**Frogger**

**Prüfungsstudienarbeit zur Vorlesung „Java Programmierung“**Prüfer: Prof. Dr. Andreas Berl

**Name: Tobias Hassebrock**

**Matrikelnummer: 00743731**

**Email:** [**tobias.hassebrock@stud.th-deg.de**](mailto:tobias.hassebrock@stud.th-deg.de)**,**

**Studiengang: Angewandte Volkswirtschaftslehre**

**Semester:** SS 2020

**Datum der Abgabe:** 28.06.2020

**Erreichte Punktezahl:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Note:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Erstkorrektur Unterschrift:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Zweitkorrektur Unterschrift:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Inhalt

[Kurzbeschreibung des Spiels 2](#_Toc42073663)

[Implementierung des Spiels 4](#_Toc42073664)

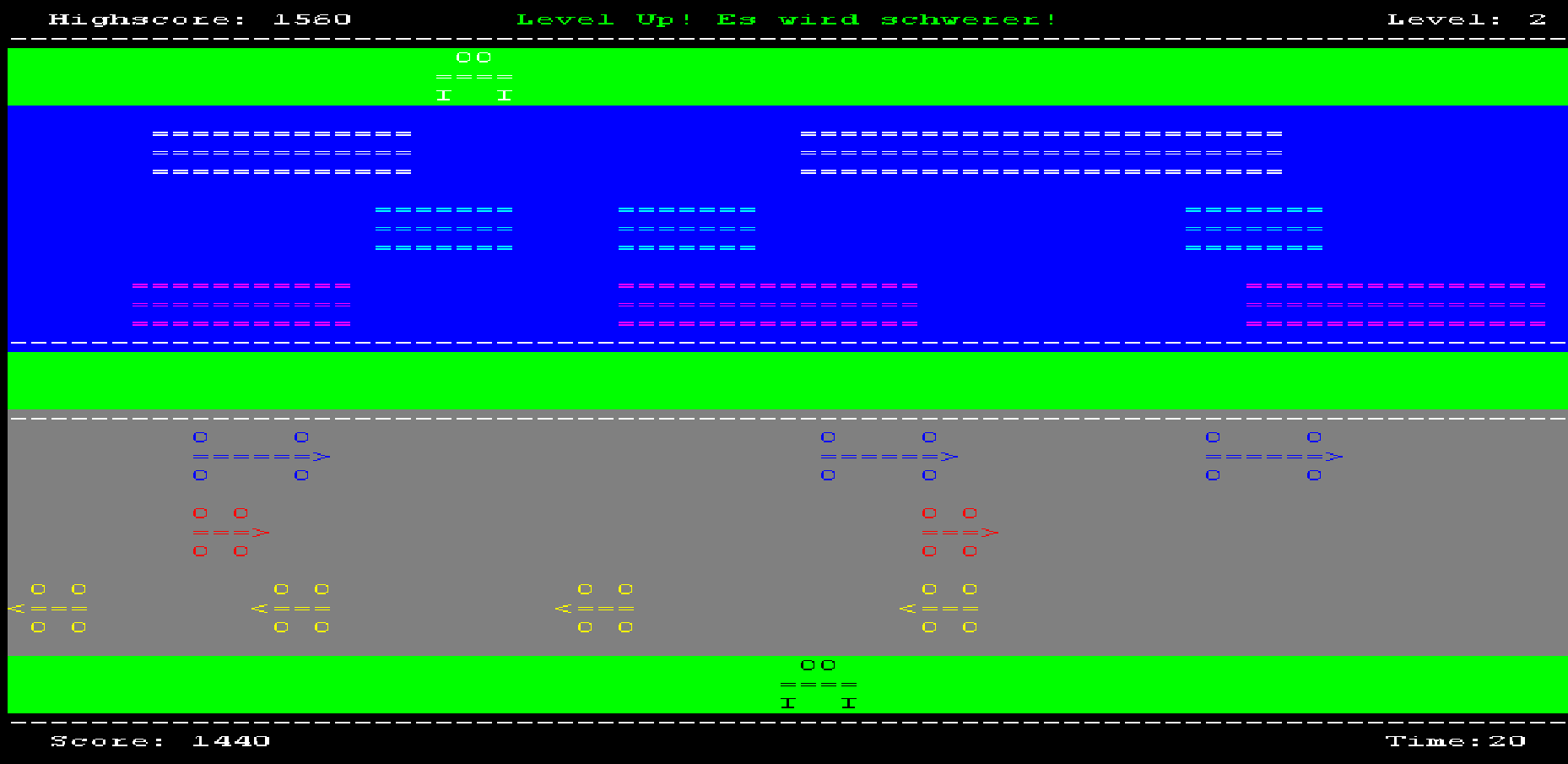
[Statistiken 9](#_Toc42073665)

[Beschreibung einer besonderen Herausforderung bei der Umsetzung des Spiels 10](#_Toc42073666)

