technische universität
dortmund

Fachprojekt Report

Interactive Real-Time Gaming

Lukas Bünger
Thilaksan Kodeeswaran
Dilsan Mahadeva
30.09.2020

Supervisors:
Prof. Dr. Jian-Jia Chen
M.Sc. Junjie Shi

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12 (Eingebettete Systeme)
http://ls12-www.cs.tu-dortmund.de

Inhaltsverzeichnis

1 Einleitung

Hier gehen wir kurz darauf ein, wie das Projekt zustande gekommen ist und in welcher Entwicklungsumgebung gearbeitet wurde.

1.1 Motivation und Hintergrund

Durch die Vergabe des Projektes war uns klar ein kleineres Retro Spiel zu programmieren, nach einiger Überlegung und Diskussion zu einfachen aber leicht erweiterbaren Spielen kamen wir auf das Snake. Jedoch war unser Ziel darüber hinaus mehr als nur das klassische Spiel zu implementieren. Es kamen sehr schnell viele Erweiterungsvorschläge wie mehr und variierte Food-Elemente oder spezielle Level oder Mehrspielermodi. Insbesondere gab es sogar noch weitere Ideen zu neuen Leveln und weiteren Spielmodi, die aber wegen zeitlichen Begrenzungen des Projekts vernachlässigt wurden und dafür mehr dafür gesorgt wurde, dass die implementierten Erweiterungen reibungslos funktionieren.

1.2 Aufbau und Umgebung der Arbeit

Die Vorgabe ein Echtzeitbetriebssystem (engl. RTOS) an der Universität im Labor zu nutzen, war wegen der Corona-Pandemie eine unmögliche Aufgabe. Auf Grund dessen wurde das gesamte Fachprojekt online abgehalten. Dadurch war ein Emulator, welcher solch ein Echtzeitbetriebssystem simuliert, das Grundwerkzeug und Grundumgebung, womit das Projekt realisiert wurde. Durch das Einarbeiten in den Code des Emulators wurde es mit der Zeit immer verständlicher den Aufbau des Simulators zu verstehen um das Spiel zu implementieren. Unter anderem hat die Versionsverwaltung mit GitHub die Zusammenarbeit von uns sehr vereinfacht. Dadurch konnte strukturiert und effektiv an dem Projekt gearbeitet werden. Durch den WebClient BigBlueButton konnten wir digitale Treffen abhalten, in denen Probleme und Ideen als Gruppe besprochen wurden.

2 Spiel

In diesem Abschnitt wird das Snake Spiel beschrieben und die Funktionen von Elementen im Gameplay oder Buttons erklärt. Dabei gehen wir auf die Funktionsweise des Spieles ein, die Implementierung folgt im Kapitel 4 : Spiellogik

2.1 Spiel - Hauptmenü

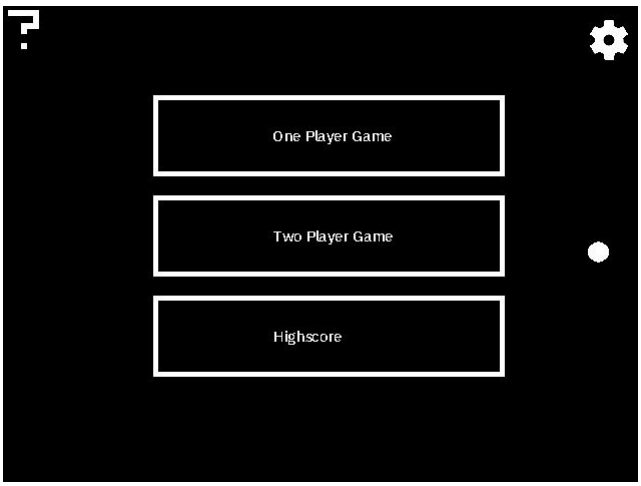

Abbildung 2.1: Hauptmenü

Das Hauptmenü (Bild 2.1) erscheint beim starten des Spiels und hat fünf Buttons. Liegt der Mauszeiger über dem Fragezeichen-Button wird eine Anleitung des Spiels angezeigt. Dort wird die Steuerung für die jeweiligen Spieler angezeigt und spezielle Elemente aus dem Mehrspielermodus erklärt.

