MMProg: Praktikum: WiSe 2018/19: Pong01

Wechseln zu:Navigation, Suche

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, das Spiel Pong zu implementieren.

2 Vorbereitung

Importieren Sie das leere Git-Projekt Pong01 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:8888/BENUTZER/Pong01.git
```

In Ihrem Projekt finden Sie eine Web-Anwendung: `src/index00.html`

https://glossar.hs-augsburg.de/beispiel/tutorium/2018/pong/WK_Pong01/web/index00.html

Diese entspricht im Wesentlichen der Musterlösung der Aufgabe 3 des Tutoriums Ball03. Allerdings wurde für die Spielbühne eine feste Größe gewählt, das Bild des Balls sowie das CSS wurden verändert und ein Splashscreen wurde eingeführt. Dieser wird mittels CSS-Transitionen und JavaScript-`await`-Befehlen in der Initfunktion der Datei `app00/app.js` implementiert. Die `await`-Befehle sowie die zugehörigen Splaschscreen-Befehle ändern sich im Laufe des Tutoriums nicht. Sie sollten sie aber trotzdem studieren, wenn Sie daran interessiert sind, wie sie funktionieren.

Beachten Sie, dass die Klasse `ModelCircle` vier Methoden `left`, `right`, `top` und `bottom` enthält. Mit diesen Methoden werden die Ränder des Kreises ermittelt. Damit kann man Kollisionsberechnungen etwas einfacher formulieren.

Bei den Methoden `left`, `right`, `top` und `bottom` handelt es sich nicht um normale Methoden, sondern um so genannte Getter-Methoden (MDN web docs: Getter). Getter-Methoden sind Methoden ohne Parameter, vor die das Schlusselwort `get` geschrieben wird. Sie werden zur Berechnung von Attributwerten verwendet werden.

```
get left() { return this.x - this.r; }
// Der linke Rand eines Kreise ist gleich
// seiner Position x abzüglich seines Radius r.
```

Eine normale Methode würde man mittels `console.log(myCircle.left());` aufrufen. Beim Zugriff auf eine Getter-Methode verwendet man dagegen die klammerfreie Attributzugriff-Syntax:

```
console.log(myCircle.left);
```

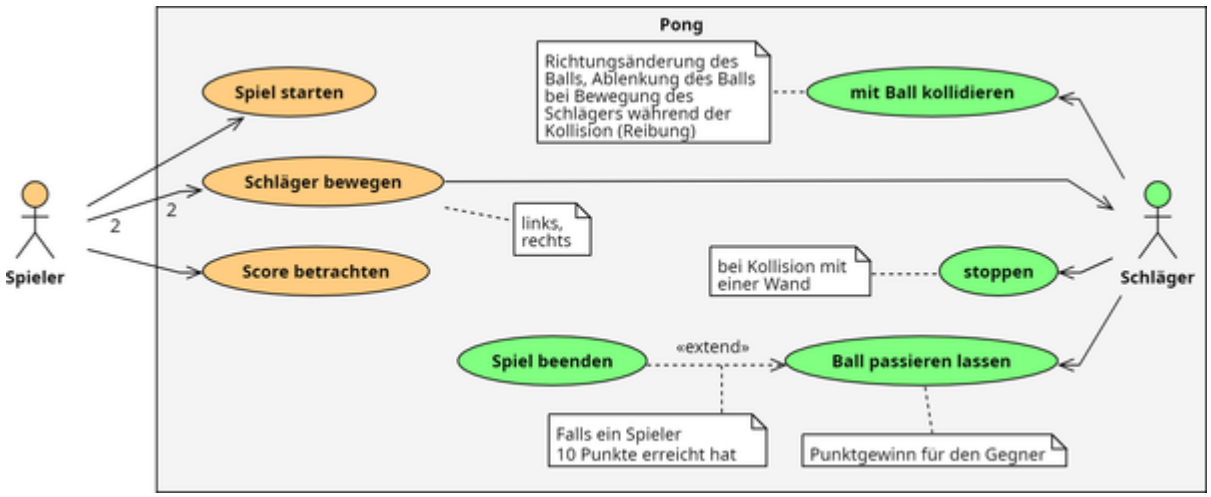
Neben den Getter-Methoden gibt es auch noch Setter-Methoden, die dazu dienen, berechnete Attribute zu verändern (MDN web docs: Setter). Diese haben genau einen Parameter, der den Wert enthält, der gespeichert werden soll:

```
set left(p_x) { this.x = p_x + this.r; }
// Anstelle des linken Randes wird die Position des Kreises
geändert,
// und zwar so, dass der linke Rand an der gewünschten Position zu
// liegen kommt.
```

Zum Aufruf der Setter-Methoden kommt ebenfalls die Attributzugriff-Syntax zum Einsatz:

