Tutorial 03 - Phaser-Einstieg

In der Fortsetzung unserer Tutorial-Reihe zum Einstieg in Phaser werden wir grundlegendes Ändern - nämlich die Programmstruktur mit Objektorientierung in eine "elegantere" Struktur bringen und wieder einiges an Funktionalität dazufügen.

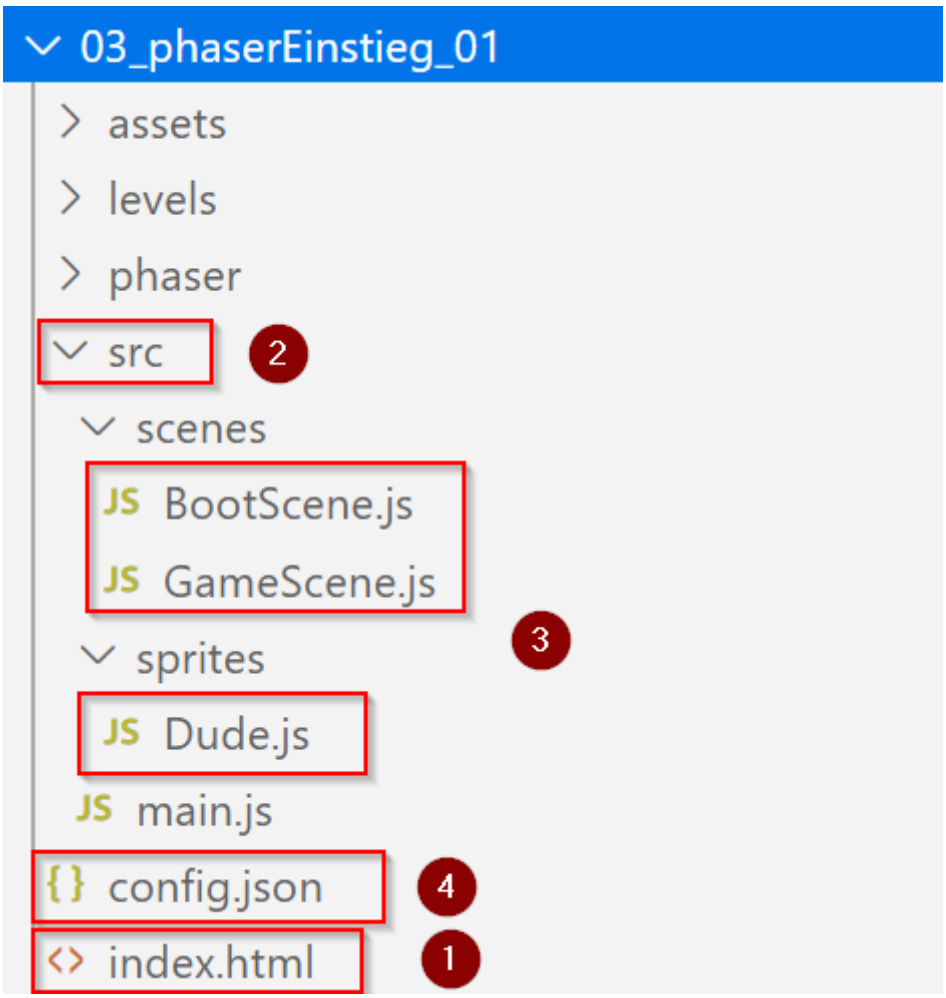
Das Tutorial ist in 3 Kapitel mit Änderungen/Erweiterungen gegliedert. Zu jedem Kapitel gibt es den fertigen Projektordner (mit vollständigem Code), d.h. man muss nichts mehr einfügen/erweitern usw. Natürlich könnt ihr Dinge ausprobieren, ändern, ...

