MMProg: Praktikum: WiSe 2018/19: Ball02

Dieser Artikel erfüllt die GlossarWiki-Qualitätsanforderungen **nur teilweise**:

Korrektheit: 3 (zu größeren Teilen überprüft)	**Umfang**: 4 (unwichtige Fakten fehlen)	**Quellenangaben**: 3 (wichtige Quellen vorhanden)	**Quellenarten**: 5 (ausgezeichnet)	**Konformität**: 3 (gut)

Vorlesung MMProg

Inhalt | EcmaScript01 | EcmaScript02 | EcmaScript03 | Ball 01| Ball 02 | Ball 03 | Pong 01

Musterlösung: Web-Auftritt (Git-Repository)

Inhaltsverzeichnis

1 Vorbereitung

Importieren Sie das leere Git-Projekt Ball02 in WebStorm. Laden Sie anschließend mittels `npm i` alle benötigten Node.js-Module in das Projekt.

Sie können Ihr Projekt zur Übung auch in Ihrem Git-Repository speichern. Das ist aber nicht so wichtig. Falls Sie dies machen möchten, müssen Sie es zuvor von meinem (schreibgeschützten) Repository lösen:

```
git remote remove origin
git remote add origin https://gitlab.multimedia.hs-augsburg.de:8888/BENUTZER/Ball02.git
```

2 Ziel

Ziel dieser Praktikumsaufgabe ist es, die “HTML5 Creation Engine” PixiJS v4 kennenzulernen.[1] Bei PixiJS handelt es sich um eine sehr mächtige (und damit auch sehr große) JavaScript-Bibliothek zur Realisierung von 2D-Animationen und -Spielen.

Mit HTML5 wurde das Canvas-Element (Leinwand-Element) eingeführt.[2] „Auf“ einem Canvas-Element können Grafiken gezeichnet werden. Aktuelle Browser unterstützen üblicherweise sowohl den `CanvasRenderingContext2D` zum Erstellen von 2D-Grafiken[3] als auch `WebGL` zum Erstellen von 3D-Grafiken (auf Basis von OpenGL)[4]. Die Erstellung von Grafiken mit WebGL ist wesentlich effizienter als die Erstellung von Grafiken mit dem `CanvasRenderingContext2D`, zumindest dann wenn, eine Grafikkarte zur Verfügung steht, deren Prozessor (GPU) OpenGL nativ unterstützt. Dies ist insbesondere dann von Bedeutung, wenn Animationen erstellt werden, d. h., wenn 60 mal pro Sekunde (also alle 16,7 ms) eine neue Grafik auf dem Canvas angezeigt werden soll.

PixiJS ist als Ersatz für den `CanvasRenderingContext2D` gedacht. Die Bibliothek stellt im Wesentlichen dieselben Grafik-Befehle wie der 2D-Kontext zur Verfügung, erstellt aber, wann immer möglich, die 2D-Grafiken mit Hilfe von WebGL. Nur wenn diese Schnittstelle vom Browser nicht zur Verfügung gestellt wird, erfolgt als Fallback das Rendering mit Hilfe des 2D-Kontextes. Dies hat allerdings meist einen deutlichen Performanz-Einbruch zur Folge.

3 Aufgaben

In Ihrem Projekt finden Sie wiederum mehrere Web-Anwendungen: `index01.html` verwendet die gepackte Version von `app01.js`, die ihrerseits das Spiel `game01.js` einbindet. Et cetera.

Schreiben Sie Ihre Lösungen der Aufgabe i in die Datei `game`i`.js`. Am einfachsten ist es, wenn Sie jeweils die Lösung der vorangegangenen Aufgabe kopieren und diese Kopie dann weiterentwickeln.

3.1 Aufgabe 1

Diese Aufgabe entspricht Aufgabe 1 der Praktikumsaufgabe MMProg: Praktikum: WiSe 2017/18: GameLoop01: *Lassen Sie die Eule horizontal vom linken bis zum rechten Fensterrand des Browsers fliegen.*

Im Projekt `WK_Ball02_Empty` ist eine mögliche Lösung dieser Aufgabe bereits enthalten, wobei allerdings ein Kreis „fliegt“ anstelle einer Eule. Ihre Aufgabe ist es nun, sich die Unterschiede zur Lösung der ersten Praktikumsaufgabe `WK_Ball01`, die sich durch die Verwendung von PixiJS ergeben, klar zu machen.