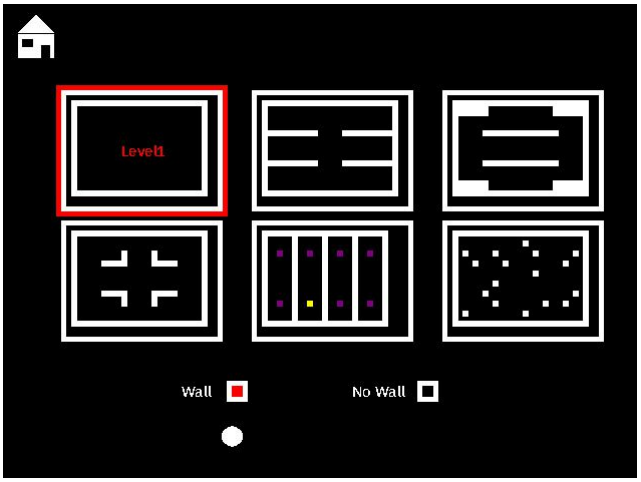

Abbildung 2.2: Einstellungen

Wird auf das Zahnrad rechts oben geklickt, öffnet sich das Einstellungsmenü (Bild 2.2), indem verschiedene Levels ausgewählt werden können und die Außenwand aktiviert und deaktiviert werden kann. Ist die Außenwand deaktiviert, kann die Schlange beim kollidieren mit der Außenwand nicht sterben, sondern kommt aus der gegenüberliegenden Wand wieder heraus. Durch klicken auf das Haus links oben kehrt der Spieler zum Hauptmenü zurück.

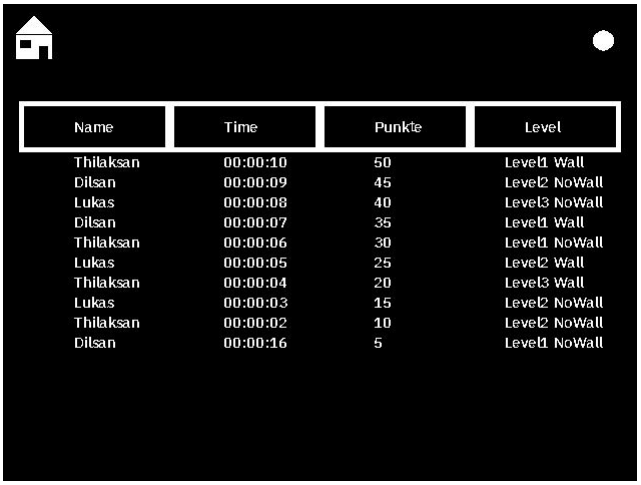

Abbildung 2.3: Highscore

Wird im Hauptmenü auf den Highscore-Button geklickt, wird der Highscore (Bild 2.3) aus den bisher gespielten Spielen angezeigt. Dabei sind die Highscores primär nach Punkten und sekundär nach Zeit sortiert. Außerdem steht in jeweils der letzten Spalte der Highscores das Level, indem gespielt wurde. Darüber hinaus kann die Sortierung durch Klicken auf das Punkte-Label umgekehrt werden. Mit Klicken auf das Haus links oben

kehrt der Spieler zurück zum Hauptmenü.
Klickt der Spieler auf den „One Player Game“-Button öffnet sich ein Menü, indem der Spieler seinen Namen eintragen kann. Dies kann durch Klicken auf die jeweiligen Buttons auf dem Bildschirm oder durch Eingabe über die Tastatur geschehen. Das Spiel wird dann gestartet, wenn der StartGame-Button betätigt wurde.
Wählt der Spieler den Mehrspielermodus, durch Betätigen des „Two Player Game“-Button im Hauptmenü, öffnet sich wieder ein Menü zum Eintragen der Spielernamen und das Spiel kann durch das Betätigen des StartGame-Buttons gestartet werden.