# Kurzbeschreibung des Spiels

Im Spiel „Frogger“ steuert der Spieler einen Frosch auf einem Spielfeld, dass sich aus einer von Autos befahrenden Straße, einen Fluss mit darin treibenden Baumstämmen und einem Grünstreifen zwischen Fluss und Straße, auf dem sich der Frosch ausruhen kann, besteht. Das Ziel des Spielers ist, mit dem Frosch auf die andere Seite der Straße und des Flusses zu gelangen. Dazu muss der Frosch zuerst den Autos auf der Straße ausweichen und anschließend von Baumstamm zu Baumstamm springen, ohne dabei in den Fluss zu fallen. Der Frosch kann dazu in alle vier Himmelsrichtungen hüpfen. Zu beachten ist, dass sich der Frosch gemeinsam mit dem Baumstamm bewegt, wenn der Frosch auf diesem landet. Schafft es der Spieler einen Frosch auf die andere Seite zu bringen, erreicht er ein neues Level. In diesem startet der Spieler mit einem weiteren Frosch von der Startposition auf der unteren Seite des Spielfelds. Gleichzeitig bewegen sich die Hindernisse (Autos und Baumstämme) im neunen Level schneller. Der Schwierigkeitsgrad des Spiels wird zusätzlich durch eine Reduzierung der Länge der Baumstämme, während sich gleichzeitig die Länge der Autos erhöht, variiert. Wird der Frosch von einem Auto überfahren, fällt in den Fluss oder verlässt das Spielfeld auf der linken oder rechten Seite (z.B. in dem er zulange auf einem Baumstamm mitfährt) ist das Spiel verloren. Zudem gibt es verschiedene Soundtracks für das Spiel selbst, das Menu und verschiedene Spielereignisse (z.B. das Erreichen der anderen Seite oder Unfälle).

Abb. 1: Ansicht des Spielfelds mit Fluss, Straße und Hindernissen sowie verschiedenen Statistikzählern. Der Spieler hat mit seinem Frosch (unten Mitte) gerade ein neues Level erreicht und startet von der Startposition neu (Quelle: Eigene Programmierung Frogger)

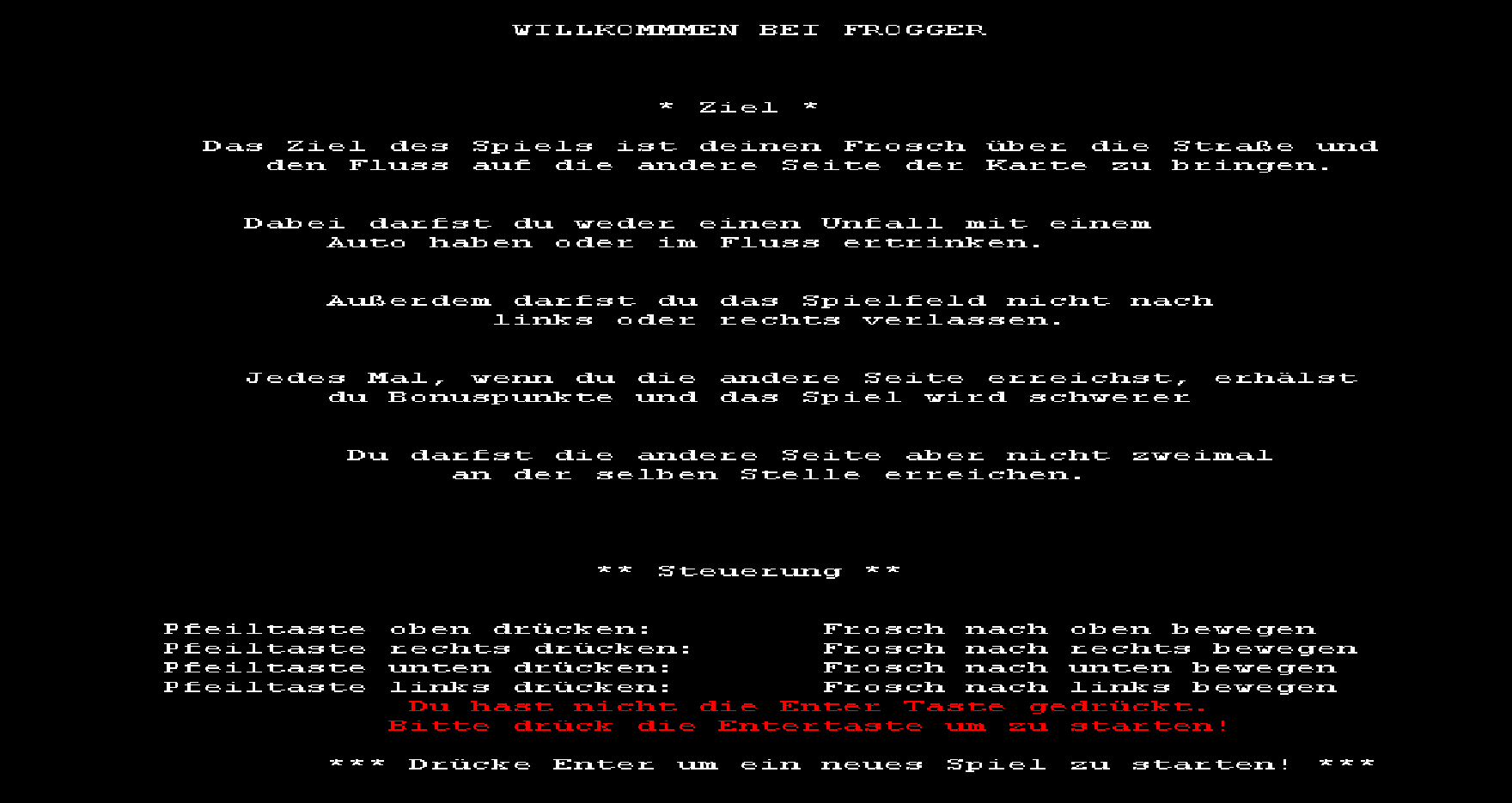


Für zurückgelegte Meter aller Frösche erhält der Spieler zudem Punkte gutgeschrieben. Im Hintergrund des Spielfeldes wird der aktuelle Score, der bisherige beste Highscore, das aktuelle Level und die vergange Zeit ausgegeben. Diese Features sind sowohl in historischen Verionen von Frogger als auch in der Prüfungsstudienarbeit vollständig implementiert. In den historischen Versionen des Spiels werden zudem Bonuspunkte für das Erreichen der anderen Seite vergeben. Die Version der Studienarbeit hingegen verteilt keine Bonuspunkte, sondern erhöht die vergebene Punktzahl für jeden zurückgelegten Meter Strecke im nächsten Level. Im Vergleich zur historischen Version zählt die Zeit zudem nicht runter, sondern hoch, sodass der Spieler über kein Zeitlimit verfügt.

Weiterhin müssen in der historischen Verion das Spiel die Frösche meist in eine spezielle Box auf der anderen Seite des Flusses gesteuert werden, während im vorliegenden Programm der zu erreichende Abschnitt des sicheren Ufers nicht spezifiert ist. Dies wird jedoch dadurch eingeschränkt, dass keine zwei Frösche am selben Abschnitt ankommen dürfen.

In der historischen Version gibt es auf dem Weg zudem einige Tiere als Hindernisse. So schwirren auf einigen Autos oder Grünstreifen z.B. tödliche Insekten, während in einigen Zielboxen oder im Fluss Krokodile schwimmen können, denen ausgewichen werden muss. Dieses Feature wurde als einziges Feature der historischen Version in dieser Studienarbeit nicht implementiert. Hingegen werden zusätzlich im Vergleich zum historischen Spiel die Regeln und die Steuerung zu Beginn erklärt. Zudem erhält der Spieler ein visuelles Feedback, falls eine falsche Taste gedrückt wurde.

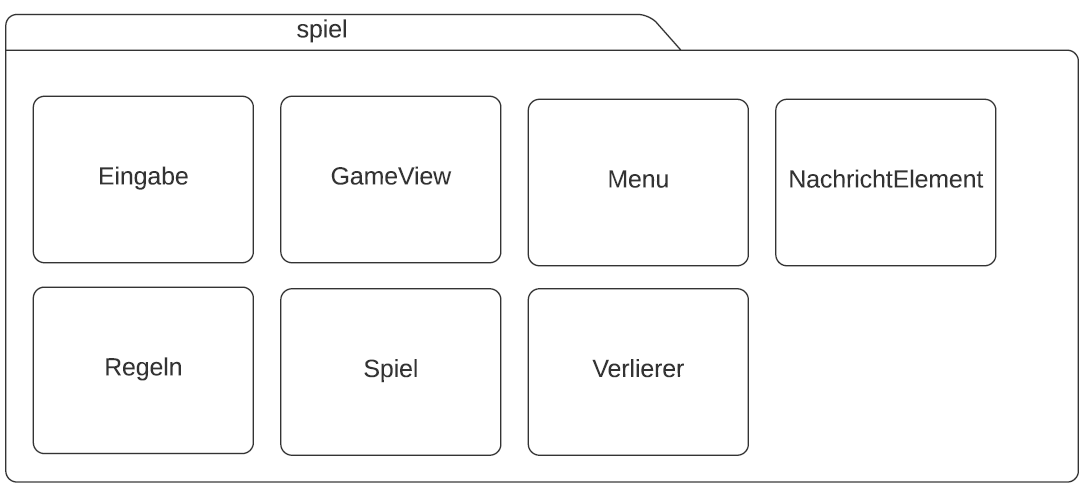
Abb. 2: Startbildschirm mit Regeln und Steuerungshinweisen über das Spiel. Der Spieler erhält hier einen dynamischen Hinweis, dass er die falsche Taste zum Starten des Spiels gedrückt hat (Quelle: Eigene Programmierung Frogger)



# Implementierung des Spiels

Die grundsätzliche Struktur des Programms wird durch die Package Struktur wiedergespiegelt. Im Package „Spiel“ befinden sich die Klassen der Spielsteuerung. Die „Eingabe-Klasse“ verarbeitet Benutzereingaben auf Basis der „Gameview-Klasse“, welche wiederum das grundsätzliche Frameworks des Spiels vorgibt. Die „Menu-Klasse“ gibt den Startbildschirm aus, startet neue Spiele und erklärt gemeinsam mit der „Regel-Klasse“ das Spiel. Die „Spiel-Klasse“ steuert den eigentlichen Spielablauf und beendet das Spiel auch wieder. Dazu verwendet es die Klasse „Verlierer“, welche verschiedene Abbruchbedingungen, bei deren Erfüllung das Spiel verloren wäre, testet. Das „Nachrichtenelement“ behandelt gemeinsam mit der „Eingabe-Klasse“ falsche Nutzereingaben und gibt dem Spieler visuelles Feedback bei falschen Eingaben oder Spielereignissen.

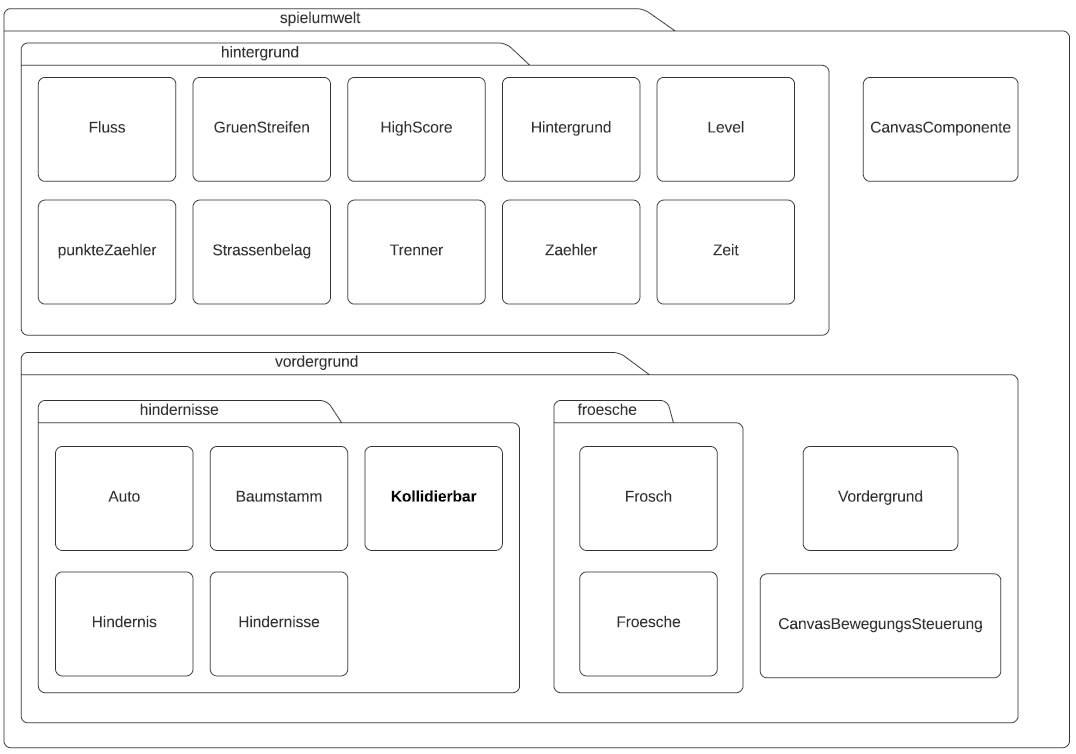
Abb. 3: Das Package „Spiel“ (Quelle: Eigene Programmierung Frogger)



Das Package „Spielumwelt“ enthält mehrheitlich ausgebbare Klassen, welche für den Nutzer im Fenster sichbar sind. Im Vergleich zum „Spiel-Package“ sind hier somit UI-Elemente zusammengefasst, welche die Spielumwelt für die Steuerungsklasse „spiel.Spiel“ bilden. Die Spielumwelt ist wiederum aufgeteilt in ein Package „Hintergrund“, welches die User-Interface-Elemente (UI-Elemente) des Hintergrunds (z.B. Fluss) und verschiedene Zähler enthält (z.B. Highscore). Der „Vordergrund“ enthält zum einen das „Hindernisse-Package“ mit den Klassen „Hindernis“, „Baumstamm“ und „Auto“ für die Modellierung der Hindernisse sowie die zusätzliche Steuerungsklasse „Hindernisse“. Zum anderen enthält der „Vordergrund“ das „Froesche-Package“, welches die Klassen des Frosches enthält.

Die Klassen „Vordergrund“ und „Hintergrund“ sind hingegen zusammenfassende Klassen, die die verschiedenen Instanzierungen der UI Klasssen zusammenfügen, sodass in diesen nach dem Bausteinprinzip variable Spielfelder erstellt werden können.

Abb. 4: Das Package „Spielumwelt“ (Quelle: Eigene Programmierung Frogger)

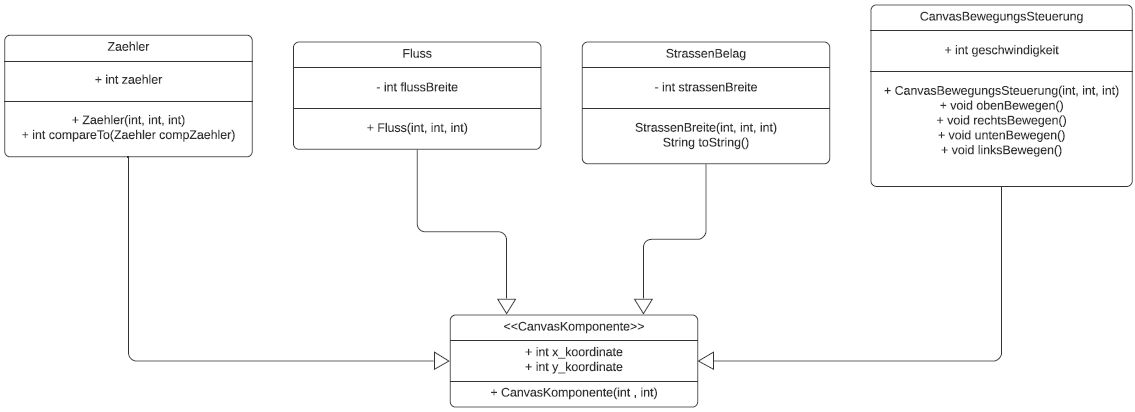


Abgerundet wird das Programm durch eine „Main-Klasse“, welche das Programm startet und ein „Ressourcen-Package“, welches verschiedene Sounds und Icons enthält.

Bei Betrachtung der Software-Architektur wird zudem deutlich, dass das „Spielumwelt-Package“ vom Paradigma der Vererbung geprägt wird und das „Spiel-Package“ vom Kompositionsparadigma geprägt wird.

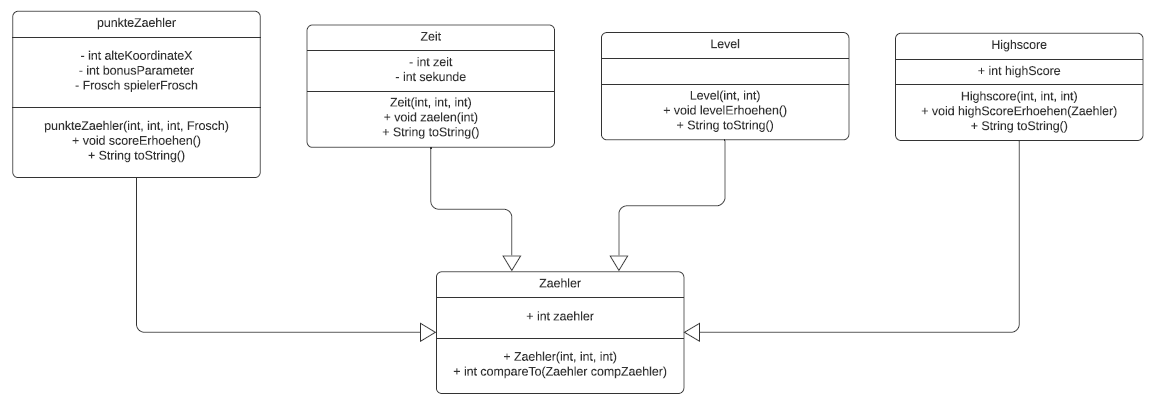
Der Grund dafür ist, dass jedes ausgebbare UI-Element von einer abstrakten Basisklasse „CanvasKomponente“ abstammt, dass den UI Elementen Koordinanten vererbt. Im nächsten Schritt entstehen reine UI-Elemente, die keine Daten verarbeiten und nur ausgegeben werden (z.B. der Straßenbelag). Zudem entsteht aus der „CanvasKomponente“ mit der „Zählerklasse“ und der „CanvasBewegungsSteuerung“ zwei weitere Basisklassen.

Abb. 4: Vererbung aus der Basisklasse „CanvasKomponente“ (Quelle: Eigene Programmierung Frogger)



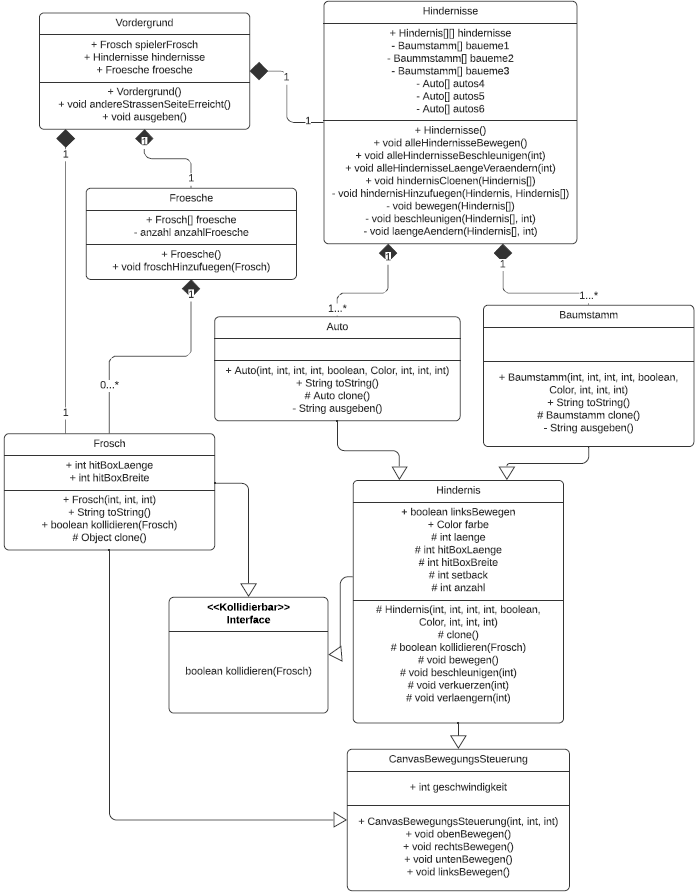
Die Basisklasse „Zaehler“ vererbt eine Zählervariable an verschiedene Zähler (z.B. „Zeit, „Highscore“) und stellt eine Methode zum vergleichen verschiedener Zähler zur Verfügung.

Abb. 5: Vererbung der Zaehler Klasse (Quelle: Eigene Programmierung Frogger)



Die Klasse „CanvasBewegungsSteuerung“ ermöglicht es von ihr abstammenden Klassen sich mit einer variablen Geschwindigkeit in alle Richtungen zu bewegen. Die „Hindernis“ und die „Frosch-Klasse“ erben dies und geben den UI-Komponenten zusätzlich die Eigenschaften für das Hitbox-System des Spiels (detailiertere Erklärung im 3. Kapitel). Die „Hindernis-Klasse“ ergänzt zusätzlich verschiedene Methoden, welche es der „Spiel-Klasse“ ermöglichen bei einem Level up das Spiel schwieriger zu gestalten (indem sich z.B. die Hindernisse schneller bewegen). Von dieser „Hindernis-Klasse“ stammen schließlich auch die Klassen „Auto“ und „Baumstamm“ ab, welche bei der Spielbereitstellung geclont werden. Zudem stellt das Interface „Kollidieren“ sicher, dass die Hindernisse und Frösche die „kollidieren()“ Funktion implementieren, sodass ein Kollision evaluiert werden kann.

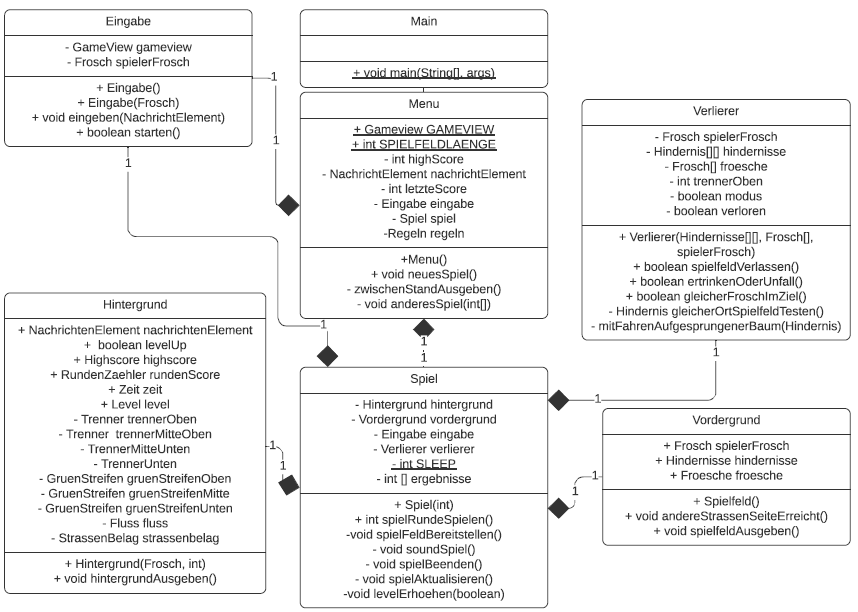
Abb. 6: Vererbung der Klasse „CanvasBewegungsSteuerung“ und die daraus folgende Zusammensetzung des Vordergrunds (Quelle: Eigene Programmierung Frogger)



Aus dieser Abbildung wird auch ersichtlich, dass die „Vordergrund-Klasse“ die verschiedenen UI-Elemente in einer Komposition zusammenfasst. Dasselbe Prinzip gilt für die „Hintergrund-Klasse“.

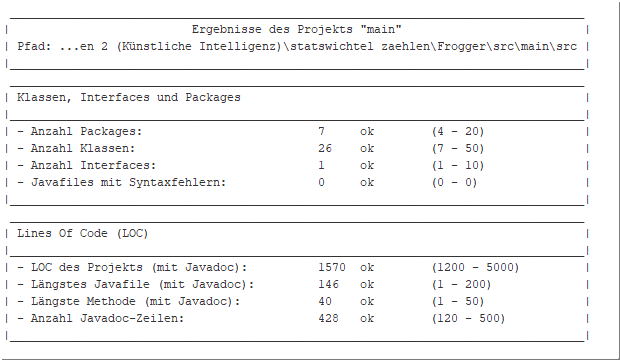
Das Package „Spiel“ beinhaltet die Spielsteuerung. Hier ist besonders auffällig, dass die Klasse „Spiel“ eine Komposition der Spielumgebung mit Hinter- und Vordergrund sowie den Klassen „Eingabe“ und „Verlierer“ ist.

Abb. 7 Spielkomposition: Die zentrale Steuerungklasse „Spiel“ und ihre Hilfsklassen kombinieren Vordergrund und Hintergrund und steuert das Spiel. (Quelle: Eigene Programmierung Frogger)



Die Klasse „Spiel“ steuert „Frogger“ über die Funktion „spielRundeSpielen()“. Diese koordiniert den Spielablauf. Dazu stellt sie zuerst das Spielfeld bereit und aktiviert Sound-Ausgaben. Anschließend führt sie das Spiel auf Rundenbasis durch. Dabei werden die Hindernisse des Vordergrunds bewegt. Dies beinhaltet auch das Aufrufen der Eingabefunktionen der „Eingabe-Klasse“ und die daraus resultierende Bewegung des Frosches. Es werden zudem die verschiedenen Verlierer-Bedingungen der „Verlierer-Klasse“ geprüft sowie durch Aufrufen der „Vordergrund.andereStrassenSeiteErreicht()“ Funktion getestet, ob ein Frosch die andere Straßenseite erreicht hat. Abhängig von den Ergebnissen dieser Prüfungen werden anschließend die Zähler des Hintergrunds aktualisiert oder das Spiel beendet und zum Menu zurückgekehrt. Die „Spiel-Klasse“ ist also der funktionale Taktgeber des Spiels, während ein Großteil der konkreten Prüfungen und der Darstellungen in Hilfsklassen ausgelagert werden, welche dann in der „Spiel-Klasse“ instanziert werden.

# Statistiken



|  |  |
| --- | --- |
| **Umsetzung von Inhalten der Vorlesung** | **Wie/Wo/Warum eingesetzt?** |
| Selbst geschriebene Interfaces | * Verwendet um die Implementierung einer „kollidieren()“ Funktion bei Froeschen und Hindernissen sicherzustellen * Z.B. „ Kollidierbar“ |
| Exceptions | * Verwendet um falsche Benutzereingaben zu behandeln und dem Benutzer ein visuelles Feedback zurückzugeben * Z.B. „eingeben()“ und „starten()“ in der „Eingabe-Klasse“ |
| toString() | * Verwendet in jeder Top Level-Klasse im Package Spielumwelt, die ein UI-Element darstellt * Gibt ASCII Zeichen als UI-Elemente aus * Z.B. „toString()“ in den Klassen „StrassenBelag“, „Auto“ |
| equals() | * Hätte genutzt werden könnnen, um festzustellen, ob sich Frosch und Hindernis am selben Ort befinden * Aufgrund der Komplexitität der Prüfung wurde sich dazu entschieden, ein eigenes Interface „Kollidierbar“ statt der „equal()“ Funktion anzulegen |
| clone() | * Verwendet in der „Auto- und Frosch-Klasse“, um identische UI-Objekte des Vordergrunds mit variabler Anzahl zu erstellen * Erstellt ein identisches Objekt mit identischen Eigenschaften des betrachteten Autos oder Froschs und ermöglicht gleichzeitg ein sicheres Manipulieren der x-Koordinate jedes identischen Objektes * Z.B. „clone()“ in der Frosch- oder Auto-Klasse |
| Nutzung von Comparator oder Comparable | * Verwendet in der „Zaehler-Klasse“ um zu prüfen, ob bei Abschluss des Spiels eine neue Highscore erreicht wurde * Berechnet die Differenz von zwei Zaehlern. Ist diese positiv, ist eine neue HighScore erreicht * Z.B. „compareTo(Zaehler)“ in der „Zaehler-Klasse“ |
| StringBuilder | * Verwendet in jeder Top Level-Klasse im Package Spielumwelt, die ein UI-Element darstellt * Stellt ASCII Zeichen zu ausgebbaren UI-Elementen zusammen * Z.B. „toString()“ in den Klassen „StrassenBelag“, „Auto“ |

# Beschreibung einer besonderen Herausforderung bei der Umsetzung des Spiels

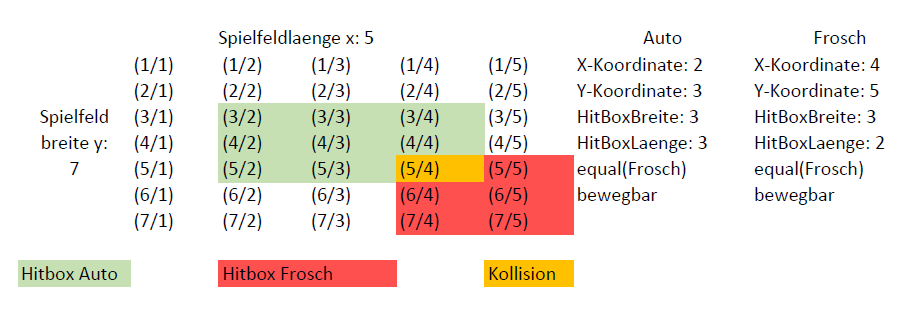
Ein Algorihmus testet in jeder Spielrunde, ob sich der gesteuerte Frosch und ein Hindernis (Auto oder Baumstamm) in derselben Position befindet.

Die Vorraussetzung dafür bildet ein Hitboxsystem. Jedes bewegbare Element des Vordergrunds verfügt über eine x-Koordinate, eine Y-Koordinate, eine Hitboxbreite und eine Hitboxlänge. Dadurch entsteht eine 2D-Fläche, die sich ausgehend von der X-Koordinate nach links und ausgehend von der Y-Koordinate nach unten erstreckt. Diese genau definierbare Fläche lässt sich somit mit einer Matrix beschreiben, die sich innerhalb der 2D Fläche des gesamten Spielfeldes bewegts, welches sich wiederum als eine Matrix beschreiben lässt.

Zudem sind die kollabierbaren Objekte des Vordergrunds bewegbar und können sich somit im Spielfeld bewegen. Außerdem verfügen sie über das Interface „Kollidieren()“, wodurch das Überschreiben der Funktion kollidieren() sichergestellt wird, in der der eigentlichen Ortstest vorgenommen werden soll.

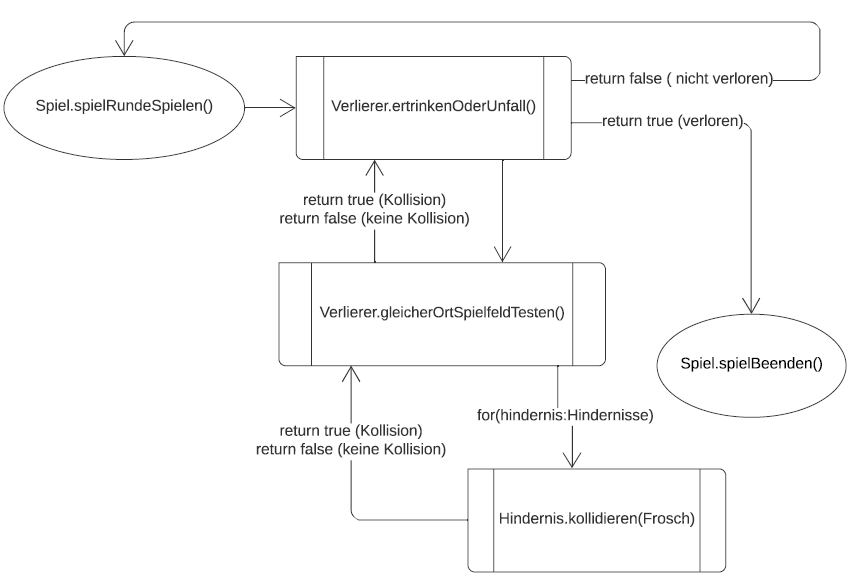
Basierend auf diesem System wird in jeder Runde getestet, ob Kollisionen vorliegen. Das System wird insbesondere dadurch komplexer, dass sich die Gewinnbedingung abhängig von der Position des Frosches ändert. So muss für eine Fortführung des Spiels im Flussbereich eine Kollision vorliegen, da ansonsten der Frosch nicht auf einem Baumstamm stehen würde und in den Fluss gefallen wäre. Im unteren Bereich hingegen darf keine Kollision vorliegen, da dies einen Unfall mit einem Auto anzeigen würde.

Abb. 3: Die Logik des verwendeten Hitbox-Systems. Ausgehend von der Kordinate (y=3/x=2) wird dem Auto eine Hitbox von 9 Punkten zugewiesen (Hitboxbreite=3 x Hitboxlänge = 3). Gleiches erfolgt für den Frosch. Equal(Frosch) gibt in diesem Fall für die Kombination 5/4 ein true zurück und zeigt eine Kollision an (Quelle: Eigene Programmierung Frogger)



In einer vereinfachten Darstellung ruft die den Spielablauf korrdinierende Funktion „Spiel.spielRundeSpielen()“ die Funktion „Verlierer.ertrinkenOderUnfall()“ auf. Diese prüft, ob der Frosch sich im Flussbereich oder im Straßenbereich befindet. Anschließend ruft sie die „Verlierer.gleicherOrtSpielfeldTesten()“ Funktion auf, welche durch alle Hindernisse des Spielfelds iteriert und für jedes Hindernis die „kollidieren()“ Funktion aufruft.

Abb. 4: Vereinfachter Programmablauf zur Prüfung, ob sich ein Hindernis und der gesteuerte Frosch in der gleichen Position befinden. (Quelle: Eigene Programmierung Frogger)



Die „kollidieren()“ Funktion führt dann die eigentliche Prüfung aus. Bei dieser wird für jede Kombination der X-Koordinate und der Y-Koordinate innerhalb der Hitbox des Hindernisses geprüft, ob diese Koordinatenkombination identisch mit einer Koordinatenkombination der Hitbox des aktuell gesteuerten Frosches ist. Ist dies für eine Kombination der Fall, wird die Prüfung abgebrochen und true zurückgegeben.

Die „Verlierer.gleicherOrtSpielfeldTesten()“ Funktion evaluiert diesen Rückgabewert. Wurde vorher festgestellt, dass sich der Frosch im Straßenbereich befindet, wird bei einer Koordinatenübereinstimmung das Spiel aufgrund einer Kollision mit einem Auto beendet. Wurde hingegen festgestellt, dass sich der Frosch im Flussbereich befindet, beginnt eine neue Spielrunde, da der Frosch in diesem Fall auf einem Baumstamm gelandet ist. Ein sehr ähnlicher Algorithmus prüft, ob an der Stelle des rettenden Ufers bereits ein anderer Frosch angekommen ist.

Ein anderer komplexer Algorithmus legt fest, wie weit die Hindernisse voneinander entfernt sind. Dieser Algorithmus ist zwar aus technischer Sicht nicht komplex, aber aus der Sicht eines Gamedesigners schwieriger. So wird das Spiel z.B. zu schwer, wenn die Hindernisse zu nah aneinander stehen oder zu einfach, wenn sie zu weit weg voneinander sind. Gleiches gilt auch für die Geschwindigkeit. Werden die Abstände in höheren Leveln verlängert oder verkürzt, verkomplexiert sich dieser Zusammenhang weiter. Um gut spielbare Abstände zu ermöglichen, wurde in der Funktion „Hindernisse.addHindernis()“ eine Variable „int zurueck“ implementiert. Wenn ein neues Spielobjekt beim Starten des Spiels erstellt wird, werden die Hinderniss Objekte ebenfalls instanziert. Anschließend werden diese Hindernisse geclont und die X-Koordinate jedes geclonten Hindernis um den Faktor „zurueck“ nach hinten verschoben. Der Faktor „zurueck“ ist dabei abhängig von der Hindernislänge, der Hindenisgeschwindigkeit, der Hindernisanzahl, einem manuellen setback und einem Zufallsfaktor. Durch den Zufallsfaktor wird auch erreicht, dass sich jedes neu gestartete Spiel vom vorherigen Spiel ein wenig unterscheidet, aber die Abstände trotzdem immer passend für ein gutes Spielgefühl sind.

Abb. 5: Vereinfachter Programmablauf zur Berechnung der Abstände zwischen verschiedenen geclonten Objekten. Durch „ThreadLocalRandom.current().nextint(1, 5+1)“ ist der Abstand zwischen den Hindernissen etwas unterschiedlich. (Quelle: Eigene Programmierung Frogger)