`app01.js`

Es wird zusätzlich die in der JavaScript-Bibliothek `pixi.js` enthaltene Klasse `Application`

importiert. Beachten Sie, dass hier diejenige Klasse der PixiJS-Bibliothek importiert wird, die auch wirklich benötigt wird. Man könnte auch `import * as PIXI from 'pixi.js';` schreiben und damit alle PixiJS-Klassen, -Funktionen und -Objekte in der Variablen `PIXI` speichern. Es ist allerdings besser, nur die benötigten Teil der PixiJS-Bibliothek einzubinden, da webpack dann die Chance hat, im Web-Ordner kleinere JavaScript-Dateien zu erzeugen. Dies kann eine deutliche Reduktion der Ladezeiten der Web-App zur Folge haben.
Anstelle des GameLoop-Objektes könnte auch der PixiJS-Ticker verwendet werden. Dieser stellt jedoch nicht sicher, dass das Model garantiert 60 mal pro Sekunde aktualisiert wird. Wenn Sie irgendwann PixiJS-Animationen einsetzen möchten, müssen Sie den PixiJS-Ticker allerdings zusätzlich starten, da ansonsten die Animationen nicht animiert werden.
Das Root-Objekt der PixiJS-Anwendung wird erstellt. Die wichtigsten Argumente sind die Breite und Höhe der Bühne sowie das Canvas-Element, auf dem die Grafiken gezeichnet werden. (Wenn man der Anwendung kein Canvas-Element zuteilt, erstellt sie selbst eines, das man anschließend per JavaScript in das HTML-Dokument einfügen muss.) Darüber hinaus gibt es diverse weitere Optionen, wie z. B. die Festlegung, ob die Leinwand transparent ist, damit der HTML-Hintergrund durchscheint.[5]
Das Spiel wird mit Hilfe einer Methode `init` initialisiert, bevor die Game Loop gestartet werden kann. Das ist notwendig, da die Grafikobjekte, die später animiert werden sollen, zunächst erstellt werden müssen. In der ersten Praktikumsaufgabe wurde diese Tätigkeit per CSS (und nicht per JavaScript) erledigt.

`game01.js`

Der Ordner `ball` wurde in `game` umbenannt und die Datei `ball01.js` in `game01.js` unbenannt, da hier nicht nur das Ball-Objekt, sondern auch alle andere Aspekte des „Spiels“ „Ein Ball bewegt sich hin und her“ definiert werden.
Es wird die in der PixiJS-Bibliothek enthaltene Klasse `Graphics` importiert, um damit eine View für das Ball-Objekt zu zeugen.
Es gibt eine Initialisierungsfunktion (Initfunktion), die zu Beginn der Anwendung aufgerufen wird, um sie zu initialisieren.
Darüber hinaus gibt es auch noch eine Resetfunktion, die jedes Mal aufgerufen wird, wenn die Anwendung neu gestartet wird, z. B. weil ein neues Spiellevel begonnen werden soll. (Diese Funktion gab es im Gegensatz zur Initfunktion auch schon in der ersten Praktikumsaufgabe.) Diese Funktion ist für wiederkehrende Initialisierungaufgaben verantwortlich, wohingegen die Initialisierungsfunktion nur ein einziges Mal zu Spielbeginn aufgerufen wird. Um Code-Duplikationen zu vermeiden, ruft die Initialisierungsfunktion auch die Resetfunktion auf.
Das Modell der Bühne (`v_stage`) wird nur teilweise initialisiert. Die Breite und Höhe wird erst später von der Initfunktion festgelegt.
Das Ballobjekt `v_ball` wird erst durch die Resetfunktion initialisiert. Diese wird erstmals von der Init-Funktion aufgerufen und kann später von der Game Loop bei Spielstart oder -neustart damit zurückgesetzt werden.
Das Ballobjekt beinhaltet schon die y-Koordinate und die y-Geschwindigkeit des Balls. In der ursprünglichen Aufgabe fehlten diese Attribute noch. Sie wurden erst in der zweiten Aufgabe („Ball vertikal fliegen lassen“) eingeführt. Außerdem wurde der Durchmesser (`d`) durch den Radius (`r`) ersetzt.
Der Ankerpunkt des Balls wurde von der linken oberen Ecke der Eule ins Zentrum des Balls verschoben. Dies hat Auswirkungen auf die Implementierung der Kollisionserkennung und -behandlung.
Die Renderfunktion unterscheidet sich leicht. In der ursprünglichen Aufgabe wurden CSS-Attribute des animierten `div`-Elements verändert. Nun müssen die Attribute des PixiJS-Grafikobjekts, das in der Init-Funktion erzeugt wurde, an die aktuellen Modellwerte angepasst werden.