1. OOP - Einsatz von Klassen / Szenen in Phaser

> Projektordner: 03_phaserEinstieg_01

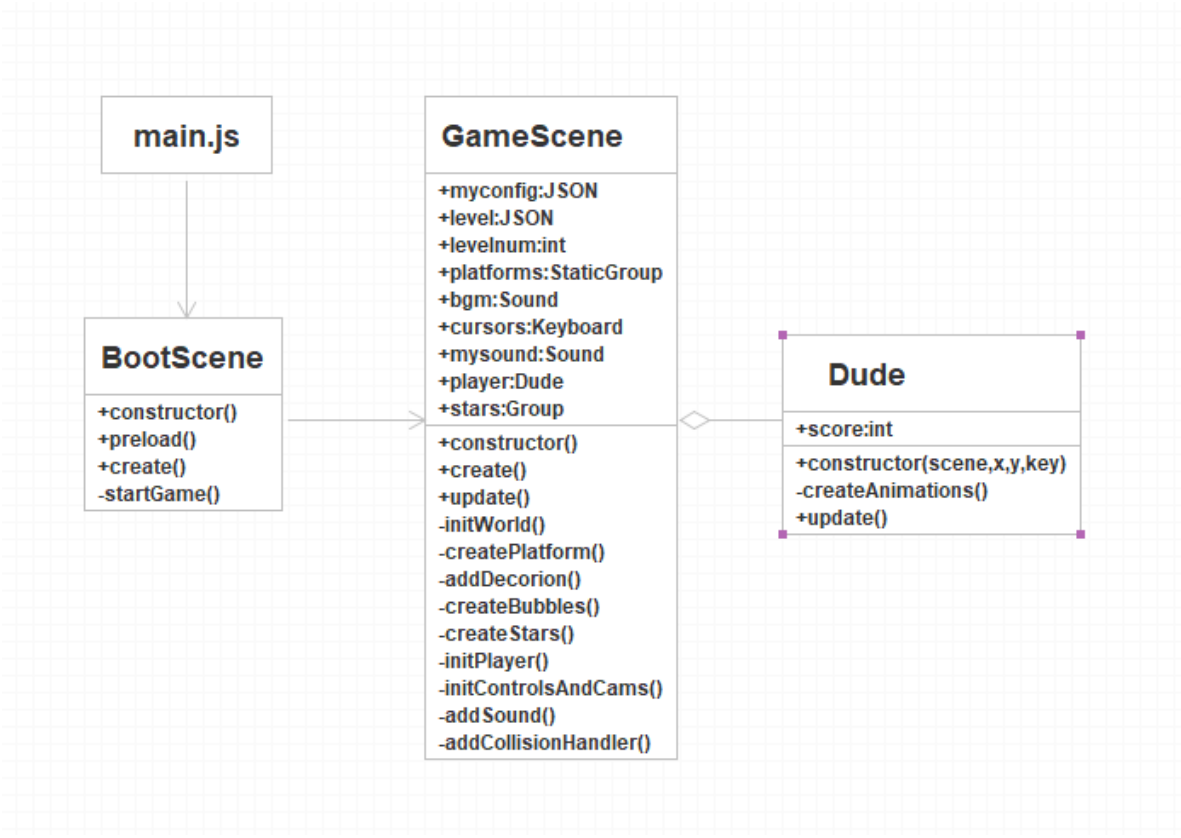
1.1. Grundlegende Struktur

Zuerst einmal ist in dieser Version die Programmstruktur grundlegend anders. Der Code wurde in mehrere Dateien aufgeteilt sodass das Spiel den heutigen Ansätzen der Objektorientierung entspricht. Auch einige Namen (Ordner/Dateien) wurden umbenannt und es sind zusätzliche Assets dazugekommen. Deswegen auch im Assets-Ordner eine Struktur mit Unterordnern (audio/fonts/images/sprites). Dadurch hat man einen leichteren Überblick über die eigenen Dateien.



Nr.	Beschreibung
1	game.html wurde zu index.html umbenannt. Grund: Dadurch wird die Datei vom Browser direkt geladen ohne den Namen angeben zu müssen. Eine kleine Änderung im Inhalt: main.js wird jetzt mit dem Zusatz `type="module"` geladen. Nur dadurch kann man innerhalb von main.js andere Code-Dateien importieren. `<script src="src/main.js" type="module"></script>`
2	Da es in vielen Projekten üblich ist den Source-Code in einem Ordner `src` zu haben, wurde von mir dieser Ordner jetzt auch umbenannt. Dadurch man sich an diesen Standard (vorheriger Name war ja `js`)
3	`main.js` enthält jetzt nur mehr die Konfiguration und den Start des Spiels (Initialisierung des Game-Objekts). Der andere Code wurde - strukturiert - in entsprechende Dateien aufgeteilt. Dies wird aber unten noch genauer erklärt.
4	Zusätzlich wird jetzt eine Konfigurationsdatei für das Spiel geladen, in der man Einstellungen machen kann (siehe Kapitel JSON im vorherigen Tutorial)

Die Zusammenhänge der neue Struktur wird im folgenden Klassendiagramm grafisch dargestellt:



1.2. Szenen in Phaser 3

Wir haben ja bis jetzt immer mit einer Hauptszene gearbeitet. Diese haben wir genutzt um z.B. Grafiken oder Audio-Dateien zu laden und hinzuzufügen. Szenen sind also in Phaser das Herz der Funktionalität.

In der neuen Version setzen wir zwei Szenen ein:

Szene	Beschreibung
BootScene	In dieser Szene werden die Assets geladen (`preload()`). Wenn das Laden fertig ist wird in der Funktion `create()` die nächste Szene geladen/gestartet (In der Funktion `startGame()`)
GameScene	Das ist jetzt die Hauptszene, die unsere Welt aufbaut und das Spiel steuert Sie enthält die wichtigen Funktionen `create()` und `update()`

Woher weiß Phaser, dass es diese Szenen gibt und in welcher Reihenfolge sie gestartet werden sollen?

Die folgende Grafik zeigt dies:
Die beiden Code-Dateien werden zuerst einmal mit einem import-Befehl geladen. In reinem JavaScript ist es dazu notwendig den genauen relativen Pfad von der Hauptdatei aus gesehen beginnend mit einem `./` anzugeben.

Als nächstes werden im `config-Objekt` in der Eigenschaft `scene` die Namen der Szenen angegeben (die importierten Klassennamen), die in diesem Spiel verwendet werden.

```
import BootScene from './scenes/BootScene.js';
import GameScene from './scenes/GameScene.js';
```

```
var config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 300 },
            debug: true
        }
    },
    scene: [
        BootScene,
        GameScene
    ]
};

//Globale Variablen!
var game = new Phaser.Game(config);
```

Grundsätzlich wird die erste angegebene Szene von Phaser als erstes geladen (hier die `BootScene`). Danach kann man Szenen per Befehl laden. In unserer `BootScene` passiert das nach dem `preload` (Dateien wurden geladen) im `create`. Dort wird die `GameScene` gestartet. Prinzipiell können in Phaser aber Szenen auch automatisch gestartet werden und parallel ablaufen. Dadurch kann man Inhalte auf unterschiedliche Szenen aufteilen und manche Szenen immer wieder einspielen.

In der `BootScene` sieht der Aufruf der `GameScene` dann folgendermaßen aus:

```
1       startGame() {
2           this.scene.start("GameScene",{level:1,score:0});
3       }
```

Neben des Namen der aufrufenden Szene kann man auch ein Datenobjekt übergeben. Wir machen das hier gleich mit dem Startlevel (1) und dem Startscore (0) - das brauchen wir in den folgenden Erweiterungen.