2.2 Spiel - Einzelspielermodus

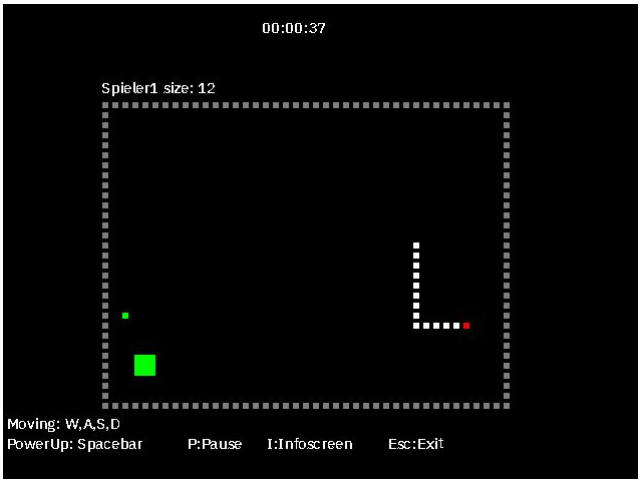

Abbildung 2.4: Einzelspieler Spiel

Der Einzelspielermodus ist im Grunde das traditionelle Snake Game. Die Schlange lässt sich steuern durch Betätigen der Tasten „W“, „A“, „S“, „D“ und bewegt sich zu Anfang alle fünf Frames. Die grünen Elemente sind das Essen der Schlange, welches die Schlange wachsen lässt. Dabei wächst die Schlange um eine Größe beim Verzehren des kleinen Food-Elements und um zwei beim Verzehren des großen Superfood-Elements. Alle 25 Punkte wird die Schlange schneller, welches die Schwierigkeit des Spiels erhöht. Das Spiel kann durch die entsprechenden Levels erschwert werden. Level fünf hat ein lilafarbenes Teleport-Element, welches die Schlange zu einem anderen Teleport-Element teleportiert. Dabei teleportieren die oberen Elemente zum nächsten rechten Element und die unteren Elemente zum nächsten linken Element. Level 6 verändert sein inneres Wandmuster nach dem Verzehren des Food-Elements, jedoch nicht nach Verzehren des Superfood-Elements. Das Spiel endet, wenn die Schlange mit der Wand oder mit sich selber kollidiert.

2.3 Spiel - Mehrspielermodus

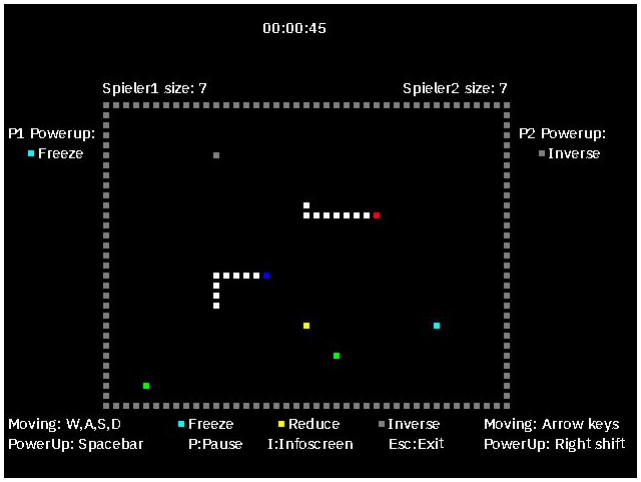

Abbildung 2.5: Mehrspieler Spiel

Beim Mehrspielermodus spielen zwei Spieler gegeneinander und versuchen das Spiel durch töten der Schlange des Gegenspielers zu gewinnen. Spieler eins bewegt die rote Schlange mit den Tasten „W“, „A“, „S“, „D“ und Spieler zwei bewegt die blaue Schlange mit den Pfeiltasten. Im Mehrspielermodus gibt es außerdem Spezial-Elemente, wie das cyanfarbene Freeze-Element, das gelbe Reduce-Element und das graue Inverse Element. Diese Elemente können aufgesammelt werden und mit Leertaste für Spieler eins und mit Shift für Spieler zwei eingesetzt werden. Dabei kann nur ein Element gleichzeitig in der Tasche eines jeweiligen Spielers sein. Beim Einsatz des Freeze-Elementes kann die gegnerische Schlange sich für 15 Schritte nicht bewegen. Beim Einsatz des Reduce-Elementes verringert sich die Länge der gegnerischen Schlange um eine Größe. Das Inverse-Element invertiert die Steuerung des Gegenspielers für 50 Schritte. Das Spiel ist beendet, wenn eines der Spieler verliert. Kollidiert die Schlange eines Spielers mit der Wand, sich selber oder dem Körper der gegnerischen Schlange, verliert derjenige Spieler. Kollidieren beide Schlangen Kopf an Kopf gewinnt derjenige Spieler mit der größeren Schlange.

2.4 Spiel - Spielende

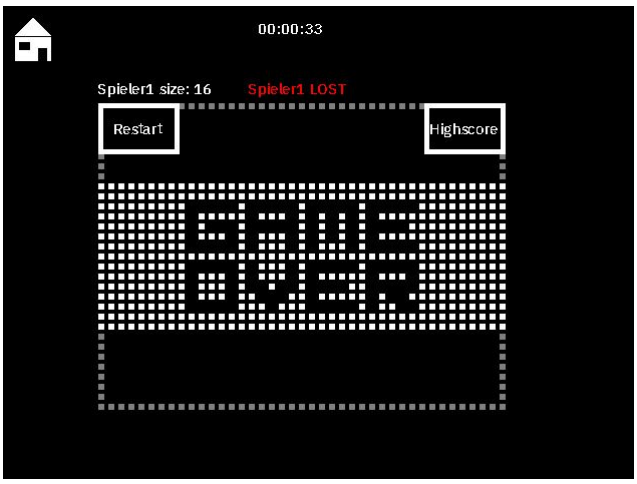

Abbildung 2.6: Spielende

Ist das Spiel zu Ende erscheint eine kleine Animation und mehrere Buttons. Der Restart-Button startet das Spiel neu und der Highscore-Button zeigt den Highscore an. Außerdem steht über dem Spielfeld welcher Spieler gewonnen bzw. verloren hat.