Die Initfunktion nimmt folgende Aufgaben wahr:

Breite und Höhe des Bühnenmodells werden an die Breite und Höhe der PixiJS-Anwendung angepasst. Die Modelle des Spiels werden initialisiert.
Die View des Balls wird initialisiert und auf der PixiJS-Bühne platziert. Dies kann erst jetzt erfolgen, da das Application-Objekt der PixiJS-Anwendung erst jetzt zur Verfügung steht. Mit Hilfe der PixJS-Klasse `Graphics`wird zunächst ein Grafikobjekt erzeugt. Dieses wird mit Inhalt gefüllt (einem Kreis von bestimmter Farbe mit einem Rand in einer anderen Farbe) und zu guter Letzt zur PixiJS-Bühne (d. h. im Prinzip zum zugehörigen Canvaselement) hinzugefügt. Wenn sich später bestimmte Attribute des Grafikobjektes – wie z. B. die Position oder die Größe – ändern, wird die graphische Darstellung des Objektes auf dem Canvas automatisch von PixiJS angepasst. Die Änderung der Grafikattribute werden in der Funktion `render` vorgenommen. Dort wird derzeit allerdings nur die Position aus dem Model in die View übertragen. Andere Attribute der View ändern sich in dieser Anwendung nicht.

3.2 Aufgabe 2

Erstellen Sie eine erste Version der Lösung der Aufgabe 2, indem Sie die Lösung der Aufgabe 1 kopieren:

Kopieren Sie `src/js/game/game01.js` nach `src/js/game/game02.js`
Kopieren Sie `src/js/app01.js` nach `src/js/app02.js` und ersetzen Sie in dieser Datei `'./game/game01.js'` durch `'./game/game02.js'`.
Kopieren Sie `src/index01.html` nach `src/index02.html` und ersetzen Sie in dieser Datei `<title>Ball02 (app01)</title>` durch `<title>Ball02 (app02)</title>`. (Einen Verweis auf die JavaScript-Datei `app02.js` gibt es in dieser Datei nicht. Der korrekte Verweis wird von webpack automatisch beim Erstellen der Dateien des Web-Ordners injiziert.)
Starten Sie `npm run watch` neu, damit auch die neu erstellten Dateien automatisch von webpack übersetzt werden.

Lösen Sie Aufgabe 2 der ersten Praktikumsaufgabe mit Hilfe von PixiJS. Das heißt, ändern Sie Ihre Version von Aufgabe 2, die Sie gerade durch Kopieren erstellt haben so ab, dass Folgendes passiert: *Lassen Sie den Ball nicht waagerecht, sondern in der horizontalen Mitte des Browserfensters senkrecht von Fensterrand zu Fensterrand fliegen.* Anstelle einer Eule verwenden Sie bitte analog zu Aufgabe 1 einen farbigen „Ball“ mit Rand.

Vergessen Sie nicht auch die Kollisionserkennung und -behandlung für die beiden Bildschirmseiten `top` und `botton` zu implementieren.

Und denken Sie daran, `npm run dev` oder besser noch `npm run watch` zu verwenden. :-)

3.3 Aufgabe 2a

Erstellen Sie zunächst wie in Aufgabe 2 beschrieben eine Kopie der Lösung von Aufgabe 2 und nennen Sie sie 2a (`index02a.html`, `app02a.js`, `game02a.js`).

Ändern Sie nun die View des Balls: Anstelle eines farbigen Kreises stellen Sie bitte eine Eule oder einen Pinguin dar. Importieren Sie zunächst das gewünschte graphische Objekt in die Datei `game02a.js`:

```
import imgBall from '../../img/owl-150.png';
```

oder

```
import imgBall from '../../img/tux-150.png';
```

(Genau genommen weisen Sie webpack mit dieser Anweisung an, das Grafikobjekt im Ordner `web/img` zu erzeugen und die zugehörige URL in der Variablen `imgBall` zu speichern. Ohne webpack müssten Sie das Bild von Hand in diesem Ordner ablegen und dem Spriteobjekt – siehe unten – die korrekte URL übergeben.)