1.3. Klassenkonzept in JavaScript

JavaScript ist - was Objektorientierung anbelangt - ein "Sonderfall", denn es ist grundsätzlich `Prototypen-Basiert`, d.h. es werden/wurden keine Klassendefinitionen wie in anderen Programmiersprachen eingesetzt, sondern sofort Objektinstanzen angelegt und mit Inhalt belebt (Eigenschaften, Methoden). Seit der Version `ES6 (EcmaScript 6)` gibt es aber auch die Möglichkeit `Klassen mit Methoden und Eigenschaften` explizit zu definieren. Dies setzen wir in unserem Spiel ein. *(Anmerkung: Ein drittes Konzept um mit Klassen zu arbeiten bietet Phaser auch noch selber an. Damit ihr die Unterschiede kennt (die sich durchaus auch in den Beispielen im Web wiederfinden), werden wir in einem späteren Tutorial noch auf diese eingehen (Prototypen-basiert vs. ES6-Klassen vs. Phaser-Klassen).)*

Schauen wir uns dazu einmal die Datei BootScene.js an:

```
class BootScene extends Phaser.Scene {
    constructor(test) {
        super({
            key: 'BootScene'
        });
        this.audiopath = "assets/audio/";
        this.imgpath = "assets/images/";
        this.spritepath = "assets/sprites/";
    }
```

Mit dem Schlüsselwort `class` und dem Namen wird diese Klasse definiert. Außerdem erbt sie die Grundfunktionalitäten von der Phaser-Klasse `Phaser.Scene` (Schlüsselwort `extends`).

Ein wesentlicher Bestandteil von Klassen ist ja die Konstruktor-Methode, welche in JavaScript den Namen `constructor` hat. In Phaser kann/sollte man beim Konstruktor als Parameter auch eine config-Variable angeben (hier test genannt).

Ein Aufruf des Konstruktors der Vererbenden Klasse (Phaser.Scene) ist in vielen Fällen wichtig und auch notwendig. Dies passiert in JavaScript mit dem Schlüsselwort `super`. In einer Phaser-Szene sollte man zumindest den `key`, d.h. den Namen der Scene-Klasse übergeben. Danach kann man Eigenschaften (Variablen) definieren bzw. belegen (initialisieren) oder Startfunktionen einsetzen. Eigenschaften der Klasse bzw. Objekte werden in JavaScript immer mit `this.` definiert bzw. eingesetzt.

Wir haben hier in der BootScene die Pfade zu den einzelnen Asset-Typen festgelegt. Diese Variablen setzen wir dann im Preload immer ein. Vorteil: Pfade und Ordnernamen können leicht geändert werden.

Das Schlüsselwort `function` fällt innerhalb von Klassendefinitionen bei den Funktionen weg. Auch demrAufruf von Funktionen der Klasse muss jetzt mit einem vorangestellten `this.` erfolgen: z.B. `this.startGame()`

Ein wichtiger Befehl in jeder unserer Klassen ist am Schluss (nach der Klassendefinition) der `Export-Befehl`. Dadurch kann die Klasse bzw. die Datei erst mit `import` in einer anderen Datei importiert werden (bei uns im `main.js`).

```
export default BootScene;
```

1.4. Config-Datei - Spiele-Einstellungen

```
{
    "levels": 2,
    "stars": 28,
    "bgmusic": false
}
```

In der neuen Version laden wir auch eine Config-Datei im JSON-Format. Damit können wir die Grundeinstellungen für das Spiel steuern. In dieser Variante bereits integriert:

- `bgmusic` - false oder true - Hintergrundmusik aktivieren oder deaktivieren (Damit ihr eure Eltern/Geschwister nicht so nervt)

Die Eigenschaften `stars` und `levels` brauchen wir erst in der nächsten Variante (03_phaserEinstieg_02).

Dein Einsatz der Eigenschaft `bgmusic` seht ihr in der `GameScene` in der Methode `addSound()`:

```
if (this.myconfig.bgmusic) {
    this.bgm = this.sound.add('backgroundmusic');
    this.bgm.play();
}
```

Die Hintergrundmusik wird nur geladen und abgespielt wenn `this.myconfig.bgmusic` `true` ist.

1.5. Dude

```
class Dude extends Phaser.Physics.Arcade.Sprite {
    constructor(scene,x,y,key) {
        super(scene, x, y, key);
        scene.physics.world.enable(this);
        scene.add.existing(this);
        this.setCollideWorldBounds(true);
        this.setBounce(0.2);
        this.createAnimations(scene);
        this.anims.play('stand');
        this.score = 0;
    }
```

Gerade bei der Klasse für den Player sieht man, dass die Objektorientierung zu dazu beitragen kann, Code verständlicher und geordneter zu gestalten.

Wie schon mehrfach erwähnt kann man ja eine Klasse - laienhaft - als eine Art Drehbuch sehen. So können wir auch in unserem Spiel alles, was der Dude können soll (Methoden) und seine Eigenschaften in einer Klasse zusammenfassen.

Die Klasse Dude wird von der Phaser-Kasse Phaser.Physics.Arcade.Sprite abgeleitet. Damit erhält sie gleich die Grund-Funktionalitäten, die in einem Arcade-Spiel für ein Sprite-Objekt notwendig sind.

Der Konstruktor für ein Sprite in Phaser benötigt als Parameter die Szene, die Koordinaten (x,y) und den key ("dude"), d.h den Namen des Grafik-Objektes, wie wir (bzw. Phaser) es intern ansprechen.

Wichtig ist auch hier der Aufruf des Konstruktors der Super-Klasse, d.h. jener Klasse von der unsere Klasse erbt (Phaser.Physics.Arcade.Sprite) der wir diese Eigenschaften gleich weiter übergeben.

Außerdem müssen wir jetzt für das Sprite (mussten wir vorher in der "Funktionsorientierten" Variante nicht) die Physics-Engine aktivieren `scene.physics.world.enable(this)`. Das Schlüsselwort `this` verweist in diesem Fall ja auf den Dude. Auch zur Szene dazufügen müssen wir das Objekt mit `scene.add.existing(this).`

Die restlichen Dinge bleiben unverändert. Eine schöne Sache ist, dass wir die Animationen, die für diese Objekt-Klasse gedacht sind auch hier anlegen. Dazu setzen wir hier unsere eigene Funktion `this.createAnimations()` ein. Trotzdem sind die Animationen aber Unterobjekte der Szene.