3 Struktur

In diesem Abschnitt wird beschrieben, wie das Spiel in seiner Struktur aufgebaut ist. Dabei werden die Verhältnisse und die Relationen einzelner Elemente miteinander erläutert.

3.1 Zustände

Das Spiel läuft über drei Zustände, da das Spiel über den RTOS-Emulator läuft und der Emulator mit Zuständen arbeitet. Jeder Zustand arbeitet wie ein Prozess (Task), das bedeutet es kann jeweils nur einer dieser Zustände gleichzeitig arbeiten und die anderen kommen in ein Wartebereich.
Im ersten Zustand wird das Hauptmenü angezeigt. Dabei werden die Voreinstellungen und der Spielmodus gewählt. Obwohl in diesem Prozess verschiedene Anzeigen, wie z.b. Levelauswahl oder Informationbildschirm, des Bildschirms existieren, wird lediglich nur der Bildschirm aktualisiert ohne den Zustand zu wechseln.
Der zweite Zustand wird erreicht, indem das wählen des Spielmodus und Eingabe des Namens vom Spieler eingegeben wurde. Somit handelt dieser Prozess um das Spiel an sich, das bedeutet, dass die Spielregeln beachtet und jegliche Veräderungen im Spiel hier verarbeitet werden.
Der letzte Zustand verarbeitet die Highscoreliste. Hierbei werden die besten zehn Spieler mit den höchsten Punkten angezeigt. Dieser Zustand hält die Liste der besten zehn Spieler auf dem aktuellsten Stand und diese Daten werden in einer Datei („Highscore.txt“) gespeichert und exportiert. Insbesondere wird die Datei auch importiert zum Ausgeben der besten Spieler.
Jeder Zustand kann durch Klicken von Buttons erreicht werden und der Spieler merkt diese Übergänge in der Regel nicht.

3.2 Verhältnisse der Elemente

Die Verhätnisse der Elemente werden im Bezug zu den Zustände beschrieben.
Im Hauptmenü werden Elemente wie Knöpfe und Symbole verwendet. Durch Drücken eines der Buttons können mehrere Informationen den Nutzer angezeigt werden. Dabei wird der Bildschirm lediglich erneuert, indem der Hintergrund neu gefüllt wird und dann die neuen Informationen dargestellt werden. Dasselbe geschieht mit den Voreinstellung des

Levels. Die Maus ist auf dem Bildschirm von der Darstellungsebene ganz vorne gesetzt, da die Maus über jedem Objekt sein kann. Entsprechend, falls die Maus sich über einer der Buttons befindet, ändert sich die Farbe des Buttons zu grau, um visuell zu zeigen, dass die Maus sich in diesem Bereich befindet.
Der Highscore Zustand arbeitet sehr ähnlich wie das Hauptmenü. Falls sich die Liste ändert wird der Bildschirm mit dem Hintergrund neugefüllt und die Liste danach visualisiert. Jedes mal beim Betreten dieses Zustandes wird die Liste durch die Datei „Highscore.txt“ importiert, um die aktuellste Liste darzustellen.
Im Spiel werden mehrere Elemente verwendet, wie z.B. das Food-Element, das Wandelement, die Schlange und das Inventar des Spielers. Dabei ist klar je nach Voreinstellung, dass bei einer Kollision von der Schlange mit der Wand, das Spiel zu Ende sein kann. Damit ist die Schlange den Wandelementen untergeordnet. Insbesondere ist die Schlange auch innerhalb des Spielfeldes von der Wand eingeschlossen. Im Fall das die Wände deaktiviert wurden, springt die Schlange einfach in die gegenüberliegende Spielfeldseite. Das Teleport-Element in Level 5 besitzt den selben Rang wie die Wandelemente, da durch zusammentreffen mit der Schlange, die Schlange nur an einer anderen Position verschoben wird. Dagegen sind die PickUp-Elemente wie Food-Element, SuperFood-Element, Reduce-Element, Freeze-Element und Inverse-Element von der Schlange abhängig, weil die Schlange diese Elemente aufnehmen kann. Darüberhinaus dürfen die PickUp-Elemente nicht auf der Schlange erzeugt werden, somit sind diese untergeordneter als die Schlange. Die jeweiligen Schlangen haben jeweils eine Liste als Datenstruktur, damit es einfacher ist Elemente hinzuzufügen und zu entfernen. Durch aufnehmen der Elemente werden einige in ein Inventar gepackt, dabei werden immer die letzten aufgenommen Elemente behalten. Diese Elemente werden jeweils an der Seite des Spielers angezeigt.