Darüber hinaus müssen Sie dafür sorgen, das von `pixi.js` nicht mehr die Klasse `Graphics`, sondern stattdessen die Klasse `Sprite` importiert wird.

Ändern Sie in der Initfunktion die Befehle zur Erstellung der View so ab, dass kein Kreis mehr gezeichnet, sondern das Bild der Eule oder des Pinguins dargestellt wird.

```
let
  ...
  v_ball_view = Sprite.fromImage(imgBall); // imgBall contains the URL of
the image
```

In der Initfunktion wird nur noch der Ankerpunkt in die Mitte des Viewobjekts verschoben. Sie müssen die vier Befehle zur ERzeugung des Grafikobjektes also löschen. Dieses Objekt wird dann wieder zur PixiJS-Bühne hinzugefügt.

```
v_ball_view.anchor.set(0.5); // center the anchor
p_pixi_app.stage.addChild(v_ball_view);
```

3.4 Aufgabe 3

Erstellen Sie zunächst wie in Aufgabe 2 beschrieben eine Kopie der Lösung von Aufgabe 2a und nennen Sie sie 3 (`index03.html`, `app03.js`, `game03.js`).

Lösen Sie Aufgabe 3 der ersten Praktikumsaufgabe mit Hilfe von PixiJS: *Lassen Sie der Ball von der Mitte des Browserfensters aus schräg über den Bildschirm fliegen. In x-Richtung soll er doppelt so schnell sein (200 Pixel/s)* wie in y-Richtung (100 Pixel/s). (Diesmal sollten Sie gleich eine Eule oder einen Pinguin fliegen lassen und nicht erst einen Kreis.)

3.5 Aufgabe 3a

Erstellen Sie zunächst wie in Aufgabe 2 beschrieben eine Kopie der Lösung von Aufgabe 3 und nennen Sie sie 3a (`index03a.html`, `app03a.js`, `game03a.js`).

Einen Nachteil hat die Lösung der Aufgaben 2a und 3: Das Bild des Vogels wird mittels `Sprite.fromImage` synchron vom Web-Server geladen. Das heißt, solange bis das Bild geladen wurde, ist die Anwendung „eingefroren“. Bei einem kleinen Bild ist das sicher kein Problem, wenn aber eine Vielzahl von oder große Medien geladen werden müssen – wie dies bei realen Web-Anwendungen der Fall ist –, ist dies sicher keine Option mehr.

Nehmen Sie an Ihrer Kopie 3a folgende Änderung vor: Laden Sie das Bild des Vogels asynchron mit Hilfe eines PixiJS-Loader-Objektes (MDN web doc: PIXI.loaders.Loader , GitHub: resource-loader)[6][7].

Zunächst ein wenige Theorie:

3.5.1 Asynchronität in EcmaScript 2017

Eine wesentliche Änderung, die sich wegen der Asynchronität ergibt, ist, dass der Ladevorgang später endet als die Ausführung der Initialisierungsfunktion. Wenn man direkt im Anschluss an die Ausführung der Initialisierungsfunktion die Game Loop startet, ist die Wahrscheinlichkeit groß, dass die Bilder noch gar nicht geladen wurden und die Anwendung daher abstürzt. Abhilfe schafft hier der Einsatz von modernen asynchronen JavaScript-Anweisungen:

`Promise`-Objekte (MDN web docs: Promise)
`async`-Funktionen (MDN web docs: async function)
`await`-Operator (MDN web docs: await)

Diese Konstrukte wurden eingeführt, da man in einem klassischen ES5-Programm häufig den Wald vor lauter Bäumen, d. h. den Code vor lauter Callback-Funktionen nicht mehr gesehen hat, wenn eine Vielzahl von asynchronen Anweisungen benötigt wurden.