```
myCircle.left = 100;
console.log(myCircle.left) → 100
```

2.1 Use Cases



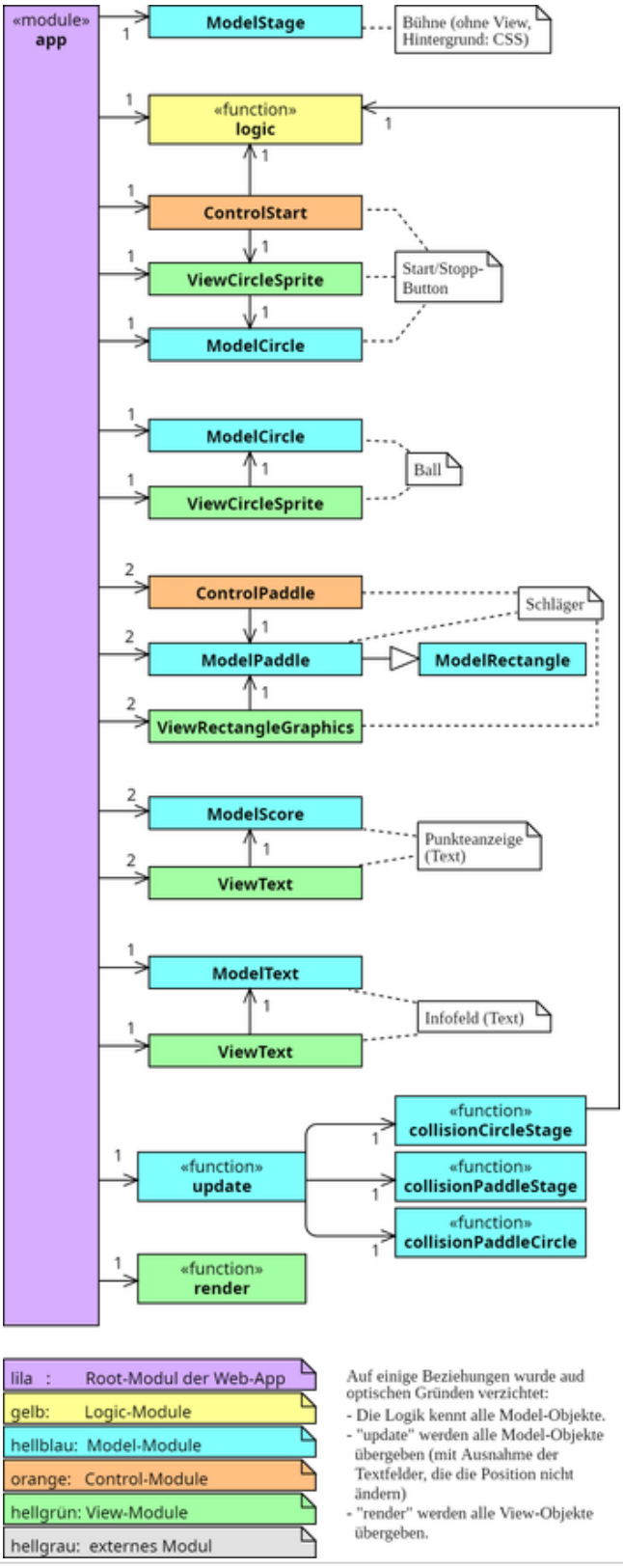
Use Cases des Spiels Pong

Der Aufgabe liegt das Modell Pong/Modellierung zugrunde. Allerdings wurden bei der Umsetzung einige Änderungen vorgenommen. Am Use-Case-Diagramm fällt auf, dass der Use Case „Spiel abbrechen“ fehlt. Es wurde darauf verzichtet, da nicht – wie ursprünglich geplant – ein Start-/Stopp-Knopf (als HTML-Button) außerhalb der Bühne platziert wird, sondern nur ein Start-Knopf innerhalb der Bühne. Dieser wird ausgeblendet, solange das Spiel läuft.

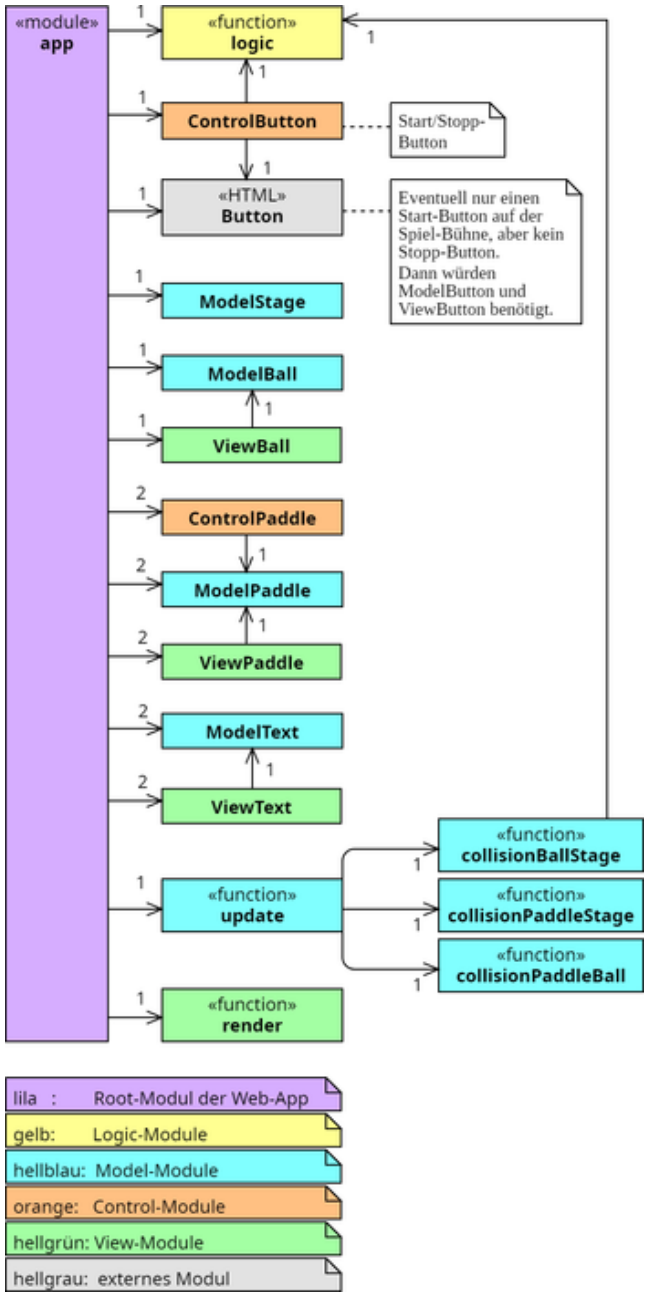
Ich habe das implementierte Modell ganz bewusst gegenüber der ursprünglichen Planung abgeändert, um den dynamischen Prozess zu verdeutlichen, den ein Modell durchläuft. Es ändert sich ständig: Elemente werden ergänzt, verfeinert, ersetzt oder auch ersazlos gestrichen. Ich kenne niemanden, der zu Beginn eines Projektes ein perfektes Modell aufstellt, dass nicht ein paar Dutzend mal geändert werden muss.

2.2 Klassendiagramm (Moduldiagramm)

[Klassendiagramm von `Pong01` (zweite Version)



[Klassendiagramm von `Pong01` (erste Version)



Dieses Klassendiagramm ist ebenfalls gegenüber dem Klassendiagramm von Pong/Modellierung weiterentwickelt worden. Anstelle eines HTML-Elements gibt es jetzt einen kreisförmigen Start-Button. Die Klassen `ModelCircle` und `ViewCircle` werden auch für den Ball verwendet (Wiederverwendung). Die Klasse `ModelPaddle` wurde als Unterklasse einer (wiederverwendbaren) Klasse `ModelRectangle` definiert. Für die Punkteanzeige wurde eine eigene Klasse `ModelScore` definiert, da in Objekten der Klasse `ModelText` keine Zahlen, sondern nur Text gespeichert werden

können. Das ist für die Rechnung mit Punkten, die jeweils mittels ++ erhöht werden, etwas unpraktisch. Zu guter Letzt wurde noch ein Textfeld eingeführt, das dazu genutzt werden kann, bei Spielende die Spieler über den Sieger zu informieren.

3 Aufgabe

Implementieren Sie Pong gemäß obigem Klassendiagramm.

3.1 Aufgabe 1

(Musterlösung: Gitlab: `WK_Pong01`, `app01`, index01.html)

Sie sollten zunächst eine Kopie Ihrer Web-App erstellen:

Erstellen Sie eine Kopie des Ordners `src/js/app00` unter dem Namen `src/js/app01`.
Erstellen Sie eine Kopie der Datei `src/index00.html` unter dem Namen `src/index01.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`).

In der Musterlösung `app00` aus Ball 03 gibt es schon einige Module, die Sie wiederverwenden können:

`model/ModelStage`
`model/ModelCircle`
`model/collisionCircleStage`
`model/update`
`view/ViewCircleGraphics`
`view/ViewCircleSprite`
`view/render`
`app`

Implementieren Sie als nächstes folgende Module, um die beiden Schläger darstellen zu könnnen:

`model/ModelRectangle` (`ModelPaddle` folgt später)
`view/ViewRectangleGraphics`

3.1.1 Konfiguration des Balls

Platzieren Sie den Ball zu Beginn des Spiels in der Mitte der Bühne, indem Sie die Konfigurationsdatei entsprechend abändern. Sie können auch noch die Ballgröße anpassen, indem Sie ein beispielsweise einen Radius von nur 12 Pixeln wählen:

```
"ball":
{ "r":   15,
  "x":   15,
  "y":   15,
  "vx": 200,
  "vy": 150
}
```

3.1.2 Klasse `ModelRectangle`

Die Klasse `ModelRectangle` hat folgende Attribute:

`width` (anstelle von `r` in `ModelCircle`)
`height` (anstelle von `r` in `ModelCircle`)
`x` (wie `ModelCircle`)
`y` (wie `ModelCircle`)
`vx` (wie `ModelCircle`)
`vy` (wie `ModelCircle`)
`ax` (vgl. Praktikum Ball01, Aufgabe 6)
`ay` (vgl. Praktikum Ball01, Aufgabe 6)
`left` (analog zu `ModelCircle` ohne `this.r`)
`right` (analog zu `ModelCircle`, aber mit `this.width`)
`top` (analog zu `ModelCircle` ohne `this.r`)
`bottom` (analog zu `ModelCircle`, aber mit `this.width`)

Darüber hinaus enthält diese Klasse zwei Methoden:

`reset` (analog zu `ModelCircle`)
`update` (analog zu `ModelCircle`, allerdings muss auch die Geschwindigkeit abhängig von der Beschleunigung angepasst werden; vgl. Praktikum Ball01, Aufgabe 6)

Beachten Sie bei der Berechnung der der Begrenzungen `left`, `right`, `top` und `bottom` eines Rechtecks, dass der Ankerpunkt üblicherweise in der linken oberen Ecke des Rechtecks liegt. Beim Kreis liegt er dagegen im Mittelpunkt.

3.1.3 Klasse `ViewRectangleGraphics`

Die Klasse `ViewRectangleGraphics` wird analog zu `ViewCircleGraphics` implementiert. Der wesentliche Unterschied ist, dass das grafische Objekt nicht mit `drawCircle`, sondern mittels `drawRect` gezeichnet wird (PIXI.Graphics).

3.1.4 `config.json` und Modul `app.js`

Jetzt können Sie zwei Schläger in Ihre Anwendung einbinden. Erweitern Sie zunächst die Kondifurationsdatei `config/config.json`. Fügen Sie **in** das Modelobjekt die Modelkonfigurationen der beiden Paddle ein:

```
"paddles":
[ { "x":        5,
    "y":      170,
    "vy":     150,
    "ay":    1000,
    "width":   10,
    "height":  60
  },
  { "x":      585,
    "y":      170,
    "vy":     150,
    "ay":    1000,
    "width":   10,
    "height":  60
   }
]
```

Und in das Viewobjekt fügen Sie **eine** View-Konfiguration ein, die für beide Paddle verwendet werden kann.