Den Spielstand für unseren Spieler den speichern wir auch direkt in unserem Spielerobjekt (`score`), und auch die Steuerung, d.h. bei welcher Taste soll unser Spieler was machen (Geschwindigkeit, Animation), `kapseln` wir in die Dude-Klasse.

Angelegt wird das Playerobjekt dann in der `GameScene` durch Aufruf der Funktion `initPlayer()`. Diese bekommt als Parameter den score übergeben, initialisiert das Objekt in dem es mit `new Dude(this,300,100,'dude')` den Konstruktor aufruft. Der Score wird auch gleich der Eigenschaft `player.score` zugewiesen.

```
initPlayer(score) {
    this.player = new Dude(this,300,100,'dude');
    this.player.score = score;
}
```

Damit der Player während des Spiels auch auf Tasten reagieren kann, bzw. entsprechende Sound starten muss die Methode `update()` der Klasse Dude in der update-Methode der GameScene-Klasse aufgerufen werden.

Das Update des Player-Objektes benötigt dafür das Objekt für die Tasten (`cursors`) und das Sound-Objekt (`mysound`) - um darauf zugreifen zu können. Für den Zugriff einer Klasse (durch Übergabe als Parameter) auf Objekte/Eigenschaften einer anderen Klasse wird oft auch der allgemeine Begriff "`Schnittstelle`" verwendet.

```
update() {
    this.player.update(this.cursors, this.mysound);
}
```

1.6. Struktur der GameScene-Klasse

In der GameScene-Klasse wird die Dude-Klasse importiert und im Konstruktor einmal alle Variablen/Eigenschaften definiert, die global benötigt werden.

```
import Dude from '../sprites/Dude.js';

class GameScene extends Phaser.Scene {
    constructor(test) {
        super({
            key: 'GameScene',
        });
        this.myconfig;
        this.level;this.levelnum;
        this.platforms;
        this.bgm;this.cursors;this.mysound;
        this.player;
    }
```

Die `create-Funktion` wurde bereinigt, indem Befehle in Gruppen zusammengefasst in eigene Funktionen ausgelagert wurden. Zum Beispiel `addSound()` für alles was mit dem Laden von Audio-Dateien zu tun hat oder in `initControlsAndCams()` alle Befehle bezogen auf Controller und Kameras.

Der Vorteil dieser Modularisierung liegt in einer besseren `Verständlichkeit/Übersichtlichkeit` und einer leichteren `Wartbarkeit`.

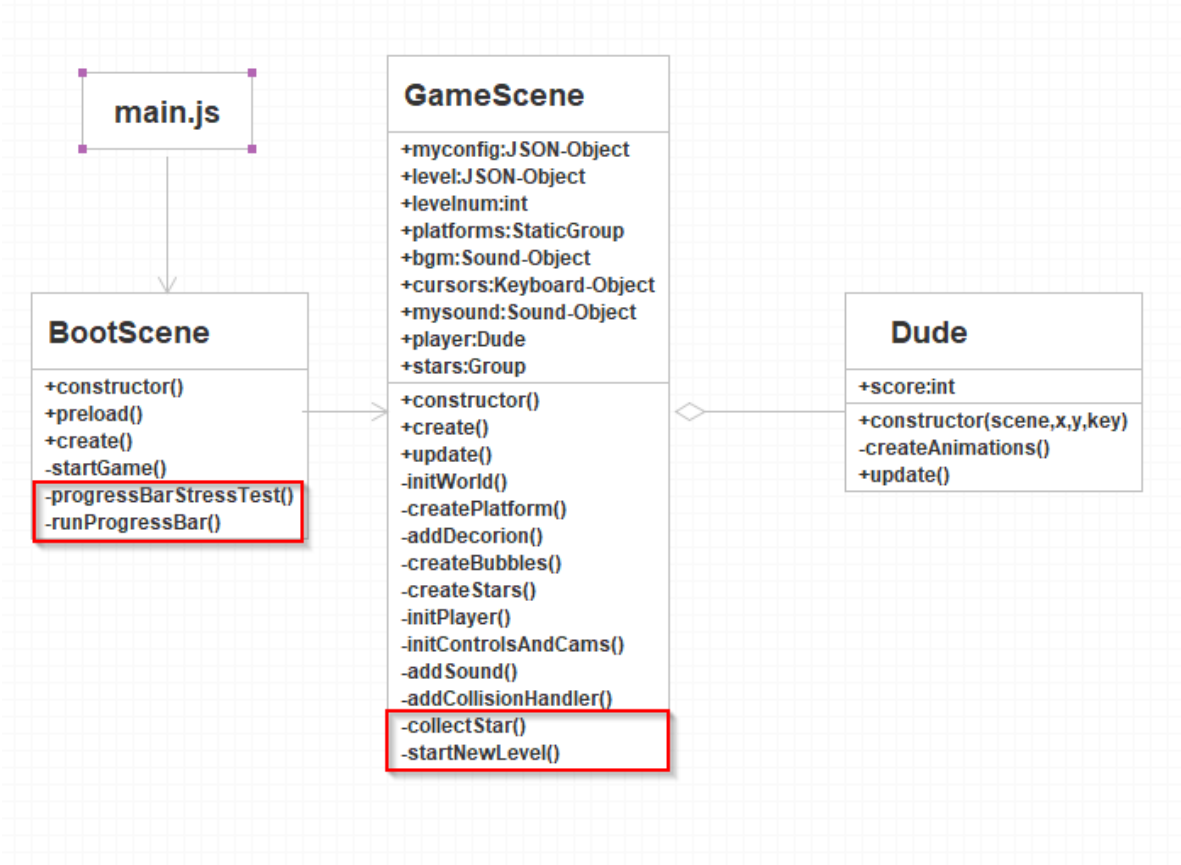
```
create(data) {
    this.myconfig = this.cache.json.get('config');
    this.levelnum = (data.level <= this.myconfig.levels) ? data.level : 1;
    this.initWorld();
    this.initPlayer(data.score);

    this.initControlsAndCams();
    this.addSound();

    this.addCollisionHandler();
}
```

2. Ladebalken, Startbutton, Stars

> Projektordner: 03_phaserEinstieg_02



2.1. Ladebalken - ProgressBar

Beim Laden von Assets in Spiele vergeht - gerade bei Web-Spielen - oft ein gewisser Zeitraum, bis das Spiel dann wirklich gestartet werden kann. Dies wird oft mit der Anzeige von Ladebalken überbrückt.