4 Spiellogik

In diesem Kapitel gehen wir auf die Hauptfunktion des eigentlichen Spielablaufs, *vGameScreen*(), und der dafür benötigten einzelnen Funktionen ein. Wie in den meisten Computerspielen besteht die Hauptfunkion im groben aus einer einzigen großen Schleife, in der zum einen der aktuelle Spielzustand abhängig von den Eingaben der Spieler geändert wird, und zum anderen der aktuelle Spielzustand grafisch aufgearbeitet und auf den Bildschirm gebracht wird. Am Anfang jedes Spiels findet noch eine Initialisierung statt, welche den Spielzustand auf den Anfang des Spiels setzt.

4.1 Initialisierung

Das erste, was nach Starten des Spiel geschieht, ist das Erstellen einer Art *seed* für die *rand*() Funktion der C - Programmiersprache. Da die *rand*() Funktion die verstrichene Zeit seit Aufruf der Funktion in der sie aufgerufen wird als Basis für die Berechnung der Pseudozufallszahl nimmt, geben die ersten Aufrufe der *rand*() Funktion immer die gleichen Werte zurück, weshalb die Schlangen der Spieler sowie das erste $foodElement$ immer an der gleichen Stelle starten. Um dem entgegenzuwirken berechnet die Funktion als erstes die letzten beiden Ziffern des Produkts der x- und y-Koordinaten der Mausposition beim klicken des Start-Buttons und dekrementiert diese Zahl bis zur null. Durch diese minimale verstrichene Zeitspanne ändern sich die Ergebnisse der ersten *rand*()-Aufrufe und damit der Startzustand des Spielfelds, vorrausgesetzt man drückt beim Start nicht auf den exakt gleichen Pixel wie bei vorherigen ersten Spielen. Danach werden die beiden Variablen *inital* und *levelInitialized* auf *true* bzw. $false$ gesetzt. Diese werden in der nun startenden Hauptschleife benötigt. Die Schleife beginnt, da die eben benannten Initialvariablen so gesetzt wurden, damit, je nach gewähltem Level, die Wände sowie alle weiteren festen Elemente des Levels an ihre entsprechenden Stellen ins $fieldArray$ geschrieben werden. Daraufhin werden die Variablen für die Position der beiden Spieler, $p1X, p1Y, p2X$ und $p2Y$, sowie das erste $foodElement$, auf zufällige Positionen im Spielfeld gesetzt.
Dafür werden die beiden Funktionen $getRandomFreeField$, welche mit der Methode beschrieben im Kapitel 5.2 : Elemente Erzeugen eine freie Stelle auf dem Spielfeld zurückgibt, und $createRandomFood$(), welches auf einer zufälligen Stelle des Feldes ein $foodElement$ erschafft, also die zugehörige Stelle im $fieldArray$() mit der Nummer 10 beschreibt sowie die Variablen $food1X, food1Y$ bzw. $food2X, food2Y$ auf die zugehörigen Koordinaten auf

dem Bildschirm setzt, benutzt. Außerdem werden die zugehörigen verketteten Listen für die Schlangen mit bereits 2 weiteren „Schlangenteilen“ initialisiert. Dies geschieht auch im Einzelspieler-Modus für die Schlange von Spieler 2, da der Code ansonsten aufgrund einer möglicherweise nicht initialisierten Spieler-2-Schlange nicht kompiliert. Einzig der Eintrag im $fieldArray()$ wird im Singleplayer-Modus ausgelassen, da offensichtlich kein Spieler 2 existiert.
Zum Schluss werden noch die boolean-Variablen $p1Ready$ und $p2Ready$ auf $false$ sowie $levelInitialized$ auf $true$ gesetzt, da das Spielfeld jetzt soweit initialisiert ist, dass das Spiel beginnen kann. Da die Variable $initial$ aber noch auf $false$ steht, startet das Spiel nicht direkt, statdessen wird nur der Startzustand, also die Startposition der Schlange sowie die ersten $foodElemente$ und eine kurze Animation gezeigt, in der das Level gezeichnet wird. Dadurch können sich die Spieler zuerst mit dem Level vertraut machen und können in Ruhe ihre Startrichtung ausmachen, so dass das Spiel erst startet, wenn alle bereit sind. Wenn nun der oder die Spieler eine ihrer Richtungstasten betätigen, wird zum einen die Startrichtung für die Schlangen gesetzt, sowie die zum jeweiligen Spieler gehörige $ready$-Variable auf $true$ gesetzt. Erst wenn die $ready$-Variablen aller Spieler auf $true$ stehen, wird die $initial$-Variable auf $false$ gesetzt und das eigentliche Spiel startet, die Schlangen fangen an sich zu bewegen.

