenn die `Q`-Taste gedrückt wird, wird die Schlag-Animation abgespielt. Während der Animation wird eine Flagge gesetzt, um weitere Animationen zu verhindern. Nach Abschluss der Schlag-Animation wird diese Flagge zurückgesetzt, und die Methode `hitEnemyWithPunch()` wird aufgerufen, um den Angriff zu verarbeiten.
  - Wenn die `E`-Taste gedrückt wird, wird eine Wurf-Animation abgespielt, und ähnliche Logik wie beim Schlag wird verwendet, um nach Abschluss der Animation den Angriff zu verarbeiten (`hitEnemyWithThrow()`).
- **Spielende:**
  - Wenn die `F`-Taste gedrückt wird und der Spieler sich in der Nähe des Aufzugs befindet, wird die Spieler-Sprite entfernt, und der Aufzug spielt eine Schließ-Animation ab. Nach Abschluss dieser Animation wird die Punktzahl angezeigt, und die Animation-Flagge wird zurückgesetzt.

2.4 Animation und Grafik

In diesem Abschnitt wird erläutert, wie Keyframe-Animationen erstellt und grafische Gestaltungselemente in das Spiel integriert werden. Der folgende Code zeigt die wichtigsten Schritte der Implementierung:

2.4.1 Laden der grafischen Ressourcen

```
preload(){
    // Load elevator sprite atlas and JSON data
    this.load.atlas('elevator', 'assets/elevator.jpg', 'assets/elevator.json');
    // Load player sprite atlas and JSON data
    this.load.atlas('player', 'assets/Character.jpg', 'assets/player.json');
    // Load enemy sprite atlas and JSON data
    this.load.atlas('enemy_001', 'assets/Enemy_001.jpg', 'assets/enemy_001.json');
}
```

Listing 3: Laden der Sprite-Atlas-Daten

Zuerst werden die benötigten grafischen Ressourcen geladen. Die `load.atlas` Methode lädt die Sprite-Atlas-Dateien, die die Grafiken für den Aufzug, den Spieler und die Feinde enthalten. Diese Daten umfassen sowohl die Bilddateien als auch die zugehörigen JSON-Dateien, die die Struktur der Frames definieren.

2.4.2 Erstellen von Keyframe-Animationen

```
create(){
    this.anims.create({
        key: 'open',
        frames: this.anims.generateFrameNames('elevator', { prefix: 'open', end: 2,
        frameRate: 3, // Frame rate for the animation
        repeat: 0 // Play the animation once
    });

    this.anims.create({
        key: 'close',
        frames: this.anims.generateFrameNames('elevator', { prefix: 'close', end: 2
        frameRate: 3, // Frame rate for the animation
        repeat: 0 // Play the animation once
    });
}
```

Listing 4: Erstellen von Animationen für den Aufzug

Hier werden Keyframe-Animationen für den Aufzug erstellt. Die `this.anims.create` Methode definiert die Animationen für das Öffnen und Schließen des Aufzugs. Die Frames für jede Animation werden aus dem `'elevator'` Atlas geladen und die `frameRate` gibt die Geschwindigkeit der Animation an. Mit `repeat: 0` wird festgelegt, dass die Animation nur einmal abgespielt wird.

2.4.3 Erstellen und Konfigurieren von Spieler- und Feindanimationen

```
loadPlayerAnims() {
    this.anims.create({
        key: 'punch',
        frames: this.anims.generateFrameNames('player', { prefix: 'punch', end: 5,
        frameRate: 12, // Frame rate for punch animation
        repeat: 0 // Play the animation once
    });
    this.anims.create({
        key: 'throw',
        frames: this.anims.generateFrameNames('player', { prefix: 'throw', end: 5,
        frameRate: 12, // Frame rate for throw animation
        repeat: 0 // Play the animation once
    });
    this.anims.create({
        key: 'jump',
        frames: this.anims.generateFrameNames('player', { prefix: 'jump', end: 3, z
        frameRate: 6, // Frame rate for jump animation
        repeat: -1 // Loop the animation
    });
    this.anims.create({
        key: 'run',
        frames: this.anims.generateFrameNames('player', { prefix: 'run', end: 1, ze
        frameRate: 6, // Frame rate for run animation
        repeat: -1 // Loop the animation
    });
}
```

