Inhaltsverzeichnis

Projektbeschreibung / Summary - 1 - 3
Spielentwicklung - 1.1 - 3
Engine Weiterentwicklung - 1.1.1 - 3
Initialisierung: - 2 - 4
Projektauftrag - 2.1 - 4
Zielscheibe - 2.2 - 5
Zielsetzung und Erfolgsmessung - 2.3 - 6
Erklärung: - 2.3.1 - 6
Vorgegebene Ziele - 2.3.2 - 7
Implementierungsspezifische Grundziele - 2.3.3 - 7
Implementierungsspezifische Extraziele - 2.3.4 - 8
Ergebnis - 2.3.5 - 8
Grobplanung - 3 - 9
Situationsanalyse – Engine (IST/SOLL) - 3.1 - 9
UML - 3.2 - 11
Projektstruktur & Ablaufplan - 3.3 - 12
Feinplanung & Umsetzung - 4 - 13
Dokumentation - 4.1 - 13
Spielsteuerung - 4.1.1 - 13
Gradle build - 4.1.2 - 14
Terminologie - 4.1.3 - 14
Details zur Umsetzung - 4.2 - 16
Mängel - 4.2.1 - 16
Änderungen an der Engine - 4.2.2 - 16
Highscore Format - 4.2.3 - 17
Kompatibilität - 4.2.4 - 17
Abschluss - 5 - 18
Screenshots - 5.1 - 18
Start - 5.1.1 - 18
Im Spiel - 5.1.2 - 19
Game Over - 5.1.3 - 20

Highscore - 5.1.4 - 21
Reflexion - 5.2 - 22
Allgemein - 5.2.1 - 22
Aufgetretene Probleme - 5.2.2 - 23
Anhang - 6 - 24

1 - Projektbeschreibung / Summary

1.1 - Spielentwicklung

Es soll der Klassiker «Snake» mit einigen Extras als Konsolenanwendung implementiert werden.

Das Grundkonzept:
In diesem Spiel steuert der Spieler eine Schlange in einem begrenzten Raum. Das Ziel des Spieles ist es möglichst viele Punkte zu erhalten. Punkte werden dadurch gewonnen, dass die Schlange Futter isst, welches zufällig auf dem Spielbereich verteilt wird. Indem die Schlange allerdings futter isst, wird sie länger. Das Spiel endet wenn der Spieler in sich selbst hineinsteuert.

Die Extras
Das Grundkonzept ist nicht besonders dynamisch. Der Projektleitung ist es bei genügend Zeit erlaubt das Grundkonzept beliebig zu erweitern.

Implementation
Das Spiel soll basierend auf der Spiel-Engine die zuvor von der Auftragnehmerin teilweise erstellt wurde implementiert werden.

1.1.1 - Engine Weiterentwicklung

Die derzeit namenlose Spiel-engine wurde noch nicht vollständig implementiert. Die Implementierung des Spiels erfordert die Weiterentwicklung der Spiel-Engine. Deren Weiterentwicklung ist allerdings nicht direkter Bestandteil dieses Projekts und dessen Dokumentation.

2 - Initialisierung:

2.1 - Projektauftrag

Projektname: ksnake-evolution
Auftraggeber: Graziano Felline (TEKO)
Projektleiter: Lucy von Känel
Team grösse: Einzelprojekt
Projektlaufzeit: 30.01.2019 – 30.03.2019: 23:59
Projekttyp: Innovationsprojekt
Das individuelle Projekt ist frei wählbar. Eigene Ideen sind gefragt.

Projektbeschreibung:

Ausgangslage:
Die Schüler wurden von einer Lehrperson der TEKO Olten beauftragt eine interaktive Konsolenanwendung in Kotlin im Rahmen einer Projektarbeit zu realisieren.

Sinn und Zweck:
Die Schüler erlangen praktisches Wissen im Umgang mit der Programmiersprache Kotlin.

Richtziel:
Eine interaktive, dokumentierte und funktionsfähige Kotlin Konsolenanwendung.

Endergebnisse:

- Eine funktionelle jar Datei liegt vor die das individuelle Projekt in ausführbarem Zustand enthällt.
- Der Quellcode für das Individuelle Projekt steht zur verfühung.

Erfolgskriterien:

- Das Richtziel (Projektbeschreibung) wurde erreicht
- Die individuellen Projektziele wurden erreicht

Bewilligung:
[ ] Genemigt
[ ] Nicht genemigt
Datum, Unterschrift: _____, _____

2.2 - Zielscheibe

Richtziel: Eine interaktive, dokumentierte und funktionsfähige Kotlin Konsolenanwendung.	

Endergebnisse:
• Projektdokument enthält
  ◦ Auftrag
  ◦ Zielscheibe
  ◦ Projektplanung
  ◦ Dokumentation der selbst definierten Aufgabe
  ◦ Reflexion und Lösungsansätze
  ◦ Projektstruktur
  ◦ Projektablaufplan
• Quellcode des Individuellen Projektes
• Funktionelle jar Datei mit der interaktiven Konsolenanwendung
	Kunde:
	• Graziano Felline (TEKO)

Sinn und Zweck: • Erhalt von praktischer Erfahrung mit Kotlin	**Erfolgskriterien:** • Die individuelle Aufgabe wurde zu genüge Dokumentiert • Die individuelle Aufgabe wurde funktionell und entsprechend der individuellen Zielsetzung implementiert. • Die Projektarbeit wird pünktlich abgegeben

2.3 - Zielsetzung und Erfolgsmessung

2.3.1 - Erklärung:

Es gibt 3 Kategorien um den Implementationserfolg zu messen. Jedes erreichte Ziel ergibt die Punkte die unter «Gewichtung» stehen. Die Summe aller ist hier entscheidend. Die Punkte werden während des Projektabschlusses eingetragen.

Ungenügend: **< 30 Punkte**
Mit weniger als 30 Punkten fehlen der Applikation essentielle Bestandteile.

Genügend: **>= 30 Punkte**
Punkte technisch beinhaltet dies die Erfüllung der Vorgaben und Grundziele.
Ist eines davon ein Grenzfall, so kann Genügend immer noch mit den Extrazielen erreicht werden. Grundsätzlich sollten die Extraziele erst angestrebt werden, wenn die Basisfunktionen vorhanden sind.

