MMProg: Praktikum: WiSe 2018/19: Ball03

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 Ziel

Ziel dieser Praktikumsaufgabe ist es, Lösungen des zweiten Teils des Tutoriums zu modularisieren. Die Modularisierung hat mehrere Vorteile:

Das Prinzip „Don't repeat yourself“ (DRY) wird unterstützt. Um dies zu erreichen, muss sichergestellt werden, dass viele Module in diversen Projekten wiederverwendet werden können.[1][2]
Mehrere Programmierer können gleichzeitig an einem Projekt arbeiten. Hier muss sichergestellt sein, dass möglichst saubere Schnittstellen definiert werden, damit ein Programmierer nicht ständig an die

Änderungen eines anderen Programmierers anpassen muss. (Die Definition von Schnittstellen ist eine der Kernaufgaben von Informatikern. Dabei handelt es sich um einen kreativen Prozess. Der Programmierer hingegen braucht „lediglich“ die Spezifikation umzusetzen. Das ist i. Allg. deutlich weniger kreativ.)
Jede Änderung an einem vorhandenen Code kann Fehler zur Fogle haben. Daher ist es von Vorteil, wenn ausgetestete, wiederverwendbare Module bestehen. Bei der Fehlersuche befindet sich der Code i. Allg. nicht in einem dieser Module, sondern im neu erstellten Code. Das vereinfacht die Fehlersuche deutlich.