Mit ES 2015 wurde die globale Klasse `Promise` eingeführt. Ein `Promise`-Objekt führt eine Aktion, wie das Laden eines Bildes oder das Warten auf ein Timer-Ereignis, asynchron aus. Sobald die Aktion erfolgreich beendet wird, führt sie eine Callback-Funktion aus, die ihr in der Methode `then` übergeben wurde. Der Callback-Funktion wird das Ergebnis der asynchronen Aktion als Ergebnis übergeben, so dass sie damit weiterarbeiten kann. Falls diese Funktion wieder ein `Promise`-Objekt als Ergebnis liefert, kann man den nächsten asynchronen Befehl mittels eines weiteren Aufrufs der Methode `then` anfügen. Etc. pp. Auf diese Weise konnte man eine Abfolge von asynchronen Anweisungen zumindest linear notieren:

```
myPromise
  .then(function(p_value) { ...})
  .then(function(p_value) { ...})
  .then(function(p_value) { ...})
```

Wenn man die kürzere Arrow-Schreibweise für Funktionen verwendet, die mit ES 2015 eingeführt wurde (MDN web doc: Pfeilfunktionen), wird der oder noch etwas kompakter und damit lesbarer:

```
myPromise
  .then(p_value => { ...})
  .then(p_value => { ...})
  .then(p_value => { ...})
```

Allerdings ist das auch nicht sonderlich schön zu lesen, vor allem wenn man umfangreiche asynchrone

Funktionen zu definieren hat. Und jede dieser Funktionen muss ein eigenes `Promise`-Objekt erstellen, was den Code auch nicht gerade lesbarer macht.

Daher wurden in ES 2017 die asynchronen Funktionen eingeführt:

```
async function myFunction(...)
{ ...}
```

In dieser Funktionen werden Anweisungen geschrieben, wie in jeder anderen Funktion auch. Und diese Anweisungen werden der Reihe nach abgearbeitet. Allerdings gibt es einen fundamentalen Unterschied. Der Funktionsrumpf wird asynchron ausgeführt. Das Ergebnis der Funktion ist ein (implizites) `Promise`-Objekt, dessen `then`-Funktion erst aufgerufen wird, sobald alle asynchronen Befehle im Rumpf der Funktion erfolgreich beendet wurden.

Da heißt, bei Aufruf der Funktion wird diese sofort wieder beendet. Der Browser wird durch die Funktion nicht blockiert, wie es z. B. beim synchronen Laden eines Bildes der Fall wäre. Die im Rumpf der Funktion enthaltenen Befehle werden trotzdem der Reihe nach ausgeführt, aber asynchron.

Wenn Sie die Funktion `init` in der Datei `app01.js` näher betrachten, fällt Ihnen auf, das sie als `async`-Funktion definiert wurde. In ihrem Rumpf befinden sich neben drei Standard-Anweisungen auch die Anweisung `await wait(500);`. Die Funktion `wait` wurde von mir in der Datei `src/lib/wk/util/wait.js` definiert:

```
function wait(p_time)
{ return new Promise(function(p_resolve)
                     { setTimeout(p_resolve, p_time); }
                    )
}
```

(In Wirklichkeit habe ich die modernere Arrow-Schreibweise verwendet, aber das Ergebnis ist dasselbe.)

Die Funktion `wait` erstellt ein `Promise`-Objekt, das die Funktion `setTimeout` ausführt ([https://developer.mozilla.org/de/docs/Web/API/WindowTimers/setTimeout MDN web doc: WindowTimers.setTimeout()]). Diese Funktion wartet eine gewisse Zeit `p_time` und ruft dann eine Callback-Funktion auf. Das ist die typische ES5-Vorgehensweise zur Behandlung von asynchronen Aktionen. In diesem Fall wird die Callback-Funktion `p_resolve` aufgerufen. Das ist eine Funktion, die dem `Promise`-Objekt mitteilt, dass die asynchrone Aktion erfolgreich beendet wurde (und das `Promise`-Objekt nun die `then`-Funktion ausführen könnte).

Seit ES 2017 kann man in `async`-Funktionen den `await`-Operator verwenden, um das Ergebnis eines `Promise`-Objekts zu verarbeiten. Das ist **viel** eleganter, als mit den (auch schon wieder veralteten) `then`-Funktionen. Sie schreiben einfach den Befehl `await wait(500);` in den Rumpf Ihrer `async`-Funktion. Dann wird die (asynchrone) Verarbeitung des Rumpfes dieser funktion um 500 ms (d. h. um eine halbe Sekunde) unterbrochen, bevor der nächste Befehl abgearbeitet wird.

3.5.2 Asynchrones Laden von Bildern für PixiJS

PixiJS stellt das Object `PIXI.loader` zur Verfügung (PixiJS API Documentation: PIXI.loaders.Loader), mit dem Sie Bilder (oder auch andere Medien) asynchron laden können. Die Bilder werden dabei

gleich so gespeichert, dass sie problemlos von PixiJS weiterverarbeitet werden können.