4.2 Hauptschleife

Sollte zu diesem Zeitpunkt die Animation des Levels noch nicht abgeschlossen sein, so wird diese abgebrochen und das Level einfach direkt vollständig angezeigt.
Wenn dies alles geschehen ist startet das eigentliche Spiel, und es geschieht jeden Schleifendurchlauf das gleiche, solange, bis das Spiel endet, indem einer der Spieler verliert. Zuerst wird der Zustand der Tastatur durch die Funktion $xGetButtonInput()$ abgerufen, sodass basierend auf den Tastatur-Eingaben der neue Spielzustand berechnet werden kann. Die beiden Funktionen $pause()$ und $infoscreen()$ steuern das pausieren oder weiterführen des Spiels nach dem Drücken der Taste „P“sowie das anzeigen oder verstecken des Hilfe-Textes am unteren Rand des Bildschirms nach Drücken der Taste „I“. Mit der Funktion $incrementSpeed()$ überprüft das Spiel, ob einer der Spieler die größe 25, 50, 75 oder 100 erreicht hat, und falls ja, erhöht es die Spielgeschwindigkeit. Die Funktion $checkGameOver$ überprüft, ob einer der Kriterien für ein Ende des Spiels erfüllt sind, also ob einer der Spieler verloren hat. Ist dies der Fall, stoppt die Bewegung der Schlange und die Funktion startet die Game-Over-Animation und zeigt die Buttons für einen Neustart, die Highscoreliste und das Hauptmenü ein. Außerdem wird, falls ein Highscore erreicht wurde, dieser in die Highscoreliste eingetragen.
Anschließend wird mit $playerOneGetNextDirection()$ bzw $playerTwoGetNextDirection()$ die nächste Richtung der Schlangen bestimmt, wieso dafür eine extra Funktion benutzt

wird erklären wir in Kapitel 5.1 : Bewegung der Schlange.
Der anschließende Teil der Hauptschleife hängt von den beiden Variablen $frame$ und $frameTicks$ ab. $frame$ gibt die aktuelle Zahl der Schleifendurchläufe seit dem letzten Schritt der Schlangen an und wird somit nach jedem Schleifendurchlauf inkrementiert, während frameTicks die Zahl der benötigten Schleifendurchläufe für einen Schritt angibt. Wenn also $frame == frameticks$ gilt, führen die Schlangen einen Schritt aus und $frame$ wird wieder auf 0 gesetzt. Mit Hilfe dieser beiden Variablen werden auch weitere Elemente des Spiel gesteuert: Wenn das Spiel pausiert oder verloren ist wird $frame$ nicht weiter erhöht damit die Schlangen stehenbleiben, und zur Erhöhung der Laufgeschwindigkeit der Schlangen wird $frameticks$ von anfänglich 5 immer weiter dekrementiert. Wenn im Schleifendurchlauf nun $frame == frameticks$ gilt, geschieht folgendes: Alle Spieler machen mit der Funktion *playerStep*() einen Schritt in ihre jeweiligen Richtungen. Dabei wird der Kopf der Schlange um eine Einheit in die aktuelle Richtung bewegt, ein neues Schlangenelement an der Stelle an der der Kopf war der Schlange hinzugefügt und die Stelle im $fieldArray$ mit der Spielernummer beschrieben, sowie das letzte Element der Schlange gelöscht und ihre zugehörige Stelle im $fieldArray$ wieder auf 0 gesetzt. Anschließend wird mit der Funktion *collisionDetection*() überprüft, ob eine der Schlangen mit ihrem Schritt mit etwas zusammengestoßen ist. Dafür wird der Wert des $fieldArray$ an der Stelle des Kopfes überprüft. Ist dieser Wert 1, 2 oder 3, hat der Spieler entweder die Schlange von Spieler 1 oder Spieler 2 oder die Wand getroffen und hat damit verloren. Aber auch das Aufsammeln eines $foodElements$ oder eines $PowerUps$ wird mit der Funktion überprüft. Bevor zum Schluss die Variable $frame$, wie vorher beschrieben, auf 0 gesetzt wird, wird mit der Funktion *powerUps*() der Einsatz der einzelnen $PowerUps$ überprüft. Diese Mechanik erkläre ich im nachfolgenden Unterkapitel.
Danach muss nurnoch der aktuelle Spielzustand auf dem Bildschirm gezeichnet werden ($drawBoard()$) und, falls nach Spielende der entsprechende Button gedrückt wurde, zum Hauptmenü zurückgekehrt werden ($backToMenu()$)