Überragend: **>= 40 Punkte**
Es ist davon auszugehen, dass nur ein Bruchteil der erweiterten Ziele in der vorgegebenen Zeit erreicht werden können. Das erreichen von 10 Extrapunkten stellt eine grössere Herausforderung dar.

2.3.2 - Vorgegebene Ziele

Ziel	Erfolg (Ja/Nein)	Gewichtung
Die Applikation kann mit «java -jar <Pfad> [Parameter]» auf Linux gestartet werden.	10	10
Die Applikation nimmt mindestens ein Argument an	5	5
Die Applikation speichert mindestens eine Datei mit Applikationsspezifischem Inhalt ab	5	5
Die Applikation ist interaktiv	5	5
	Erreicht:	Total:
	15	15

2.3.3 - Implementierungsspezifische Grundziele

Ziel	Erfolg (Ja/Nein)	Gewichtung
Es existiert ein sichtbarer Spielbereich	1	1
Eine Schlange wird auf dem Spielbereich dargestellt	1	1
Der Spieler kann die Schlange mit «w,a,s,d» steuern	2	2
Es existiert immer mindestens eine Futterstück auf dem Spielbereich	1	1
Die Schlange wird länger wenn sie über das Futterstück fährt	2	2
Das Futterstück verschwindet, wenn die Schlange darüber fährt	2	2
Trifft die Schlange auf eine Wand, erscheint sie auf der gegenüberliegenden Seite fortlaufend wieder.	3	3
Trifft die Schlange auf sich selber, so endet das Spiel	3	3
	Erreicht:	Total:
	15	15

2.3.4 - Implementierungsspezifische Extraziele

Ziel	Erfolg (Ja/Nein)	Gewichtung
Buffs erscheinen zufällig auf der Karte	1	1
Werden die Buffs von der Schlange gegessen, verschwinden sie und die Schlange erhält einen additiven Multiplikator für jedes Futterstück	3	3
Die Schlange wird je nach Multiplikator schneller oder langsamer	1	1
Das Spiel lässt sich pausieren	0	1
Grundeinstellungen lassen sich über ein Menü im Spiel persistent ändern	0	3
Im Spiel gibt es eine Punkteanzeige	1	1
Im Spiel gibt es eine Multiplikator Anzeige für die Buffs	1	1
Multiplayer am selben Computer	0	3
Multiplayer via Netzwerk	0	5
Multiplayer gegen einen programmierten Spieler	0	5
Es gibt eine Highscore (Name, Punkte)	3	3
Die Highscore ist persistent	1	1
Generierung von Kollisionshindernissen beim Spielstart	0	2
	Erreicht:	Total:
	11	30

2.3.5 - Ergebnis

Die Punkte wurden im vorherigen Kapitel eingetragen und farblich markiert.
Die erreichte Punktzahl beträgt: 15/15 + 15/15 + 11/30 = **41/60 Punkte.**

3 - Grobplanung

3.1 - Situationsanalyse - Engine (IST/SOLL)

Die Erstellung des Spiels in der Spiel-engine erfordert gewisse Features die jedoch noch nicht alle implementiert sind:

Grün: Zu genüge erfüllt.
Orange: Suboptimal, verhindert aber die Zielerreichung nicht.
Rot: Die Engine muss aktiv erweitert werden um das Ziel erreichen zu können.

IST (Vorhanden)	SOLL (Benötigt)	Notiz
ASCII Rendering bis zu 250›000 Frames pro Sekunde Ryzen 7 1800x - 1 Render Prozess auf 100% ausgelastet. (TMUX + Alacritty auf Voidlinux)	Mindestens ein Frame pro Sekunde	
Rendern von programmierten Entitäten auf einem Spielfeld durch Implementation von vorgegebenen Methoden.	Darstellung von beweglichen und unbeweglichen Spielobjekten.	
Nicht blockierende echtzeit Erkennung von Tastatureingaben durch den Benutzer in einer Konsole. Enter wird nicht benötigt.	Erkennung von mindestens 4 Tastenanschlägen in der Konsole während das Spiel läuft, ohne diese mit Enter bestätigen zu müssen.	
Zeitlich steuerbare Logikabfragen für die individuelle Programmierung des Spiels.	Möglichkeit mindestens einmal pro Sekunde die Spielsituation zu analysieren und darauf zu reagieren.	
Nicht vorhanden	2D Kollisionserkennung.	1
Nicht vorhanden	Vorzugsweise ein implementiertes interaktives Benutzermenü, das sich sehr einfach erweitern und abspeichern lässt, ohne eine Entität zu implementieren.	2
Möglichkeit ingame Nachrichten oder Logs chronologisch während des Spiels anzuzeigen	Vorzugsweise eine Möglichkeit Spielwerte (Bezeichnung, Wert) anzeigen zu lassen ohne dafür eine Entität zu implementieren.	3
Kryo Library zur Serialisierung vorhanden, Daten zwischen dem Compositor und der Logik werden immer serialisiert. Netzwerkoptionen sind jedoch keine vorhanden.	Senden von serialisierten Daten via Netzwerk	4