Listing 5: Erstellen von Spieler-Animationen

Zusätzlich zu den Aufzug-Animationen werden auch die Animationen für den Spieler definiert. Dies umfasst Animationen wie `'punch'`, `'throw'`, `'jump'` und `'run'`. Jede Animation besteht aus einer Serie von Frames, die mit `this.anims.create` erstellt werden. Die `frameRate` und `repeat` Parameter steuern die Geschwindigkeit und die Wiederholungsrate der Animationen.

2.4.4 Grafische Gestaltung und Integration

```
create(){
    // Add elevator sprite and scale it
    elevator = this.add.sprite(200, 560, 'elevator').setScale(1.5);

    // Play elevator opening animation and handle completion
    isAnimating = true;
    elevator.play('open').once('animationcomplete', () => {
        isAnimating = false; // Animation is complete

        // Create the player sprite with physics and scaling
        player = this.physics.add.sprite(200, 625, 'player').setScale(2).refreshBod

        // Load player animations
        this.loadPlayerAnims();

        // Set player physics properties
        player.setBounce(0.2); // Add bounce effect
        player.setCollideWorldBounds(true); // Prevent player from leaving the worl
    });
}
```

Listing 6: Hinzufügen und Skalieren der Sprites

In diesem Abschnitt wird der Aufzug zur Szene hinzugefügt und skaliert. Das gleiche gilt für den Spieler, der ebenfalls hinzugefügt und mit Physik-Engine-Eigenschaften versehen wird. Die `setScale` Methode passt die Größe der Sprites an, um sie an die gewünschte Spielgrafik anzupassen.

Der Einsatz von Keyframe-Animationen und die grafische Gestaltung sind entscheidend für das visuelle Erlebnis im Spiel. Die Keyframe-Animationen werden verwendet, um flüssige Bewegungen und Interaktionen der Spielobjekte darzustellen, während die grafische Gestaltung sicherstellt, dass diese Objekte in der Spielwelt korrekt angezeigt und skaliert werden. Die Kombination dieser Elemente trägt zu einem ansprechenden und dynamischen Spielerlebnis bei.

2.5 Kinematik und Dynamik

2.5.1 Deklaration der Spielvariablen

```
var platforms; // Plattformen-Gruppe f r statische Objekte
var player; // Spieler-Charakter-Sprite
var enemys; // Gruppe f r Feind-Sprites
```

Listing 7: Deklaration der Spielvariablen

Diese Variablen werden verwendet, um verschiedene Spielobjekte zu verwalten: Plattformen für statische Objekte, der Spieler und eine Gruppe von Feinden.

2.5.2 Erstellen der Plattformen

```
addPlatforms(x, y, width) {
    for (var i = 0; i < width / 64; i++) {
        platforms.create(x + (i * 64), y, 'ground').refreshBody();
    }
}
```

Listing 8: Funktion zum Hinzufügen von Plattformen

Hier wird eine Reihe von Plattformen erstellt, die entlang der X-Achse ausgerichtet sind. Jede Plattform ist ein statisches physikalisches Objekt, das es dem Spieler und den Feinden ermöglicht, darauf zu interagieren.

2.5.3 Erstellen der Feinde

```
addEnemies() {
    for (let i = 0; i < 20; i++) {
        let enemy = enemys.create((1000 + (i * (400*(i/3)))), 600, 'enemy_001').set
        enemy.setGravityY(6000); // Schwerkraft f r Feind setzen
    }
}
```

Listing 9: Funktion zum Hinzufügen von Feinden

Diese Funktion fügt Feinde zur Szene hinzu und setzt ihre Schwerkraft, damit sie sich physikalisch korrekt verhalten.

2.5.4 Erstellen von Animationen

```
this.anims.create({
    key: 'open',
    frames: this.anims.generateFrameNames('elevator', { prefix: 'open', end: 2, zero
    frameRate: 3,
    repeat: 0
});
```

Listing 10: Erstellen von Elevator-Animationen

Hier werden Animationen für verschiedene Spielobjekte wie den Aufzug definiert. Diese Animationen helfen dabei, visuelle Effekte wie das Öffnen und Schließen des Aufzugs darzustellen.

2.5.5 Spieler- und Feindphysik

```
player.setBounce(0.2); // Rückprall-Eigenschaft hinzufügen
player.setCollideWorldBounds(true); // Verhindert das Verlassen der Weltgrenzen
```

Listing 11: Physik-Eigenschaften des Spielers

```
enemy.setGravityY(600); // Schwerkraft für Feind setzen
```

Listing 12: Physik-Eigenschaften der Feinde

Diese Einstellungen fügen Rückprall-Eigenschaften zum Spieler hinzu und setzen die Schwerkraft für Feinde, um ein realistisches physikalisches Verhalten zu gewährleisten.

2.5.6 Kollisionsabfragen

```
this.physics.add.collider(platforms, enemys);
```

Listing 13: Kollision zwischen Plattformen und Feinden

```
this.physics.add.collider(player, enemys, this.hitByEnemy, null, this);
```

Listing 14: Kollision zwischen Spieler und Feinden

Diese Funktionen fügen Kollisionen zwischen Plattformen und Feinden sowie zwischen dem Spieler und den Feinden hinzu.

2.5.7 Spielersteuerung und Animationen

```
if (aKey.isDown) {
    player.setVelocityX(-360); // Spieler nach links bewegen
    player.flipX = true; // Spieler-Sprite spiegeln
}
```

Listing 15: Bewegung des Spielers nach links

```
if (wKey.isDown && player.body.touching.down) {
    player.setVelocityY(-360); // Spieler springen
    player.anims.play('jump', true); // Spring-Animation abspielen
}
```

Listing 16: Springen des Spielers

Diese Codesnippets zeigen, wie der Spieler bewegt und animiert wird, basierend auf den Tasteneingaben.

2.5.8 Überprüfen der Kollisionen und Aktionen

```
hitByEnemy(player, enemy) {
    playerHealth -= 1; // Gesundheit des Spielers reduzieren
    if (playerHealth <= 0) {
        gameOver(score); // Spiel beenden
    }
}
```

Listing 17: Spieler wird von Feind getroffen

Diese Funktion behandelt die Kollision zwischen dem Spieler und den Feinden, reduziert die Gesundheit des Spielers und überprüft, ob das Spiel beendet werden muss.

2.5.9 Werfen von Projektilen

```
let thr = this.physics.add.sprite(player.x + 15, player.y - 25, 'thr').setScale(0.2
thr.setVelocityX(player.flipX ? -1500 : 1500); // Projektilgeschwindigkeit setzen
```

Listing 18: Projektil werfen

```
this.physics.add.overlap(thr, enemys, (thr, enemy) => {
    enemy.destroy(); // Feind zerst ren
    thr.destroy(); // Projektil zerst ren
});
```

Listing 19: Kollision des Projektils mit Feinden

Diese Funktionen zeigen, wie ein Projektil erstellt und geworfen wird, sowie wie es mit Feinden kollidiert.

2.6 Audio und Sound

2.6.1 Laden und Initialisieren von Audio

```
this.load.audio('jumpSound', 'assets/sounds/jump.mp3');
this.load.audio('hitSound', 'assets/sounds/hit.mp3');
this.load.audio('bgMusic', 'assets/sounds/bgMusic.mp3');
```

Listing 20: Laden von Audio-Dateien

Hier werden die Audio-Dateien für den Sprung-Sound, den Treffer-Sound und die Hintergrundmusik geladen. Diese Dateien müssen im Verzeichnis ‘assets/sounds/‘ vorhanden sein.

2.6.2 Wiedergabe von Sounds bei Aktionen

```
jump() {
    this.sound.play('jumpSound'); // Sprung-Sound abspielen
    player.setVelocityY(-300); // Spieler springt
}
```

Listing 21: Abspielen des Sprung-Sounds

In dieser Funktion wird der Sprung-Sound abgespielt, wenn der Spieler springt. Dies bietet unmittelbares akustisches Feedback für die Aktion des Spielers.

```
hitByEnemy(player, enemy) {
    this.sound.play('hitSound'); // Treffer-Sound abspielen
    playerHealth -= 1; // Gesundheit des Spielers reduzieren
    if (playerHealth <= 0) {
        gameOver(score); // Spiel beenden
    }
}
```

Listing 22: Abspielen des Treffer-Sounds

Wenn der Spieler von einem Feind getroffen wird, wird der Treffer-Sound abgespielt, um das Feedback zu verstärken.

2.6.3 Hintergrundmusik

```
create() {
    this.bgMusic = this.sound.add('bgMusic'); // Hintergrundmusik hinzuf gen
    this.bgMusic.setLoop(true); // Hintergrundmusik wiederholen
    this.bgMusic.play(); // Hintergrundmusik abspielen
}
```

Listing 23: Hintergrundmusik starten

Die Hintergrundmusik wird hinzugefügt und so konfiguriert, dass sie kontinuierlich wiederholt wird. Dies schafft eine stimmungsvolle Atmosphäre im Spiel.

2.6.4 Audio-Steuerung

```
adjustVolume() {
    this.bgMusic.setVolume(0.5); // Lautst rke der Hintergrundmusik auf 50% setzen
    this.sound.setVolume(0.8); // Lautst rke aller anderen Sounds auf 80% setzen
}
```

Listing 24: Lautstärke und Audio-Einstellungen

Hier wird die Lautstärke für die Hintergrundmusik und andere Sounds angepasst, um das Spielerlebnis zu optimieren.

2.6.5 Audio-Verzögerung und -Wiederholung

```
playSoundWithDelay() {
    this.time.delayedCall(1000, () => {
        this.sound.play('hitSound'); // Treffer-Sound mit 1 Sekunde Verz gerung ab
    });
}
```

Listing 25: Verzögerung bei der Wiedergabe eines Sounds

Dieser Code zeigt, wie man einen Sound mit einer Verzögerung abspielt, um zusätzliche Effekte oder zeitliche Synchronisationen zu ermöglichen.

2.6.6 Audio-Effekte bei Kollisionen

```
this.physics.add.collider(player, enemys, (player, enemy) => {
    this.sound.play('hitSound'); // Treffer-Sound bei Kollision abspielen
    enemy.destroy(); // Feind zerst ren
});
```

Listing 26: Audio-Effekte bei Kollisionen

Hier wird der Treffer-Sound bei der Kollision zwischen dem Spieler und einem Feind abgespielt.

3 Projektergebnisse

3.1 Funktionalität und Spielelemente

Das fertige Spiel umfasst verschiedene funktionale Elemente und Spielelemente:

- **Hintergrund und Umgebungsobjekte**: Der Hintergrund wird als Tile-Sprite hinzugefügt und die Umgebung umfasst Plattformen und Türen, die durch die Methoden `addPlatforms` und `addDoors` erstellt werden.
- **Spielcharaktere**: Der Spieler und Feinde werden als Sprites geladen. Der Spieler kann durch die Welt navigieren, springen, schlagen und werfen, während die Feinde die Spielerposition verfolgen und angreifen.
- **Animationen**: Verschiedene Animationen wurden für den Spieler und die Feinde definiert, darunter Lauf-, Sprung-, Schlag- und Wurfanimationen.
- **Audio-Feedback**: Es gibt Hintergrundmusik und Soundeffekte für Aktionen wie Schläge, Würfe und Feindangriffe.
- **Spielmechanik**: Der Spieler sammelt Punkte, indem er Feinde besiegt und kann die Spielrunde durch Erreichen eines Fahrstuhls beenden.

3.2 Herausforderungen und Lösungen

Während der Entwicklung des Spiels traten mehrere Herausforderungen auf, die gelöst werden mussten:

3.2.1 Herausforderung 1: Hintergrund

```
1 this.bg = this.add.tileSprite(0, 0, 12000, 900, 'bg').setOrigin(0);
```

Listing 27: Code zur Problembehebung beim Hintergrund

Ein Problem trat auf, als der Hintergrund nicht korrekt angezeigt wurde. Dieses Problem wurde gelöst, indem der Hintergrund als Tile-Sprite hinzugefügt wurde, was sicherstellt, dass der Hintergrund kontinuierlich über die gesamte Bildschirmbreite dargestellt wird.

3.2.2 Herausforderung 2: Feinde spawnen

```
addEnemies() {
        for (let i = 0; i < 20; i++) { // Add 20 enemy loads
            for (let e = 0; e < i; e++){
                let enemy = enemys.create((1000 + (i * (400*(i/3)))+(e*40)), 600, '
                enemy.health = 1; // Set enemy health
                enemy.flipX = true; // Flip enemy sprite horizontally

                enemy.isAttacking = false; // Initialize attacking state

                enemy.play('enemyIdle'); // Play idle animation
                this.physics.add.collider(enemy, player, this.hitByEnemy); // Colli
                enemy.setGravityY(6000); // Set gravity for enemy
            }

        }
    }
```

Listing 28: Code zum Hinzufügen von Feinden

Die Formel für die Berechnung der x-Position des Feindes lautet:

$$x = 1000 + (i \cdot 400)$$

Ein Problem beim Feind-Spawn bestand darin, dass die Feinde nicht korrekt positioniert wurden. Die Lösung bestand darin, die Position der Feinde dynamisch zu berechnen und sicherzustellen, dass sie innerhalb der Grenzen des Spielbereichs erzeugt werden. Es werden pro Spawn von Feinden ein weiterer Feind hinzu geladen.

3.2.3 Herausforderung 3: Feinde verfolgen durch die Luft

```
this.physics.moveToObject(enemy, player, 300);
```

Listing 29: Problematische Funktion: Feinde verfolgen durch die Luft

Ein weiteres Problem war, dass die Feinde durch die Luft verfolgten, da `moveToObject` den kürzesten Weg berechnet und dabei die Physik ignoriert. Dies führte dazu, dass Feinde durch Objekte hindurch bewegten. Eine mögliche Lösung ist die Implementierung einer benutzerdefinierten Verfolgungslogik, die Hindernisse berücksichtigt und alternative Bewegungsstrategien verwendet, um die Feinde auf dem Boden zu halten. Dies würde allerdings noch nicht getätigt.

```
updateEnemies() {
    enemys.children.iterate((enemy) => {
        if (enemy) {
            // Logik zur Vermeidung von Hindernissen
            const distance = Phaser.Math.Distance.Between(player.x, player.y, enemy
            if (distance < 800) {
                // Bewege Feinde auf den Spieler zu, aber verhindere das Durchdring
                if (distance < 50) {
                    enemy.setVelocityX(0);
                    enemy.anims.play('enemyAttack', true);
                } else {
                    enemy.anims.play('enemyRun', true);
                    this.physics.moveToObject(enemy, player, 300);
                }
            } else {
                enemy.anims.play('enemyIdle', true);
                enemy.setVelocityX(0);
                enemy.setVelocityY(0);
            }
        }
    });
}
```

Listing 30: Verbesserte Feind-Bewegung

3.3 Assets Quellen

- Menü Image: Reddit
- Background Sound: Youtube
- Player Assets Craftpix.net
- Zombie Assets Craftpix.net
- Throw a Knife Sound pixabay.com
- Knife Imagevecteezy.com
- Door Image: Reddit
- Punch Sound pixabay.com
- Zombie Die/Bite Sound pixabay.com

4 Fazit

4.1 Erkenntnisse aus dem Projekt

Bei der Durchführung des Projekts haben sich mehrere wichtige Erkenntnisse herauskristallisiert. Besonders auffällig war, dass die in der Konzeptentwicklung vorgestellte Idee in dem vorgesehenen Umfang und Detailgrad leider zu umfangreich für eine Einzelperson war. Dies lag vor allem an der begrenzten Zeit des kurzen Sommersemesters, welches nicht ausreichte, um alle geplanten Funktionen und Features in der Tiefe zu realisieren. Die Komplexität und der Umfang des Projekts erwiesen sich als zu herausfordernd, um sie innerhalb des vorgegebenen Zeitrahmens vollständig umzusetzen.

Diese Erkenntnis verdeutlicht die Bedeutung einer realistischen Planung und das Management von Projektumfang und Zeitressourcen. In zukünftigen Projekten sollten die Planung und die Zielsetzung entsprechend angepasst werden, um den realistischen Rahmen für die Umsetzung zu gewährleisten.

4.2 Mögliche Weiterentwicklungen

Aufgrund der Erkenntnisse aus dem Projekt ergeben sich mehrere mögliche Weiterentwicklungen, die dazu beitragen können, die ursprünglichen Konzeptideen in einer praktikableren Form umzusetzen:

- **Priorisierung der Funktionen:** Eine detaillierte Priorisierung der Funktionen und Features kann helfen, die wichtigsten und realisierbarsten Aspekte des Projekts frühzeitig umzusetzen. Dadurch wird sichergestellt, dass die Kernfunktionen des Spiels auch innerhalb eines begrenzten Zeitrahmens erfolgreich realisiert werden können.
- **Erweiterung des Teams:** Um den Umfang und die Komplexität des Projekts besser bewältigen zu können, wäre es sinnvoll, zusätzliche Teammitglieder hinzuzufügen. Dies würde es ermöglichen, Aufgaben effizienter zu verteilen und spezialisierte Kompetenzen einzubringen, die zur Verbesserung des Projekts beitragen können.
- **Modularisierung der Projektentwicklung:** Eine modulare Herangehensweise an die Entwicklung könnte helfen, das Projekt in überschaubare Teile zu gliedern. Jedes Modul könnte unabhängig entwickelt und getestet werden, was die Komplexität verringert und die Fehleranfälligkeit reduziert.
- **Realistischere Zeitplanung:** Für zukünftige Projekte sollte eine realistischere Zeitplanung erstellt werden, die den tatsächlichen Aufwand für die Umsetzung der verschiedenen Funktionen berücksichtigt. Dies könnte durch detaillierte Zeitabschätzungen und Pufferzeiten für unvorhergesehene Herausforderungen unterstützt werden.
- **Iterative Entwicklung:** Ein iterativer Entwicklungsansatz, bei dem regelmäßig Feedback eingeholt und Anpassungen vorgenommen werden, kann dazu beitragen, die Entwicklung effektiver zu steuern und notwendige Anpassungen frühzeitig zu identifizieren. Dies kann auch helfen, das Projekt besser an die verfügbaren Ressourcen und Zeitrahmen anzupassen.

Durch die Umsetzung dieser Weiterentwicklungen könnte das Projekt in zukünftigen Phasen erfolgreicher und effektiver realisiert werden. Die Erkenntnisse aus der aktuellen Erfahrung bieten wertvolle Lektionen, die als Grundlage für die Verbesserung der Projektplanung und -durchführung dienen können.