# Funktionale und objektorientierte Programmierkonzepte
# Übungsblatt 13

TECHNISCHE UNIVERSITÄT DARMSTADT

**Prof. Karsten Weihe**

| | |
|---|---|
| Übungsblattbetreuer: | Ruben Deisenroth |
| Wintersemester 22/23 | v1.0.8 |
| Themen: | JavaFX, MVC |
| Relevante Foliensätze: | 10 |
| Abgabe der Hausübung: | 10.02.2023 bis 23:50 Uhr |

## Hausübung 13 — Gesamt: 42 Punkte

*Space Invaders*

**Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.**

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h13` und ggf. `src/test/java/h13`.

**Verbindliche Anforderungen für die gesamte Hausübung:**

- In der Datei `src/main/java/h13/controller/GameConstants.java` sind alle Konstanten des Spiels gespeichert. Ihre Implementierung muss auch dann funktionieren, wenn die Werte der Konstanten verändert werden.

**Anmerkungen:**

- Insofern nicht per `@Nullable` annotiert, dürfen Sie davon ausgehen, dass alle Rückgabewerte und Parameter von Methoden niemals `null` sind.
- Sie dürfen nach Belieben Attribute, Methoden und Hilfsklassen hinzufügen, sofern die Anforderungen erfüllt werden. (bei einigen Aufgaben ist dies auch notwendig)
- Alle „sinnvollen“ Datenstrukturen und Hilfsmethoden der Standardbibliothek dürfen verwendet werden, sofern Sie sich an die Anforderungen halten. (Keine Reflections/Bytecode-Hacks/externen Bibliotheken,...)

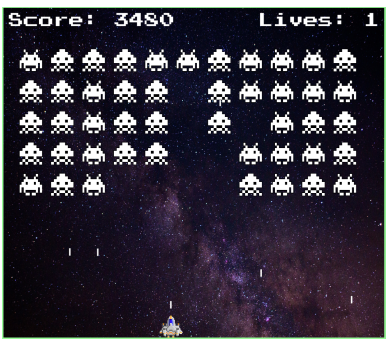


Abbildung 1: Beispielhafte Spielsituation von Space Invaders

## Einleitung

In dieser Hausübung werden Sie eine vereinfachte Version des 2D-Arcade-Klassikers *Space Invaders* entwickeln.

### Don't Panic

Da eine vollständige Implementierung des Spieles den Umfang einer Hausübung deutlich übersteigt (jedenfalls wenn man es *ordentlich* machen will!), werden schon große Teile der Implementierung vorgegeben und das Spielprinzip leicht vereinfacht.

Auch wenn diese Hausübung auf den ersten Blick sehr komplex erscheinen mag, da es viele verschiedene Klassen gibt, die miteinander interagieren, ist es nicht nötig, sich in allen Details zu verlieren. Es reicht, wenn Sie die einzelnen Klassen und ihre Aufgaben grob verstehen und die vorgegebenen Methoden implementieren können.

### Spielprinzip (nicht 100% Originalgetreu)

Der Spieler steuert eine Kanone, die er am unteren Bildschirmrand nach links und rechts fahren kann.

Jede Runde beginnt mit mehreren Reihen regelmäßig angeordneter Aliens, die sich ständig horizontal und dabei nach und nach abwärts bewegen und den Spieler mit Geschossen angreifen.

Der Spieler selbst hat einen unbegrenzten Munitionsvorrat, kann aber erst dann ein neues Geschoss abfeuern, wenn das vorige vom Bildschirm verschwunden ist. (das gilt auch für die Geschosse der Aliens)

Wenn es einem der Aliens gelingt, den unteren Bildschirmrand zu erreichen und neben der Kanone zu landen, ist das Spiel vorbei. (im Original verliert der Spieler nur ein Leben pro Alien, dass den unteren Bildschirmrand erreicht)

### Struktur

Zunächst sollten sie sich etwas mit der Struktur der Vorlage vertraut machen. Das Prinzip MVC (Model View Controller) kennen Sie bereits aus den Folien.

Auf diese Hausübung bezogen haben Model, View und Controller die folgende Aufgaben:

- Model: Die Spielobjekte, die vom Controller verwaltet werden.
- View: Die grafische Darstellung der Spielobjekte.
- Controller: Die Spiellogik, die die Spielobjekte verwaltet.