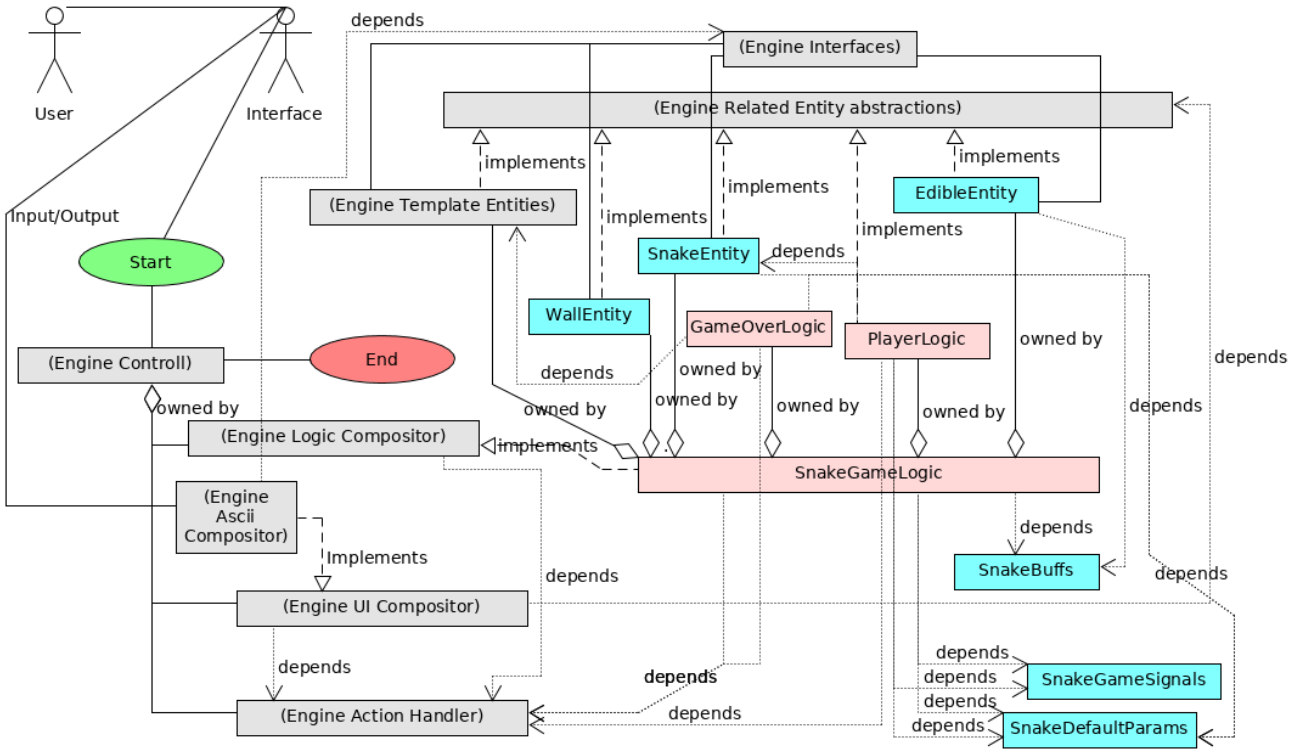
Notizen:
1: Die Kollisionserkennung ist essentiell für die Umsetzung und deren Implementation stellt eine nicht ganz triviale Herausforderung dar. Ist jedoch Zeitlich machbar.

2 & 3: Es handelt es sich um Dinge die zwar durch eine spezifische Implementation realisierbar sind, jedoch aus Performance gründen und potentiell mehreren Verwendungszwecken besser in der Engine auf eine generische Art und Weise implementiert werden.

4: Wird lediglich für ein gewisse Extraziele benötigt. Grundsätzlich fehlt der Engine eine Client-Server Funktionalität, wobei die effektive Implementierung des Servers etwas Spiel spezifisches wäre. Es gäbe mehrere einfache Möglichkeiten das ganze sauber zu implementieren. Es sprengt jedoch den Rahmen diese hier zu erläutern und ich bewahre mir das für den Fall auf, das dafür Zeit bleibt.

3.2 - UML

Da das UML mit sämtlichen Funktionen und Properties nicht in das Dokument passt und den nötigen Aufwand für den Rahmen des Projektes sprengt ist die Darstellung stark vereinfacht und auf die Klassen limitiert. Graue Einträge sind zudem Teil der Game Engine und nicht direkter Bestandteil des Projektes. Sie dienen lediglich zur Übersicht.



Bez: Start und Ende:
Der Benutzer Input wird von der jeweiligen Compositor Implementation über den Action Handler verschickt. Das Signal zum beenden ist universell SIGINT. Jeder Prozess kann darauf hören und sich beenden. Die Engine schaltet sich erst aus wen nichts mehr läuft.

3.3 - Projektstruktur & Ablaufplan

Arbeitspaket	Teilaufgaben	No.	Abhängig	Fertig bis	verf. Zeit*
Dokumentation	- Template erstellen	1		19.01.2019	5t
	- Projektdokument Struktur	2	1	20.01.2019	6t
	- Zielscheibe	3	2	27.01.2019	7t
	- Projektauftrag	4	2	27.01.2019	7t
	- Erklärung der Spielsteuerung	5	2,8,14	24.03.2019	1m
	- Reflexion / Schwierigkeiten	6	2,22	30.03.2019	0-1t
Planung	- Situationsanalyse (IST/SOLL)	7	2,8	27.01.2019	0-1t
	- Zielsetzung prüfbar auf Erfolg	8	2	27.01.2019	0-1t
	- UML	9	2,7,8	03.02.2019	7t
	- Planung Ende	*10*			
Programmierung	- Spielstart	11	12	10.02.2019	0-1t
	- Spielcontroller	12	10	10.02.2019	7t
	- Wände	13	12	17.02.2019	7t
	- Spielsteuerung	14	12	24.02.2019	14t
	- Teleportation	15	12,13,20	31.02.2019	7t
	- Futter	16	12,20	31.02.2019	7t
	- Buffs	17	12,20	31.02.2019	7t
	- Highscore / Game Over	18	12,20	17.03.2019	21t
	- Programmierung Ende	*19*			
Erweiterung der Engine	- Kollisionserkennung	20	12	24.02.2019	14t
	- Template Entitäten Text	21		07.03.2019	~3m
	- Performance/Kompatibilität fixes	22		30.03.2019	~3m
	- Erweiterung der Engine Ende	*23*			
Abschluss	- Finish der Dokumentation	24	19	24.03.2019	7t
	- Exportieren der Resultate	25	19,23,24	30.03.2019	0-1t
	- Abgabe	26	25	30.03.2019	0-1t

**Verfügbare Zeit: Zeit von der letzten Abhängigkeit bis zum Enddatum des Arbeitspakets*

4 - Feinplanung & Umsetzung

Um der Praxis treu zu bleiben und weil das Projekt nur von einer einzigen Person durchgeführt wurde, wurde die Applikation während der Umsetzung im Detail geplant.

4.1 - Dokumentation

4.1.1 - Spielsteuerung

Starten/Beenden:
Das Spiel kann mit «java -jar datei.jar --compositor ascii» gestartet werden.
Beenden ist jederzeit mit «CTRL + C» möglich. Alternativ wird das Spiel beendet, wenn am Ende in der Highscore Ansicht «Enter» gedrückt wird.

Das Spiel ist zu Ende, wenn die Schlange in sich selber fährt.

Bewegung:
Die Schlange kann mit **w** (oben), **a** (links), **s** (unten), **d** (rechts) gesteuert werden

Buffs / Multiplikator:
In einem bestimmten Intervall tauchen neben Futter bestimmte Buffs auf der Karte auf. Gegessene Buffs verändern das Tempo der Schlange und die Punkte die es für jedes Futterstück gibt.
Darstellung:

- `-` Mittlerer Tempoverlust (*0.8)
- `!` Hoher Tempoverlust (*0.6)
- `+` Mittlerer Tempogewinn (*1.2)
- `^` Hoher Tempogewinn (*1.4)

Essen:
Die Schlange wird länger mit jedem Futterstück das sie isst. Entsprechend erhöht sich die Punktezahl mit jedem Futterstück basierend auf dem wert 50 multipliziert mit dem Punktemultiplikator.
Darstellung:

- `*` Essen

Highscore:
Nachdem die Schlange gestorben ist, wird die Highscore mit den top 10 Resultaten angezeigt. Ist der Spieler unter diesen, wird er an der entsprechenden Position aufgefordert einen Namen einzugeben. Bestätigt wird mit «Enter»

- Ist derselbe Namen zweimal vorhanden wird dieser überschrieben, wenn die neue Punktzahl höher ist als die alte.
- Wird kein Namen eingeben, wird «Anonymous» als Name abgespeichert.
- Ist der Punktestand zu tief, gibt es nichts zu tun, ausser das Spiel mit «Enter» zu beenden.

4.1.2 - Gradle build

Die Engine mit der Applikation wird mit dem gradlewrapper erstellt. Das Endresultat ist eine «fatjar» die sämtliche Abhängigkeiten beinhaltet.

HTML Dokumentation:	«./gradlew dokka»
Das Projekt:	«./gradlew build»
Ausführen:	«java -jar ./build/libs/ksnake-1.0-SNAPSHOT.jar --compositor ascii»

4.1.3 - Terminologie

Einige Begriffe die im Zusammenhang mit in den Code Kommentaren und der Dokumentation öfter verwendet werden:

Field

Ein «Field» ist nichts weiter als eine Liste aus Entitäten, die das aktuelle Spielgeschehen darstellen. Es wird für die Kommunikation zwischen dem «LogicCompositor» und dem «UICompositor» verwendet.

Entity

Eine «Entity» oder «Entität» ist die abstrakteste Bezeichnung für ein Spielobjekt das auf dem Spielfeld existieren kann. Grundsätzlich ist alles was letztendlich angezeigt wird oder Spiel relevant ist eine Entität. Auf dem Spiellevel ist eine Entität je nach Spielobjekt anders implementiert. Spielentitäten implementieren Interfaces und überschreiben abstrakte Funktionen damit sie von der Engine gehandhabt werden können.

Eine Entität ist ein komplexeres Model, welches ein einzelnes Spielobjekt mit mehreren Layern darstellen kann. Nimmt man als Beispiel ein eine Schlange, so heisst das, dass die Entität für jedes Segment der Schlange zuständig ist und der Engine sagt wie die Schlange als ganzes dargestellt werden soll, oder wie sie auf kollisionen reagiert.

Beispiele in dem entwickelten Spiel sind die Klassen: «EdibleEntity», «SnakeEntity», «WallEntity» und die Engine presets: «TextEntity» sowie «TextPairEntity»

EntityLogic

Eine Entität ansich ist «single threaded» und sollte sich selber nicht wirklich mit dem Handling von Interaktionen befassen. Die EntityLogic ist dazu da diesen Teil zu übernehmen. Sie lauft in einem separaten Thread und reagiert auf eine Anfrage des Hauptthreads des Spiels (LogicCompositor derivat) oder auf interaktionen die von der Engine getriggert werden. Normalerweise handhabt sie eine zugewiesene Entität.

Beispiele in dem entwickelten Spiel sind die Klassen: «PlayerLogic» und «GameOverLogic»

UICompositor

Klassen die den UICompositor erweitern, befassen sich mit der Darstellung des Spielfeldes. Damit eine Entität auf einem Spielfeld in einem entsprechenden Compositor dargestellt werden kann, sollte diese ein zugehöriges Interface implementieren.

Beispiel in dem entwickelten Spiel ist der «ASCIICompositor» respektive das «ASCIISupport» Interface. Der «DummyCompositor» ist zwar funktionsfähig, allerdings nur zu Debug Zwecken da.

LogicCompositor

Die einzige Klasse die den LogicCompositor erweitern sollte ist die individuelle Spiellogik. Der LogicCompositor ist dafür zuständig, dass das Spielfeld mit jedem «Tick» neu berechnet wird und danach an den entsprechenden UICompositor geschickt wird. Kollisionen, Interaktionen, Reaktionen und Spieler eingaben müssen entweder dort oder in einer separaten EntityLogic gesteuert werden.

Beispiel in dem entwickelten Spiel ist die Klasse «SnakeGameLogic»

ActionHandler

Der ActionHandler ist ein essentieller Bestandteil der Engine. Grundsätzlich solle er als Signalgeber fungieren. Ein Thread kann sich für Notifikationen für bestimmte Events registieren und wird später entsprechend benachrichtigt wenn diese eintreten. Solche Events sind:

- Kollisionen
- Benutzereingaben
- Signale (SIGINT, SIGTERM, etc)
- Game Signale (Sind Spielabhängig)
- Framerate Cap (Zur änderung der Framerate)
- Neue Logic/UI Compositoren (Die Engine kann theoretisch «multiplexen»)
- Ingame Logs

Es können dabei grundsätzlich diverse Daten mitgesendet werden. Es gibt keine feste Regel, was genau mitgeschickt wird. Abgesehen von Engine Internen Komponenten sollten allerdings nur ~primitive wie z.B. Zahlentypen, Chars und Strings benutzt werden.

RigidBody / *Collider

RigidBody ist ein Interface das von Entitäten implementiert werden kann, welche in der Lage sein sollen mit anderen Entitäten zu kollidieren. Ein Collider wiederum ist der aktive Teil. Es wird jeweils überprüft ob ein Collider mit einem RigidBody kollidiert. Die Kollisionsimplementation ist Teil der jeweiligen Entität die den Collider Implementiert, wobei es hierfür vordefinierte Hilfefunktionen gibt, die die meisten Fälle abdecken.

Die Kollision wird danach von der zugehörigen «EntityLogic» gehandhabt. Informiert wird diese über den «ActionHandler» ausgelöst vom «LogicCompositor». Die Kollisionserkennung ist asynchron.

4.2 - Details zur Umsetzung

4.2.1 - Mängel

Kollisionserkennung - Verzögerung:
Die Kollisionserkennung wird von der Engine aus Performancegründen asynchron durchgeführt. Das hat zur folge, dass die entsprechende Meldung erst verzögert bei den entsprechenden Game Threads eintrifft. Optimalerweise würden diese die Differenz korrigieren. Dazu fehlte allerdings die Zeit. Die Folge davon ist ein kleiner grafischer «glitch». (Ein Segment der Schlange ist in der Wand zu sehen)

Eine alternative wäre die Kollisionserkennung synchron durchzuführen. Dies würde aber wiederum je nach Anwendung zu Performanceeinbussen führen.

4.2.2 - Änderungen an der Engine

Implementierung von Kollisionserkennung:
Da zu Beginn des Projektes kein generischer Weg bestand Kollisionen zu erkennen, wurde die Engine entsprechend erweitert. Hierzu wird ein «RigidBody» sowie «*Collider» Interface benutzt. Entitäten die einen RigidBody implementieren können grundsätzlich mit anderen Objekten kollidieren. Wenn eine Entität die ein Collider Interface implementiert hat auf ein RigidBody Objekt trifft, wird dies mit der Quelle und dem Ziel der Entität die den Collider implementiert hat gemeldet.

Grundsätzlich kann die genaue Kollisionserkennung von jeder Entität individuell implementiert werden. Es wurden jedoch Hilfefunktionen erstellt, die für die meisten Fälle benutzt werden können.

Jedoch ist die einzig funktionierende Kollisionsfunktion eine positionale. Sprich ob sich ein Punkt in einem 3D Rechteck befindet.

Implementierung einer «TextPair» Entität:
Dies war sowohl für die Spielwerte und die Highscore hilfreich.
Die beiden Dinge haben gemeinsam, dass sie grundsätzlich aus einer Liste von Bezeichnungen und Werten bestehen.

Da eine solche Entität ein Use Case für mehrere Applikationen darstellt, wurde sie als Preset von der Engine implementiert. Sie bietet support für den ASCII Compositor und ist relativ flexibel was den Inhalt angeht.

4.2.3 - Highscore Format

Die Highscore wird unter «$HOME/.local/share/ksnake/highscore.json» im json Format gespeichert bzw. von dort geladen:

4.2.4 - Kompatibilität

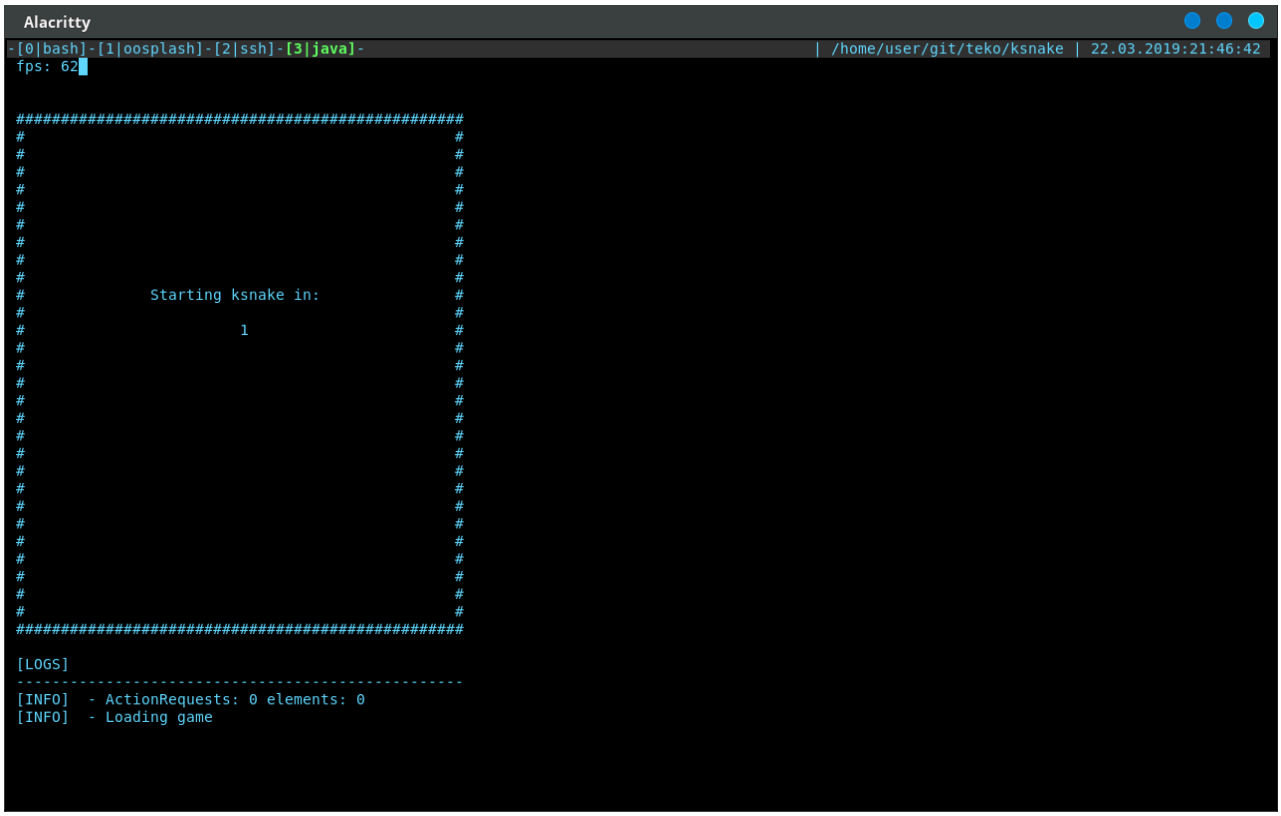
Das Spiel benutzt den ASCII Compositor der Engine. Dieser wiederum benötigt ein Terminal das die ASCII Kontroll Sequenzen unterstützt. Auf Linux ist das in den meisten Fällen der Fall. Auf Windows ist dies erst seit Windows 10 möglich, muss jedoch von der Anwendung aktiviert werden. Ein entsprechender Fix wurde implementiert.