4.3 PowerUps

Es gibt 3 PowerUps im Spiel, welche während des Spiels von den Spielern aufgesammelt und dann aktiviert werden können, wordurch sie verbraucht werden und folgende Effekte auf die gegnerische Schlange haben: Einfrieren, Umkehrung der Steuerung oder Verkürzung der Schlange. Für alle 3 PowerUps gibt es jeweils einzelne $-exists, -X, -Y$ und $-Player$ Variablen, welche jeweils angeben, ob das jeweilige PowerUp aktuell auf dem Spielfeld existiert, deren X- und Y-Koordinaten sowie welcher Spieler aktuell davon betroffen ist. Außerdem gibt es die beiden Variablen $p1PowerUp$ und $p2PowerUp$, welche angeben welches PowerUp jeder Spieler gerade trägt.
Innerhalb der *powerUps*()-Funktion wird zuerst für jedes PowerUp, sowie für das *superFood-*

Element, welches ähnlich funktioniert, überprüft ob es bereits auf dem Spielfeld existiert. Falls nicht, wird eine Pseudozufallszahl berechnet und mit dieser mit einer gewissen Wahrscheinlichkeit (1:50) ein neues PowerUp an einer zufälligen Stelle auf dem Feld erstellt. Falls nun ein Spieler ein PowerUp aufsammelt, was in der *collisionDetection*() festgestellt und bearbeitet wird, wird die entsprechende PowerUp-Variable des Spielers auf den Wert des PowerUps gesetzt. Wenn nun ein Spieler ein PowerUp besitzt und aktiviert, setzt *powerUps*() diese Variable wieder auf 0, sowie die entsprechenden Variablen des PowerUps auf ihre Aktivierungswerte. So wird zum Beispiel zum Einfrieren von Spieler 1 $playerOneFrozen$ auf 15 gesetzt. Solange $playerOneFrozen > 0$ gilt, wird für Spieler 1 anstelle eines Schrittes der Schlange der Wert von $playerOneFrozen$ dekrementiert. Die Schlange von Spieler 1 bewegt sich demnach 15 Schritte lang nicht, ist also eingefroren. Ähnlich funktioniert die Invertierung der Steuerung, hierbei wird die Variable $inverseControlOne$ auf 50 gesetzt und ebenfalls mit jedem Schritt invertiert. Solange die Variable größer 0 ist, wird in der Funktion $playerOneGetNextDirection$() die entgegengesetzte Richtung der eigentlich per Tastendruck angegebenen als $nextDirection$ gesetzt. Das Reduzieren der Schlange funktioniert auf gleicher Weise, hierbei wird lediglich die Variable auf 1 gesetzt und im nächsten Schritt wieder zurück auf 0 gesetzt, zusammen mit der Verkürzung der Schlange.

5 Probleme und Lösungen

Im folgendem werden einige Probleme, die bei der Implementierung des Projektes aufgekommen sind, beschrieben und des weiteren werden verschiedene Lösungsansätze dazu vorgestellt.

5.1 Bewegung der Schlange

Bei der Implementierung der Schlange wurde am Anfang erst ein Quadrat erstellt, danach musste die Bewegung hinzugefügt werden. Da am Anfang noch keine Bildschirmrate gab, konnte die Schlange in jede Richtung in Echtzeit laufen, je nach Eingabe der Richtung. Um die Schlange in direkter Diagonalen Richtungen zu vermeiden, wurde die Bewegung in Horizontaler- und Vertikaler-Achse beschränkt. Das Problem war, dass die Schlange in die entgegengesetzte Richtung laufen kann, d.h. die Schlange konnte durch sich selbst durchlaufen.
Der erste Lösungsansatz war mit Hilfe von einer Variable die aktuelle Richtung zu speichern um je nachdem in die andere Achse zu laufen. Damit wird verhindert direkt in die entgegengesetzte Richtung zu laufen. Jedoch wurde das Problem noch nicht behoben, da durch das gleichzeitige Benutzen von mehreren Richtungseingaben die Variable eine falsche Richtung erhält und somit die entgegengesetzte Richtung erlaubt. Dies kann so schnell passieren, sodass die Schlange die zweite Richtung nicht wahrnimmt und sofort in die entgegengesetzte Richtung läuft.
Deshalb gab es einen endgültigen Lösungsansatz, welcher mit zwei Variablen und mit der Methode, die jeden Schritt der Schlange verarbeitet, arbeitet. Die Variablen sind dabei einmal $p1Direction$ (für Player1 und Player2 $p2Direction$) für die aktuelle Richtung und $p1NextDirection$ (für Player1 und Player2 $p2NextDirection$) für die nächste Richtung. Die Idee mit dem Verhindern der Achse bleibt. Jedoch wird die aktuelle Richtung erst auf die neue Richtung gesetzt, sobald die Schlange sich um ein Feld bewegt.

