Tutorial 02 - PhaserEinstieg

1. Einleitung

Ausgangspunkt für die neuen Ergänzungen ist der neue Stand 02_phaserEinstieg - hier sind zusätzliche Dateien enthalten und bereits ein paar kleine Ergänzungen eingepflegt.

Wichtiger Hinweis:

Manchmal werden Änderungen im Browser - auch wenn man in der Codedatei auf Speichern geht - nicht so leicht vom Browser übernommen, da er sich viele Dinge (z.B. Grafiken) aus einem Cache holt - damit werden dann Seiten auf denen du oft bist schneller geladen.

In der Entwicklung stört das, aber was kann man dagegen tun:

- Browser schließen und wieder öffnen
- Shift-F5 - hier allerdings Chrome hartnäckig und nimmt oft trotzdem Dinge aus dem Browser (da ist Firefox besser zum Testen)
- Cache beim Browser deaktivieren (Bei Chrome in der Entwickleransicht - Network mit Disable Cache möglich)
- Eventuell hat man aber auch einen Fehler im Code - F12-Konsole - Schauen ob Fehlermeldungen erscheinen

---

2. Player-Ergänzungen - Touching Down / Bounce

Bereits vorhandene Ergänzungen betreffen die zwei folgenden Funktionalitäten

In der `initPlayer`-Funktion ist für den Player ein Bounce-Effekt gesetzt, d.h. beim Landen auf einer der Plattformen (nach einem Sprung) prallt er noch einmal ab. Dies geht mit folgendem Befehl:

```
1  player.setBounce(0.2);
```

Außerdem wurde im `update` in der Abfrage ob die Pfeil-Hinauf-Taste gedrückt wurde auch noch eine zweite Bedingung hinzugefügt (Mit einer Und-Verknüpfung). Jetzt springt er nur ab, wenn Pfeil-Hinauf gedrückt wurde und der Body des Players auf einem Untergrund steht (Plattform oder Weltgrenze) bzw. diesen berührt.

```
1  if (cursors.up.isDown && player.body.touching.down) {
2      player.setVelocityY(-330);
3  }
```

3. Sound

Nun fügen wir Sound dazu:

1. Hintergrundsound (Hintergrundmusik)

2. Musikeffekte bei bestimmtem Verhalten (z.B. Springen)

Zuerst brauchen wir wieder zwei globale Variablen (oben bei den anderen Variablen). `bgm` für die Hintergrundmusik (Backgroundmusic) und `myounds` für alle anderen Töne.

```
var bgm, mysounds;
```

In der `preload`-Funktion müssen natürlich auch die entsprechenden Audio-Dateien geladen werden. Wie bei allen anderen Assets geht das mit `this` (die Szene) und dem entsprechenden `load`-Befehl. Hier ist es `this.load.audio`. Wiederum wird ein interner `key` (Name) dafür vergeben und dann der Speicherort angegeben. Eine tolle Funktionalität bei Phaser ist es, dass man hier ein Array von Musikdateien - wie hier bei der Hintergrundmusik - angeben kann und es wird dann jenes Dateiformat (hier ogg oder mp3) genommen, welches der Browser wirklich verarbeiten kann.

```
this.load.audio('backgroundmusic', [
        'assets/bgm.ogg',
        'assets/bgm.mp3'
    ]);
this.load.audio('jump','assets/jump.wav');
this.load.audio('collect', 'assets/coin.wav');

```

In der `create`-Funktion werden dann die Sound-Variablen (bzw. Sound-Objekte) initialisiert. Dabei ist hier `mysound` ein Objekt, das (momentan) zwei Variablen/Eigenschaften beinhaltet, die jeweils einen anderen Sound laden. Die Variable `bgm` wird mit der Hintergrundmusik (`key` 'backgroundmusic') initialisiert. Mit `bgm.play()` kann man dann auch sofort die Hintergrundmusik starten.

```
    mysound = {
        jump: this.sound.add("jump"),
        collect: this.sound.add("collect")
    };
    bgm = this.sound.add('backgroundmusic');
    bgm.play();
```

Einen zweiten Soundeffekt ergänzen wir auch: Im update setzen wir beim Sprung den Befehl `mysound.jump.play()` ein. Jetzt sollte unser Player beim Springen einen Ton erzeugen.

```
if (cursors.up.isDown && player.body.touching.down) {
    player.setVelocityY(-330);
    mysound.jump.play();
}
```

Dies war nur ein kurzer Einstieg in die Welt des Sounds in Games. Da gibt es - auch bei Phaser - noch viele viele Dinge und Einstellungen, die man damit machen bzw. entdecken kann.