Getestete Terminals:
- Alacritty (Linux) + tmux
- Gnome-Terminal (Linux)
- Terminology (Linux)
- cmd (Windows 10)

5 - Abschluss

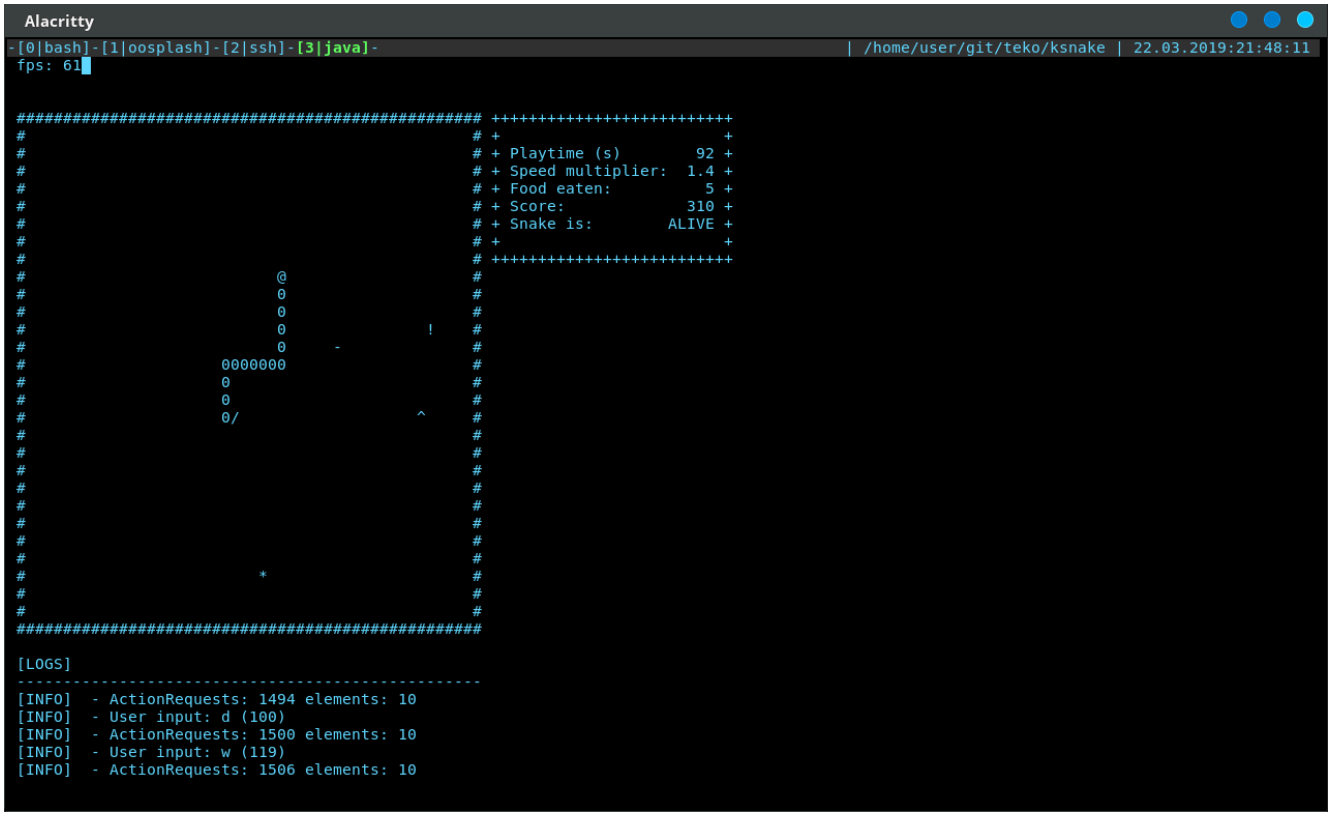
5.1 - Screenshots

5.1.1 - Start



Ein Ladebildschirm wird beim Spielstart angezeigt. Dieser dient mehr dazu dem Spieler einige Sekunden zu geben um sich auf den Start gefasst zu machen.

5.1.2 - Im Spiel



Das gerenderte Spielfeld mit der Schlange die sich je nach Benutzereingaben bewegt. Das @ Symbol ist der Kopf, der Schwanz je nach Bewegung /, \ oder I.

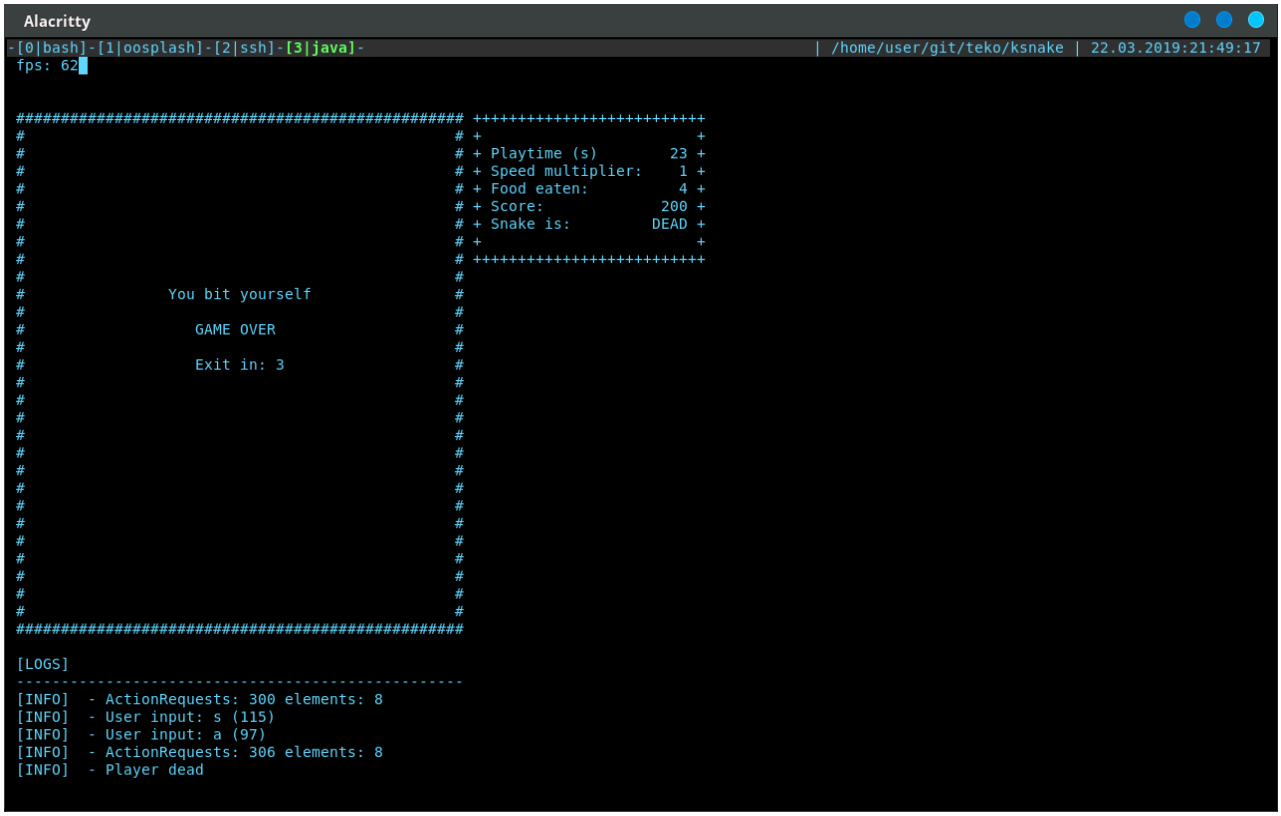
Die Futterstücke werden mit * und die Buffs mit !, -, +, ^ dargestellt.

Rechts sind die aktuellen Spielwerte zu sehen:

- Spielzeit
- Geschwindigkeitsmultiplikator
- Gegessenes Futter
- Punkte
- Tot/Lebendig

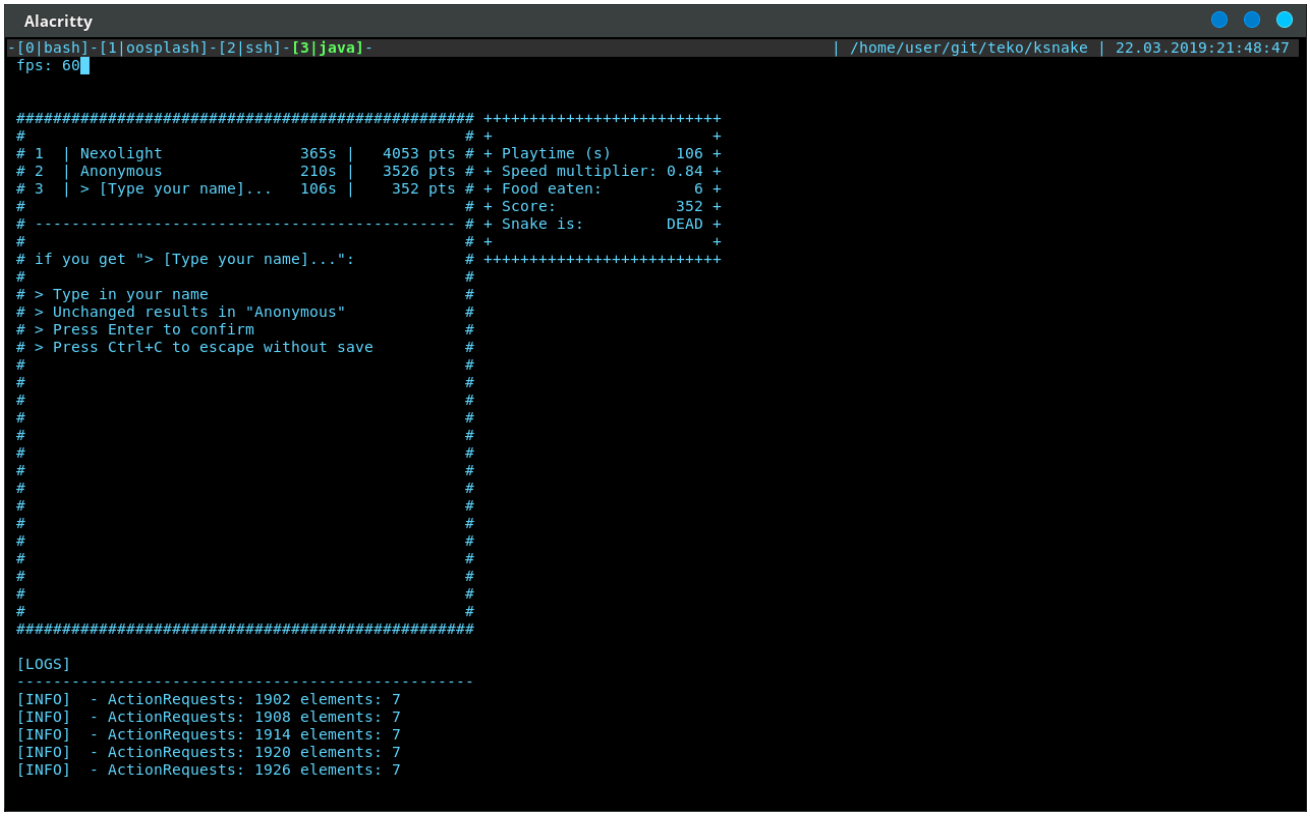
Unten sind einige ingame Logs zu sehen. Diese dienten ursprünglich zum Debuggen. Dort werden einige Spielevents angezeigt.

5.1.3 - Game Over



Der Game Over Bildschirm erscheint, nachdem sich die Schlange selber gebissen hat für einige Sekunden.

5.1.4 - Highscore



Die Highscore wird am Spielende angezeigt. Es gibt Platz für 10 Einträge. Die Einträge müssen einmalig sein.

Der Spieler wird aufgefordert seinen Namen einzugeben, wenn er es in die Top 10 geschafft hat, mit Enter zu bestätigen oder mit CTRL+C abzubrechen.
Existiert der Name bereits, wird dieser im Falle einer höheren Punktzahl überschrieben.
Wird nichts eingetragen, wird der Name «Anonymous» verwendet.

Die Datei wird auf allen Systemen unter «$HOME/.local/share/ksnake/highscore.json» gespeichert bzw. geladen.

5.2 - Reflexion

5.2.1 - Allgemein

Grundsätzlich habe ich kaum Probleme eine neue Programmiersprache in wenigen Tagen zu lernen. Unter anderem auch weil ich bereits eine Ausbildung als Applikationsentwicklerin und viel Berufserfahrung hatte, konnte mir das Modul nicht all zu viel neues bieten.

Ich habe allerdings die Chance genutzt Kotlin meinem Repertoire hinzuzufügen und mich mit der Programmiersprache eingehend beschäftigt.

Das führte zu einem Projekt das grösser war als ich es eigentlich vorhatte und einer Projektarbeit die darauf basierte. Ich sehe solche Projekte als eine Chance meine Programmierskills zu verbessern und die Dinge besser zu machen die ich zuvor in anderen Projekten in anderen Sprachen bemängelt hatte.

Natürlich unter Berücksichtigung der vorgegebenen Zeit, die hier zu knapp war für das was ich mir vorgenommen hatte.

5.2.2 - Aufgetretene Probleme

Die Dokumentation

Ich bin der Meinung das eine Projektdokumentation wie diese hier ansich ungeeignet für ein Einzelprojekt in diesem Rahmen ist. Daher habe ich dieser nicht besonders viel Aufmerksamkeit geschenkt. Das hat sich insofern ausgewirkt, dass ich effektiv die Idee die ganze Zeit in meinem Kopf hatte, das Projekt entsprechend umgesetzt und die Dokumentation anschliessend ergänzt habe um sie nicht nonstop anzupassen und die Arbeit doppelt und dreifach zu machen.

Das führte aber dazu, dass ich nun ein Grossteil der Dokumentation im Stress schreibe und etwas überfordert bin, das Ausmass des Projekts in der Dokumentation festzuhalten. Daher habe ich mich darauf limitiert nur das Spiel selber zu Dokumentieren und die genaue Funktionsweise der Engine so gut wie möglich auszulassen.

Fehlende Engine Features

Wie bereits in einem vorherigen Kapitel erwähnt, war zu der Zeit wo die Projektarbeit startete noch nicht jedes Feature in der Engine implementiert, dass ich für meine Idee effektiv benötigte. Ich war mir dessen bewusst, analysierte was genau fehlte und wie realistisch es ist, diese Features noch hinzuzufügen.

Grundsätzlich war es nicht viel und in der vorgegebenen Zeit gut machbar. Das grösste oder besser gesagt komplexeste Problem war die Kollisionserkennung. Die Engine selber sollte in der Lage sein diese in einem 3D Raum zu handhaben. Das erfordert bessere Mathe Skills als die die ich derzeit habe. Ich war allerdings in der Lage die Implementation soweit zu vereinfachen, dass sie für einfache 3D Spiele funktionsfähig wäre.

Die Zeit

Grundsätzlich habe ich mir zuviel für das Projekt vorgenommen und das war mir auch durchaus bewusst. Da ich mehr Zeit benötigte entschied ich mich dazu den Unterricht weitgehend zu ignorieren und mich in der Zeit mit der Implementation zu beschäftigen. Ich lerne Grundsätzlich nicht indem mir jemand anderes etwas beibringt, sondern indem ich es nachschlage und selber umsetze.

Während dieses Prozesses bin ich praktisch über jedes Thema das auch im Unterricht behandelt wurde gestossen und habe es auf meine eigene Art und Weise gelernt während dem ich auch etwas zum Projekt beigetragen hatte.

IST vs SOLL

Durch die Mangelnde Zeit war von anfang an klar, dass ich nicht in der Lage sein werde meine ganze Idee umzusetzen. Daher stellte ich mir eine Liste mit Features zusammen und habe sie in verschiedene Kategorien eingestellt.Das gab mir einen Überblick worauf ich mich am meisten fokussieren sollte und was ich notfalls auslassen kann.

UML Diagramm

Das Projekt umfasst mehr als 4000 Zeilen Code. Es wurde nicht basierend auf einem UML aufgebaut. Daher wurde es schwierig dieses nachträglich zu konstruieren. Ich investierte mehrere Stunden um einen geeigneten UML Generator zu finden. Leider Schlug dieser Ansatz fehl, da sämtliche Tools entweder manuell funktionieren, weder mit Eclipse noch Intellij laufen oder Kotlin nicht unterstützen.

Für die grösse des Projektes würde eine manuelle Erstellung mehrere Tage in Anspruch nehmen und da manuell erstellt, bei kleinen Änderungen schnell nicht mehr mit dem Projekt übereinstimmen.

Daher der kam der Entschluss das UML auf die Klassen zu reduzieren und die gesamte Darstellung stark zu vereinfachen.

Flackern bei der Darstellung

Insbesondere wenn viele Elemente auf einmal dargestellt werden kann es zu einem Flickern bei der Darstellung im ASCII Compositor kommen. Bis jetzt ist nicht wirklich bekannt woher das Problem kommt.

Der erste Gedanke war, dass der Prozess zu CPU intensiv ist. Allerdings ist die CPU Auslastung niedrig und der Compositor stellt einfach die letzten Frames erneut dar, wenn in der zwischenzeit keine neuen berechnet wurden. Zudem ist das Flackern bei niedrigen Frameraten stärker als bei hohen. Gute Resultate wurden immer mit der Bildwiederholfrequenz des Monitors erzielt.

Erst nach einer Optimierung des ASCII Compositors die zu einer Leistungssteigerung von (experimentall getestet) +150,000 FPS führte, wurde klar, dass das Problem abhängig vom benutzten Terminal ist. Zwar wurde das Problem nicht endgültig gelösst aber der ASCII Compositor kann nun 350,000+ FPS darstellen.

Die Leistungssteigerung wurde durch eine Reduktion der print calls erzielt.

6 - Anhang

- 4x Kompetenzkarten
- 1x Ausführbare jar Datei
- 1x Generierte Code Dokumentation als zip gepackt
- 1x Programmcode als zip gepackt