2 Vorbereitung

Importieren Sie das leere Git-Projekt Ball03 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 das 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/BENUTZER/Ball03.git
```

3 Aufgaben

In Ihrem Projekt finden Sie drei Web-Anwendungen: `src/index01.html`, `src/index02.html` und `src/index03.html`. Allerdings existiert derzeit nur der JavaScript-Ordner `src/js/app01` für die Web-App `index01.html`. Die fehlenden JavaScript-Ordner `src/js/app02` und `src/js/app03` werden erst zu einem späteren Zeitpunt erstellt.

Im Ordner `src/js/app01` befinden sich derzeit zwei Dateien: `app.js` und `game.js`. Diese Dateien enthalten eine App, bei der sich ein Ball schräg über die Bühne bewegt. Diese App ist eine Mischung der Musterlösungen von Teil 2 des Ball-Tutoriums: Bei der View des Balls wurde die `Graphics`-Version aus der ersten Teilaufgabe dieser Tutoriumsaufgabe genommen, die Update-Funktionalität und die Kollisionserkennung entstammt dagegen dem vierten Teil.

Beachten Sie, dass sich die Ordnerstruktur gegenüber Teil 2 des Tutoriums geändert hat. Im Ordner `src/js` gibt es für jede Web-App einen Ordner mit Namen `appXY`, in dem eine Datei `app.js` enthalten sein muss, die die App initialisiert. Es können im selben Ordner beliebig viele weitere Dateien und Unterordner enthalten sein, die von `app.js` direkt oder indirekt importiert werden. Die Umstrukturierung ist sinnvoll, da ab sofort eine Web-App aus deutlich mehr als drei Dateien bestehen wird.

Zu jedem Web-App-Ordner `src/js/appXY` muss es eine zugehörige Datei `src/indexXY.html` geben.

Anmerkung: Die Datei `webpack.config.js` wurde speziell für diese Struktur entwickelt. Die im Teil 2 enthaltene Datei gleichen Namens ist damit nicht kompatibel.

3.1 Aufgabe 0: Analyse von `app01`

Es gibt in `WK_Ball03` eine `app01` mit zwei Modulen `app` und `game`, die in zwei Dateien `app.js` und `game.js` implementiert wurden.

Sehen Sie sich zunächst diese App an und analysieren Sie, wie diese funktioniert. Beachten Sie insbesondere Folgendes:

Es wird ein Ball-Objekt erzeugt, mit Attributen „Radius“, „Position“ und „Geschwindigkeit“.
Dieser Ball wird durch einen einfarbigen Kreis mit Rand visualisiert.
Der Ball bewegt sich gemäß seinen Initialparametern schräg über die Bühne.
Bei einer Kollision mit dem Bühnenrand ändert er seine Bewegungsrichtung.

Die gesamte beschriebene Spiellogik ist in einer Datei enthalten: `game.js`. So eine Datei bezeichne ich als **Moloch**. Das Prinzip „Implementiere keinen Moloch“ nennt man fachsprachlich „Modularisierung“.

Initiales Moduldiagramm von `app01`

Eigentlich ist das Programm gar nicht so molochartig, wie es zunächst scheint. Es kommen einige Module zum Einsatz:

`index01.html`: Eine HTML-Seite zum Starten der eigentlichen Web-Anwendung, sobald sie vom Browser geladen wird.
`head.css`: Eine CSS-Datei, die das Layout der HTML-Datei festlegt (insbesondere die Hintergrundfarbe), solange die App geladen wird. Diese Datei wird mittels eines `link`-Elements direkt von `index01.html` eingebunden. Mittels webpack wird der Inhalt dieser Datei komprimiert und in die zugehörige HTML-Datei injiziert, indem das `link`-Element durch ein `script`-Element ersetzt wird.
`app.css`: Eine CSS-Datei, die das Layout der Web-App festlegt. Diese Datei wird von `app.js` importiert. Mittels webpack wird dafür gesorgt, dass der JavaScript-Code von `app.js` eine komprimierte Version vom Inhalt der CSS-Datei in `index01.html` injiziert wird, sobald die Web-App vollständig geladen wurde.
`GameLoop`: Eine Game-Loop-Klasse, die Ihnen im Rahmen des Praktikums zur Verfügung gestellt wird. Dieses Modul benutzt weitere Module der WK-Bibliothek: `Automaton` und `EventDispatcher`.
`wait`: Eine asynchrone Funktion, die in der WK-Biliothek bereitsteht, um eine gewisse Zeitspanne zu warten, bevor eine Aktion durchgeführt wird. Hier wird sie verwendet, um den Start der App ein paar Millisekunden zu verzögern, nachdem die App vollständig geladen und sichbar gemacht wurde.
PixiJS: Eine sehr mächtige 2D-Grafik-Bibliothek, die sehr modular aufgebaut ist. Aus dieser Biliothek werden zwei Klassen verwendet (auch diese verwenden – wie die `GameLoop` – zahlreiche weitere Module, um ihre Aufgaben zu erfüllen; es würde allerdings das Diagramm vollkommen unlesbar machen, alle diese Module hier aufzuführen):

- `PIXI.Application`: Die PixiJS-Root-Klasse. Das zugehörige Objekt enthält alle Elemente, um graphische Elemente effizient auf einem HTML-Canvas-Element darzustellen.
- `PIXI.Graphics`: Eine PixiJS-Klasse zum Zeichnen einfacher geometrischer Formen wie Kreisen, Rechtecken, Linien etc.

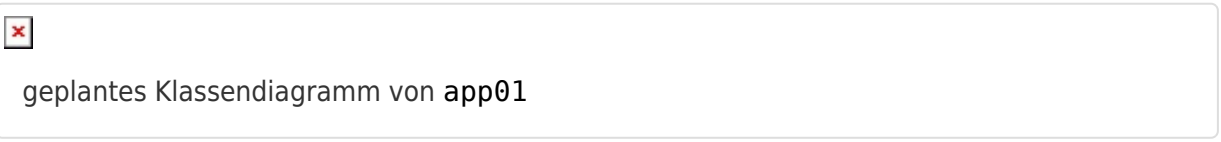
`app`: Das Hauptmodul. Es stellt mit Hilfe der zuvor genannten Module die Spielumgebung zur Verfügung: Eine PixiJS-Bühne, eine Game Loop sowie eine CSS-Datei. Sobald die Umgebung erstellt und initialisiert wurde, startet diese Modul leicht verzögert die Update- und die Render-Funktion des eigentlichen Spielmoduls `game`.
`game`: Das immer noch molochartige Spielmodul, das im Rahmen des Praktikums in diverse

Einzelmodule aufgeteilt werden wird.

3.2 Aufgabe 1: Modularisierung ders Game-Moduls