> **Hinweis:**
>
> Wenn Sie ihre Implementierung testen wollen, bevor Sie alle Aufgaben bearbeitet haben, können Sie Zeile `//Student.setCrashEnabled(false);` in der Klasse `SpaceInvaders` einkommentieren.

## H1: Modellierung der Sprites (Model) ?? Punkte

Zuerst kümmern wir uns um die Modellierung. Diese bietet dann die Grundlage für die Darstellung und die Steuerung der Spielobjekte.

> **Hinweis (**Für die gesamte H1**):**
>
> Die Maße des `GameBoard`s betragen **immer** genau `GameConstants.ORIGINAL_GAME_BOUNDS`. Sie müssen keinerlei „Zoomfaktor“ o.ä. beachten, darum kümmern sich die Klassen `GameBoard` und `GameScene` aus H2. Implementieren Sie also alle Methoden so, dass sie mit diesen Maßen funktionieren.

### H1.1: Klasse Sprite ?? Punkte

Die Klasse `Sprite` bildet die Grundlage für alle Objekte im Spiel. Sie enthält die Position, die Größe und die Geschwindigkeit des Objekts. Außerdem enthält sie die Methoden zum Bewegen des Objekts. Die Klasse `Sprite` ist abstrakt, da sie nicht instanziiert werden soll. Sie wird nur von ihren Unterklassen verwendet. Außerdem nutzt sie Methoden der Klasse `Utils`, die ebenfalls in H1.1 implementiert werden soll.

Implementieren Sie die folgenden Methoden:

- **Methode `damage`**:

  Die Methode `damage` verringert die Lebensanzahl der `Sprite` um die gegebene Menge.

- **Methode `die`**:

  Die Methode `die` setzt die Lebensanzahl der `Sprite` auf 0.

- **Methode `isDead`**:

  Die Methode `isDead` liefert dann `true` zurück, wenn die `Sprite` keine Leben mehr besitzt, sonst `false`.

- **Methode `clamp`**:

  Die Methode `clamp` der Klasse `Utils` soll die gegebene Position so begrenzen, dass die `Sprite` noch vollständig auf das `GameBoard` passen würde. Dazu wird für außerhalb liegende Positionen die am nächsten gelegene Position innerhalb des `GameBoard`s zurückgegeben. Für eine innen liegende Position soll die gegebene Position unverändert zurückgegeben werden. Also würde z.B. der Punkt $(4000, 4000)$ für eine Sprite mit Höhe $10$ und Breite $10$ auf $(245, 213)$ abgebildet werden, wenn das `GameBoard` die Breite $256$ und die Höhe $224$ hat. Hier eine Skizze (mit anderen Werten):


(a) Vor dem Clampen (b) Nach dem Clampen

Abbildung 2: Beispiele für das Clampen

> **Hinweis:**
>
> Als Rückgabe eignet sich z.B. die Klasse `BoundingBox`, welche die abstrakte Klasse `Bounds` implementiert.

- **Methode `getNextPosition`**:

  Die Methode `getNextPosition` der Kasse `Utils` soll die Position zurückgeben, zu der sich die `Sprite` während des nächsten Updates bewegen soll. Dabei gilt für die neue Position $p'$ folgendes:

  $$p' = p + \vec{dir} \cdot v \cdot \Delta t$$

  Hierbei ist $p$ die aktuelle Position, $\vec{dir}$ der aktuelle Richtungsvektor, $v$ die aktuelle Geschwindigkeit und $\Delta t$ die verstrichene Zeit seit dem letzten Aufruf von `update` in Sekunden.

- **Methode `update`**:

  Implementieren sie die aus Interface `Updatable` geerbte Methode `update` so, dass die `Sprite` sich an die nächste Position bewegt. Nutzen Sie dafür die zuvor implementierte Methode `getNextPosition` der Klasse `Utils` sowie die Methoden, `setX` und `setY` der Klasse `Sprite`.

> **Verbindliche Anforderung:**
>
> Eine Sprite muss sich zu **jedem** Zeitpunkt vollständig innerhalb des GameBoards befinden. Benutzen Sie dafür die Methode `clamp`.

## H1.2: Klasse Bullet ?? Punkte

Als nächstes folgt die Klasse `Bullet`. Sie repräsentiert die Schüsse der Spieler und der Gegner. Eine `Bullet` hat ein `BattleShip owner` und eine `Direction direction`.

> **Definition — Hitboxen** Eine Hitbox ist der Bereich einer Sprite, bei dem eine Kollision festgestellt werden kann. In unserem Fall ist die Hitbox eine Rechteck, basierend auf Position, Breite und Höhe der Sprite. (Theoretisch könnte man zwar dafür sorgen, dass die Hitbox genau so groß ist, wie die nicht-transparenten Pixel, allerdings wäre das deutlich aufwendiger...)

Implementieren Sie die folgenden Methoden:

- **Methode `canHit`**:

  Methode `canHit` soll Überprüfen, ob eine Kollision der `Bullet` mit einem gegebenen `BattleShip` stattfinden kann.
  Dabei gelten die folgenden Regeln:

  - `Bullet`s können keine anderen `Bullet`s treffen. (wird durch Parametertyp bereits abgedeckt)
  - Bullets können nur lebende `BattleShip`s treffen, die ihrem Besitzer feindlich gesinnt sind.
  - Eine Kollision findet statt, gdw. sich die Hitboxen der `Bullet` und des `BattleShip`s überschneiden.

> **Hinweise:**
>
> - Ob zwei `BattleShip`s feindlich gesinnt sind, können Sie mit der Methode `isEnemy` von `BattleShip` ermitteln.
> - Um Mehrfachkollisionen zu vermeiden, müssen Sie die bisherigen Treffer zwischenspeichern.
> - Die Hitbox einer Sprite kann man mit der Methode `getBounds` von `Sprite` ermitteln.

- **Methode `hit`**:

  Methode `hit` soll die Kollision der `Bullet` mit einem `BattleShip` behandeln. Dabei sollen sowohl der `BattleShip` als auch die `Bullet` 1 Schaden erleiden. Sie können davon ausgehen, dass die Methode nur aufgerufen wird, wenn eine Kollision möglich ist (siehe `canHit`).

- **Methode `update`**:

  Überschreiben sie die geerbte `update`-Methode so, dass sie zusätzlich zur Bewegung des Super-Aufrufs überprüft, ob sich die `Bullet` gerade aus dem `GameBoard` heraus bewegen würde (ohne clamping). Falls ja, soll die Methode `die` aufgerufen werden.

> **Hinweis:**
>
> Je nach Implementierung müssen Sie möglicherweise die `update`-Methode in `Sprite` anpassen.

## H1.3: Klasse BattleShip ?? Punkte

Die Klasse `BattleShip` ist die `super`-Klasse für `Player` und `Enemy`. Ein `BattleShip` kann schießen und hat Freunde bzw. Feinde.

Implementieren Sie die folgenden Methoden:

- **Methode `shoot`**:

  Methode `shoot` soll eine neue `Bullet` zentriert auf dem `BattleShip` erzeugen. Die erzeugte `Bullet` soll in die übergebene Richtung fliegen. Außerdem wird die `Bullet` per `setBullet` dem `BattleShip` zugewiesen und in die `toAdd`-Liste des `GameState` eingefügt. Wenn das `BattleShip` bereits eine `Bullet` besitzt und der Wert von `ApplicationSettings.instantShootingProperty()` gleich `false` ist, soll die Methode nichts tun. Wenn das `BattleShip` allerdings bereits eine `Bullet` besitzt und der Wert von `ApplicationSettings.instantShootingProperty()` gleich `true` ist, soll die Methode trotzdem eine neue Bullet erzeugen und in die `toAdd`-Liste des `GameState` eingefügt, die alte Bullet wird dabei aber nicht gelöscht.

  > **Anmerkung:**
  >
  > Wenn die `Bullet` nicht in die `toAdd`-Liste, sondern direkt dem `GameState` hinzugefügt werden würde, könnte es zu einer *ConcurrentModificationException* kommen. (Race Condition)

- **Methode `isFriend`**:

  Methode `isFriend` soll überprüfen, ob ein `BattleShip` einem anderen `BattleShip` freundlich gesinnt ist. Dabei gelten die folgenden Regeln:

  - `BattleShip`s sind gegenüber sich selbst freundlich gesinnt.
  - Ein `BattleShip` $b_1$ ist gegenüber einem anderen `BattleShip` $b_2$ freundlich gesinnt, wenn $b_2$ eine Instanz von $b_1$ ist. In unserem Fall müssen Sie dies nur für die Unterklassen `Player` und `Enemy` von `BattleShip` überprüfen. Konkret heißt das, dass `Player` bzw. `Enemy` gegenüber anderen `Player` bzw. `Enemy` freundlich, jedoch gegenseitig feindlich gesinnt sind.

## H1.4: Klasse Enemy ?? Punkte

Klasse `Enemy` repräsentiert die computergesteuerten Gegner („Aliens“). Die Bewegung der `Enemy`-Objekte wird durch die Klasse `EnemyMovement` gesteuert, und das Schussverhalten verwalten die `Enemy`-Objekte selbst.

Implementieren Sie die folgende Methode:

- **Methode `update`**:

  Überschreiben Sie die geerbte `update`-Methode so, dass sie zusätzlich zur Bewegung des Super-Aufrufs zufällig die Methode `shoot` aufruft. Dabei gelten die folgenden Regeln:

  - Der erste Schuss darf erst nach dem in `ApplicationSettings.enemyShootingDelayProperty()` festgelegten Zeitintervall abgegeben werden.
  - Zwischen zwei Schüssen erneut `ApplicationSettings.enemyShootingDelayProperty()` Millisekunden gewartet werden.
  - Wenn ein Schuss möglich ist, beträgt die Wahrscheinlichkeit bei jedem Aufruf der Methode `update` genau den in `ApplicationSettings.enemyShootingProbabilityProperty()` festgelegten Wert (siehe `ApplicationSettings`). Nutzen Sie hierfür die Methode `Math.random()`.
  - Wenn kein Schuss möglich ist, tut die Methode nichts.

## H1.5: Klasse Player ?? Punkte

Die Klasse `Player` repräsentiert den Spieler. Sie ist eine Unterklasse von `BattleShip` und wird direkt vom `PlayerController` gesteuert.

Implementieren Sie die folgende Methode:

- **Methode `update`**:

  Überschreiben sie die geerbte `update`-Methode so, dass sie zusätzlich zur Bewegung des Super-Aufrufs die Methode `shoot` aufruft, falls `isKeepShooting` `true` ist. (ungeachtet `elapsedTime`)

## H1.6: Klasse EnemyMovement ?? Punkte

Die Klasse `EnemyMovement` modelliert die Bewegung der Gegner. Die Gegner bewegen sich immer gemeinsam als eine Gruppe. Dabei bewegen sich die Gegner Schlangenlinienförmig, erst nach rechts, dann nach unten, dann nach links und dann nach unten und dann wieder nach rechts usw. Hier ist ein Beispiel für eine solche Bewegung:

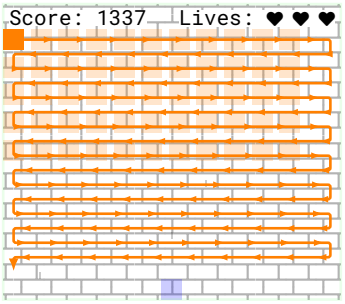


Abbildung 3: Beispielhafte Bewegung der Gegner

> **Hinweis:**
>
> Sie dürfen in ganz `EnemyMovement` davon Ausgehen, dass sich nur lebende Gegner im `GameState` befinden.

Implementieren Sie die folgenden Methoden:

- **Methode `getEnemyBounds`**:

  Methode `getEnemyBounds` soll eine `BoundingBox` zurückgeben, die die Grenzen aller `Enemy`s auf dem Spielfeld beschreibt. Dabei soll die `BoundingBox` so klein wie möglich sein, ohne `Enemy`s zu verdecken.

  > **Hinweis:**
  >
  > Sie können die Methode `getEnemyBounds` überprüfen, indem Sie die berechneten `Bounds` in `GameBoard` (siehe H2.3) zeichnen. Bitte entfernen Sie diesen Code wieder, bevor Sie Ihre Lösung abgeben, da er sonst die Bewertung beeinflussen könnte.

- **Methode `bottomWasReached`**:

  Methode `bottomWasReached` soll überprüfen, ob ein `Enemy` den unteren Rand des Spielfeldes erreicht hat.

- **Methode `nextMovement`**:

  Die Methode `nextMovement` soll die Bewegungsrichtung des `Enemy`s aktualisieren und die Bewegungsgeschwindigkeit um `ENEMY_MOVEMENT_SPEED_INCREASE` erhöhen. Außerdem soll bei einer vertikalen Bewegung das `yTarget` um `VERTICAL_ENEMY_MOVE_DISTANCE` erhöht werden.

- **Methode `targetReached`**:

  Methode `targetReached` soll überprüfen, ob das Ziel einer Bewegung erreicht wurde.

- **Methode `updatePositions`**:

  Methode `updatePositions` soll die Positionen aller `Enemy`s auf dem Spielfeld aktualisieren. Dabei soll die Position um die gegebene Verschiebung verändert werden. (Mittels `setX()` und `setY()`)

- **Methode `update`**:

  Methode `update` soll (sofern der untere Bildschirmrand noch nicht erreicht wurde) den nächsten Bewegungsschritt durchführen. Hierfür sollten Sie zunächst die nächste Position mit der Methode `getNextPosition` der Klasse `Utils` berechnen. Für den Parameter `bounds` können Sie ihre zuvor implementierte Methode `getEnemyBounds` verwenden. Anschließend müssen sie mittels der Methode `targetReached` abfragen, ob die aktuelle Bewegung ihr Ziel erreicht hat. Falls ja soll die Bewegungsrichtung mit der Methode `nextMovement` geändert werden. Anschließend soll die Position aller Gegner mit der Methode `updatePositions` aktualisiert werden (unabhängig davon, ob die Bewegung ihr Ziel erreicht hat oder nicht).

  > **Anmerkung:**
  >
  > Die Gegner bewegen sich immer als eine Einheit, das heißt alle Gegner bewegen sich gleichzeitig gleich schnell in die gleiche Richtung. Beachten Sie auch, dass Sie die Position wieder „clampen“ müssen, damit die Gegner nicht aus dem Spielfeld herauslaufen.

> **Unbewertete Verständnisfrage:**
>
> Wie wird während der Ausführung tatsächlich sichergestellt, dass sich nur lebende Gegner im `GameState` befinden?

## H2: Game Scene und Rendering (View) ?? Punkte

In dieser Aufgabe werden Sie alle Vorbereitungen treffen, um das Spiel in der `GameScene` darstellen zu können. Das wird ihnen ermöglichen, Änderungen an der Spiellogik direkt im Spiel zu testen.

**Verbindliche Anforderungen** (Für die gesamte H2)**:**

Jede Zeichenmethode, die die Einstellungen des übergebenen `GraphicsContext` verändert (Linienstärke, Zeichenfarbe, Schriftart, Transformation, ...) stellt die Originaleinstellungen von vor dem Methodenaufruf nach Abschluss ihrer eigentlichen Funktionalität wieder her.

### H2.1: GameScene ?? Punkte

Die Klasse `GameScene` stellt das `GameBoard` dar und kümmert sich um dessen Größe.

Implementieren Sie die folgende Methode:

- **Methode `initGameBoard`**:

  Implementieren Sie die Methode `initGameBoard` in `GameScene` so, dass stets die folgenden Bedingungen eingehalten werden (auch wenn sich die Fenstergröße ändert):

  – das Gameboard füllt die ganze `GameScene` maximal, wobei das Seitenverhältnis beibehalten wird und kein Pixel außerhalb der `GameScene` liegt.

  – das Gameboard ist horizontal und vertikal zentriert in der `GameScene`.

  Korrekte (a-c) und fehlerhafte (d) Beispiele:

(a) breitere Szene (b) höhere Szene (c) gleiches Seitenverhältnis (d) falsch: nicht maximal ausgefüllt

Abbildung 4: Skalierung+Zentrierung mit festem Seitenverhältnis. Das äußere Rechteck stellt die Größe der Szene dar, das gestrichelte Rechteck die berechnete Größe des Boards.

**Hinweise:**

– Sie können die in der Vorlesung vorgestellten `Bindings` verwenden.

– Die Klassen `GameScene` und `GameBoard` haben die Methoden `widthProperty()` und `heightProperty()`

– Die Klasse `GameBoard` hat die Methoden `translateXProperty()` und `translateYProperty()`, welche zur Positionierung des Boards verwendet werden können.

## H2.2: Sprite Renderer ?? Punkte

Die Klasse `SpriteRenderer` ist für das Zeichnen der `Sprite`-Objekte zuständig. Die Klasse soll niemals instanziiert werden, sondern nur statisch verwendet werden.

Implementieren Sie die folgende Methode:

- **Methode `renderSprite`**:

  Implementieren sie die statische Methode `renderSprite` der Klasse `SpriteRenderer`. Die Methode soll eine gegebene `Sprite` in dem gegebenen `GraphicsContext` zeichnen. Dabei müssen die Position und Größe der gegebenen `Sprite` beachtet werden. Zum Zeichnen betrachten Sie die folgenden Fälle:

  – Falls die gegebene Sprite eine Textur besitzt, soll diese mittels der Methode `drawImage` des gegebenen `GraphicsContext` gezeichnet werden.

  – Falls nicht, soll ein Rechteck in der entsprechenden Farbe und Fläche der Sprite mittels der Methode `setFill` des gegebenen `GraphicsContext` gefüllt werden.

## H2.3: GameBoard ?? Punkte

In der Klasse `GameBoard` ist bereits die Methode `update` implementiert, welche einen `GraphicsContext` erzeugt, der entsprechend der Größe des `GameBoard` skaliert ist und die einzelnen `draw`-Methoden in der korrekten Reihenfolge aufruft. Sie können also davon ausgehen, dass der übergebene `GraphicsContext` immer die Größe des Originalspielfeldes hat (also `GameConstants.ORIGINAL_GAME_BOUNDS`) und das kompletteGameBoard repräsentiert.

Hier eine Skizze dessen, was das Gameboard alles Zeichnen muss:

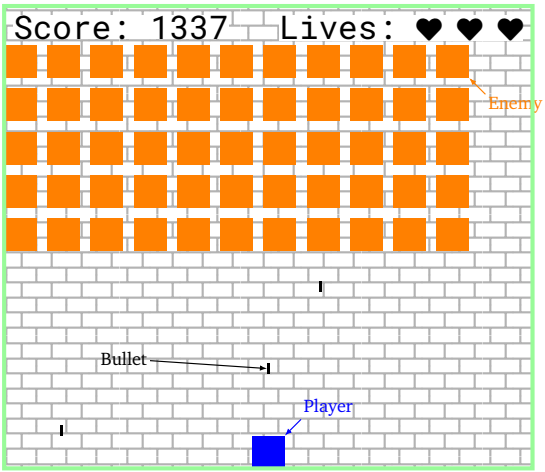


Abbildung 5: Skizze des `GameBoard`

Implementieren Sie die folgenden Methoden:

- **Methode `drawBackground`**:

  Zeichnet den Hintergrund des `GameBoard` in der entsprechenden Farbe. Falls das `backgroundImage` gesetzt ist, soll dieses mittels der Methode `drawImage` des gegebenen `GraphicsContext` gezeichnet werden. Falls nicht, soll das komplette `GameBoard` mit der methode `clearRect` des gegebenen `GraphicsContext` zurückgesetzt werden.

- **Methode `drawSprites`**:

  Zeichnet alle `Sprites` des `GameController` in der folgenden Reihenfolge (also von hinten nach vorne) mittels der Methode `renderSprite` der Klasse `SpriteRenderer`:

  1. `Bullets`
  2. `Enemys`
  3. den `Player`
  4. Alle anderen `Sprites`, die nicht `Bullets`, `Enemys` oder der `Player` sind.

  > **Hinweis:**
  >
  > Mit dem Aufruf `getGameController().getGameState().getSprites()` in `GameBoard` erhalten Sie eine Liste aller `Sprites` im Spiel.

- **Methode `drawHUD`**:

  > **Definition — Heads Up Display (HUD)** Ein Heads Up Display (HUD) ist ein grafisches Element, das Informationen über den Spielverlauf anzeigt. In unserem Fall soll das HUD die Anzahl der Leben des Spielers und die erreichte Punktzahl anzeigen.