In der nächsten Erweiterung des Spiels (03_phaserEinstieg_02) ist so ein Ladebalken in der `BootScene` (welche ja für das Laden zuständig ist) eingebaut.

Am Beginn des `preload()` wird die Funktion `runProgressBar` gestartet:

```
1  preload() {
2      this.runProgressBar();
```

In der Funktion `runProgressBar` werden Grafiken für die Bar und die Box vorbereitet (Punkt 1), d.h. Phaser bietet auch die Möglichkeit Grafiken während der Laufzeit zu erzeugen ohne auf externe Bilder zugreifen zu müssen - hier mit `this.add.graphics()`.

Auch Text kann in Phaser auf unterschiedliche Weise erzeugt werden (dazu auch noch später in diesem Tutorial bei unserem ScoreBoard). Hier ist es einmal der `loadingText` mit `this.make.text(config)`

Genauso bei den Prozenten für den Fortschritt - `percentText` - und der Anzeige, welche Datei gerade geladen wird - `assetText` (beides hier nicht abgebildet - siehe `BootScene.js`)

> Texte und Grafiken sind keine Variablen mit einfachen Datentypen, sondern Objekte

```
runProgressBar() {
    var progressBar = this.add.graphics();
    var progressBox = this.add.graphics();
    progressBox.fillStyle(0x222222, 0.8);
    progressBox.fillRect(240, 270, 320, 50);

    var width = this.cameras.main.width;
    var height = this.cameras.main.height;
    var loadingText = this.make.text({
        x: width / 2,
        y: height / 2 - 50,
        text: 'Loading...',
        style: {
            font: '20px monospace',
            fill: '#ffffff'
        }
    });
    loadingText.setOrigin(0.5, 0.5);
```

1

2

Ein wesentlicher Baustein in der Programmierung - hier insbesondere in der Spiele-Programmierung - ist es, auf Ereignisse zu reagieren. Dafür gibt es in den Programmiersprachen Ereignis/Event-Listener und Ereignis/Event-Handler.

Bei unserer ProgressBar brauchen wir jetzt einige Event-Listener. Diese werden hier mit `this.load.on` erstellt, d.h. reagiere auf Veränderungen beim Laden (Lade-Objekt). Wir benötigen drei Ereignisse:

Ereignis-Name (String)	**Parameter**	**Beschreibung**
progress	value	Wird bei jeder (prozentuellen) Veränderung des Fortschritts ausgelöst und übergibt der dem Event-Handler den Prozentwert (value)
fileprogress	file	Wird jedes mal aktiviert, wenn eine neue Datei geladen wird. Parameter ist der Dateiname
complete	-	Wird ausgelöst, wenn alle Dateien/Assets in den Speicher geladen wurden

Wir können dies nun für unser Spiel einsetzen und so die Texte und Grafiken bei jeder Änderung auch dementsprechend anpassen. Beim `complete` entfernen wir wieder alle Text- und Grafik-Objekte aus dem Speicher.

```
        this.load.on('progress', function (value) {
            percentText.setText(parseInt(value * 100) + '%');
            progressBar.clear();
            progressBar.fillStyle(0xffffff, 1);
            progressBar.fillRect(250, 280, 300 * value, 30);
        });

        this.load.on('fileprogress', function (file) {
            assetText.setText('Loading asset: ' + file.key);
        });

        this.load.on('complete', function () {
            progressBar.destroy();
            progressBox.destroy();
            loadingText.destroy();
            percentText.destroy();
            assetText.destroy();
        });
```

Damit wir unseren Ladebalken auch vernünftig testen können gibt es die Möglichkeit einmal künstlich mehr Grafiken zu laden. Dafür haben wir die Methode `progressBarStressTest()`. Diese rufen wir am Ende des `preload()` auf und diese beinhaltet eine Schleife, in der eine Grafik mit unterschiedlichem `key` (Namen) 2000 mal geladen wird. Wenn wir diese Testmethode nicht mehr benötigen können wir sie löschen oder einfach nicht mehr aufrufen.

```
1  progressBarStressTest() {
2      for (var i = 0; i < 2000; i++) {
3          this.load.image('logo'+i, this.imgpath+'bubble.png');
4      }
5  }
```

Viele Cheats in Spielen gibt es deswegen, weil Programmierer vorher Testmethoden eingebaut haben.

2.2. Startbutton

Mit einem Startbutton runden wir die Lade-Szene (`BootScene`) noch ein wenig ab. Ist das `Preload` beendet wird ja die `create`-Methode aufgerufen. Hier erstellen wir ein Textobjekt (diesmal mit `this.add.text`).