(Musterlösung: Gitlab: `WK_Ball03`, `app01`; unter `app01a` finden Sie eine Variante, in der das Update- und das Render-Modul nicht als Funktionsmodule, sondern als Klassenmodule realisiert wurden)

Ein wichtiges Programmierprinzip besagt, dass jedes Modul nur eine Aufgabe erledigen soll. Bislang ist das Game-Modul in dieser Hinsicht noch ziemlich schlecht, da es zahlreiche Aufgabe erfüllt. Dies sollen Sie ändern.



geplantes Klassendiagramm von `app01`

Als Ersatz für das Game-Modul sollen insgesamt sechs Komponenten erstellt werden „Bühne (Model)“, „Ball (Model)“, „Ball (View)“, „Kollisionserkennung und -behandlung (Model)“, „Update-Funktion (Model)“ und „Render-Funktion (View)“. Oben sehen Sie das zugehörige Klassendiagramm, bestehend aus insgesamt sieben Modulen:

Modul `app`: Zuständig für die Initialisierung der Spielumgebung und das anschließende Starten des Spiels. Dieses Modul besteht aus einer Folge von Anweisungen, die direkt beim Start der Web-App ausgeführt werden.
Klasse `ModelStage`: Jedes Objekt dieser Klasse repräsentiert das Modell einer Spielbühne. Üblicherweise gibt es nur eine Spielbühne. Es gibt aber auch Spielsituationen mit mehreren Spielbühnen (z. B. wenn neben der Hauptbühne, die nur einen Ausschnitt der Spielwelt zeigt, eine Minimap existiert, die einen Überblick auf die Spielwelt gewährt). Diese Klasse kann in vielen Anwendungen wiederverwendet werden. Im Laufe der Zeit werden allerdings sicher noch weitere Attribute und Methoden ergänzt werden.
Klasse `ModelCircle`: Eine ebenfalls sehr gut wiederverwendbare Klasse, die für kreisförmige Objekte aller Art in einem Spiel zum Einsatz kommen kann. Normalerweise gibt es zahlreiche kreisförmige Objekte in einem Spiel. Auch hier wird es im Laufe der Zeit sicher diverse Erweiterungen geben.
Funktion `collisionCircleStage`: Ein Modul, das eine Funktion zur Kollisionserkennung und -behandlung von (beweglichen) Kreisobjekten mit (unbeweglichen) Rändern der Bühne zur Verfügung stellt. Auch dieses Modul kann sehr gut wiederverwendet werden. Allerdings gibt es (wie bei allen Fragen der Kollisionserkennung und -behandlung) noch zahlreiches Verbesserungspotential.
Funktionssammlung `update`: Dieses Modul enthält zwei Funktionen: `initUpdater` und `update`. Die erste Funktion dient dazu, die Update-Funktion zu initialisieren, d. h., ihr die Objekte bekannt zu geben, die sie regelmäßig aktualisieren soll. Die zweite Funktion wird der Game Loop als Callback-Funktion übergeben, um diese Aktualisierungen regelmäßig (z. B. genau sechzig mal pro Sekunde) durchzuführen. Außerdem ist die Update-Funktion dafür zuständig, die Kollisionserkennung und -behandlung zu initiieren. Sie selbst führt diese Aufgabe allerdings nicht durch, sondern überträgt sie an geeignete Hilfsmodule (in diesem Fall das Modul `collisionCircleStage`). Das Update-Modul kann meist nicht problemlos wiederverwendet werden, da jede Web-App ihre ganz eigene Objektwelt verwaltet.
Klasse `ViewCircle`: Eine sehr gut wiederverwendbare Klasse, die die kreisförmigen Objekte visualisieren soll. In Aufgabe 1 wird der Ball mit Hilfe von PixiJS-Graphics-Befehlen gezeichnet (weshalb die Klasse vielleicht besser `ViewCircleGraphics` heißen sollte). In einer späteren

Aufgabe sollen sie die Visualisierung mit Hilfe von Bildern unter Zuhilfenahme der PixiJS-Klasse `Sprite` realisieren. (Diese Klasse sollte daher eventuell `ViewCircleSprite` genannt werden.) Funktionssammlung `render`: Dieses Modul enthält zwei Funktionen: `initRenderer` und `render`. Die erste Funktion dient dazu, die Render-Funktion zu initialisieren, d. h., ihr die Objekte bekannt zu geben, deren Visualisierung sie regelmäßig aktualisieren soll. Die zweite Funktion wird der Game Loop als Callback-Funktion übergeben, um diese Aktualisierungen regelmäßig (möglichst sechzig mal pro Sekunde) durchzuführen. Im Gegensatz zur Update-Funktion ist diese Funktion sehr gut wiederverwendbar. Ihre einzige Aufgabe ist es, für alle View-Objekte, die in einen Array enthalten sind (das ihr bei Initialisierung übergeben wurde), jeweils die Update-Funktion aufzurufen, die jedes View-Objekt bereitstellen muss.

3.2.1 Erstellen der Modul-Struktur

Legen Sie zunächst zwei Ordner an:

`src/js/app01/model`
`src/js/app01/view`

Erzeugen Sie dann innerhalb des Ordners `src/js/app01/` für jedes Modul eine leere EcmaScript-Datei:

`app.js` (diese Datei gibt es schon, sie wird im Folgenden allerdings modifiziert.
`model/ModelStage.js`
`model/ModelCircle.js`
`model/collisionCircleStage.js`
`model/update.js`
`view/ViewCircle.js`
`view/render.js`

Die Idee, jedes Modul in einer eigenen Datei zu platzieren, geht auf Java zurück. Im Gegensatz zu anderen Sprachen erzwingt Java, dass jede Klasse in eine eigene Datei geschrieben werden muss (vgl. Why is each public class in a separate file?). Außerdem gibt es in Java nur eine Art von Modulen: Klassen. Beide Restriktionen gelten für JavaScript nicht. Sie sollten sich aber angewöhnen, zumindest die Regel „ein Modul === eine Datei mit demselben Namen plus der Endung `.js`“ zu beachten.

Fügen Sie in jede Datei einen geeigneten Rahmencode ein. In ein Modul, das eine Klasse enthält (siehe obiges Diagramm), fügen Sie bitte folgenden Code ein, **wobei Sie jeweils `CLASSNAME` durch den Klassenname ersetzen müssen**.