Achtung: Noch ein wichtiger Hinweis für Web-Audio:*

Gerade in Chrome kann es sein, dass die Hintergrund nicht sofort startet, sondern erst, wenn man als Benutzer den Spieler durch Tastendruck einmal bewegt hat. Das liegt an den Einstellungen von Chrome.

In der Entwickleransicht von Chrome (Mit Taste F12 aufrufbar) erscheint in der Konsole dann auch folgender Warning:

"*The AudioContext was not allowed to start. It must be resumed (or created) after a user gesture on the page. https://goo.gl/7K7WLu*"

Damit will Chrome sinnlose Hintergrundmusik auf Webseiten - ohne das dies der Benutzer bewusst aktiviert hat - verhindern.

In Spielen kann man das mit einem Play-Button oder ähnlichen Elementen umgehen, sodass vor Start des Spiels auf jeden Fall eine Benutzerinteraktion erfolgt.

---

4. JSON

Ein wichtiges Textformat ist heutzutage das JSON-Textformat. JSON steht für Java Script Object Notation, d.h. es werden damit Datenobjekte definiert.
Dies wird in vielen Webanwendungen eingesetzt um Daten auszutauschen (z.B. Wetterdaten von einem Wetterdienst im JSON-Format abrufen).

In der Spiele-Entwicklung wird JSON sehr oft für das Speichern von Levels, Welten oder aber auch Animationen verwendet.

Wir verwenden es, damit wir nicht alle Plattformen händisch im Code erzeugen müssen, sondern beliebig viele mit ein paar Einstellungszeilen erstellen lassen können. Außerdem werden wir so auch noch Hintergrunddekoration (Blumen, Zaun, Gras) einbauen.

Unsere Datei befindet sich unter `levels/level01.json` und ist folgendermaßen aufgebaut:

Sie beginnt - wie jede JSON-Datei - mit einer geschwungenen Klammer (und am Schluss wieder die schließende dazu). Unsere enthält dann zwei Objekte: `platforms` und `decoration`. Die Namen von Variablen/Eigenschaften schreibt man in JSON als String (mit Hochkomma). Jedes dieser zwei Objekte enthält ein Array (durch die eckigen Klammern gekennzeichnet), welches wiederum Unterobjekte mit den Daten enthält. Bei den Plattformen sind das x- und y-Koordinate sowie (optional) der Skalierungsfaktor, d.h. um welchen Wert soll die Plattform vergrößert oder verkleinert werden (Kommazahlen schreibt man auch hier mit einem Punkt, also z.B. 0.5 um die Plattform halb so groß zu machen).

Bei den dekorativen Elementen haben wir auch die Koordinaten, wo das jeweilige Element in der Welt platziert werden soll (x,y) und - nachdem alle Hintergrundgrafiken in einer Spritesheet-Grafik (decor.png im Assets-Ordner) liegen - welches Frame, d.h. welche Grafik genommen werden soll (beginnend bei 0 - d.h. 2 ist das Zaunelement)

```
{
    "platforms": [
        {"x": 400, "y": 568, "scale": 2},
        {"x": 600, "y": 400},
        {"x": 50, "y": 250},
        {"x": 750, "y": 220}
    ],
    "decoration": [
        {"frame": 2, "x": 630, "y": 514},
        {"frame": 2, "x": 663, "y": 514},
        {"frame": 2, "x": 697, "y": 514},
        {"frame": 3, "x": 756, "y": 514},
        {"frame": 1, "x": 84, "y": 514},
        {"frame": 0, "x": 252, "y": 514},
```

```
        {"frame": 4, "x": 462, "y": 362}
    ]
}
```