Importieren Sie diesen Loader in Ihre Datei `app03.js`. Ersetzen Sie

```
import {Application} from 'pixi.js';
```

durch

```
import {loader, Application} from 'pixi.js';
```

Fügen Sie außerdem folgenden Import-Befehl in die Datei `app03.js` ein:

```
import imgBall from '/img/tux-150.png';
```

Diesen Befehl kennen Sie schon. Er befindet sich noch in der Datei `game03a.js`. Dort sollten Sie ihn löschen (In der Datei `game03.js` dürfen sie ihn dagegen nicht löschen.) Wie Sie wissen, bewirkt dieser Import-Befehl, dass die URL des Bildes `/img/tux-150.png` in der Variablen `imgBall` gespeichert ist (und das Bild in den Web-Ordner kopiert wird).

Fügen Sie nun vor der Initfunktion den folgenden Befehl ein.

```
loader.add('imgBall', imgBall);
```

Damit teilen Sie dem Loader mit, welches Bild er später asynchron laden soll. Als Argumente erwartet einen Namen, mit dem später auf das Bild zugegriffen werden kann, sowie die URL des Bildes, under der er es im Web-Auftritt findet. Es ist durchaus üblich, eine Vielzahl von Medien auf einmal (d. h. parallel) zu laden:

```
p_pixi.loader
  .add(...)
  .add(...)
  .add(...)
  .add(...)...
```

Jetzt könnten Sie die Bilder mittels `loader.load(geeigneteCallbackFunktion)` asynchron laden. Da PixiJS keine Proimises unterstützt, müsste man hier auf die alte ES-5-Methode zurückgreifen.

Unter `/wk_pixi/loader/asyncLoader` finden Sie eine kleine Funktion, die das eleganter mit Hilfe eines `Promise`-Objektes macht. Importieren Sie diese Funktion:

```
import loadResources from '/wk_pixi/loader/asyncLoader';
```

Jetzt können Sie die Ressourcen ganz elegant mit `await` laden. Fügen Sie folgenden Code als erste Zeile in den Rumpf der Initfunktion ein:

```
const c_resources = await loadResources();
```

Damit wird die asynchrone Ausführung der Initfunktion solange unterbrochen, bis alle Bilder geladen wurden. Danach wird der nächste Befehl ausgeführt. Das ist der Befehl, der das Spiel initialisiert:

```
game.init(c_pixi_app);
```

Ersetzen Sie ihn durch:

```
game.init(c_pixi_app, c_resources);
```

Das heißt, übergeben Sie der Initfunktion nicht nur das PixiJS-App-Objekt, sondern auch die Ressourcen (das Bild), das Sie geladen haben. Jetzt müssen Sie nur noch die Initfunktion in der Datei `game03a.js` anpassen.

Ersetzen Sie dort in der `let`-Anweisung

```
v_ball = {},
v_ball_view =  ...;
```

durch

```
v_ball = {},
v_ball_view;
```

Das heißt, das Ball-View-Objekt wird nicht mehr hier erzeugt, sondern erst in der Initfunktion. Fügen Sie zu diesem Zweck den Parameter `p_resources` in die Parameterliste der Initfunktion ein:

```
function init(p_pixi_app, p_resources)
```

In diesem Parameter werden der Funktion die Ressourcen (Bilder) übergeben, die Sie zuvor asynchron) geladen haben. Damit können Sie jetzt die View erzeugen.

Fügen Sie vor die beiden Befehle

```
v_ball_view.anchor.set(0.5),
p_pixi_app.stage.addChild(v_ball_view);
```

den Befehl

```
v_ball_view = new Sprite(p_resources['imgBall'].texture);
```

Es wird, wie auch schon in den Aufgaben 2a und 3, ein `Sprite`-Objekt erstellt. Diesmal wird jedoch das Bild nicht aus einer Datei gelesen (`Sprite.fromImage...`)) sondern aus dem vom Loader erstellten Ressourcen-Objekt unter dem Namen `'imgBall'` extrahiert. Warum man nicht direkt das Objekt `p_resources['imgBall']` verwenden kann, sondern dessen Textur verwenden muss, ist eines von den unergründlichen Geheimnissen von PixiJS.