Auch hier ist wieder ein `Event-Listener` wichtig. mit `startButton.on('pointerdown', this.startgame, this)`, bringen wir unserem Programm bei, dass es bei Klick auf das Objekt `startButton` die Methode `startGame()` aufrufen soll. Und damit gelangen wir in die nächste Szene - die `GameScene`, unsere Hauptszene.

```
    create() {
        var textConfig = {
            color: '#000000',
            fill: '#ffffff',
            backgroundColor: '#0000ff',
            padding: 20
        }
        var startButton = this.add.text(400, 300, 'Spiel starten', textConfig).setOrigin(0.5,0.5);
        startButton.setInteractive({ useHandCursor: true });
        startButton.on('pointerdown', this.startGame, this);
        //this.startGame();
    }

    startGame() {
        this.scene.start("GameScene",{level:1,score:0});
    }
```

2.3. Collect Stars

Bei den Sternen fehlt uns noch, das der Spieler diese auch wirklich einsammeln kann. Dazu ergänzen wir die Methode `addCollisionHandler()` um eine neue Art von Collider. Wir nehmen hier die Methode `overlap` - diese wird nicht schon bei Berührung, sondern erst bei signifikanter Überschneidung von Gegenständen aktiviert. Hier legen wir fest, dass diese Methode bei Überlappung des Players mit einem der Objekte der Gruppe `this.stars` ausgelöst werden soll. Wichtig ist diesmal auch der 3. Parameter. Dieser definiert eine sogenannte `Callback-Methode`, d.h. eine Methode, welche aufgerufen wird, wenn das Ereignis eintritt - `this.collectStar`.

```
this.physics.add.overlap(this.player, this.stars, this.collectStar, null, this);
```

Diese Callback-Methode bekommt die beiden betreffenden Objekte übergeben, d.h. den Spieler und das konkrete Sternobjekt und kann nun mit den `collect-Sound` abspielen (dieser wurde bei der Definition von `mysound` dazugefügt - siehe `addSound()`) und den Stern "zerstören". Mit `star.destroy()` wird das jeweilige Objekt deaktiviert und aus der Szene genommen.

```
collectStar(player, star) {
    this.mysound.collect.play();
    star.destroy();

```

2.4. Neues Level starten / Kamera-Effekte

Wir bleiben bei der `collectStar-Methode` und schauen uns an, wie man in das nächste Level wechseln kann. Dazu brauchen wir auch einen Kamera-Effekt.

Zuerst einmal eine Abfrage, die festlegt, wann dieses Ereignis eintritt: Nämlich wenn es keine Sterne mehr gibt. Die Methode `this.stars.countActive(true)` liefert die Anzahl der aktiven Objekte dieser Sternengruppe. Um nicht ewig lange spielen zu müssen, damit man das testen kann, wurde in der `config.json` die Anzahl der Sterne deswegen auf 2 gesetzt.

```
    if (this.stars.countActive(true) == 0) {
        this.startNewLevel(this.levelnum,player.score);
    }
```

Wenn dies zutrifft wird die Methode `startNewLevel` aufgerufen. Diese bekommt das aktuelle `level` und den aktuellen `Player-Score` übergeben. (Um den Score kümmern wir uns erst in der nächsten Erweiterung des Spiels).

Interessant ist hier, dass Kameras unterschiedliche interessante Effekte bieten. Beim Wechseln des Levels setzen wir hier den `Fade-Effekt` der Hauptkamera - `this.cameras.main` - ein. Der Fade-Effekt bewirkt, dass die Kamera in einer angegebenen Zeitdauer auf eine bestimmte Farbe (hier schwarz) wechselt.

Parameter bei Fade:

- Dauer in ms - 1000
- R (0), G (0), B (0) - Parameter für RGB-Farbwerte
- false - hier könnte eine Callback-Funktion eingesetzt werden.

Um den Effekt auch wirklich angezeigt bekommen müssen wir wieder einen `Event-Listener` hinzufügen, diesmal der Kamera. Ist der Fade-Effekt fertig (`camerafadeoutcomplete`) wird das Level um 1 erhöht und mit `this.scene.restart` wird die aktuelle Szenen (`GameScene`) neu gestartet. Wir übergeben die neue `Level-Nummer` und den `Score`.

```
startNewLevel(level,score) {
    this.cameras.main.fade(1000,0,0,0,false);
    this.cameras.main.on("camerafadeoutcomplete",function() {
        var newLevel = level + 1;
        this.scene.restart({level:newLevel,score:score});
    },this);
}
```

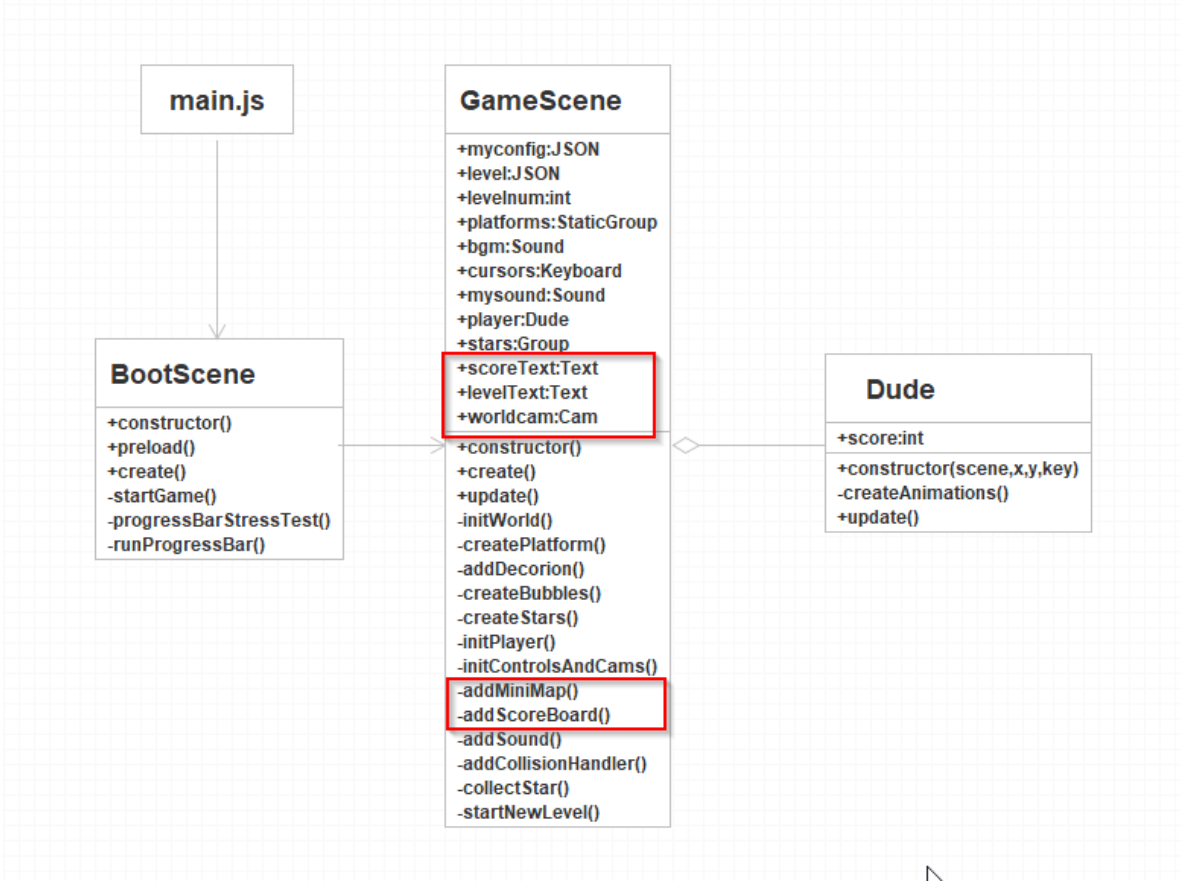
Beim Start oder Restart der Szene wird das `create` aufgerufen (und die Daten mit dem data-Objekt übergeben). In unserem Beispiel haben wir einen Sicherheitsmechanismus eingebaut, der bewirkt, dass wir nicht zu weit hinaufzählen (es gibt ja nur 2 Levels). Eine Abfrage: `Wenn data.level <= der Levelanzahl` in der Konfiguration `dann nimmt data.level`, `sonst 1`. Diese Abfrage kann mit dem `ternären Operator` abgekürzt werden: `(Bedingung) ? Wert bei true : Wert bei false`.