  Zeichnen Sie die folgenden Texte auf den GraphicsContext mit der Methode `fillText`:

  – Die Punktzahl in der oberen linken Ecke in dem Format: `"Score: <punktzahl>"`

  – Die Leben sollen oben rechts in der Ecke angezeigt werden, in dem Format: `"Lives: <lebensanzahl>"`

  Hierbei soll nach dem Doppelpunkt also genau **ein** Leerzeichen stehen, und der mit <> markierte Text durch seinen Wert ersetzt werden (ohne <>). Also z.B. `"Lives: 3"` oder `"Score: 1337"`. Außerdem verwenden Sie die Schriftart `GameConstants.HUD_FONT` und halten jeweils `GameConstants.HUD_PADDING` Abstand von der entsprechenden Ecke des `GameBoards`.

  > **Hinweis:**
  >
  > Sie können die Klasse `Text` nutzen, um die Maße des Textes festzustellen.

- **Methode `drawBorder`**:

  Die Methode `drawBorder` soll einen Rahmen um das komplette `GameBoard` mittels der Methode `strokeRect` in der in den `GameConstants` vorgegebenen Breite und Farbe zeichnen. (Dabei ist die Border halb innen und halb außen um das `GameBoard` zu zeichnen.)

## H3: Spiellogik (Controller) ?? Punkte

Jetzt kommt alles zusammen: Im Controller sollen die Spiellogik implementiert werden, Eingaben verarbeitet werden und der „game loop“ verwaltet werden.

**Definition — Game Loop** Als „Game Loop“ wird die Hauptschleife eines Spiels bezeichnet. Sie wird so lange ausgeführt, bis das Spiel beendet wird. In ihr werden alle Aktionen ausgeführt, die das Spiel ausmachen. Dazu gehören das Zeichnen der Objekte, das Verarbeiten der Eingaben und das Berechnen der Spiellogik. In unserem Fall wird der „Game Loop“ durch einen `AnimationTimer` verwaltet. Das heißt, dass die Methode `handle` jedes Mal aufgerufen wird, wenn ein neues Bild gezeichnet werden soll.

### H3.1: GameController ?? Punkte

Die Klasse `GameController` ist die Zentrale Steuerung des Spiels. Sie ist für das Erzeugen der anderen Controller zuständig und verwaltet die Spiellogik (also z.B. das Spielende, die Punkte, etc.).

Implementieren Sie die folgenden Methoden:

- **Methode `doCollisions`**:

  Methode `doCollisions` soll die Kollisionen zwischen den `Bullets` und den `BattleShips` behandeln. Dazu muss die Liste `sprites` aus `GameState` durchlaufen werden. Für jede `Bullet` soll durch die Methode `canHit` aus `Bullet` mit jedem `BattleShip` geprüft werden, ob eine Kollision möglich ist. Falls ja, soll die Methode `hit` der `Bullet` entsprechend aufgerufen werden. Sie dürfen davon ausgehen, dass sich niemals zwei Gegner überlappen werden.

- **Methode `updatePoints`**:

  Methode `updatePoints` soll die Punktzahl des `Players` aktualisieren. Dabei sollen alle getroffenen Gegner der übergebenen Liste `damaged` durchlaufen werden. Für jeden besiegten Gegner soll die Punktzahl des `Players` um den Wert des Attributs `pointsWorth` dieses Gegners erhöht werden.

- **Methode `handleKeyboardInputs`**:

  Methode `handleKeyboardInputs` soll die Tastenkombinationen für die Steuerung der `GameScene` verarbeiten. Dabei soll die Klasse `GameInputHandler` verwendet werden.

  Wenn die Taste `ESCAPE` gedrückt wird, soll das Spiel pausiert werden, und der Spieler gefragt werden, ob er aufgeben möchte. Falls er aufgeben möchte, soll die Methode `lose` aufgerufen werden. Ansonsten soll das Spiel fortgesetzt werden.

  Wenn die Taste `F11` gedrückt wird, soll der Vollbildmodus umgeschaltet werden.

- **Methode `lose`**:

  Methode `lose` soll dem Spieler anzeigen, dass er verloren hat, und eine Möglichkeit geben, unter Eingabe eines Spielernamens die Punkte in die Highscoreliste `highscores` aus `ApplicationSettungs` einzutragen. Falls der Spieler keinen Namen eingibt, soll der Name `Anonymous` verwendet werden.

  Anschließend wird dem Spieler angeboten, das Spiel neu zu starten. Falls er dies möchte, soll die Methode `reset` aufgerufen werden. Ansonsten soll das Spiel beendet werden und der Spieler zum Hauptmenü zurückkehren.

  **Hinweis:**

  Da die Implementierung eigener Dialoge in JavaFX sehr aufwendig ist, empfehlen wir die Verwendung der Klasse `TextInputDialog` aus dem Paket `javafx.scene.control` sowie der Klasse `Alert` aus dem Paket `javafx.scene.control` für die Erstellung von Dialogen.

## H3.2: PlayerController ?? Punkte

Die Klasse `PlayerController` verwaltet das `Player`-Objekt und ist für die Steuerung des Spielers zuständig.

Implementieren Sie die folgende Methode:

- **Methode `playerKeyAction`**:

  Methode `playerKeyAction` soll die Tastenkombinationen für die Steuerung des `Player`s verarbeiten. Dabei soll die Klasse `GameInputHandler` verwendet werden. Wir unterscheiden zwischen den Richtungstasten `A`, `D`, `LEFT` und `RIGHT` und der Schießtaste `SPACE`.

  Ein Spieler kann sich nur horizontal bewegen und schießen. Dafür muss er die folgenden Tasten drücken:

  - `A` oder `LEFT`: Der Spieler bewegt sich nach links.
  - `D` oder `RIGHT`: Der Spieler bewegt sich nach rechts.
  - Keine der beiden Tasten, oder zwei gegensätzliche Richtungen (z.B. sowohl `A` als auch `D` oder `LEFT` und `RIGHT`): Der Spieler bewegt sich nicht, ungeachtet anderer Richtungstasten.
  - `SPACE`: Das Attribut `keepShooting` des `Player`s wird auf `true` gesetzt, solange die Taste gedrückt wird.

  > **Hinweis:**
  >
  > Sie können die Methode `combine` des Enums `Direction` verwenden, um mehrere Richtungseingaben zu kombinieren.

## H3.3: EnemyController ?? Punkte

Die Klasse `EnemyController` initialisiert die Gegner und verwaltet sie. Außerdem ist sie dafür zuständig, neue Gegner zu erzeugen, wenn alle Gegner getötet wurden.

Implementieren Sie die folgende Methode:

- **Methode `isDefeated`**:

  Methode `isDefeated` soll genau dann `true` zurückgeben, wenn keine lebenden Gegner mehr vorhanden sind.

## H4: Einstellungsmenü ?? Punkte

Als letztes wollen wir noch ein Wenig mit „klassischem JavaFX“ arbeiten. In der Klasse `SettingsScene` soll ein Einstellungsmenü erstellt werden.

Implementieren Sie die folgende Methode:

- **Methode `init`**:

  Methode `init` soll die `SettingsScene` initialisieren und alle benötigten Elemente dem `TabPane` `contentRoot` hinzufügen. Überlegen Sie sich, welche Elemente Sie benötigen und wie Sie diese hinzufügen können. Sorgen Sie außerdem im `SettingsController` dafür, dass das alle Änderungen in der `SettingsScene` auch in `ApplicationSettings` übernommen werden, und umgekehrt (hierfür können Sie `bindBidirectional` verwenden). Pro zwei korrekt einstellbaren Werten gibt es einen Punkt. Die folgenden Werte sollen einstellbar sein:

  - `ApplicationSettings.instantShootingProperty()`: Sofortiges Schießen des Players (Checkbox)
  - `ApplicationSettings.enemyShootingDelayProperty()`: Minimales Schussintervall der Gegner (Slider)
  - `ApplicationSettings.enemyShootingProbabilityProperty()`: Schusswahrscheinlichkeit der Gegner (Slider)
  - `ApplicationSettings.fullscreenProperty()`: Spiel im Vollbildmodus starten (Checkbox)
  - `ApplicationSettings.loadTexturesProperty()`: Texturen der Sprites laden (Checkbox)
  - `ApplicationSettings.loadBackgroundProperty()`: Hintergrund laden (Checkbox)

  Sie sind hierbei in der Gestaltung der `SettingsScene` bis auf diese Anforderungen frei.