```
class CLASSNAME {
  constructor() {
  }
}

export default CLASSNAME;
```

oder auch (je nachdem, welche Klammerstruktur sie bevorzugen; bei der folgenden Struktor müssen Sie den Code allerdings `Strg-Shift-v` → `Paste without Formating` einfügen, damit sie erhalten bleibt)

```
class CLASSNAME
{ constructor()
  {
  }
}

export default CLASSNAME;
```

Für Funktionsmodule, die mehr als eine Funktion exportieren (hier sind das die Module `update` und `render`; siehe oben), sollten Sie folgenden Code einfügen (wobei Sie die Funktionsnamen natürlich geeignet ersetzen müssen):

```
function FUNKTIONSNAME_1()
{
}

function FUNKTIONSNAME_2()
{
}

export { FUNKTIONSNAME_1, FUNKTIONSNAME_2 };
```

Beachten Sie, dass Sie später diese Funktionen (z. B. in der Datei `app.js`) mittels

```
import { FUNKTIONSNAME_1, FUNKTIONSNAME_2 } from './ORDNER/MODULNAME.js';
```

importieren können.

Für Funktionsmodule, die nur eine Funktion exportieren (hier ist das das Modul `collisionCircleStage`), sollten Sie folgenden Code einfügen (wobei Sie den Funktionsnamen natürlich ebenfalls geeignet ersetzen müssen):

```
function FUNKTIONSNAME()
{
}

export default FUNKTIONSNAME;
```

Da hier die Funktion als Default-Objekt exportiert wird, können Sie diese Funktion später mittels

```
import FUNKTIONSNAME from './ORDNER/MODULNAME.js';
```

importieren. Hier können bzw. müssen Sie beim Importieren also auf die geschweiften Klammern

verzichten. Sollten Sie die geschweiften Klammern unbedingt verwenden wollen, müssten Sie die Exportanweisung anlog zum Modul mit mehreren Funktionsobjekten definieren.

Starten Sie nun `npm run watch` und überprüfen Sie , ob die Web-App `app01` ausführbar ist. Das sollte auf jeden Fall funktionieren, da Sie ja an der App (d. h. an den Dateien `app.js` und `game.js` bislang gar nichts geändert haben.

Importieren Sie nun die fünf Module, auf die das Modul `app` im obigen Diagramm verweist, in die Datei `app.js`. Beachten Sie, dass die Syntax der Importanweisungen von den Exportanweisungen abhängen, die im jeweiligen Modul verwendet wurde (siehe Anmerkungen zum Funktionscode zuvor).

Wenn Sie keine syntaktischen Fehler gemacht haben, sollte webpack die App immer noch fehlerfrei übersetzen und ausführen können. WebStorm warnt Sie allerdings bei den Import-Anweisungen, dass Sie Objekte improtieren, die Sie gar nicht verwenden. Das stimmt derzeit ja auch, also ist alles in Ordnung.

3.2.2 Implementierung der Komponenten

3.2.2.1 `ModelStage`

Attribute und Methoden der Klasse `ModelStage`

Implementieren Sie zunächst die Klasse `StageModel`. Im UML-Klassendiagramm sehen Sie, dass jedes Objekt dieser Klasse vier Attribute hat: `left`, `right`, `top`, `bottom`. Es sollen jeweils Zahlen darin gespeichert werden. Allerdings ist JavaScript untypisiert, so dass Sie diese Bedingungen höchstens in JSDoc-Kommentaren formulieren können. Erschwerend kommt hinzu, das man in EcmaScript-Klassen derzeit Attribute nur mit Hilfe von Getter- und Setter-Methoden definieren kann. Direkt kann man sie zurzeit nur innerhalb des Konstruktors erzeugen.

Also implementieren Sie nur den Konstruktor:

```
constructor({left:   p_left   = 0,
             right:  p_right  = 0,
             top:    p_top    = 0,
             bottom: p_bottom = 0
            })
{ this.left   = p_left;
  this.right  = p_right;
  this.top    = p_top;
  this.bottom = p_bottom;
}
```

Hier kommt eine Parametersyntax zum Einsatz, die in EcmaScript 6 unter dem Namen Destructuring eingeführt wurde.

In EcmaScript 5 hätten Sie das noch folgendermaßen schreiben müssen:

```
constructor(p_config)
{ this.left   = p_config.left    == null ? 0 : p_config.left;
  this.right  = p_config.right   == null ? 0 : p_config.right;
  this.top    = p_config.top     == null ? 0 : p_config.top;
  this.bottom = p_config.bottom  == null ? 0 : p_config.bottom;
}
```

In beiden Fällen erwartet der Konstruktor ein Objekt als Argument, das die Initialwerte für die vier Attribute enthält. Sollte ein Wert nicht angegeben werden, so wird er mit dem Defaultwert `0` initialisiert.

Probieren Sie aus, ob ihr Code funktioniert. Fügen Sie in die Datei `app.js` folgende Befehle ein und sehen Sie nach, was im Konsolfenster des Browsers ausgegeben wird, wenn Sie die Web-App starten.