Wie binden wir die JSON-Datei ein und erstellen die jeweiligen Elemente?
Zuerst wieder eine globale Variable, in die wir diese JSON-Daten laden wollen:

```
var level;
```

Im `preload` ergänzen wir zwei Codezeilen, nämlich das Laden der JSON-Datei - wieder nach dem gleichen Schema wie andere Assets mit `this.load.json(key, Speicherort)` und das Spritesheet - welches diesmal keine Animation, sondern unterschiedliche Grafiken enthält.

```
this.load.json('level1', 'levels/level01.json');
this.load.spritesheet('decoration', 'assets/decor.png', { frameWidth: 42, frameHeight: 42});

```

Nun passen wir auch noch die Funktion `initWorld()` an. Das händische erstellen von Plattformen (platforms.create) können wir auskommentieren, das brauchen wir so jetzt nicht mehr. Dafür initialisieren wir jetzt die Level-Variable mit `scene.cache.json.get(key).` Nun sind die beiden Objekte `platforms` und `decoration` unserer JSON-Datei direkt verfügbar - zur Erinnerung das sind jetzt zwei Arrays mit Unterobjekten. JavaScript bietet jetzt einen praktischen Nutzen: Man kann mit `forEach` alle Objekte/Werte eines Arrays durchgehen. Dabei muss man zwei Parameter angeben: Eine Funktion, die für jedes der Objekte aufgerufen werden soll (z.B. createPlatform) und das jeweilige Unterobjekt übergibt man durch die Angabe des Schlüsselwortes `this`.

```
function initWorld() {
    //this.add.image(400,300,'sky');
    scene.add.image(0,0,'sky').setOrigin(0,0);
    platforms = scene.physics.add.staticGroup();

    /*
    //Grund
    platforms.create(400, 568, 'platform').setScale(2).refreshBody();

    //Plattformen
    platforms.create(600, 400, 'platform');
    platforms.create(50, 250, 'platform');
    platforms.create(750, 220, 'platform');
    */
    level = scene.cache.json.get('level1');
    level.platforms.forEach(createPlatform, this);
    level.decoration.forEach(addDecoration, this);
}
```

Die jeweiligen Funktionen (auch callback-Funktion genannt - da wir sie nicht direkt aufrufen, sondern dies die Funktion `forEach` erledigt) können wir direkt unter das `initWorld()` geben. Beide brauchen einen Parameter in Klammer, damit wir das übergebene Objekt wo abspeichern (der Name ist frei wählbar).

Nun können wir sagen, was wir mit dem jeweiligen Objekt machen wollen. Das Objekt p enthält ja x,y-Koordinate und einen eventuellen Scale-Faktor (scale). Diese Eigenschaften/Werte nehmen wir jetzt einfach für unseren `platforms.create`-Befehl her.