> Den ternären Operator setzt man dort ein, wo der Wert einer Variable entsprechend einer Bedingung unterschiedlich gesetzt wird.

```
create(data) {
    this.myconfig = this.cache.json.get('config');
    this.levelnum = (data.level <= this.myconfig.levels) ? data.level : 1;
```

3. Kameras - Minimap / ScoreBoard, Texte

> Projektordner: 03_phaserEinstieg_03



3.1. Kameras

Mit Kameras können nicht nur interessante Effekte erzielt werden, wie im Kapitel 2 gezeigt, sondern es können auch mehrere Kameras für unterschiedliche Zwecke eingesetzt werden. In diesem Kapitel zeige ich euch den Einsatz einer zweiten Kamera als Mini-Map - eine kleine Ansicht der gesamten Welt.

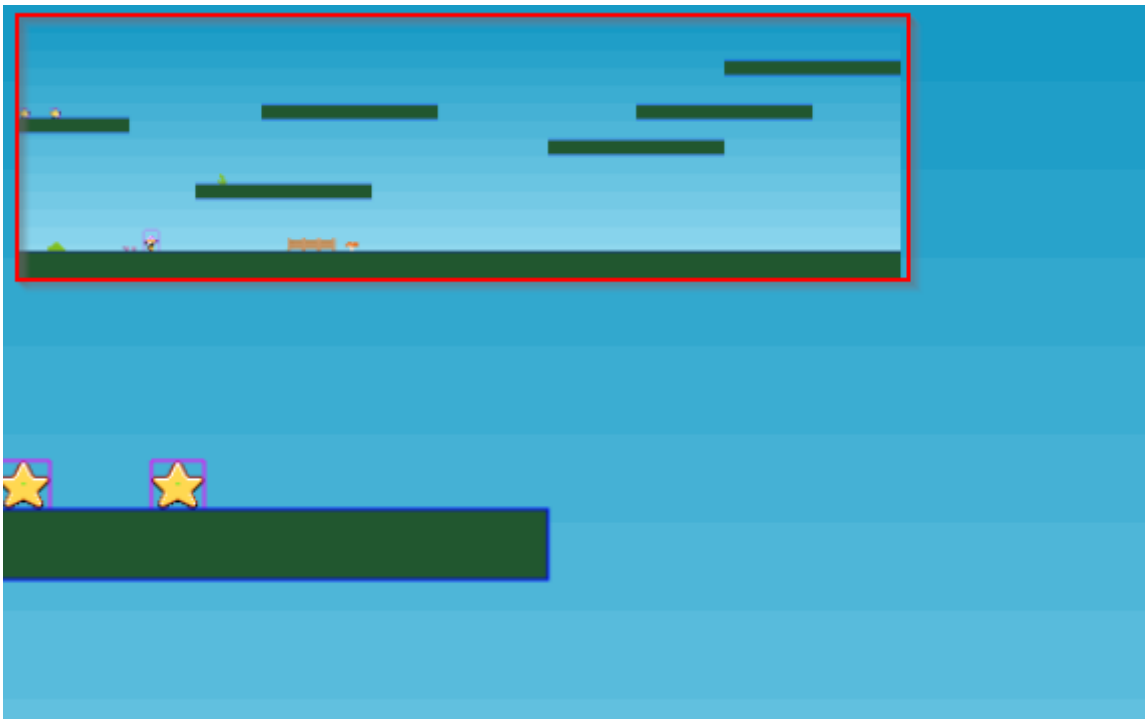
Dazu wird in der Methode `initControlsAndCams()` der Aufruf `addMiniMap` dazugefügt. Diesen kann man später auch wieder auskommentieren.

Darunter sieht man dann wie man eine neue Kamera in einer neuen Eigenschaft/Variable speichert - this.worldcam - und die entsprechenden Einstellungen dazu. Wichtig ist `setZoom` um die Welt kleiner zu zoomen (hier auf 20% - 0.2) und die Koordinaten und Größenangaben (`x,y,width,height`).

```
initControlsAndCams() {
    this.cursors = this.input.keyboard.createCursorKeys();
    //Größere Welt als der Bildausschnitt - Kamera folgt dem Player
    this.cameras.main.startFollow(this.player);
    this.physics.world.setBounds(0,0,2000,600);
    this.cameras.main.setBounds(0,0,2000,600);
    this.addMiniMap();
}

addMiniMap() {
    //Just another cam!
    this.worldcam = this.cameras.add(10, 10, 400, 120).setZoom(0.2).setName('mini');
    this.worldcam.scrollX = 800;
    this.worldcam.scrollY = 240;
    //this.worldcam.setBounds(0,0,2000,600);
}
```

Und schon ist unsere Minimap fertig - und man sieht links oben einen verkleinerten Ausschnitt des gesamten Spielgeschehens:



Weitere Infos zu Kameraeffekten und -einstellungen findet man in der API-Doku von Phaser (siehe Kapitel 4). Wenn man da unter `Classes - Phaser.Cameras.Scene2D.Camera` - https://photonstorm.github.io/phaser3-docs/Phaser.Cameras.Scene2D.Camera.html findet man alle Methoden und Eigenschaften einer 2D-Kamera in Phaser.

3.2. Scoreboard / Texte / Bitmaptext

Ähnlich kurz und einfach ist es, ein ScoreBoard zu erstellen. Im `create` fügen wir aus diesem Grund den Aufruf von `addScoreBoard()` hinzu. Die Methode selber erstellt dann mit `this.add.text` Textobjekte, die wir in globalen Variablen (`scoreText` und `levelText`) abspeichern. In beiden Texten hängen wir auch gleich die tatsächlichen Werte dazu (`player.score` und `levelnum`). Weitere Parameter sind x- und y-Position gleich zu Beginn und nach dem Text in einem Konfigurationsobjekt Schriftgröße und Füllfarbe für die Schrift.

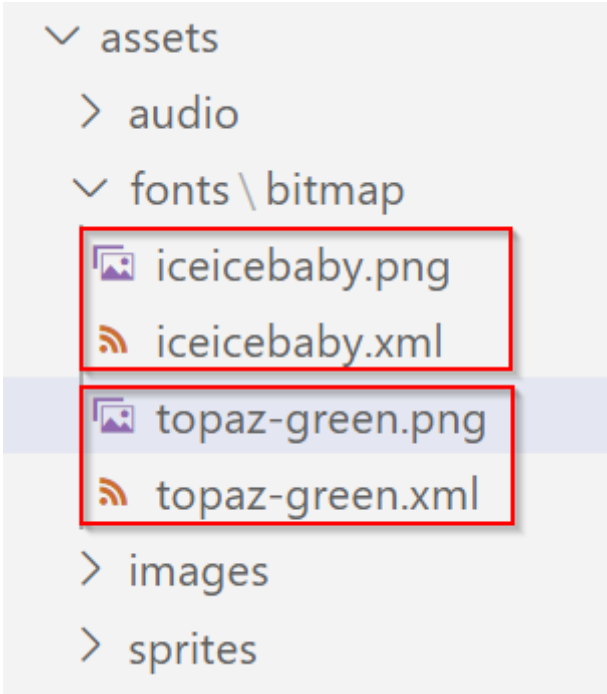
Wichtig ist auch noch der Befehl `setScrollFactor(0,0)`. Dieser bewirkt, dass sich bei Bewegung der Kamera (diese folgt ja bei uns dem Player) die Texte entsprechend dort bleiben, wo sie sind, d.h. nicht einfach wegscrollen.

```
addScoreBoard() {
    this.scoreText = this.add.text(20, 550, 'Score: '+this.player.score,
                                   { fontSize: '32px', fill: '#FFF' });
    this.scoreText.setScrollFactor(0,0);
    this.levelText = this.add.text(400, 550, 'Level: '+this.levelnum,
                                   { fontSize: '32px', fill: '#FFF' });
    this.levelText.setScrollFactor(0,0);
}
```

In der Methode `collectStar` fügen wir jetzt noch dazu, dass bei jedem Aufheben eines Sternes 10 Punkte dazugefügt werden und im `scoreText` immer wieder der neue Spielstand - mit `setText()` - angezeigt wird.

```
this.player.score += 10;
this.scoreText.setText('Score: ' + this.player.score);
```

Um noch schönere Texte zu bekommen, gibt es auch die Möglichkeit Grafiken für die einzelnen Zeichen einzusetzen. Das geht mit sogenannten Bitmap-Textobjekten. In den `assets` von unserem Spiel findest du zwei Beispiele: `iceicebaby` und `topazgreen`. Eine Schriftart besteht dabei immer aus zwei gleichnamigen Dateien: Zunächst einmal die Grafik mit den Zeichen und dann eine XML-Steuerdatei, die angibt, welche Grafik (angegeben durch Koordinaten, Breite und Höhe) für welches Zeichen steht.



Ein Beispiel dazu findest du bei den Phaser-Beispielen - wenn du ausgehend von der Phaser-Hauptseite (phaser.io) folgendem Menüpfad folgst:

`phaser.io - Examples - Game Objects - Bitmaptext - Dynamic - Canvas-Text`

Hier der direkte Link dazu:

http://phaser.io/examples/v3/view/game-objects/bitmaptext/dynamic/canvas-text

4. Quellen und Weiterführende Infos

Link	Beschreibung
Beispiele	Hier findest du viele Beispiele von den Entwicklern nach Kategorien geordnet. Diese kann man abspielen oder aber in einer sogenannten Phaser-Sandbox editieren und herumprobieren. Auch dieses Beispiel stammt vom Prinzip her aus den Beispielen und wurde von mir abgewandelt/erweitert.
API-Docs	Die Dokumentation der Klassen/Schnittstellen. Es gibt folgende Kategorien -Namespaces - Auflistung nach Namensräumen -**Classes** - Auflistung der Klassen! -**Events** - Welche Event-Listener gibt es für welche Klassen -**GameObjects** - Welche Arten von Game-Objekten gibt es -**Physics** - Welche Physic-Engines gibt es -**Scene** - Welche Eigenschaften/Objekte hat eine Scene
Tutorials	Eine große Anzahl an Tutorials in unterschiedlicher Qualität