```
const
  c_model_stage =
    new ModelStage
        ({ "left":   0,
           "right":  document.documentElement.clientWidth,
           "top":    0,
           "bottom": document.documentElement.clientHeight
        });

console.log(c_model_stage);
```

Den `console.log`-Befehl sollten Sie anschließend wieder löschen, der wird nicht mehr benötigt. Die Konstante sollte dagegen bestehen bleiben.

Übrigens, mittels geeigneten JSDoc-Kommentare, können Sie zumindest dokumentiren, dass in den Attributen eines Stageobjektes nur Zahlen gespeichert werden dürfen:

Folgender Kommentar sollte vor der Definition der Klasse stehen.

```
/**
 * @class ModelView
 *
 * @property {number} left   - x position of the left border
 * @property {number} right  - x position of the right border
 * @property {number} top    - y position of the top border
 * @property {number} bottom - y position of the bottom border
 */
```

Und folgender Kommentar sollte vor der Definition des Konstruktors stehen:

```
/**
 * @param {Object} p_config
 * @param {number} [p_config.left = 0]
 * @param {number} [p_config.right = 0]
 * @param {number} [p_config.top = 0]
 * @param {number} [p_config.bottom = 0]
 */
```

3.2.2.2 `ModelCircle`

Attribute und Methoden der Klasse `ModelCircle`

Implementieren Sie die Klasse `ModelCircle` analog zur Klasse `ModelStage`.

Im UML-Diagramm dieser Klasse wird allerdings noch die Methode `update` mit dem Inputparameter `p_delta_s` aufgeführt. Implementieren Sie diese Methode ebenfalls. Im Rumpf der Methode müssen Sie die ersten beiden Zeilen der Funktion `update` einfügen, die Sie in der Datei `game.js` vorfinden und den Bezeichner `v_ball` geeignet umbenennen.

Testen Sie diese Klasse, indem Sie folgende Konstante in die Datei `app.js` einfügen:

```
const
  c_model_ball =
    new ModelCircle
    ({ "r":   75,
       "x":   75,
       "y":   75,
       "vx": 400,
       "vy": 300
    });
```

Fügen Sie anschließend folgende Befehle in `app.js` ein und überprüfen Sie im Konsolfenster des Browsers, ob alles funktioniert.

```
console.log(c_model_ball);
c_model_ball.update(1/60);
console.log(c_model_ball);
```

Diese drei Testbefehle sollten Sie anschließend wieder löschen.

3.2.2.3 `collisionCircleStage`

Funktion `collisionCircleStage`

In die Funktion `collisionCircleStage` in der Datei `collisionCircleStage.js` fügen Sie

zunächst die beiden Parameter ein, die Sie im obigen Diagramm vorfinden. In den Rumpf fügen Sie die restlichen Befehle aus der Funktion `update` der Datei `game.js` ein. Das sind die Befehle zur Kollisionserkennung und -behandlung. Beachten Sie, dass die Objekte nicht mehr `v_ball` und `v_stage` heißen, sondern `p_circle` und `p_stage`.

Diese Funktion wird nicht sofort, sondern später gemeinsam mit den Funktionen des Updatemoduls getestet.

3.2.2.4 `update`

Funktionen des Moduls `update`

In diesem Modul müssen Sie zwei Funktionen implementieren.

Zunächst sollten Sie aber das Module `collisionCircleStage` importieren. Danach sollten Sie **vor** den beiden Funktionen zwei Variablen definieren, in denen die Objekte gespeichert werden, auf die regelmäßig von der Update-Funktion zugegriffen werden muss:

```
let v_circle, v_stage;
```

Nun müssen Sie dafür sorgen, dass die Funktion `initUpdater` die Werte, die ihr in den Parametern `p_circle` und `p_stage` übergeben werden, in diesen beiden Variablen gespeichert werden.

Als letztes implementieren Sie die Funktion `update`. Diese muss die Position des Kreisobjekts `v_circle` mittels der Updatemethode der Klasse `ModelCircle` aktualisieren. Natürlich muss dabei das Argument, das ihr im Parameter `p_delta_s` übergeben wurde, an die Update-Methode der Klasse `ModelCircle` weitergeleitet werden.

Im zweiten Schritt muss sie die zuvor importierte Kollisionsfunktion für die beiden Objekte `v_circle` und `v_stage` ausführen.

Wenn Sie die Klasse implementiert haben, können Sie in der Datei `app.js` im Anschluss an die Konstantendefinitionen schon mal die Update-Loop initialisieren.