BTW: Man bräuchte das Objekt `p_resources` gar nicht als Argument an die Initfunktion übergeben. Man könnte auch in der Datei game3a.js, auch wieder den Loader importieren

```
import {loader, Sprite} from 'pixi.js';
```

und dann mittels `load.resources` auf dieses Objekt zugreifen. Allerdings ist die Kommunikation zweier Module über ein globales Objekt (wie das Objekt `loader`) *deprecated*, da man nicht weiß, wann und ob überhaupt das darin enthaltene Ressourcenobjekt sauber initialisiert wurde. Eine direkte Kommunikation mittels Parametern und Argumenten ist dem vorzuziehen.

3.6 Aufgabe 3b

Erstellen Sie zunächst wie in Aufgabe 2 beschrieben eine Kopie der Lösung von Aufgabe 3a und nennen Sie sie 3b (`index03b.html`, `app03b.js`, `game03b.js`).

Erweitern Sie Ihre Lösung von Aufgabe 3 so, dass 5 unterschiedliche Vögel auf unterschiedlichen Routen über die Bühne fliegen. (Hier benötigen Sie Ihr Wissen über den Umgang mit Arrays: siehe Praktikumsaufgaben EcmaScript01 und EcmaScript02.)

Gehen Sie dazu folgendermaßen vor:

Laden Sie in der Datei `app03b.js` die beiden Bilder `/img/owl-150.png` und `/img/owl-150.png` asynchron mittels des PixiJS-Loaders.
Bearbeiten Sie nun die Datei `game03b.js`*
Legen Sie in einer Variablen oder Konstanten `v_ball_nr` die Anzahl der Vögel fest, die fliegen sollen. Initialisieren Sie diese mit der Zahl `5`.
Ersetzen Sie `v_ball` und `v_ball_view` durch `v_balls` und `v_ball_views`. Diese Variablen sollen jeweils ein Array mit `v_ball_nr` Model-Objekten bzw. `v_ball_nr` zugehörigen View-Objekten beinhalten. Die Variable `v_ball_views` wird in der Initfunktion initialisiert und die Variable `v_balls` in der Resetfunktion.
Initialisieren Sie das Array `v_ball_views` in der Initfunktion mit Hilfe einer Schleife und des Push-Befehls. Sie müssen `v_ball_nr` Sprite-Objekte erzeugen und ins Array einfügen. Als Bild wählen Sie per Zufallsentscheidung (`Math.random() < 0.5`) jeweils die Eule oder Tux.
Initialisieren Sie das Array `v_balls` in der Resetfunktion mit Hilfe einer Schleife und des Push-Befehls. Der Radius aller fünf Bälle sei 75. Die Position wählen Sie zufällig `Math.random()` innerhalb der Bühne, die Geschwindigkeit in x- und y-Richtung ebenfalls zufällig jeweils zwischen 100 und 400.
Sorgen Sie in der Updatefunktion mit Hilfe einer Schleife dafür, dass all 5 Objekte im Array `v_balls` aktualisiert werden.

Sorgen Sie in der Renderfunktion mit Hilfe einer Schleife dafür, dass all 5 Objekte im Array `v_ball_views` rerendert werden.

3.7 Aufgaben 4 bis 10

Nun können und sollten Sie zur Übung die Aufgaben 4 bis 10 der ersten Praktikumsaufgabe ebenfalls mit Hilfe von PixiJS lösen. Laden Sie dabei die Bilder wieder asynchron und verwenden Sie JSON zur Konfiguration. Sie sollten ruhig auch mal mehr als einen Vogel animieren, z. B. indem Sie mehrere Eulen hintereinander die Wand entlang fliegen lassen.

4 Quellen

1. PixiJS v4: The HTML5 Creation Engine
2. MDN web docs: Canvas API
3. MDN web docs: CanvasRenderingContext2D
4. MDN web docs: Einführung in WebGL
5. PixiJS API Documentation: PIXI.Application
6. PixiJS API Documentation: PIXI.loaders.Loader
7. [https://github.com/englercj/resource-loader GitHub: resource-loader]

1. **Kowarschick (MMProg)**: Wolfgang Kowarschick; Vorlesung „Multimedia-Programmierung“; Hochschule: Hochschule Augsburg; Adresse: Augsburg; Web-Link; 2018; Quellengüte: 3 (Vorlesung)