```
"paddle":
{ "border": 0,
  "color":  { "color": "#999999" }
}
```

Wenn Sie keinen Syntaxfehler gemacht haben, lässt sich die `app01` immer noch fehlerfrei übersetzen.

Nun ist es an der Zeit die Datei `app.js` anzupassen:

Importieren Sie die beiden neu erstellten Klassen `ModelRectangle` und `ViewRectangleGraphics` analog zu `ModelCircle` und `ViewCircleGraphics`. Auch hier gilt: Wenn Sie in beiden Dateien keine Syntaxfehler gemacht haben, lässt sich die App immer noch fehlerfrei übersetzen.
Fügen Sie nach der Konstanten `c_config_view_ball` zwei Konstanten ein, mit denen Sie auf die beiden Konfigurationsobjekte zugreifen können, die Sie zuvor in die Datei `config.json` eingefügt haben:

```
c_config_models_paddle = c_config_model.paddles,
c_config_view_paddle   = c_config_view.paddle,
```

Nun müssen Sie im Anschluss an die Erzeugung von `c_model_ball` die Models der beiden Schläger erzeugen (wären es mehr als zwei Objekte, würde man das Array natürlich mit Hilfe einer Schleife füllen):

```
c_models_paddle =
  [ new ModelRectangle(c_config_models_paddle[0]),
    new ModelRectangle(c_config_models_paddle[1])
  ]
```

Als nächstes müssen Sie das soeben erzeugte Array in das Konfigurationsobjekt vom Funktionsaufruf der Funktion `initUpdater` einfügen, da der Updater natürlich auch die Positionen der Schläger regelmäßig neu berechnen muss:

```
initUpdater({stage:   c_model_stage,
             ball:    c_model_ball,
             paddles: c_models_paddle
           });
```

Jetzt fehlen noch die Views. Diese müssen im Rumpf der Initfunktion im Anschluss an die Ballview erzeugt werden (auch hier gilt: das Array würde mit Hilfe eine Schleife befüllt werden, wenn es sich um mehr als zwei Schläger handeln würde):

```
c_views_paddle =
[ new ViewRectangleGraphics
      (c_pixi_app, c_models_paddle[0],c_config_view_paddle),
  new ViewRectangleGraphics
      (c_pixi_app, c_models_paddle[1], c_config_view_paddle)
]
```

Diese Views müssen vom Renderer regelmäßig neu gezeichnet werden (hier kommt wieder ES-6-Destructuring-Syntax zum Einsatz, um den Inhalt des Arrays `c_views_paddle` in ein anderes Array einzugügen; in ES 5 müssten Sie `initRenderer([c_view_ball].concat(c_views_paddle));` schreiben)

```
initRenderer([c_view_ball, ...c_views_paddle]);
```

Jetzt sollte die Anwendung wieder laufen und die beiden Schläger sollten zu sehen sein.

3.1.5 Modul `update`

Sie sollten auch noch das Update-Modul `update` aktualisieren. Die Schläger bewegen sie noch nicht, aber sie sollten sich bewegen, da in der Konfigurationsdatei für beide Schläger eine Geschwindigkeit und eine Beschleunigung in y-Richtung eingetragen wurde. Das heißt, das Update-Modul sollte dafür sorgen, dass für beide Schläger regelmäßig die Updatefunktion aufgerufen wird.

Fügen Sie den den Parameter `paddles: p_paddles` in das Config-Objekt der Parameterliste der Funktion `initUpdater` ein.

Speichern Sie das im Parameter `p_paddles` übergebene Array mit den `ModelRectangle`-Objekten in einer globalen (aber privaten) Variablen `v_paddles` des Moduls.
Fügen Sie in die Methode `update` eine For-Loop ein, die für jedes Objekt im Array `v_paddles` die zugehörige Update-Methode (mit einem geeigneten Argument) aufruft.
Testen Sie, ob sich das Programm noch korrekt übersetzen lässt. Wenn Sie es laufen lassen, sollten die beiden Schläger fluchtartig die Bühne verlassen:
https://glossar.hs-augsburg.de/beispiel/tutorium/2018/pong/WK_Pong01/web/index01.html

3.2 Aufgabe 2

(Musterlösung: Gitlab: `WK_Pong01`, `app02`, index02.html)

Sorgen Sie jetzt dafür, dass die Kollisionen der Paddle mit dem Ball und der Bühne erkannt und behandelt werden.

Fügen Sie in die Klassen `ModelCircle` und `ModelRectangle` Setter-Methoden für die Attribute `left`, `right`, `top` und `bottom` ein. Diese sollen dafür sorgen, dass jeweils die x- bzw. die y-Koordinate des Objektes so gesetzt wird, dass die Getter-Methode des modifizierten Randattriutes den gewünschten Wert als Ergebnis liefert.
Definieren Sie die Klasse `ModelPaddle` als Subklasse der Klasse `ModelRectangle`. Der Konstruktor `constructor(p_config)` übergibt das Konfigurationsobjekt `p_config` einfach an seine Superklasse: `super(p_config);`. Überschreiben Sie anschließend die Resetfunktion. Diese soll die Breite, Höhe und Position des Schlägers genauso wie die Klasse `ModelRectangle` auf die im Konfigurationsobjekt übergebenen Werte zurücksetzen. Die Geschwindigkeit und die Beschleunigung soll sie allerdings jeweils auf setzen, da die Schläger sich zu Beginn nicht bewegen.
Verwenden Sie im Modul `app` die Klasse `ModelPaddle` anstelle der Klasse `ModelRectangle` zum Erzeugen der beiden Schläger-Models.
Fügen Sie in die Klasse `ModelPaddle` drei Methoden `down`, `up` und `stop` ein.
Die Methode `down` setzt die Geschwindigkeit und die Beschleunigung in y-Richtung auf die im Konfigurationsobjekt übergebenen Werte, **sofern die Geschwindigkeit beim Aufruf gleich ist**.
Die Methode `up` setzt die Geschwindigkeit und die Beschleunigung in y-Richtung auf die im Konfigurationsobjekt übergebenen Werte, allerdings mit negativem Vorzeichen, **sofern die Geschwindigkeit beim Aufruf gleich ist**.
Die Methode `stop` setzt die Geschwindigkeit und die Beschleunigung in y-Richtung auf , **sofern die Geschwindigkeit beim Aufruf ungleich ist**.
Definieren Sie eine Modul `collisionPaddleStage(p_paddle, p_stage)`, das das übergebene Paddle mittels der zuvor definierten Methode `stop` anhält und vollständig zurück auf die Bühne verschiebt, sobald es mit einem Bühnenrand kollidiert. Sie können hierzu die zuvor definierten Setter-Methoden verwenden: `p_paddle.top = p_stage.top;` bzw. `p_paddle.bottom = p_stage.bottom;`.
Wenn Sie möchten, können Sie auch noch gleich die Kollisionserkennung und -behandlung von Kreis und Bühne (`collisionCircleStage`) vereinfachen, indem Sie die zuvor definierten Attribute `left`, `right` ... verwenden.
Schreiben Sie eine Kollisionererkennung- und behandlung `collisionPaddleCircle(p_paddle, p_circle)` für Schläger und Ball. Folgender Code funktioniert, aber nur schlecht, falls der Ball mit einer Ecke eines Schläger kollidiert. Hier bestehnt noch erhebliher Verbesserungsbedarf:

```
function collisionPaddleCircle(p_paddle, p_circle)
{ if (p_circle.y + 0.8*p_circle.r >= p_paddle.top &&
      p_circle.y - 0.8*p_circle.r <= p_paddle.bottom
     )
  { if (p_circle.vx > 0 &&  // The ball is moving from left to
right.
        p_circle.right >= p_paddle.left && p_circle.left <
p_paddle.right
       )
    { p_circle.right = p_paddle.left;
      p_circle.vx    = -p_circle.vx;
    }

    if (p_circle.vx < 0 &&  // The ball is moving from right to
left.
        p_circle.left <= p_paddle.right && p_circle.right >
p_paddle.left
       )
    { p_circle.left = p_paddle.right;
      p_circle.vx   = -p_circle.vx;
    }
  }
}
```

Importieren Sie die beiden Funktionen `collisionPaddleStage` und `collisionPaddleCircle` und das Update-Modul und runfen Sie die beiden Funktionen innerhalb der Schleife der Update-Funktion geeignet auf.

Wenn sich alles übersetzen lässt, sollte die App wieder laufen. Die Schläger sollten sich nicht bewegen, aber der Ball sollte davon abprallen, sobald er mit einem kollidiert. Sie können testhalber als letzen Befehl (d. h. nach dem Start der Game Loop) folgenden Befehl in den Rumpf der Init-Funktion im Modul `app` einfügen:

```
c_models_paddle[1].up();
```

Damit sollte sich – sofern Sie Ihr Array mit den beiden Schläger-Models `c_models_paddle` genannt haben – der rechte Schläger nach oben bewegen, sobald das Spiel gestartet wurde. Und er sollte stehen bleiben, sobald er mit dem oberen Rand kollidiert:
https://glossar.hs-augsburg.de/beispiel/tutorium/2018/pong/WK_Pong01/web/index02.html

Sie sollten testweise auch den anderen Schläger nach oben oder unten bewegen.

3.3 Aufgabe 3

(Musterlösung: Gitlab: `WK_Pong01`, `app03`, index03.html)

Als nächstes sollten Sie eine Contoller für die Schäger einbauen:

Fügen Sie in die Konfigurationsdatei nach `"model"` und `"view"` ein drittes Objekt ein, das beschreibt, mit welchen Tasten die beiden Schläger gesteuert werden sollen:

```
"control":
{ "paddles":
  [ { "up":   "w",
      "down": "x"
    },
    { "up":   "ArrowUp",
      "down": "ArrowDown"
    }
  ]
}
```

Legen Sie eine neue Datei `control/ControlPaddle.js` an, die einen Klasse `ControlPaddle` definiert und eportiert. Diese Klasse enthält lediglich einen Kostruktor. Dieser definiert zwei interne Funktionen `o_start_moving` und `o_stop_moving` (`o_` steht für Observer Pattern '''O'''bserver), die als Event Handler für Tastaturereignisse verwendet werden (vgl. Hello-World-Tutorium, Teil 3, Aufgabe 6). Wenn eine der beiden Tasten im Konfigurationsobjekt **gedrückt** werden (`window-keydown`-Ereignis), wird der Schläger mittels einer der zuvor definierten Methoden `up` oder down n die gewünschte Richtung bewegt. Sobald die Taste wieder losgelassen wird (`window-keyup`-Ereignis) wird der Schläger mittels der Methode `stop` wieder angehalten.

```
constructor(p_paddle,
            { up: p_up = 'ArrowUp', down: p_down = 'ArrowDown'} =
{}
           )
{ function o_start_moving(p_event)
  { if (p_event.key === p_up)
    { p_paddle.up(); }
    else if (p_event.key === p_down)
    { p_paddle.down(); }
  }
  function o_stop_moving(p_event)
  { if (p_event.key === p_up || p_event.key === p_down)
    { p_paddle.stop(); }
  }
  window.addEventListener("keydown", o_start_moving);
  window.addEventListener("keyup",   o_stop_moving);
}
```

Nun müssen Sie noch diController für die beiden Schläger im Modul `app` erstellen und initialisieren: Importieren Sie zunächst die neu definierte Klasse `ControlPaddle`.

Definieren Sie dann analog zu den anderen Konstanten, in denen Sie bestimmte Teilobjekte der Konfigurationsdatei speichern, zwei Konstanten `c_config_control = config.control` sowie `c_config_control_paddles = c_config_control.paddles`. In der zweiten Konstante wird das Konfigurationsobjekt für den Paddle-Controller gespeichert, das Sie zuvor in die Datei `config.json` eingefügt haben.
Erzeugen und initialisieren Sie nun im Anschluss an die Konstantendefinitionen die beiden Controller. Beschten Sie, dass es nicht notwendig ist, die beiden Objekte zu speichern, da kein Modul wieder darauf zugreifen wird.

```
new ControlPaddle(c_models_paddle[0],
c_config_control_paddles[0]);
new ControlPaddle(c_models_paddle[1],
c_config_control_paddles[1]);
```

Wenn Sie jetzt die Anwendung starten, sollten Sie die beiden Schläger mittels der Tasten `w` und `x` bzw. `ArrowUp` und `ArrowDown` bewegen können. Der Ball sollte abprallen, wenn er mit einem kollidiert:
https://glossar.hs-augsburg.de/beispiel/tutorium/2018/pong/WK_Pong01/web/index03.html

3.4 Aufgabe 4

(Musterlösung: Gitlab: `WK_Pong01`, `app04`, index04.html)

Nun ist es an der Zeit, die Spiellogik zu implementieren.

Fügen Sie ein paar neue Informationen in die Konfigurationsdatei ein:

`"model"`: Hier werden Informationen über die beiden Score-Textfelder erfasst: Position und Startwert:

```
"scores":
[ { "x":     20,
    "y":     20,
    "score":  0
  },
  { "x":    580,
    "y":     20,
    "score":  0
  }
]
```

`"view"`: Hier wird festgelegt, wie eine (Score-)Textfeld formartiert (`"style"`) und positioniert (`anchor`; `0.5` entspricht zentriert) werden soll:

```
"text":
{ "style": { "fontFamily": ["Verdana", "Helvetica", "sans-serif"],
             "fontSize":   "20px"
           },
  "anchor": {"x": 0.5, "y": 0}
}
```

`"logic"`: Ein neuer Eintrag an Ende der Konfigurationsdatei, in der definiert wird, wie viele Runden das Spiel dauern soll:

```
{ "winScore": 3 }
```

Ersetzen Sie in der Konfigurationsdatei nun noch im Ballobjekt die festen Geschwidigkeitswerte `"vx": 200, "vy": 150` durch die nachfolgenden Objekte. Diese können später in der Resetfunktion von `ModelCircle` mittels `concretize` durch Zufallswerte in den angegebenen Bereichen ersetzt werden. Ermittelt wird jeweils einen Zufallszahl im Intervalll 150 bis 300 und wird mit 50-prozentiger Wahrscheinlichkeit mit einen negativen Vorzeichen versehen (vgl. Praktikumsaufgabe Ball03, Aufgabe 2):

```
"vx": {"@min": 150, "@max": 300, "@positive": 0.5},
"vy": {"@min": 150, "@max": 300, "@positive": 0.5}
```

Damit das Spiel wieder funktioniert, müssen Sie die Klasse `ModelCircle` so erweitern, dass die Resetfunktion bei jedem Aufruf die Funktion `concretize` auf das Konfigurationsobjekt anwendet, bevor sie die Attribute des Ballobjektes auf die Initialwerte zurücksetzt. Improtieren Sie zunächst die Funktion `/wk/util/concretize` in das Modul. Anschließend müssen Sie den Konstruktor und die Resetfunktion so umschreiben, dass die Zerlegung des Konfigurationsobjektes (Destructuring) nicht mehr im Konstruktor, sondern erst in der Resetfunktion durchgeführt wird, **nachdem `concretize` darauf agewendet wurde (dabei wird auf die Attribute `this.rConfig`, `this.xConfig` etc. verzichtet; stattdessen wird das ganze Konfigurationsobjekt in `this.config` dauerhaft gespeichert):**

```
class ModelCircle
constructor(p_config = {})
{ this.config = p_config;
  this.reset();
}
reset()
{ const {r=0, x=0, y=0, vx=0, vy=0} = concretize({config:
this.config});
  this.r  = r;
  this.x  = x;  this.y  = y;
  this.vx = vx; this.vy = vy;
}
...
```

Jetzt sollte das Spiel wieder funktionieren, mit dem Unterschied, dass der Ball bei jedem Aufruf von der Mitte aus in eine zufällige Richtung losfliegt.
Wenn Sie möchten, können und sollten Sie (aus Symmetriegründen) auch die Klassen `ModelRectangle` und `ModelPaddle` so umschreiben, dass das Destructuring erst in der Resetfunktion stattfindet.
Nun müssen Sie die beiden Klassen `ModelScore` und `ViewText` zur Verwaltunge der Score-Textfelder erstellen (und im jeweiligen Modul auch exportieren!).
Der Konstruktor von `ModelScore` hat einen Parameter `p_config` in dem ihm ein Konfigurationsobjekt übergeben wird, das zuvor in der Konfigurationsdatei definiert wurde (beispielsweise `{ "x": 20, "y": 20, "score": 0}`). Diese Objekt wird wie auch schon in der Klasse `ModelCircle` im Attribut `this.config` gespeichert und in der Resetfunktion zum Initialisieren der drei Attribute `this.score`, `this.x` und `this.y` verwendet. Auf den Einsatz der Funktion ycodecretize kann hier verzichtet werden. (Es würde aber auch keine Probleme verursachen, wenn sie dennoch aufgerufen werden würde.)
Neben dem Konstruktor, und der Resetfunktion gibt es noch eine Gettermethode `text`, mittels der die Renderfunktion auf den Stringwert des Scores zugreifen kann. (Im Attribut `this.code` wird ein Zahlwert gespeichert, damit dieser mittels des ++-Operators einfach hochgezählt werden kann.

```
get text()
{ return this.score.toString(); }
```

Die Klasse `ViewText` ist im Prinzip wie die Klasse `ViewSpriteCircle` aufgebaut. Die wesentlichen Unterschiede sind:
Es wird die PixiJS-Klasse `Text` nstelle der Klasse `Sprite` importiert.
Das Konfigurationsobjekt, das dem Konstruktor übergeben wird, enthält kein Attribut `image`, sondern die Attribute `style` und `anchor`.
Im Model-Objekt wird ein Objekt übergeben, dass in einem Attribut `text` den Text bereitstellt, der visualisiertwerden soll. Die Objekte der Klasse `ModelScore` haben so ein Attribut.
Das Sprite-Objekt wird mittels `const c_sprite = new Text(p_model.text, p_style)` erstellt, wobei `p_model` der Parameter ist, indem dem Konstruktor das Score-Model übergeben

wurde, und `p_style` das gewünschte Styleobjekt aus der Konfigurationsdatei enthält. In diesem Objekt dürfen alle Styleattribute verwendet werden, die PixiJS unterstützt: {http://pixijs.download/dev/docs/PIXI.TextStyle.html PIXI.TextStyle]. Anschließend muss noch der Anchor des Textobjektes korrekt gesetzt werden: `c_sprite.anchor = p_anchor;`.
Der Rest des Konstruktors stimmt mit dem Rest des Konstruktors der Klasse `ViewSpriteCircle` überein.
Die Renderfunktion muss nicht nur das Text-Objekt korrekt platzieren, sondern auch noch den gewünschten Text darstellen. Fügen Sie daher den folgenden Befehl ein:

```
this.sprite.text = this.model.text;
```

Erzeugen und initialisieren Sie die beiden Score-Textfelder wie üblich im App-Modul:
`ModelScore` und `ViewText` importieren,
`c_config_models_score` und `c_config_view_text` wie üblich definieren,
`c_models_score` analog zu `c_models_paddle` definieren (mit `ModelScore` als Konstruktor),
`c_views_score` analog zu `c_views_paddle` definieren (mit `ViewText` als Konstruktor).
Fügen Sie das Array `c_views_score` analog zu `c_view_paddle` in das Array ein, das der Funktion `initRenderer` als Argument übergeben wird, damit künftig auch die Textfelder gerendert werden.
Sie haben nun soviele Models definiert, dass es sich rentiert, eine Konstante `c_models` zu definieren, die alle Models enthält und diese Konstante sowohl an fie Funktion `initUpdater` zu übergeben (`initUpdater(c_models);`) als auch später an die Logic (sobald diese definiert ist):

```
c_models =
  { stage:   c_model_stage,
    ball:    c_model_ball,
    paddles: c_models_paddle,
    scores:  c_models_score
  }
```

Als nächstes müssen Sie die Logik in der Datei `logic/logic.js` implementieren. Diese stellt wie üblich eine Funktion `initLogic` bereit, in der ihr alle wesentlichen Objekte übergeben werden, die Sie kennen muss, um das Spiel zu verwalten. Das sind die Gam Loop (da sie das Spiel starten und stoppen können sollte), die Models sowie das Konfigurationsobjekt `logic` aus der Konfigurationsdatei. Wie üblich muss diese Funktion die ihr übergebenen Werte in (dateiinternen) Variablen speichern. Anschließend sollte sie das Spiel starten:

```
function initLogic(p_game_loop,
                   {ball: p_ball, paddles: p_paddles, scores:
p_scores},
                   {winScore: p_win_score = 10} = {}
                  )
{ v_game_loop = p_game_loop;
  v_ball      = p_ball;
  v_paddles   = p_paddles;
  v_scores    = p_scores;
  v_win_score = p_win_score;

  // start the game
  v_game_loop.start();
}
```

Darüber hinaus muss die Logik eine Funktion implementieren, die die Kollissionserkennung aufrufen kann, wenn ein Spieler einen Ball passieren lässt. Diese muss die Punkte des Gegners erhöhen, die Game Lopp stoppen, wenn das Spiel vorbei ist oder eine neue Runde starten, wenn das Spiel noch nicht vorbei ist.

```
function ballLoss(p_player)
{ // the score of the other player is raised
  const c_player = p_player === 'left' ? 1 : 0;
  v_scores[c_player].score++;
  if (v_scores[c_player].score === v_win_score) // game over
  { v_game_loop.stop();
    v_ball.reset();
  }
  else // the next round starts
  { v_ball.reset(); }
}
```

Beide Funktionen müssen exportiert werden.
Als nächstes müssen Sie dafür sorgen, dass die Kollisionsfunktion `collisionCircleStage` den Ball nicht mehr an den Wänden hinter den Schlägern abprallen lässt. Anstatt dessen muss der Ball die Bühne verlassen und die Logik muss informiert werden, welcher Spieler (= Schläger) den Ball verlohren hat. Improtieren Sie dazu in diesem Modul die zuvor definierte Funktion `ballLoss` aus der Logik und ersetzen Sie die bisherigen Kollsisionsabfragen für die linke und die rechte Wand durch folgenden Code:

```
if (p_circle.right < p_stage.left)
  { ballLoss('left'); }
  if (p_circle.left > p_stage.right)
  { ballLoss('right'); }
```

Integrieren Sie die Logik nun auch noch ins Modul `app`:
Importieren Sie die Funktion `initLogic`.
Definieren Sie `c_config_logic` wie üblich.
Ersetzen Sie zum Schluss den Start der Game Loop am Ende der Initfunktion durch die Initialisierung der Logik:

```
initLogic( new GameLoop({update: update, render: render}),
             c_models,
             c_config_logic
           );
```

Wenn jetzt alles funktioniert, sollten Sie ein erstes Spiel wagen können.
https://glossar.hs-augsburg.de/beispiel/tutorium/2018/pong/WK_Pong01/web/index04.html

3.5 Aufgabe 5

(Musterlösung: Gitlab: `WK_Pong01`, `app05`, index05.html)

Verfeinern Sie das Spiel weiter, indem Sie einen Startbutton integrieren, der das Spiel startet, sobald Sie auf ihn klicken. Sie brauchen dazu ein Bild (`imgStart` das Sie in `config.json` konfigurieren und in `app.js` laden müssen. Das Bild eines kreisrunden Buttons finden Sie im Bilder-Ordner: `start-200.png`.

Sie müssen in der Datei `config.json` außerdem das Model und die View des Startbuttons geeignet konfigurieren und die zugehörigen Model- und Viewobjekte im App-Modul definieren und initialisieren. Da das Bild kreisrund ist, sollten Sie die Kalssen `ModelCircle` und ViewCircleSprite verwenden. Vergessen Sie nicht, das Model in das Objekt `c_models` einzufügen. Außerdem müssen Sie die View in das Array von `initRenderer` einfügen. Im Update-Modul müssen Sie dafür sorgen, dass die Update-Funktion des Startbuttons regelmäßig aufgerufen wird.

Das bisherige Vorgehen ist etwas problematisch, da Sie die Game Loop bei Spielende nicht mehr anhalten können. Der Grund ist, dass auch der Button mit Hilfe der Loop aktualisiert und gerendert wird. Es wäre zwar kein Problem darauf zu verzichten, aber es ist gar nicht schlecht, die Game Loop dauerhaft laufen zu lassen, um auch nach Spielende bestimmte Elemente auf der Bühne animieren zu können.

Das heißt aber, dass Sie die Klasse `ModelCircle` un drei Methoden erweitern müssen, mit denen Sie den Ball jederzeit in der Mitte des Spielfelds platzieren, mit zufälliger Flugrichtung starten und auch wieder anhalten können.

```
moveTo({x=0, y=0} = concretize({config: this.config}))
{ this.x = x; this.y = y; }
start()
{ const {vx=0, vy=0} = concretize({config: this.config});
  this.vx = vx; this.vy = vy;
}
stop()
{ this.vx = 0; this.vy = 0; }
```

Jetzt müssen Sie in der Logik eine neue Startfunktion definieren, außerdem müssen Sie `ballLoss` so umschreiben, dass das Spiel nicht mehr mittels einens Stopps der Game Loop angehalten wird:

```
import wait from '/wk/util/wait';

let
  v_game_loop, v_start_button, v_ball, v_paddles, v_scores,
v_win_score;

function initLogic(p_game_loop,
                   { startButton: p_start_button,
                     ball:        p_ball,
                     paddles:     p_paddles,
                     scores:      p_scores
                   },
                   { winScore: p_win_score = 10
                   } = {}
                  )
{ v_game_loop    = p_game_loop;

  v_start_button = p_start_button;
  v_ball         = p_ball;
  v_paddles      = p_paddles;
  v_scores       = p_scores;
  v_win_score    = p_win_score;
  v_ball.moveTo();        // move the ball to its start point
  v_ball.stop();
  v_start_button.reset(); // make the start button visible
  v_game_loop.start();
}

async function startGame()
{ v_scores[0].score = 0;
  v_scores[1].score = 0;
  v_start_button.x = -2*v_start_button.r; // move the start button
```

```
outside
                                          // the stage to make it
invisible
  await wait(1000);
  v_ball.start();                        // start the game
}

function ballLoss(p_player)
{ // the score of the other player is raised
  const c_player = p_player === 'left' ? 1 : 0;
  v_scores[c_player].score++;
  if (v_scores[c_player].score === v_win_score) // game over
  { v_ball.moveTo();        // Put the ball in the middle of the
stage
    v_ball.stop();          // so that no collision with the outside
of
                            // the stage is detected.
    v_start_button.reset(); // Make the start button visible again.
  }
  else // the next round starts
  { v_ball.reset(); }
}

export {initLogic, startGame, ballLoss};
```

Beachten Sie, dass die Startfunktion asynchron definiert wurde, um nach den Klicken des Startbuttons noch mittesl `await wait(1000)` etwas zu warten zu können, bis der Spieler die Maus aus dem Spielfeld bewegt hat.

Als letztes benötigen Sie einen Controller für den Startbutton, der die zuvor in der Logik definierte Startfunktion aufruft, sobald der Spieler den Startbutton betätigt. Wenn man die PixiJS-Events verwendet, die bei einem Klick auf ein grafisches Element verschickt werden, ist es ganz einfach einen derartigen Controller zu implementieren:

```
import {startGame} from '../logic/logic';

function controlStart(p_start)
{ p_start.addEventListener('pointerdown', () => startGame()); }

export {controlStart};
```

Jetzt brauchen Sie die Funktion `controlStart` nur noch in das App-Modul zu importieren und für die View des Startbuttons aufrufen, sobald diese View definiert wurde.

Wenn alles funktioniert, können Sie nun mehrere Runden Pong spielen:
https://glossar.hs-augsburg.de/beispiel/tutorium/2018/pong/WK_Pong01/web/index05.html

3.6 Aufgabe 6

(Musterlösung: Gitlab: `WK_Pong01`, `app06`, index06.html)

Erweitern Sie das Spiel so, dass bei Spielende der Name des Siegers („linker Spieler“ oder „rechter Spieler“) in einem Textfeld ausgegeben wird, bevor der Startbutton wieder eingeblendet wird: https://glossar.hs-augsburg.de/beispiel/tutorium/2018/pong/WK_Pong01/web/index06.html

4 Quellen

1. **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