```
initUpdater({stage: c_model_stage, circle: c_model_ball});
```

Damit ändert sich zunächst einmal nichts, da die Update-Loop noch nicht (in der Game Loop) gestartet wird. Aber sie sehen, ob Sie irgenwelche Syntaxfehler gemacht haben, die entweder WebStorm oder webpack oder die Browser-Konsole meldet.

3.2.2.5 `ViewCircle`

Attribute und Methoden der Klasse `ViewCircle`

Die Aufgabe der Objekte dieser Klasse ist es, Kreise mittels der `Graphics`-Klasse von PixiJS zu viausalisieren. Daher müssen Sie zunächst diese Klasse importieren:

```
import {Graphics}  from 'pixi.js';
import stringToHex from '/wk_pixi/util/stringToHex';
```

Wie Sie sehen, wird auch noch eine Utility-Funktion `stringToHex` importiert, die Hex-Strings der Art `'#FFAA00'` in Integerzahlen umwandelt. Dabei entspricht der Hexwert der resultierende Integerzahl dem Hex-String. Für das obige Beispiel ergäbe sich der Wert `0xFFAA00`. Diese Übersetzung ist notwendig, da PixiJS nur mit Integerzahlen umgehen kann, aber in JSON-Dateien, die künftig zur Initialisierung von Modulen eingesetzt werden sollen, keine Integerzahlen im Hex-Format gespeichert werden können.

Der Konstruktor der Klasse `ViewCircle` muss ein `Graphics`-Objekt erzeugen, initialisieren und im Attribut `this.sprite` speichern. Die Erzeugung und Initialisierung erfolgt analog zur Erzeugung der View in der Funktion `init` der Datei `game.js`. Beachten Sie jedoch: In der Datei `game.js` werden beim Aufruf der `Graphics`-Methoden konstante Werte wie die Zahl `5` oder der Wert `0xFFAA00` verwendet. Hier müssen Sie natürlich auf die Parameter des Konstruktors zurückgreifen. (Und denken Sie daran, dass Sie Hex-Strings mittels der Funktion `stringToHex` in Integerzahlen verwanden müssen.)

Das neu erzeugte `Graphics`-Objekt muss anschließend im Attribut des PixiJS-`Application`-Objekts `p_pixi_app` als Kind-Objekt gespeichert werden:

```
p_pixi_app.stage.addChild(this.sprite);
```

Zu guter Letzt (oder auch ganz zu Beginn), muss das dem Konstruktor übergebene Model-Objekt `p_model` im Attribut `this.model` gespeichert werden, damit die Renderfunktion darauf zugreifen kann. Im Model-Objekt sind die Position, die Größe und evtl. weitere Informationen gespeichert, die beim Rendern des Objektes berücksichtigt werden müssen.

In der Klasse `ViewCircle` muss noch die Methode `render` implementiert werden. Hier kann im Prinzip die Renderfunktion übernommen werden, die sich in der Datei `game.js` befindet. Man muss nur beachten, dass die Objekte `v_view_ball` und `v_ball` hier `this.sprite` und `this.model` heißen. Jetzt können und sollten Sie einen **Aufruf** der Funktion `render` (genauer: `this.render`) als letzten Befehl in den Konstruktor einfügen, damit die View von Anfang an auf der Bühne an der korrekten Position angezeigt wird.

Testen Sie die Klasse, indem Sie in die Datei `app.js` folgende Konstante einfügen:

```
const
  c_view_ball =
    new ViewCircle
        ( c_pixi_app,
          c_model_ball,
          { "border":      5,
            "borderColor": "#AAAAAA",
            "color":       "#FFAA00"
          }
        );
```

Wenn alles klappt, sollten auf im Browser zwei Bälle zu sehen sein, einer, der sich bewegt (`game.js`) und einer, der unbewegliche in der linken oberen Ecke des Browsers steht (`ModelBall`, `ViewBall`), da die Update- und die Renderfunktionen noch nicht von der Game Loop aufgerufen werden. Andern Sie in der Konstantendefinition ruhig einmal die Initialisierungswerte `border`, `borderColor` und/oder `color` und sehen Sie sih das Ergebnis im Brwoser an..

3.2.2.6 `render`

Funktionen des Moduls `render`

Jetzt fehlt nur noch das Rendermodul. Dieses muss analog zum Updatemodul realisiert werden.

Die Initfunktion `initUpdater` speichert das ihr im Parameter `p_views` übergebene Array in einer Variablen `v_views`. Die `render`-Funktion durchläuft das Array `v_views` und ruft für jedes darin enthaltene Element die Methode `render` (siehe Klasse `ViewBall`) auf.

Wenn Sie fertig sind, testen Sie diese Methode: Initialisieren Sie in der Datei `app.js` im Anschluss an die Konstanten-Definition den Renderer. Der Funktion `initRender` wird ein Array mit genau einem Element übergeben, das regelmäßig gerendert werden soll: `c_view_ball`.

```
initRenderer([c_view_ball]);
```

Wenn sich das Programm fehlerfrei übersetzen lässt, können Sie prüfen, ob alles funktioniert: Löschen Sie den Importbefehl für die Datei `game.js` und ersetzen Sie in der Game Loop die Funktionen `game.update` und `game.render` durch die importierten Funktionen `update` und `render`. Außerdem müssen Sie in der Initfunktion den Aufruf von `game.init(...)` löschen.

Wenn jetzt alles läuft, dann ist die Modularisierung erfolgreich abgeschlossen Jetzt könnten sie auch noch die Datei `game.js` löschen. Aber vermutlich ist es besser, sie aufzuheben, um die nicht-modularisierte Version mit der modularisierten zu vergleichen.

3.3 Aufgabe 2: Konfiguration der Anwendung

mittels JSON

(Musterlösung: Gitlab: `WK_Ball03`, `app02`; unter `app02a` finden Sie eine Variante, in der die Funktion `/wk/lib/util/concretize` verwendet wurde, um die Breite und Höhe des Browser-Fensters nachträglich in das JSON-Objekt einzufügen)

Sie sollten zunächst eine Kopie Ihrer Web-App erstellen:

Erstellen Sie eine Kopie des Ordners `src/js/app01` unter dem Namen `src/js/app02`.
Erstellen Sie eine Kopie der Datei `src/index01.html` unter dem Namen `src/index02.html` und ändern Sie den Titel in dieser Datei entsprechend.
Starten Sie gegebenenfalls `npm run watch` neu (Abbruch des Watchers: `Strg-c` bzw. `Crtl-c`).

Erstellen Sie nun die JSON-Datei `src/js/app02/config/config.json` und fügen Sie folgendes JSON-Objekt in diese Datei ein:

```
{ "model":
  { "stage":
    { "left":   0,
      "right":  500,
      "top":    0,
      "bottom": 500
    },

    "ball":
    { "r":   75,
      "x":   75,
      "y":   75,
      "vx": 400,
      "vy": 300
    }
  },

  "view":
  { "ball":
    { "border":      10,
      "borderColor": "#000000",
      "color":       "#00AAFF"
    }
  }
}
```

Importieren Sie das in dieser Datei enthaltene Objekt in Ihre Anwendung `app02/app.js`:

```
import config from './config/config.json';
```

Über die Variable `config` sind nun alle alle Informationen zugänglich, die in der JSON-Datei enthalten sind. Beispiel.

```
console.log(config.view.ball)
→
{ "border":      10,
  "borderColor": "#000000",
  "color":       "#00AAFF"
}
```

Ersetzen Sie nun in den Konstruktoren der Objekte `c_model_stage`, `c_model_ball` und `c_view_ball` den darin enthaltenen JSON-Code durch den entsprechenden JSON-Code im Objekt `config`.

Testen Sie Ihre Anwendung und committen Sie die Änderungen, wenn alles läuft.

Ein (aus Sicherheitsgründen gewolltes) Problem von JSON ist, dass dort keine Funktionen definiert werden können. Das heißt, die Höhe und Breite der Bühne kann nicht mittels JSON dynamisch an die Breite des Brwoserfensters angepasst werden. In der Datei wurden die Höhe und die Breite auf jeweils 500 (Pixel) festgelegt. Um die Höhe und Breite an die Fenstergröße anzupassen, müssen Sie sie das Stage-initialisierungs-Objekt in der Datei `app02/app.js` ändern, **nachdem** Sie die JSON-Datei importiert haben:

```
const
  c_config_model_stage = config.model.stage,
  c_document_element   = document.documentElement;

c_config_model_stage.right  = c_document_element.clientWidth;
c_config_model_stage.bottom = c_document_element.clientHeight;
```

Anmerkung: Es ist vorteilhaft, Objekte, die mehrfach verwendet werden, zunächst in einer Konstanten zu speichern. Der folgende Code würde auch funktionieren, hätte aber ein paar kleine Nachteile.

```
config.model.stage.right  = document.documentElement.clientWidth;
config.model.stage.bottom = document.documentElement.clientHeight;
```

Nachteile:

Der Code ist nicht DRY.
Die Ausführungszeit ist evtl. etwas länger, da dieselben Objekte mehrfach aus eine komplexeren Objektstruktur extrahiert werden müssen (außer man hat einen richtig guten JavaScript-Compiler, der diese redundaten Befehlsschritte wegoptimieren kann).
Der Code ohne Konstanten kann weniger stark komprimiert werden, da lokale Konstantennamen vom Uglyfier durch kurze Namen ersetzt werden können. Attributnamen können hingegen nicht verkürzt werden.

Es gibt auch noch eine zweite Möglichkeit, das geladen JSON-Objekt nachträglich zu verändern. Importieren Sie die Funktion `/wk/lib/util/concretize`. Diese Funktione dient dazu, in einem JSON-Objekt Strings, die mit einen @-Symbol beginnen, nachträglich durch andere Werte zu ersetzen.

Beispielsweise liefert `concretize({"@min": 10, "@max": 20})` eine Zufallszahl zwischen 10

und 20, `concretize(["@some", "karo", "herz", pik", "kreuz"]})` *liefert zufällig einen der vier Strings* `"karo"`, `"herz"`, `pik"`, *und* `"kreuz"`, *die im Array enthalten sind. Eine Dokumentation des Befehls finden Sie im Ordner* `doc/jsdoc` *der Musterlösung.*

Um `conrretize` verwenden zu können, müssen Sie zunächst das Stage-Objekt on der JSON-Datei `config.json` ändern:

