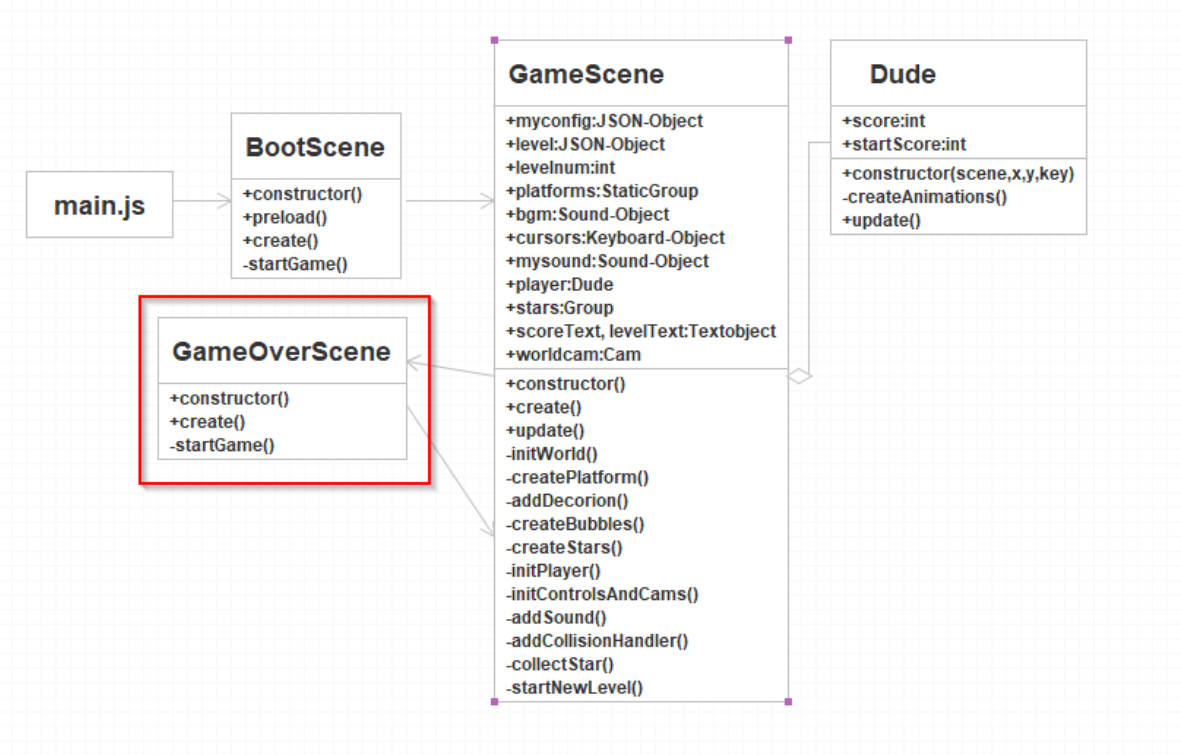
Tutorial 04 - Phaser-Einstieg

1. GameOverScene



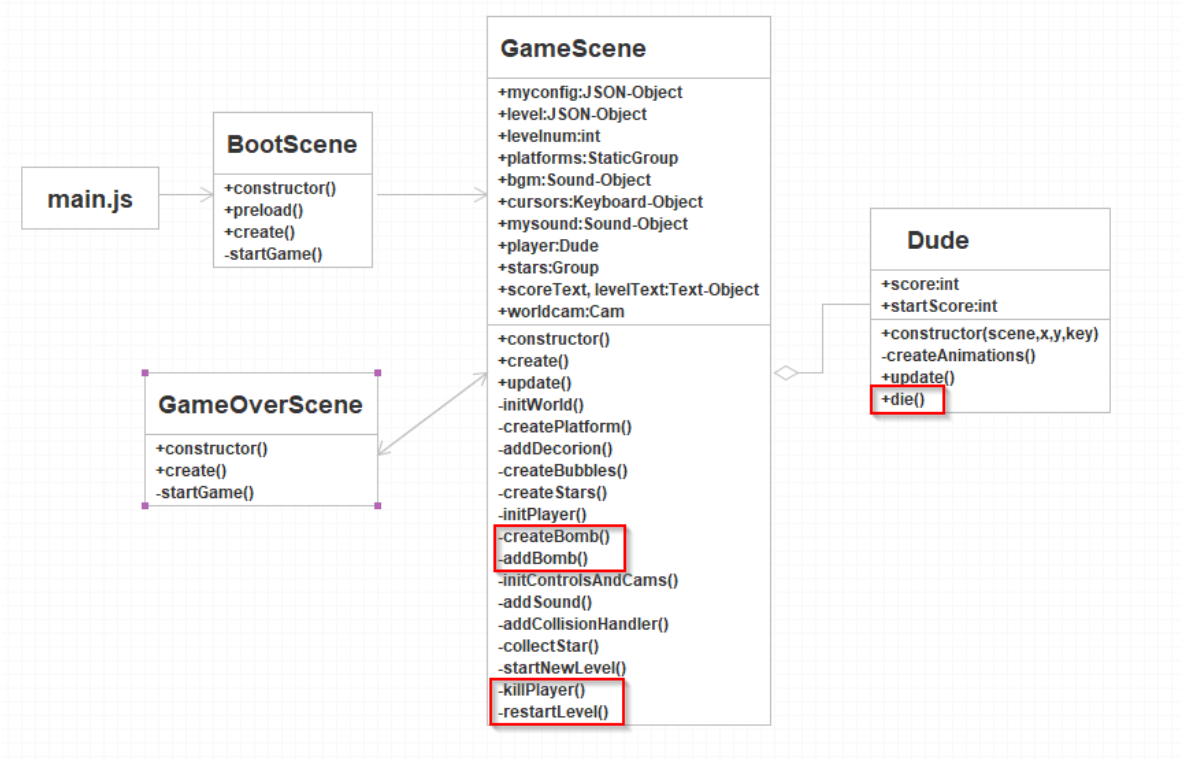
Für das Ende des Spiels - wenn man das letzte Level erfolgreich beendet hat - wird eine neue Szene eingefügt. Dafür wurde der Code um eine neue Klasse `GameOverScene` (`scenes/GameOverScene.js`) ergänzt, welche sehr ähnlich aufgebaut ist, wie die `BootScene`. Allerdings müssen keine Assets mehr geladen werden, sondern nur die Texte und der Restart-Button angezeigt werden.

In der `GameScene` wird die Methode `startNewLevel` um eine Abfrage ergänzt: Wenn die neue Level-Nummer größer als die Anzahl der Levels ist, dann wird die `GameOverScene` gestartet (hier im else-Zweig):

```
startNewLevel(level,score) {
    this.cameras.main.fade(1000,0,0,0,false);
    this.cameras.main.on("camerafadeoutcomplete",function() {
        var newLevel = level + 1;
        if (newLevel <= this.myconfig.levels) {
            this.scene.restart({level:newLevel,score:score});
        } else {
            this.scene.start('GameOverScene');
        }
    },this);
}
```

2. Bombe

Nun brauchen wir noch Spielelemente, die den Player daran hindern sein Ziel zu erreichen. Wir starten einmal mit einer Bombe.



2.1. Bombe einfügen

Die Grafik wird in der `BootScene` geladen:

```