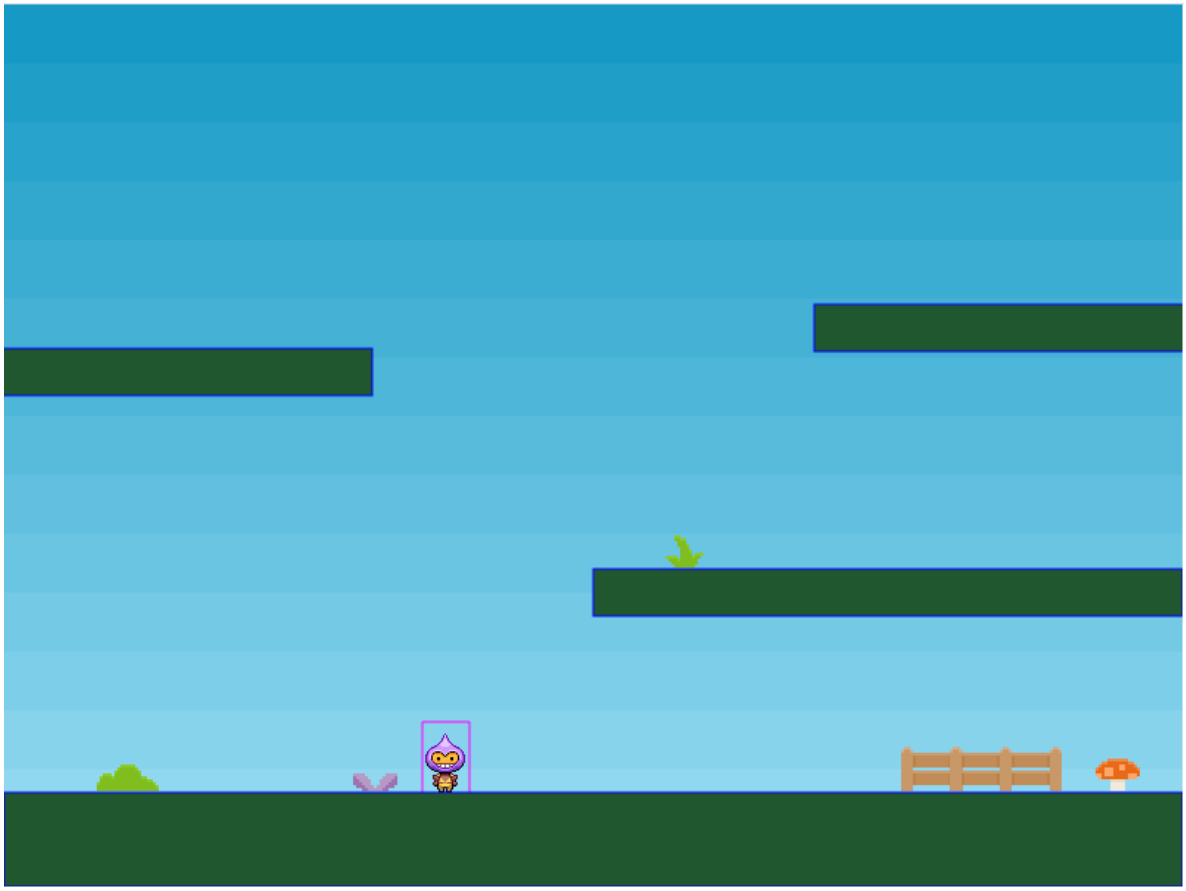
```
function createPlatform(p) {
    platforms.create(p.x,p.y, 'platform').setScale(p.scale).refreshBody();
}

```

Analog funktioniert das auch bei der Dekoration. Hier lautet die Variable `decodata` (frei wählbar) und die darin enthaltenen Werte (`decodata.x`, `decodata.y` und `decodata.frame`) werden in den `add.image`-Befehl für die Scene eingesetzt. Natürlich braucht dieser Befehl auch noch den Verweis auf die zu verwendende Grafik: 'decoration' (Der Key, den wir im Preload festgelegt haben).

```
function addDecoration(decodata) {
    scene.add.image(decodata.x, decodata.y, 'decoration', decodata.frame);
}
```

Nun haben wir alle Plattformen und auch einige Hintergrundgrafiken mit wenigen Zeilen Code erzeugt und in der Level-Datei (level01.json) können wir beliebig viele dieser Elemente hinzufügen. Wir könnten unseren Spielern sogar einen Level-Editor bauen, mit der sie (per Drag&Drop) selber Welten gestalten könnten.

Dein Spiel sollte jetzt so aussehen:



5. Group - repeat

Wir haben ja schon über `groups` oder `staticGroups` (z.B. die Plattformen) gesprochen. In diesem Kapitel werden wir eine Reihe von Sternen erzeugen ohne viele Befehle oder gar eine Schleife einsetzen zu müssen.

Zuerst brauchen wir für unsere Sternengruppe wieder eine globale Variable

```
1  var stars;
```

Im `preload` laden wir wieder die entsprechende Grafik-Datei (key: star, Datei: assets/star.png)

```
1  this.load.image('star', 'assets/star.png');
```

In der Funktion `create` setzen wir - zwischen initWorld und initPlayer - den Aufruf für eine neue Funktion `createStars` ein (Zeile 5). Durch diese modulare Programmierweise ist ja unser Code - wie schon des öfteren besprochen - übersichtlicher und besser wartbar.

Außerdem fügen wir (hier in Zeile 9) auch gleich einen Collider ein, sodass die Sterne auf den Plattformen zu liegen kommen (und nicht aus der Welt fliegen - teste dies indem du Zeile 9 dann weglässt).

```
1  function create() {
2      scene = this;
3      initWorld();
4      //Aufruf der Funktion createStars
5      createStars();
6      initPlayer();
7      this.physics.add.collider(player, platforms);
8      //Auch ein Collider muss wieder erzeugt werden:
9      this.physics.add.collider(stars, platforms);
10     ...
11 }
```

Nun brauchen wir aber auch nur unsere Funktion `createStars`:

Die (globale) Variable wird mit dem Befehl `scene.physics.add.group` initialisiert. Dieser Methode übergeben wir ein Konfigurationsobjekt, welches folgende Parameter festlegt:

- key - Der Name der einzufügenden Grafik (wie wir ihn im `load` festgelegt haben)
- repeat - Wie viele Elemente dieser Grafik sollen auf einmal (wiederholt) eingefügt werden
- setXY - An welcher Position sollen die Grafiken eingefügt werden:
    - x - Startwert für die x-Position
    - y - Startwert für die y-Position
    - stepX - Abstand der einzelnen Grafiken zueinander horizontal

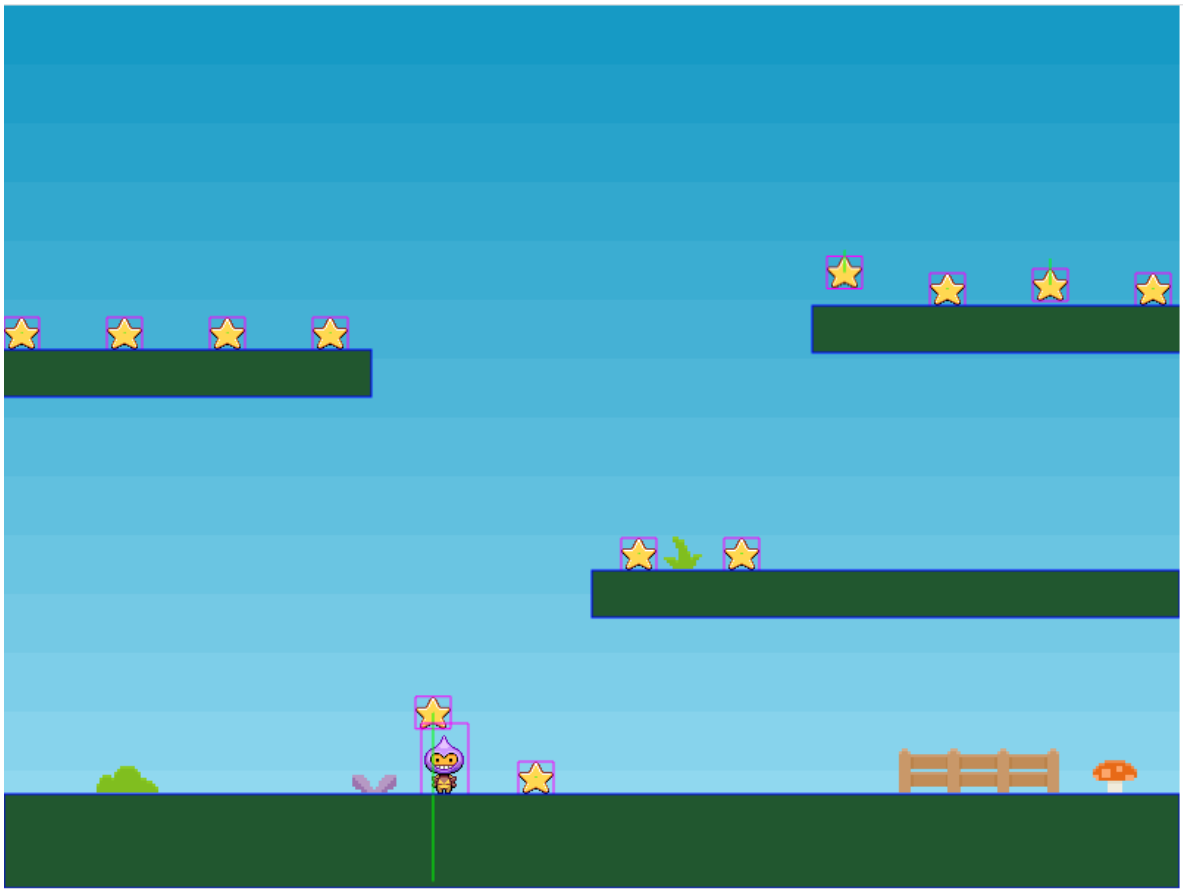
Auch hier würde es noch einige andere Einstellungsmöglichkeiten geben (z.B. stepY um die Elemente auch nach unten zu versetzen)

Jetzt wollen wir allerdings allen Sternen noch einen (unterschiedlichen) Bounce-Effekt beim Aufkommen auf die Plattformen geben. Dazu bietet Phaser die Möglichkeit mit einem sogenannten Iterator alle Elemente einer Gruppe zu durchlaufen. Mit `stars.children.iterate` werden alle Sterne durchlaufen und für jeden Stern eine sogenannte anonyme Funktion aufgerufen, die den jeweiligen Stern übergeben bekommt. (Eine anonyme Funktion ist eine Funktion ohne Name, die man in eine andere Funktion einbaut).

Diese setzt für jeden Stern einen Bounce-Effekt in Y-Richtung (setBounceY) und greift dazu auf eine Methode der Phaser-Mathematik-Klasse (Phaser.Math) zu. Mit FloatBetween wird eine zufällige Gleitkommazahl erzeugt - hier zwischen 0.4 und 0.8. Der Effekt ist jener, dass jeder Stern jetzt unterschiedlich stark abprallt.

```
function createStars() {
    stars = scene.physics.add.group({
        key: 'star',
        repeat: 11,
        setXY: { x: 12, y: 0, stepX: 70 }
    });
    stars.children.iterate(function (stern) {
        //  Jedem Stern einen unterschiedlichen Bounc-Effekt geben:
        stern.setBounceY(Phaser.Math.FloatBetween(0.4, 0.8));
    });

}
```

Hier der nunmehrige Stand in unserem Spiel:



6. Particle / Particle Emitter

Eine letzte Funktionalität, die wir uns dieses mal betrachten ist jene, für diverse optische Effekte in Welten sogenannte Partikel-Emitter einzusetzen. Beispiele dafür wären: Rauchschwaden, Flammen, Meteoritenschauer, ... - bei uns werden es Seifenblasen.

Wiederum laden wir im `preload` eine neue Grafik:

```
this.load.image('seifenblase', 'assets/bubble.png');
```

Am Ende der Funktion `create` rufen wir auch gleich wieder eine neue Funktion auf, die wir dazu nutzen werden unseren Emitter und unsere Partikel zu erzeugen: `createBubbles()`.

```
function create() {
    ...
    createBubbles();
}
```

Im `createBubbles` erzeugen wir zuerst einmal eine Variable für die Partikel und fügen die Seifenblasen-Grafik dazu.

Mit `particles.createEmitter` erzeugen wir einen Emitter (also ein aussendendes Objekt) bei dem wir als Parameter diverse Einstellungen (in einem Objekt) übergeben

```
function createBubbles() {
    var particles = scene.add.particles('seifenblase');

    var emitter = particles.createEmitter({
        x: 600, //x-Position des Emitters
        y: 300, //y-Position
        speed: 100, //Startgeschwindigkeit eines Partikels
        quantity: 1, //Wieviele Partikel werden pro "Ausstoß" erzeugt
        frequency: 10, //Nach welcher Zeitspanne werden neue Partikel ausgestoßen
        gravityY: -10, //Welche Gravitation wirkt auf die Partikel
        //Lebensdauer eines Partikels zwischen 1000 und 2000 ms
        lifespan: { min: 1000, max: 2000 },
        //Es gibt auch unterschiedliche Modi, wie Partikel grafisch erzeugt werden
        blendMode: 'ADD'
    });
}
```

Nun sollten an der oben angegebenen Position jede Menge Seifenblasen erzeugt werden. Probiere ruhig mit den Einstellungen herum, um ein Gefühl für die Möglichkeiten zu erhalten.