```
"stage":
{ "left":   0,
  "right":  "@right",
  "top":    0,
  "bottom": "@bottom"
}
```

Die Strings `"@right"` und `"@bottom"` haben keinerlei spezielle Bedeutung. Sie wurden (von mir) frei gewählt. Sie sollen später mittels `concretize` durch konkrete Werte (die Bühnenbreite bzw. -höhe) ersetzt werden.

Löschen Sie nun in der DAtei `app02/app.js` wieder die zuvor eingefügten Befehle:

```
const
  c_config_model_stage = config.model.stage,
  c_document_element   = document.documentElement;

c_config_model_stage.right  = c_document_element.clientWidth;
c_config_model_stage.bottom = c_document_element.clientHeight;
```

Wenden Sie dafür die Funktion `concretize` auf das JSON-Objekt an, dass Sie dem Konstruktor `ModelStage` übergeben.

```
c_model_stage =
  new ModelStage(concretize
                 ({config:      config.model.stage,
                   environment:
                   { '@right':  document.documentElement.clientWidth,
                     '@bottom': document.documentElement.clientHeight
                   }
                 })
                ),
```

Sie sehen, dass hier der Funktion `concretize` nicht nur das zu modifizierende JSON-Objekt übergeben wird, sondern auch ein so genanntes Environment-Objekt (*environment* = Umgebung, Umwelt, Ausstattung), das festlegt, durch welche Werte die beiden Strings `"@right"` und `"@bottom"` ersetzt werden sollen.

3.4 Aufgabe 3: Ersetzen der `Graphics`-View durch eine `Sprite`-View

(Musterlösung: Gitlab: `WK_Ball03`, `app03`; unter `app02a` finden Sie eine Variante, in der zwar alle erlaubten Bilder in der `app.js` aufgeführt sind, aber nur diejenigen geladen werden, die in der Konfigurationsdatei unter `init.pixiImages` aufgelistet werden)

Ersetzen Sie nun die `Graphics`-View des Balls durch eine `Sprite`-View. (Sie können dazu auch eine Version `app03` erstellen, wenn Sie später die beiden Versionen nochmals vergleichen möchten. Tipp: Im Dateibaum von WebStorm zwei Dateien selektieren, wie z. B. `app02/app.js` und `app02a/app.js` und dann <coded>Strg-d bzw. `Ctrl-d` oder auch Mausklick rechts → `Compare files`.)

Gehen Sie folgendermaßen vor, um eine `Sprite`-View zu erstellen und zu verwenden:

Nennen Sie `ViewCircle.js` in `ViewCircleGraphics.js` um (Mausklick rechts → `Refactor` → `Rename`, alle Optionen selektieren; innerhalb der Datei muss der Name allerdings noch angepasst werden).
Definieren Sie eine Klasse `ViewCircleSprite` analog zu `ViewCircleGraphics`. Diese muss die PixiJS-Klasse `Sprite` anstelle von `Graphics` importieren und geeignet initialisieren (siehe Ball02, Aufgabe 2a). Insbesondere benötigt sie dazu das Ressources-Objekt, das zuvor vom PixiJS-Loader dynamisch geladen wird (siehe Ball02, Aufgabe 3a). Für das Laden der Bilder ist das Modul app.js zuständig. Dem Konstruktor der Klasse `ViewCircleSprite` wird es in einem zusätzlichen Parameter `p_ressources` übergeben. In der Konfigurationsdatei stehen nicht mehr die Attribute für das `Graphics`-Objekt, sondern der Name des Bildes, das angezeigt werden soll (und in `p_ressources` enthalten sein muss).
Importieren Sie nun die neue Klasse code>ViewCircleSprite anstelle von `ViewCircleGraphics` in das Modul `app.js`. Und nun müssen Sie wie im Tutorium Ball02, Aufgabe 3a vorgehen (vgl. Musterlösung):

- Gewünschte Bilder, PixiJS-Loader sowie `/wk_pixi/loader/loadResources` importieren.
- Importierte Bilder zum PixiJS-Loader hinzufügen.
- Die Ressourcen asynchron **im Rumpf** der Init-Funktion laden.
- **Danach** (d. h. auch im Rumpf der Init-Funktion), das Objekt `c_view_ball` mittels des neuen Konstruktors `ViewCircleSprite` erzeugen (erst jetzt gibt es das Ressources-Objekt).
- Danach den Renderer initialisieren (erst jetzt existiert das zugehörige View-Objekt).

4 Quellen

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

Kategorien:
Multimedia-Programmierung/Tutorium
Praktikum:MMProg:WiSe 2018/19