this.load.image('bomb', this.imgpath+'bomb.png');
```

In der `GameScene` wird im `create` eine neue Methode `createBomb()` aufgerufen, in der wieder eine Gruppe erzeugt wird:

```
createBomb() {
    this.bombs = this.physics.add.group();
    this.addBomb();
}
```

Die Methode `addBomb()`, die darin aufgerufen wird ermittelt einen x-Wert, der sich ungefähr an den Koordinaten des Players orientiert. Dazu wird wieder der Ternär-Operator - das abgekürzte if - eingesetzt: Ist der Spieler im linken Bereich, dann wird die Bombe im rechten Bereich geworfen und umgekehrt.

Mit `setBounce(1)` wird bewirkt, dass die Bombe ewig "herumspringt", ohne an Geschwindigkeit zu verlieren. `setVelocity` setzt den Startvektor für die Geschwindigkeit.

```
addBomb() {
    var x = (this.player.x < 400) ?
        Phaser.Math.Between(400, 800) : Phaser.Math.Between(0, 400);
    var bomb = this.bombs.create(x, 16, 'bomb');
    bomb.setBounce(1);
    bomb.setCollideWorldBounds(true);
    bomb.setVelocity(Phaser.Math.Between(-200, 200), 20);
}
```

2.2. Player erweitern - die()

Auch der Player bekommt eine zusätzliche Funktionalität. Die Methode `die()` soll bei Aufruf die Grafik rot einfärben und die Animation auf "stand" festlegen.

```
die() {
    this.setTint(0xff0000);
    this.anims.play('stand');
}
```

2.3. Collision-Handling

Die Methode `addCollisionHandler()` wird um die neuen Collider-Aufrufe ergänzt. Zunächst einmal muss die Bombe mit den Plattformen interagieren:

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

Wichtig ist vor allem, dass bei "Überlappung" zwischen Player und Bombe etwas passiert. Da wird nämlich die Callback-Methode `killPlayer()` aufgerufen.

```
this.physics.add.overlap(this.player, this.bombs, this.killPlayer, null,
this);
```

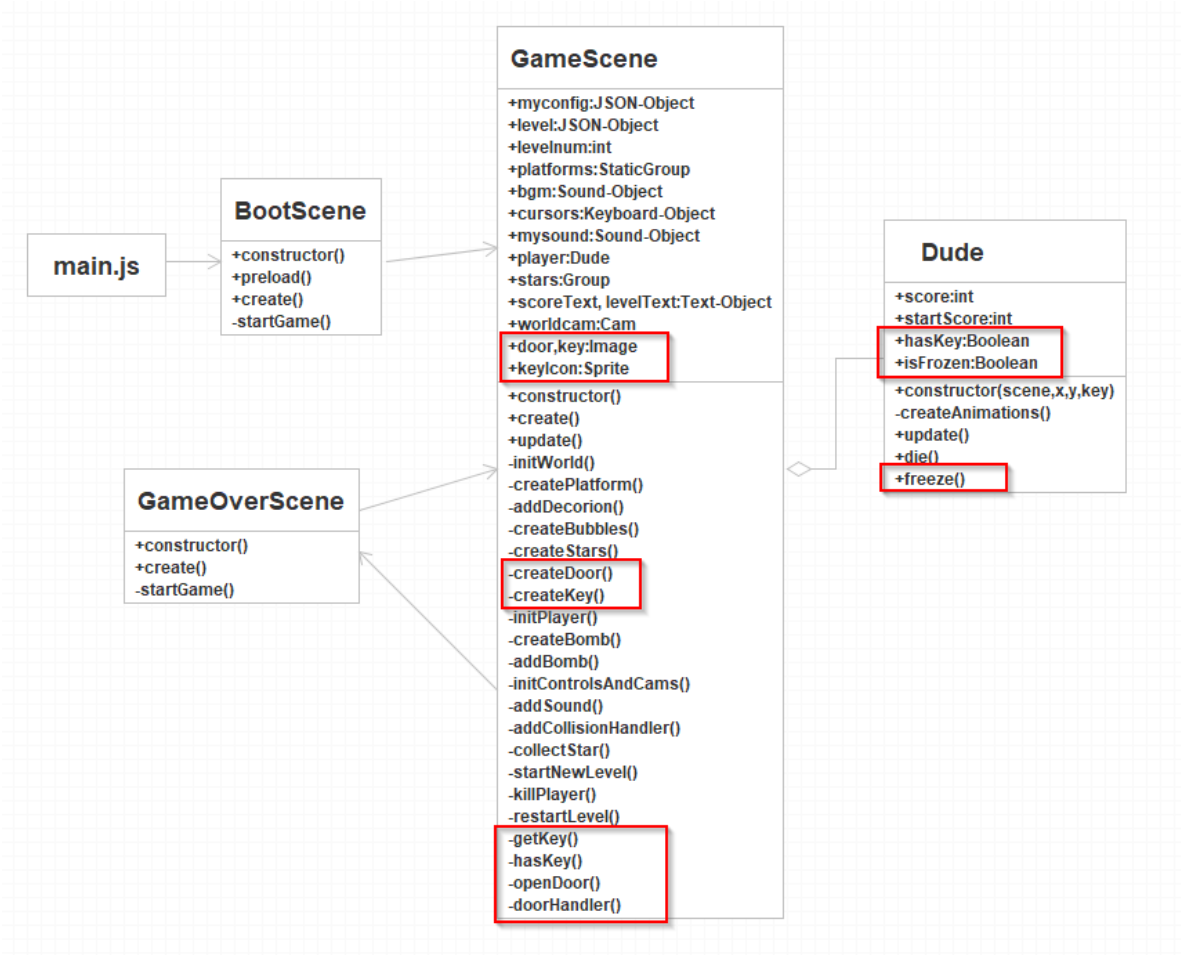
Diese Methode ruft zunächst einmal die `die()-Methode` des Players auf (Rot einfärben und stehen bleiben) und danach die Methode `restartLevel()`.

```
killPlayer() {
    this.player.die();
    this.restartLevel(this.levelnum,this.player.startscore);
}
```

Die Methode `restartLevel()` erhält als `Parameter` die aktuelle `Levelnummer` und den `Score` übergeben (nämlich den Startspielstand dieses Levels, welcher auch direkt im Player-Objekt gespeichert wurde). Dann wird wieder einmal ein Kameraeffekt eingesetzt. Mit `shake` und der Dauer in Millisekunden wird die Kamera so richtig durchgeschüttelt - wie es bei einer Bombenexplosion so üblich ist. Über einen `Eventlistener (camerashakecomplete)` wird nach Beendigung dieser Animation die Szene mit `restart` wieder neu gestartet.

```
restartLevel(level, score) {
    this.cameras.main.shake(1000);
    this.cameras.main.on("camerashakecomplete",function() {
        this.scene.restart({level:level,score:score});
    },this);
}
```

3. Tür zum nächsten Level



3.1. Schlüssel als Türöffner - Tweens

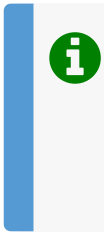
In der `BootScene` in der Methode `preload()` werden wieder die entsprechenden Assets geladen: `key`, `keyIcon` (für das Scoreboard) und der `Sound` für das nehmen des Schlüssels - dieser wird in der `GameScene` zum `mysound-Objekt` dazugefügt.

```
this.load.image('key', this.imgpath+'key.png');
```

```
this.load.spritesheet('keyicon', this.spritepath+'key_icon.png',
                      { frameWidth: 34, frameHeight: 30});
```

```
this.load.audio('getkey', this.audiopath+'key.wav');
```

In der `GameScene` wird im `create()` die Methode `createKey()` aufgerufen. In der wird das Bild des Schlüssels hinzugefügt und, damit der Schlüssel nicht fällt die Eigenschaft `allowGravity` auf `false` gesetzt. Außerdem wird eine sogenannte Tween-Animation erstellt. Diese bewirkt, dass sich der Schlüssel (Auswahl welches Element mit `targets`) in `y-Richtung` bewegt und zwar um `6 Pixel`. Dies dauert `800 ms` und mit `ease` wird die Art der Bewegung festgelegt (d.h. wie genau verläuft die Bewegung - hier: easeInOut - Am Anfang beschleunigen, zum Schluss abbremsen - siehe z.B. unter https://easings.net/de). Diese Bewegung wird auch wieder in die andere Richtung gemacht (`yoyo auf true`) und unendlich oft wiederholt (`loop auf -1`).



Tweens werden für sich wiederholende Animationen/Transformationen eingesetzt. In Phaser gibt es dazu eine eigene Tween-Klasse. Mit scene.tweens.add (bzw. this.tweens.add) kann man einen entsprechenden Tweeneffekt zu einem Element hinzufügen.

```
createKey() {
    this.key = this.physics.add.image(this.level.key.x, this.level.key.y,
"key");
    this.key.body.allowGravity = false;
    this.tweens.add({
        targets: this.key,
        y: this.key.y+6,
        duration: 800,
        ease: 'Sine.easeInOut',
        yoyo: true,
        loop: -1
    });
}
```

In der Methode `addScoreBoard()` wird das `keyIcon` platziert, welches mit 2 unterschiedlichen Frames anzeigen soll, ob man den Schlüssel schon hat. Auch hier wird der `ScrollFactor auf 0,0` gestellt, sodass sich das Bild mit der Kamera mit bewegt und somit statisch erscheint.

```
this.keyIcon = this.add.image(300,570,'keyicon',0);
this.keyIcon.setScrollFactor(0,0);
```

Damit sich der Player auch merkt, ob er den Schlüssel schon hat oder nicht wird in der Klasse `Dude` im Konstruktor eine Eigenschaft `hasKey` definiert und auf `false` gesetzt.

```
this.hasKey = false;
```

3.2. Tür hinzufügen

Auch für die Tür wird zuerst in der `BootScene` das entsprechende `SpriteSheet` geladen.

```
this.load.spritesheet('door',this.spritepath+'door.png',
                      { frameWidth: 42, frameHeight: 66});
```

Auch der `Sound` für das Öffnen der Tür wird geladen und in der `GameScene` zum `mysound-Objekt` hinzugefügt

```
this.load.audio('opendoor', this.audiopath+'door.wav');
```

Im `create()` der `GameScene` wird die Methode `createDoor()` aufgerufen, welche die Grafik der Tür an der richtigen Stelle platziert (wie im Level, d.h in der JSON-Datei festgelegt).

```
createDoor() {
    this.door = this.physics.add.image(this.level.door.x,
                                       this.level.door.y,
"door").setOrigin(0.5,1);
    this.door.body.allowGravity = false;
}

```

3.3. Spieler erweitern - Freeze

Damit der Spieler beim Durchschreiten der Türe (Eine Animation - siehe Kapitel Collision-Handling) nicht mehr mit den Pfeil-Tasten gesteuert werden kann muss er kurz "eingefroren" werden. Dazu bekommt die `Dude-Klasse` im Konstruktor die Eigenschaft `isFrozen (Startwert false)` zugewiesen.

```
this.isFrozen = false;
```

Im `update()` werden alle Aktionen des Spielers nur ausgeführt, wenn er nicht gefroren ist - `!this.isFrozen`.

```
if (!this.isFrozen) {
    ...
}
```

Die Funktion `freeze()` deaktiviert die physikalischen Eigenschaften des Objekts und setzt die Eigenschaft `isFrozen` auf `true`.

```
freeze() {
    this.disableBody(true,false);
    this.isFrozen = true;
}
```

3.4. Collision-Handling

Schlussendlich müssen wir noch die einzelnen Collision-Handler festlegen und implementieren.

3.4.1. Key

Key und Player sollen bei `Überlappung` auslösen und die Methode `getKey()` aufgerufen werden.

```
this.physics.add.overlap(this.player, this.key, this.getKey, null, this);
```

In dieser Methode wird die Schlüsselgrafik vom Bildschirm genommen und der Player bekommt die Eigenschaft `hasKey` auf `true` gesetzt. Der Sound wird abgespielt und im `Scoreboard` wird die `keyIcon-Grafik` auf das `Frame 2 (Index 1)` gesetzt - ein voller Schlüssel.

```
getKey(player, key) {
    this.key.destroy();
    this.player.hasKey = true;
    this.mysound.getkey.play();
    this.keyIcon.setFrame(1);
}
```

3.4.2. Door - Process-Callback-Funktion

Auch für den Player und die Türe brauchen wir einen `Overlap-Collider`. Allerdings darf dieser erst wirklich reagieren, wenn der Player auch wirklich den Schlüssel schon hat. Um Bedingungen für Collider-Ereignisse abfragen zu können gibt es - neben der `Collide-Callback-Funktion` `openDoor()` - eine sogenannte `Process-Callback-Funktion`. Diese heißt bei uns `hasKey()`.

```
this.physics.add.overlap(this.player, this.door, this.openDoor, this.hasKey,
this);
```

Diese Funktion hat die Aufgabe `false` (darf nicht ausgeführt werden - hat den Schlüssel noch nicht) oder `true` (darf ausgeführt werden - hat den Schlüssel) zurück zu liefern. Dies wird hier damit realisiert, dass die Boolean-Werte `player.hasKey` (true oder false) mit `player.body.touching.down` (berührt den Boden ja oder nein - In der Luft, wenn er gerade springt soll er die Türe ja nicht öffnen können) verknüpft werden und das Ergebnis zurück geliefert.

```
hasKey(player, door) {
    return (player.hasKey && player.body.touching.down);
}
```

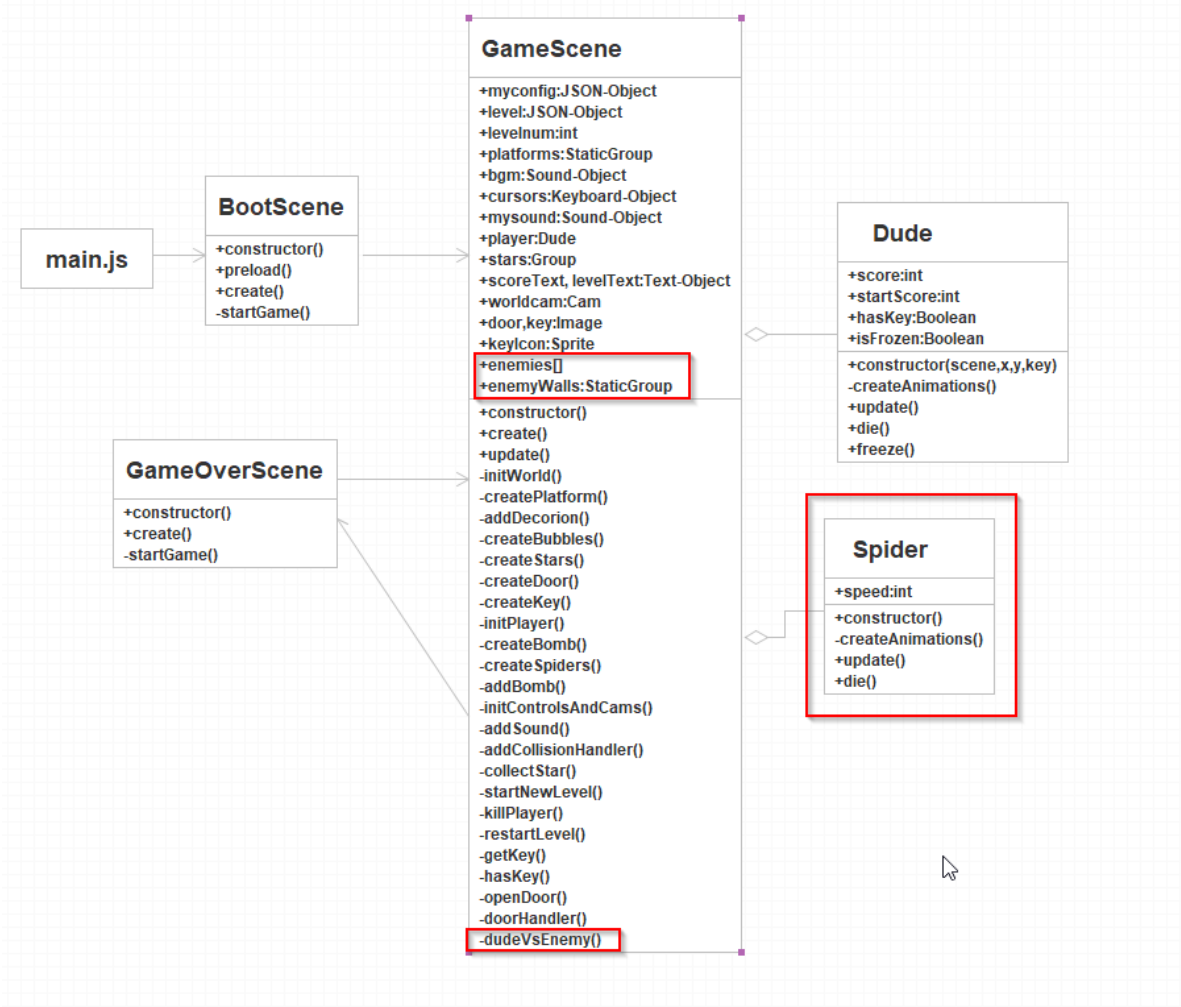
Erst wenn `hasKey true` zurück liefert wird `openDoor()` aufgerufen. Zuerst wird einmal die physikalische Eigenschaft der Türe deaktiviert (`disableBody(true,false)`), damit `openDoor` nur einmal aufgerufen wird. Die Grafik der Tür wird auf Frame 2 (Index 1) gesetzt. Beim Player wird `freeze` aktiviert, damit dieser solange er das Level wechselt nicht mit der Steuerung bewegt werden kann. Dann wird zum Spieler eine Tween-Animation hinzugefügt, in der sein Alpha-Wert sich von 100% auf 0 verändert und er sich der Türe zubewegt. D.h. es sieht so aus, als wenn er durch die Türe verschwinden würde. Erst wenn diese Animation beendet ist (`Event-Listener: onComplete`) wird die Methode `doorHandler()` aufgerufen, die das nächste Level startet.

```
openDoor(player, door) {
    door.disableBody(true,false);
    door.setFrame(1);
    this.mysound.opendoor.play();
    this.player.freeze();
    this.tweens.add({
        targets: this.player,
        duration: 500,
        alpha: 0,
        x: this.door.x,
        onComplete: this.doorHandler,
        onCompleteParams: [ this ]
    })
}
```

Methode `doorHandler()`:

```
doorHandler(tween, targets, scene) {
    scene.startNewLevel(scene.levelnum, scene.player.score);
}
```

4. Noch ein Feind - Spider



4.1. Spider-Klasse

4.1.1. Spider.js

Auch hier starten wir mit dem Laden der `Spritesheet-Grafik` in der `BootScene`, welche die Animationen für die Spinne enthält.

```
this.load.spritesheet('spider',this.spritepath+'spider.png',
                      { frameWidth: 42, frameHeight: 32});
```

Für die Spinne erstellen wir wieder eine eigene Klasse, da diese ja mehr können soll. Im Konstruktor sind da vor allem drei Dinge wichtig, damit das mit der Physik auch hinhaut: Man muss das Objekt mit `group.add(this)` zur Gruppe hinzufügen, die physikalischen Eigenschaften/Methoden aktivieren (`physics.world.enable(this)`) und außerdem das Objekt zur Szene hinzufügen (`scene.add.existing(this)`). Ansonsten ist die Vorgehensweise ähnlich wie bei anderen Sprites.

```
class Spider extends Phaser.Physics.Arcade.Sprite {
    constructor(scene, x, y, key,group) {
        super(scene, x, y, key);
        group.add(this);
        scene.physics.world.enable(this);
        scene.add.existing(this);
        this.setCollideWorldBounds(true);
        this.createAnimations(scene);
        this.anims.play('krabble');
        this.speed = 100;
        this.body.setVelocityX(this.speed);
    }
```

Neben der normalen Animation "krabble" ist kommt hier eine weitere Variante der Erstellung von Animationen in der Methode `createAnimations()` zum Einsatz. Die Animation `die()` führt zuerst 3 mal die Frames 0 und 4 hintereinander aus und danach 6 mal das Frame mit der ID 3.

```
scene.anims.create({
    key: 'die',
    frames: scene.anims.
        generateFrameNumbers('spider', {frames: [0,4,0,4,0,4,3,3,3,3,3,3]}),
    frameRate: 12
});
```

Beim `update()` der Spinne wird darauf geachtet, ob diese rechts oder links einen Körper berührt oder durch einen Körper blockiert wird (`blocked`). Dann wird die x-Geschwindigkeit des Bewegungsvektor umgedreht, d.h. die Spinne versucht in die andere Richtung zu gehen.

```
update() {
    if (this.body.touching.right || this.body.blocked.right) {
        this.setVelocityX(-this.speed);
    } else if (this.body.touching.left || this.body.blocked.left) {
        this.setVelocityX(this.speed);
    }
}
```

Die Methode `die()` deaktiviert die Spinne, führt die `"die-Animation"` aus - und wenn diese beendet ist nimmt sie das gesamte Objekt aus dem Spiel.

```
die() {
    this.body.enable = false;
    this.anims.play("die");
    this.on('animationcomplete', function() {
        this.destroy();
    }, this);
}
```

4.1.2. Level01.json / Level02.json

Um die Spinne in den Levels einzubauen kommt in den Leveldateien ein zusätzlicher Eintrag dazu, nämlich ein Eintrag mit dem Namen `"spiders"`, welches aus einem `Array` von `Positionsobjekten` besteht (jeweils `x` und `y`)

```
"door": {
    "x": 600,
    "y": 536
},
"spiders": [{"x": 121, "y": 399}, {"x": 800, "y": 362}, {"x": 500, "y": 147}]
```

4.1.3. BootScene.js

In der Bootscene laden wir im `preload` die Spritesheet-Grafik. Achtung: `frameWidth` und `frameHeight` müssen auch hier wieder passen, d.h die Grafik in Frames der richtigen Größe unterteilen!

```
this.load.spritesheet('spider',this.spritepath+'spider.png',
    { frameWidth: 42, frameHeight: 32});

```

4.1.4. GameScene.js

In der `GameScene` müssen wir folgende Ergänzungen machen, um die Spinne einzufügen:

Im `create` erstellen wir eine neue Gruppe, welche wir `enemies` nennen - hier könnten ja später auch noch andere Objekte als die Spinne dazukommen. Wenn das Grundverhalten (z.B. `Collision-Handler`) gleich ist, dann kann man unterschiedliche Objekte in eine Gruppe geben.

```
this.enemies = this.physics.add.group();
this.createSpiders();
```

Im `createSpiders()` gehen wir in einer for-Schleife alle Positionsobjekte durch, die im `level` unter dem Namen `"spiders"` vorhanden sind (fehlt dieser Eintrag, weil diese Art von Feinden im Level nicht vorkommt macht das auch nichts, dann bricht die Schleife sofort ab). Für jedes dieser Elemente erstellen wir ein neues Spider-Objekt und übergeben die `Koordinaten`, den `Schlüssel` (Name) und die Enemy-Gruppe (Das Objekt wird ja direkt im Konstruktor in die Gruppe dazugefügt)

```
createSpiders() {
    for (var data of this.level.spiders) {
        var newspider = new Spider(this,data.x,data.y,'spider',this.enemies);
    }
}
```

Auch die `update()-Methode` der `GameScene` müssen wir erweitern. Nun müssen bei jedem Bildschirmaufbau alle Enemy-Objekte durchlaufen werden - JavaScript bietet hier auch eine forEach-Funktion um Arrays von Objekten zu durchlaufen - und rufen die update()-Methode der Feind-Klasse auf.

```
update() {
    this.player.update(this.cursors, this.mysound);
    this.enemies.children.entries.forEach(function(enemy) {
        enemy.update();
    });
}
```

4.2. Unsichtbare Wände für den Feind

Oft benötigt man in Spielen unsichtbare Wände, damit sich Objekte nur innerhalb eines bestimmten Bereiches bewegen können. Bei uns wird bei jeder Plattform am Anfang und am Ende eine solche unsichtbare Wand platziert, damit sich die Spinnen, wenn sie auf einer Plattform landen, nur auf dieser hin- und herbewegen können.
Dazu braucht man in der `BootScene` natürlich wieder eine entsprechende Grafik:

```
1  this.load.image('enemy-wall', this.imgpath+'invisible_wall.png');
```

Die Methode `createPlatform()` in der `GameScene` wird um zwei Befehle erweitert. Links und rechts soll jeweils eine dieser sichtbaren Wände entstehen.

```
1  createPlatform(p) {
2      var newplatform = this.platforms
3          .create(p.x,p.y,
   'platform').setScale(p.scaleX,p.scaleY).refreshBody();
4      this.createEnemyWall(newplatform,'left');
5      this.createEnemyWall(newplatform,'right');
6  }
```

Die Funktion `createEnemyWall(platform,pos)` platziert entweder links oder rechts eine Wand. Dazu bekommt sie ein Platform-Objekt und die Position (links/rechts) als Parameter übergeben.

Aus dieser Information berechnet sie die jeweilige Position, erstellt dieses Objekt und - am Wichtigsten - stellt die Sichtbarkeit auf false - `wall.visible = false`.

```
1  createEnemyWall(platform, pos) {
2      var posx, originx;
3      if (pos=="right") {
4          posx = platform.x + platform.displayWidth/2;
5          originx = 0;
6      } else {
7          posx = platform.x - platform.displayWidth/2;
8          originx = 1;
9      }
10     var wall = this.enemyWalls
11         .create(posx, platform.y, 'enemy-
   wall').setOrigin(originx,1).refreshBody();
12     wall.visible = false;
13 }
```

4.3. Player erweitern - bounce()

Wenn der Player auf eine Spinne hüpft soll diese sterben, ein entsprechender Sound eingespielt werden und das Player zurückprallen (bounce). Das Audio-File wird wieder in der `BootScene` geladen (und in der `GameScene` zum `mysound-Objekt` hinzugefügt).

```
this.load.audio('stomp', this.audiopath+'stomp.wav');
```

In der Dude-Klasse wird die Methode bounce() eingefügt, welche dem Spieler einen "Schubs" (Impuls) nach oben gibt - er prallt ab.

```
bounce() {
    const BOUNCE_SPEED = 200;
    this.setVelocityY(-BOUNCE_SPEED);
}
```

4.4. Collision-Handling

Auch die Methode `addCollisionHandler()` wird wieder entsprechend erweitert. Zuerst müssen die Spinnen mit den Plattformen und den EnemyWalls interagieren:

```
this.physics.add.collider(this.spiders, this.platforms);
this.physics.add.collider(this.spiders, this.enemyWalls);
```

Für die Interaktion mit dem Spieler wird wieder die overlap-Methode eingesetzt, die im gegebenen Fall die Methode `dudeVsSpider()` startet.

```
this.physics.add.overlap(this.player, this.spiders, this.dudeVsEnemy, null,
this);
```

In der Methode `dudeVsEnemy()` müssen wir zwischen zwei Fällen unterscheiden:

1. Player kommt von oben: Spinne stirbt und Player prallt ab
2. Ansonsten (Player kommt von der Seite): Player stirbt

Die zugrunde liegende Abfrage testet einfach die y-Geschwindigkeit des Spielers. Ist diese > 0 bedeutet das, dass der Spieler gerade von oben herunterfällt.

```
dudeVsEnemy(player, enemy) {
    if (player.body.velocity.y > 0) {
        //Player von oben: Spinne ist tot
        enemy.die();
        player.bounce();
    } else {
        //Sonst ist leider der Player tot
        this.killPlayer();
    }
}
```