5.2 Elemente Erzeugen

Relativ früh in der Entwicklung des Spiels machten sich Performanceprobleme beim Erstellen neuer Elemente, vor allem des $foodElement$, bemerkbar. Bei zunehmend langer Schlange erhöhte sich die Rechenzeit für das Finden einer freien Stelle auf dem Spielfeld

so stark, dass zum Beispiel nach dem finden des $foodElements$, eine bemerkbare Pause entstand in der das Spiel nicht weiterlief und somit stockte. Diese Pause war allerdings nicht gleichbleibend lang, sondern erschien zufällig und mit unterschiedlicher Dauer, jedoch mit ansteigender Länge der Schlange öfter und mit erhöhter Dauer.
Dies war zurückzuführen auf unseren Algorithmus zur Bestimmung einer freien Stelle sowie die Speicherung der Schlange in einer Liste. Der Algorithmus suchte pseudo-randomisiert eine Stelle des Spielfeldes aus und überprüfte daraufhin, ob ein Element der Schlange bereits auf diesem Feld ist. Falls dies nicht der Fall war, war das Feld geeignet, ansonsten wurde einfach ein anderes Feld ausgesucht und dies solange wiederholt bis ein freies Feld gefunden wurde. Durch eine längere Schlange stieg somit die Wahrscheinlichkeit, dass ein neues Feld ausgesucht und der überprüfungsprozess deshalb mehrere Male durchgeführt werden musste. Das größere Problem hierbei war aber der Überprüfungsprozess selber, sowie die Speicherung der Schlange in einer verketteten Liste. Um zu überprüfen, ob das ausgesuchte Feld nicht schon durch die Schlange belegt war, musste die gesamte Schlange Element für Element überprüft werden. In Kombination mit der Wahrscheinlichkeit auf mehrere Überprüfungen entstand dadurch bei längeren Schlangen ein so großer Rechenaufwand, dass die Verzögerung durch den Spieler bemerkbar und somit der Spielfluss ins stocken geraten ist.
Zur Lösung des Problems wurde eine neue Variable eingeführt, welche auch in der weiteren Entwicklung noch sehr hilfreich wurde: ein zweidimensionales Array mit Integer-Werten, das $fieldArray$. Das $fieldArray$ dient als kompakte Darstellung des aktuellen Zustands des gesamten Spielfelds. Dabei repräsentierte jeder der 39 x 29 Integer-Werte eine Stelle auf dem Spielfeld und deren aktuellen Zustand. Ist der Wert 0 ist das Feld frei, ist der Wert 1 ist die Stelle von der Schlange von Spieler 1 belegt usw. . Alle Werte und zugehörigen Zustände sind in Tabelle 5.1zu sehen.
Durch das $fieldArray$ ist nun der Prozess des überprüfens einer Stelle nicht mehr abhängig von der Länge der Schlange, sondern kann immer in konstanter Zeit durch einmaliges checken des zugehörigen Wertes im $fieldArray$ erledigt werden. Das Problem der mehrmaligen überprüfungen durch Auswählen eines bereits belegten Feldes ist damit auch trivialisert, da das überprüfen nun schnell genug geht. Durch das $fieldArray$ wurde auch die weitere Implementierung des Spiels vereinfacht, da durch neue Wertezuweisungen im Array relativ einfach neue Objekte wie Wände und PowerUps in die Spiellogik eingeführt werden konnten, und das überprüfen der Stellen an denen diese Objekte sich befinden durch das $fieldArray$ direkt abgedeckt ist.

Field Array	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	0	1	0	0	0
2	0	0	0	1	0	0	0
3	3	3	1	1	0	3	3
4	0	0	0	0	0	0	0
5	3	3	0	0	0	3	3
6	0	0	0	0	0	0	0

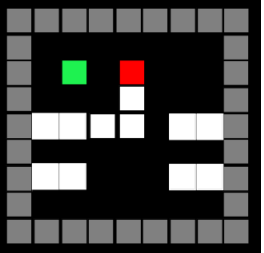

Abbildung 5.1: Darstellung des eines Spielfeldes mit dazugehörigem $fieldArray$

5.3 Spielleistung

Beim Ausführen des Spiels kam es sehr oft zu Problemen mit der Spielleistung. Insbesondere wenn das Spiel mit anderen Programmen, wie BigBlueButton, gleichzeitig lief. Dies führte in den meisten Fällen zur Verlangsamung des Spielfluss. Desweiteren flackerte das Spiel sehr stark, sodass ältere Frames anstatt des neuen Frames gezeichnet wurden. Weitere Probleme waren häufig auftretende Speicherzugriffsfehler vor allem, wenn das Spiel über eine virtuelle Maschine lief. Jener Fehler führte zur Terminierung des Spiels.

Wert	**Objekt**
0	Leer
1	Spieler 1
2	Spieler 2
3	Wand
10	Food-Element
11	Superfood-Element
12	Freeze-Element
13	Reduce-Element
14	Inverse-Element
20	Teleport

Tabelle 5.1: Fieldarray-Werte

Abbildungsverzeichnis

List of Source Codes

GitHub Repository

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift