Gymnasium am Fredenberg

Umsetzung des Spiels „Snake“ in Java

Facharbeit im Seminarfach

Informatik

von

Paul Dinkler

Betreuer: Daniel Müller

Abgabetermin: 26.03.2020

Inhaltsverzeichnis

1. EINLEITUNG

1.1. EINFÜHRUNG IN DAS COMPUTERSPIEL „SNAKE“

Das Spiel Snake gilt als Geschicklichkeitsspiel und gehört heute immer noch zu den Computer-Spiele-Klassikern. Auch wenn es bereits in den 70er bis 80er Jahren diverse Versionen des Spiels, damals genannt „Hyper Wurm“, auf den sogenannten „Arcade Automaten“ gab, wird als Erscheinungsdatum meist auf das Jahr 1997 verwiesen, da damals das Spiel vorinstalliert auf den Nokia Handys erschienen ist (Quelle 1). Selbst der Entwickler Taneli Armanto hatte nicht mit diesem Erfolg gerechnet. „When we created Snake for the Nokia 6110 in 1997, we wanted to give people an entertaining experience, but we never imagined that it would become the classic mobile game. It showed people that you could create a great game for a mobile phone." (Taneli Armanto, 2005, Quelle 2). Das Spielkonzept war damals wie heute das gleiche, auch wenn es inzwischen verschiedene Variationen bzw. Modellierungen des Spiels gibt. Das ursprüngliche Spielkonzept ist jedoch Folgendes: In dem Spiel „Snake“ geht darum auf einer Benutzeroberfläche oder auch allgemein betrachtet einem Feld, ein immer länger werdendes Objekt, in unserem Fall eine Schlange so zu bewegen, dass es weder auf eine Wand noch sich selbst trifft. Die Wand wird meistens mit dem Ende der Oberfläche gekennzeichnet. Wichtig anzumerken ist allerdings, dass sich die Schlange nur in die vier Himmelsrichtungen bewegen kann, auch wenn es mittlerweile Versionen gibt, bei denen sich die Schlange schräg bewegen kann (Quelle 3). Diese Bewegungen können auf ganz unterschiedliche Weise ausgeführt werden. Während anfangs Cursor-Pfeiltasten oder eine normale Alpha-Tastatur verwendet wurden, ermöglicht es die heutige Technologie die Schlange just mit Hilfe des „Touch-Screens“ zu bedienen (Quelle 3). Damit das Objekt bzw. die Schlange länger wird, muss es „Happen“ einsammeln, die sich irgendwo auf dem Spielfeld befinden und immer wieder neu erscheinen, sobald die vorherige gegessen wurde. Die „Happen“ haben die Größe eines Kästchens. Bei manchen Versionen des Spiels gibt es des Weiteren „Wände“, die zufällig auf dem Spielfeld platziert sind um so die Bewegungen der Schlange einzuschränken. Um die Schwierigkeit des Spiels zu erhöhen, ist auch eine Geschwindigkeitsmodellierung in manchen Versionen nicht unüblich. Gewonnen hat man schlussendlich, wenn die Schlange das volle Spielbrett bedeckt und so der maximale „Score“ erreicht wurde (Quelle 3).

1.2. Ziel der Arbeit

Das Ziel der Facharbeit ist, ein derartiges Programm mit Java für den Computer lauffähig zu entwickeln und für den Leser wissenschaftlich verständlich zu erklären. Allerdings werde ich einige Veränderungen in das ursprüngliche Spiel einbringen, die den Spielspaß erhöhen werden.

1.3. Überblick auf mein Vorgehen

Ich habe mir bei der Entwicklung meines Programms zuerst überlegt, welchen Umfang das Spiel haben soll und welche Elemente existieren müssen. Dafür habe ich im Wesentlichen eine Quelle benutzt, die mir auch Hilfestellung bei verschiedenen Details gab (Quelle 4). Im Anschluss habe ich meine Ideen mithilfe eines Struktogramms auf das Papier gebracht. Dies ist ein „grafisches Hilfsmittel bei der Programmentwicklung zur Darstellung eines Algorithmus“ (Prof. Dr. Richard Lackes, 2017). Nachdem ich mir überlegt habe, welche Funktionalitäten bzw. Eigenschaften das Programm haben soll, habe ich begonnen das Programm zu schreiben. Dabei habe ich zuerst die notwendigen Fenster erstellt und im Anschluss die Spiellogik mittels Methoden und Klassen implementiert.

2. Entwicklung des Programms

Der folgende Abschnitt 2.1 der Facharbeit dient nur zur ungefähren Darstellung des Programms. Es schließt nicht vollständig alle Methoden und Attribute der einzelnen Klassen mit ein, sondern soll lediglich einen Überblick auf die Funktionsweise des Programms und die Spiellogik liefern. Die Darstellung mit den Methoden und Klassen sowie ihre Analyse erfolgen in Abschnitt 2.3 und 2.4.

2.1. Grundlegendes Modell des „Snake-Games“

Das „Snake-Spiel“ startet mit einem Menüfenster, auf dem vier Schwierigkeitsgrade sowie „Spielregeln“ und „Beenden“ in Form von Knöpfen auszuführen sind.

Beim Starten des Spiels, also dem Ausführen eines Schwierigkeitsgrades, öffnet sich eine neue Spieloberfläche.

Nach dem Starten des Spiels, also dem Initiieren des Schlangenkopfes und eines Apfels, sollte permanent überprüft werden, ob die Schlange einen Apfel bzw. vergifteten Apfel gegessen hat oder ob sie gegen sich oder eine Wand gefahren ist.

Isst die Schlange einen Apfel, wird sie um eine Einheit länger, isst sie einen vergifteten Apfel wird sie um eine Einheit kürzer. Wenn sie gegen sich oder die Wand fährt startet das Spiel neu. Im Anschluss bewegt sich die Schlange weiter und die Aufrufe werden erneut getätigt, bis das Spiel vorbei ist.

2.2. Begründung für die Wahl von Java

Ganz grundlegend ist Java eine Objektorientierte Programmiersprache (OOP). Das bedeutet, dass der Code in Form von Klassen und Methoden geschrieben wird und so Objekte bzw. Instanzen erstellt werden können (Quelle 5). In dem Snake-Programm müssen verschiedene Objekte mit speziellen Eigenschaften erzeugt werden, weswegen ich Java als Programmiersprache gewählt habe. Ein weiterer Vorteil der OOP ist die Vererbung, also der Weitergabe von Eigenschaften und Methoden über mehrere Klassen hinweg. Zunächst einmal gibt es zwei Schlüsselwörter die sich mit der Vererbung befassen: „extends“ und „implements“. „Extends“ wird benutzt um anzugeben, dass eine neu definierte Klasse von der Klasse die sie erweitert, also der Basisklasse abgeleitet wird, um so ihre Funktionalität zu erhalten. Als Beispiel lassen sich hier die Basisklassen „JFrame“ und

„JPanel“ anführen, die ich beide in meinem Programm verwende (später mehr dazu) (Quelle 6).

Das Schlüsselwort „implements“ hingegen bezieht sich nur auf die Implementierung einer Schnittstelle, die auch als „interface“ bezeichnet und am Beispiel des „ActionListeners“ im Folgenden erläutert wird. Als Schnittstelle wird ein Klassentyp bezeichnet, der nur abstrakte Methoden enthält. Das bedeutet, dass die Methoden in dieser Klasse nur deklariert sind, jedoch nichts im Methodenrumpf steht. Anhand von Schnittstellen kann man also erkennen, was eine Klasse machen soll aber nicht wie. Nach der Implementierung solcher Klassen müssen jene Methoden mit in das Programm übernommen/überschrieben werden, sodass die Methode nun weiß, „wie“ sie eine bestimmte Handlung vollziehen soll (Quelle 7). Überschrieben wird eine Methode anhand des Schlüsselwortes „@Override“. Dieses Vorgehen wird anhand des Codes noch deutlicher (Quelle 1).

Wie im Folgenden näher beschrieben wird, benutze ich für die graphische Anwendung des Spieldesigns „Swing“. „Java Swing“ gehört zu den „Java Foundation Classes“ (JFC) und wird dazu benutzt oberflächenbasierte Anwendungen zu programmieren (Quelle 2). Die dazugehörigen Klassen befinden sich im „javax.swing.“-Paket. Zunächst sollte man sich die Frage stellen: Was ist das „Swing“-Paket? Alle in dem Swing Paket enthaltenen Elemente werden in vier unterschiedliche Kategorien unterteilt: 1. Fenster und Dialoge, 2. Menüs, 3.Container, 4. Bedienelemente (Quelle 8).

1. Fenster und Dialoge: Unter diese Kategorie fällt zum Beispiel das in diesem Programm verwendete „JFrame“. Diese Klasse bildet das Grundfenster des Programms auf dem anschließend diverse Container und Bedienelemente hinzugefügt und so zu einer funktionalen Oberfläche verknüpft werden. Dialoge werden in meinem Programm nicht verwendet, nichts desto trotz sind es Nachrichten an den Benutzer, die beispielsweise in Form einer Fehlermeldung ausgetragen werden (Quelle 9).

2. Diese Kategorie habe ich komplett weggelassen, weil sie zur Erstellung eines Snake-Spiels nicht benötigt wird. Als Beispiel lässt sich hier das „JMenu“ anführen, das einen Container darstellt, dem beliebige Auswahlmöglichkeiten zur Veränderung des jeweiligen Interfaces hinzugefügt werden können (Quelle 10).

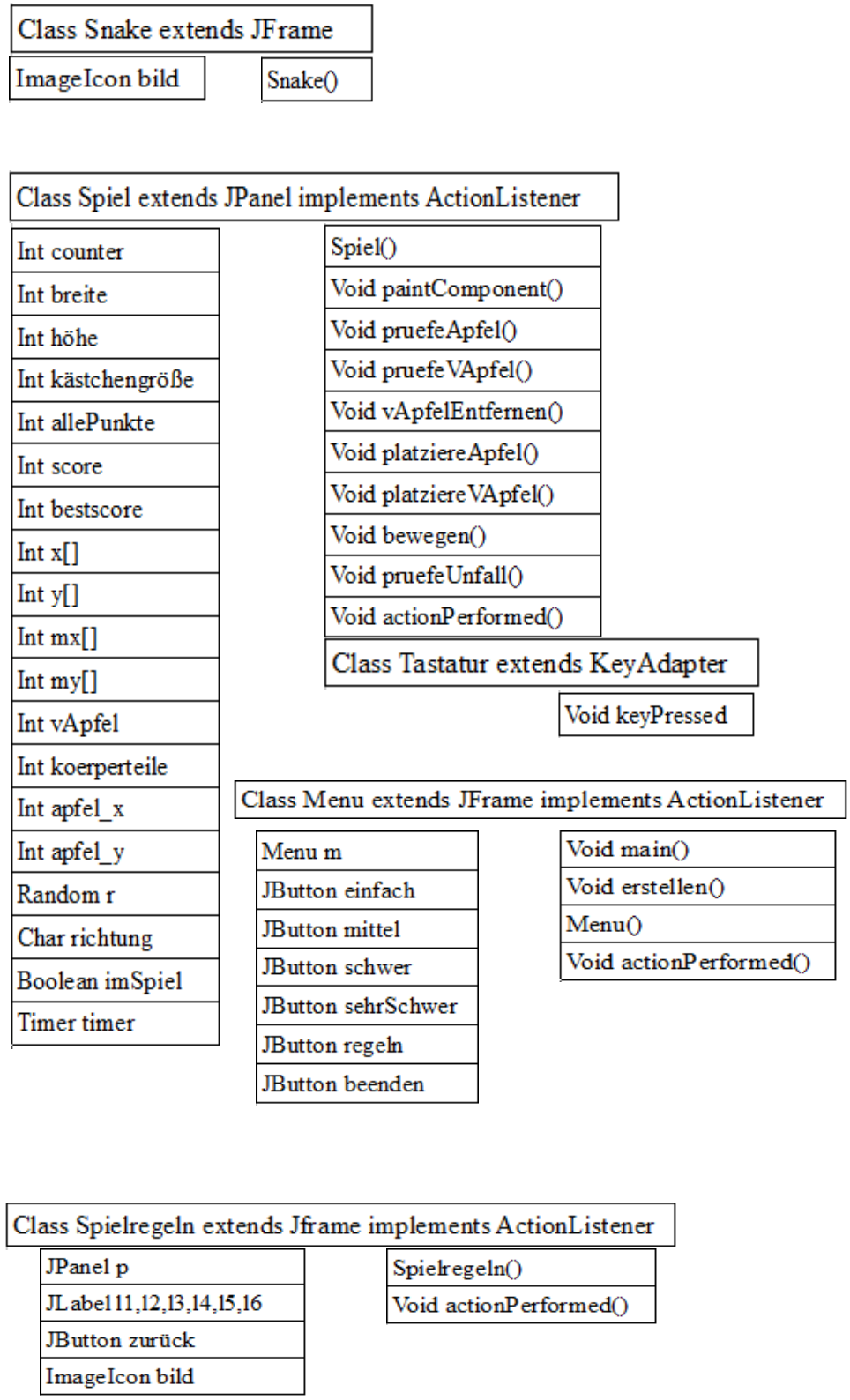
3. Container sind in jeder graphischen Applikation vorhanden, da sie zur Strukturierung der Bedienelemente dienen. In meinem Programm wird ein „JPanel“ benutzt, das als Grundlage der Grafikprogrammierung dem „JFrame“ hinzugefügt wird. Dazu ist es der Standard-Container, auf welchem die verschiedenen Bedienelemente platziert werden (Quelle 11).

4. Die vierte Kategorie stellen die Bedienelemente dar, die ebenfalls für jede graphische Benutzeroberfläche unvermeidlich sind, da sie für die Interaktion mit dem Benutzer verantwortlich sind. Da in meinem Programm nur „JButtons“ und „JLabels“ vorhanden sind, werde ich auch nur auf diese eingehen. „JButtons“ sind Knöpfe die bei Betätigung eine bestimmte Aktion ausführen sollen. „Jlabels“ hingegen sind einfach Texte, die allerdings auch nicht veränderbar sind. Beide werden dem „JPanel“ hinzugefügt, welches danach dem „JFrame“ beigefügt werden muss (Quelle 12).

In der Grundversion eines Programms haben die Bedienelemente noch keine Funktionalität, da zum Beispiel dem „JButton“ noch nicht „mitgeteilt“ wurde, welche Aktion ausgeführt werden soll, wenn er betätigt wird. Das lässt sich durch sogenannte „Events“ lösen. In meinem Programm gibt es das „Event“ „Tastatureingabe“, das für das Steuern der Schlange eingesetzt wird, und das „Event“ „Klicken der Maustaste“, das für das Bedienen der Knöpfe verwendet wird. Um ein „Event“ zu erzeugen, muss zunächst einmal der richtige „Event-Handler“ und seine erforderlichen „Event-Methoden“ implementiert werden. Also die abstrakte Klasse, die eine Verbindung zur Quelle des Ereignisses haben kann. Tritt ein Event auf, wird auch nebenbei ein Objekt erzeugt, welches zu einer der Unterklassen von dem „java.awt“-Paket gehört und somit alle notwendigen Informationen über das „Event“ hat. Im Falle eines „JButtons“ wäre jene Klasse der „ActionListener“, mit der Methode „actionPerformed()“. Zur Ausführung der gewünschten Aktion müsste jetzt nur noch die „actionPermormed()“-Methode überschrieben werden, indem abgefragt wird, ob die Quelle der jeweilige „JButton“ ist und anschließend der Code eingetragen wird. Das gleiche Prinzip gilt für die „Tastatur“, jedoch ist hier der „Event-Handler“ der sogenannte „KeyAdapter“ und die zu überschreibende Methode ist die „KeyPressed()“-Methode (Quelle 13, Quelle 14, Quelle 15).

2.3. METHODEN, KLASSEN UND ATTRIBUTE ALS MODELLIERUNG

140



2.4 Beschreibung der Klassen, Methoden und Benutzeroberflächen

Die Klassen „Snake“, „Menü“ und „Spielregeln“ werden im Folgenden kurz erklärt, jedoch nicht im Detail beschrieben, da sie lediglich den Rahmen des Programms festlegen. Die „Spiel-Klasse“ wird dagegen ausführlich dargestellt, da sie das Herzstück des Programms ist.

Die vom Programm ausgeführte „main()“-Methode ist Teil der Menüklasse, sodass es sinnvoll ist, diese Klasse zuerst zu analysieren. Zunächst einmal erweitert die Menüklasse die Klasse „JFrame“ und implementiert den „ActionListener“. Damit man allerdings nicht nur ein bloßes Fenster hat, müssen später Komponenten wie zum Beispiel „JLabels“ oder „JButtons“ hinzugefügt werden.

Die Klassenvariablen sind die sechs „JButtons“: „beenden“, „regeln“, „einfach“, „mittel“, „schwer“ und „sehrSchwer“. Des Weiteren wird eine Instanz der Klasse „Menü“ deklariert.

Widmen wir uns zu allererst dem Konstruktor der Klasse. Dieser ist lediglich für die Initiierung der „JButtons“ zuständig.

Die „main()“-Methode ist die Methode, die schließlich vom Programm ausgeführt wird. In dieser ist allerdings nur der Aufruf der „erstellen-Methode“, weswegen wir uns auch nun mit dieser beschäftigen.

Hier wird die bereits deklarierte Instanz der Klasse „Menü“ initialisiert, indem einige Standardfunktionen angewendet werden (Anhang - *Code 3)*.

Die letzte Methode ist die „actionPerformed()“-Methode, die den „JButtons“ ihre jeweilige Funktion zuweist (Anhang - *Code 4)*. Wählt man einen Schwierigkeitsgrad, wird die Klasse „Snake“ mit diesem als Parameter aufgerufen.

Fahren wir nun mit einer etwas kleineren Klasse fort, die zwar für das Spielgeschehen nicht relevant ist, jedoch das Menü trotzdem abrundet, die Klasse „Spielregeln“. Diese Klasse erweitert ebenfalls die Klasse „JFrame“ und implementiert auch die Klasse „ActionListener“. Als Klassenvariablen werden ein „JPanel“, sechs „JLabels“ und ein „JBut-

ton“ deklariert. Im Konstruktor der Klasse werden wieder einige JFrame-Standardprozeduren durchlaufen (Anhang - *Code 6)*. *Mit* dieser Standardprozedur wird der Button „zurück“ in der uns bekannten Weise initialisiert. Nun folgt die Initialisierung des „JPanels“ und der „JLabels“. Für die „JLabels“ geben wir den gewünschten Text ein und setzen ihre x-und y-Koordinaten auf die gewünschten Werte (Anhang - *Code 8)*. Die zweite und letzte Methode in der Klasse Spielregeln, ist die „actionPerformed()“-Methode, die sich auf den „JButton“ „zurück“ bezieht.

Die dritte Klasse ist die Klasse „Snake“ die das „JFrame“ erweitert. Als sehr kleine Klasse besitzt sie nur den Konstruktor, in welchem die Klasse „Spiel“ als „JPanel“ hinzugefügt wird. „Snake“ ist also lediglich das Fenster, auf welchem wieder einige typische Methoden angewendet werden (Anhang - *Code 9)*.

2.4. Kern des Programms: Klasse „Spiel“

Kommen wir nun zum Herzstück des Programms, der Klasse „Spiel“. Wie schon gesagt, ist hier die Spiellogik und das Spieldesign implementiert. Diese Klasse erweitert die Klasse „JPanel“ und implementiert den „ActionListener“, dessen Aufgabe ein wenig von der des „ActionListeners“ der anderen Klasse abweicht, dazu später. Starten wir zunächst einmal mit dem Spielfeld. Das Spielfeld an sich besteht zunächst einmal aus zwei Faktoren, der Spielfeldhöhe und der Spielfeldbreite, die theoretisch veränderbar sind. Jedoch sollte darauf geachtet werden, dass beide Variablen auf dem Wert 600 bleiben, da es sonst Formatierungsfehler mit der rechts anliegenden Leiste gibt, die eine Breite von 300 hat. Diese Variablen geben an wie viele Pixel sich in x-und in y-Richtung befinden. In dem 600 *600 Pixel Feld sind Kästchen eingebettet, in denen die Spielelemente erscheinen können. In meinem Programm habe ich die Kästchengröße auf 25 gesetzt, was bedeutet, dass ein Kästchen eine Länge und Breite von 25 Pixel hat. Da die Höhe als auch die Breite des Feldes auf 600 gesetzt sind, gibt es in x-Richtung, als auch in y-Richtung 600/25=24 Kästchen. Die nächste definierte Variable „allePunkte“ berechnet sich aus dem Produkt der Breite und der Höhe des Spielfeldes, dividiert durch die Kästchengröße zum Quadrat, in unserem Fall 600*600/(25*25)=(576). Ein weiteres wichtiges Attribut ist „geschwindigkeit“. Dieser Integer (ganzzahliger Wert) steht im direkten Zusammenhang mit dem „timer“, einer weiteren Variablen, die im Konstruktor aufgerufen wird. Die Geschwindigkeit steht in Abhängigkeit zum gewählten Spielmodus. Bei „einfach“, „mittel“, „schwer“, „sehr schwer“ sind die Geschwindigkeiten „200“, „170“, „140“, „110“. Diese

Zahlen geben in Millisekunden an wie schnell das dargestellte Spielbild „erneuert“ wird. Gemeint ist also, dass bei einer eher geringen Zahl wie 110 sich mehr Bilder pro Zeiteinheit ändern und das Spiel schneller und damit schwerer ist. Kommen wir zurück zu den Variablen. Die Attribute „counter“, „score“ und „bestscore“ sind zwar nicht für die Spiellogik essentiell, geben allerdings wichtige Informationen für das Spiel an. „counter“ zählt die Bewegungen der Schlange, „score“ gibt die Länge der Schlange an, und „bestscore“ gibt an, welche Schlangenlänge in einer Periode von Spielen maximal erreicht wurde. Eine weitere Variable ist „körperteile“, die angibt, wie lang die Schlange zum derzeitigen Stand ist. Startet das Spiel ist die Länge der Schlange 1. Es gibt also am Anfang nur den Kopf. Als nächstes werden zwei Integer-Arrays mit der Größe „allePunkte“ erstellt. Die beiden Arrays „x[]“ und „y[]“geben die x-Koordinaten und die y-Koordinaten der Körperteile der Schlange an, auf denen sich die Schlange zu einem bestimmten Zeitpunkt befindet. Die Arrays werden in der Größe „allePunkte“ erstellt. Es kann zwar aufgrund der vergifteten Äpfel nicht diese Länge an Körperteilen erreicht, allerdings ist dieser geringe Speicherplatz vernachlässigbar, weswegen ich die Länge so definiert habe. Wie bereits erwähnt, habe ich das Spiel etwas modifiziert. Ich spreche von den vergifteten Äpfeln, wobei immer einer erscheint, wenn die Schlange einen Apfel gegessen hat. Da dieser auch eine Position auf dem Spielfeld belegt, habe ich auch für ihn zwei Arrays „mx[]“ und „my[]“ mit der Größe „allePunkte“/2 angelegt. Wobei es hier zu differenzieren gilt: Falls das Spielfeld eine Breite und Höhe hat, die beide ungerade sind, ist die maximale Anzahl vergifteter Äpfel („allePunkte“-1)/ 2, da bereits ein Kästchen durch den Schlangenkopf besetzt ist und erst danach pro Apfel bzw. Körperteil ein vergifteter Apfel erscheint. Sind Breite und Höhe gerade, so gibt es („allePunkte“-2)/2 vergiftete Äpfel, da zum Schluss genau ein Kästchen frei bleibt, dass von einem Apfel besetzt wird, wodurch wir zunächst ein Kästchen für den Schlangenkopf und ein Kästchen für den letzten Apfel abziehen müssen, damit sich die Äpfel und vergifteten Äpfel genau aufteilen. Da diese Unterscheidung allerdings noch etwas Code benötigen würde, habe ich für die Größe vereinfachend „allePunkte“/2 angegeben. Aufgrund der Tatsache, dass es immer nur einen Apfel auf dem Spielfeld gibt, werden hier keine weiteren Arrays benötigt, weswegen ich nur zwei Variablen „apfel_x“ und „apfel_y“ erstellt habe, die die derzeitige Position des Apfels angeben. Diese ändern sich natürlich nachdem ein Apfel gegessen wurde. Des Weiteren gibt es die Variable „vApfel“, die anfangs mit 0 initialisiert wird und sich immer dann um eins erhöht, wenn ein weiterer vergifteter Apfel er-

scheint. Es bleiben noch drei weitere Attribute. Zunächst gibt es die Variable „im-Spiel“ (ein Boolean), die solange „true“ ist, bis der Benutzer das Spiel gewonnen hat. Auch initialisiert wird ein „Random-Objekt“. Dieses wird im folgenden Code benutzt, um diverse zufällige Werte zu generieren, die beispielsweise als Koordinaten des Apfels bzw. des vergifteten Apfels fungieren. Die letzte Variable „richtung“ ist vom Typ „Char“ und nimmt die Werte 'R', 'L', 'U' oder 'O' an. Diese Werte beschreiben die momentane Richtung des Schlangenkopfes - rechts, links, unten oder oben -, wobei sie am Anfang auf 'R' also rechts gesetzt ist.

Fangen wir nun mit der ersten Methode an, dem Konstruktor. Dieser wird natürlich erstellt, wenn wir die Klasse „Spiel“ aus der Klasse „Snake“ aufrufen. Als erstes nimmt der Konstruktor die Variable „geschwindigkeit“ entgegen, die direkt im Anschluss dem Klassenattribut „geschwindigkeit“ ihren Wert zuweist. Als nächstes wird die Größe des Spielfeldes mit den uns bekannten Werten initialisiert, wobei in x-Richtung weitere 300 Pixel hinzugefügt werden, damit, wie bereits erwähnt, dort die Nebeninformationen des Spiels angezeigt werden können. Des Weiteren wird der Hintergrund mit blau initialisiert, das Programm fokussierbar gemacht und ein „KeyListener“ hinzugegeben, der als Parameter ein Objekt der Klasse „Tastatur“ entgegennimmt. Der „KeyListener“ verhält sich ähnlich wie der „ActionListener“, der Unterschied ist jedoch, dass der „KeyListener“ nur auf Tastatureingaben reagiert und nicht auf die Betätigung verschiedener Bedienelemente auf der Benutzeroberfläche. Die „Tastatur“ ist eine Klasse, die in der Spiel-Klasse definiert ist, auf welche ich später noch eingehen werde. Anschließend werden im Konstruktor die x- und y-Koordinaten des Schlangenkopfes initialisiert. Die x-Koordinaten verlaufen steigend von links nach rechts und die y-Koordinaten verlaufen steigend von oben nach unten. Dazu erscheint der erste Apfel mit der „platziereApfel()“-Methode und der „Timer“ wird mit der Geschwindigkeit gestartet. Der „Timer“ sorgt dafür, dass nach einem bestimmten Zeitintervall, in unserem Fall der Geschwindigkeit in Millisekunden, eine bestimmte Aktion ausgeführt wird. Wir benutzen ihn, um das Bild nach dem vorgegebenen Zeitintervall zu aktualisieren. Aufgrund der Tatsache, dass die Klasse „Spiel“ den „ActionListener“ implementiert, hat sie natürlich auch eine „actionPerformed()“-Methode, die sich auf den Boolean „imSpiel“ als „ActionEvent“ bezieht. Das bedeutet solange die Variable „imSpiel“ auf „true“ gesetzt ist werden folgende Funktionen ausgeführt: prüfeApfel(), prüfeVApfel(), prüfeUnfall(), bewegen(). Nach dem Ausführen wird unabhängig von der „imSpiel“-Variable die „repaint()“-Methode aufgerufen. Dies ist eine

Methode aus dem java.awt-Package und sorgt dafür, dass alle Komponenten des Spiels wieder gezeichnet werden, indem sie die paintComponent()-Methode aufruft. Dafür werden zunächst alle graphischen Komponenten kurzzeitig gelöscht und im Anschluss mit den Veränderungen wieder gezeichnet.

Fangen wir mit der Methode „prüfeApfel()“ an. Diese Methode soll überprüfen, ob die Schlange gerade einen Apfel isst, indem sie die Koordinaten des Schlangenkopfes mit der derzeitigen Apfelposition abgleicht. Wenn dem so ist, erhöhen sich die „körperteile“ und der „score“ um eins. Anschließend wird überprüft, ob der „score“ größer als der „bestscore“ ist, um den „bestscore“ eventuell auf den „score“ zu setzten, falls der „score“ größer ist. Danach werden die Methoden „platziereApfel()“ und „platziereVApfel()“ aufgerufen. In der „platziereApfel()“-Methode werden für die x-bzw. y-Koordinate des Apfels zufällige Werte generiert. Dies geschieht, solange bis die Werte nicht mehr auf dem Schlangenkörper liegen. Genauso verhält sich auch die Methode „platziereVApfel()“. Zu differenzieren ist jedoch, dass diese Funktion immer den letzten Wert des mx[] und my[]-Arrays abruft, da diese die momentanen Koordinaten des „vergifteten Apfels“ darstellen. Des Weiteren wird hier abgeglichen, ob sich der neue vergiftete Apfel auf der Position eines Vorherigen befinden. In diesem Fall wird die Funktion nochmal ausgeführt. Die „vApfel“-Variable, die angibt wie viel vergiftete Äpfel es gibt, wird danach um eins erhöht.

Nach der Überprüfung des Apfels muss der Zusammenstoß mit einem vergifteten Apfel geprüft werden. Dies erfolgt mit der „prüfeVApfel()“-Methode. Dabei wird die Kopfposition mit den Koordinaten aller „vergifteten Äpfel“ abgeglichen. Stimmen sie über ein, vermindern sich der „score“ und die „körperteile“ um eins. Im Anschluss wird die „vApfelEntfernen()“-Methode aufgerufen. Diese versetzt die x-Koordinate des jeweiligen v-Apfels auf 1000, sodass er sich nun außerhalb des Fensters befinden und so nicht mehr sichtbar ist. Diese Methode ist zwar nicht besonders effizient, allerdings muss das Array nicht verändert werden, was die Prozedur deutlicher einfacher gestaltet.

Wichtig ist auch die Methode „prüfeUnfall()“, da diese, falls sich die Schlange gegen die Wand oder sich selbst bewegt, das Spiel abbricht und neu startet. Dafür werden die Kopfkoordinaten mit den Koordinaten aller Körperteile, die größer als drei sind verglichen, da die Schlange keine Körperteile mit Index< 4 mit dem Kopf berühren kann. Wenn sie

übereinstimmen, bedeutet das, dass das Spiel verloren ist und der „Score“, „VApfel“(Anzahl vergifteter Äpfel) und der „counter“ auf null und die Körperlänge der Schlange auf eins gesetzt werden. Der zweite Teil der Methode beschreibt den Abgleich des Schlangenkopfes mit den Koordinaten der Wände. Die Folgen dieses Abgleichs sind die Gleichen wie die des Vorherigen. Dazu kommt allerdings, dass die Richtung der Schlangenbewegung auf 'R' und die Startposition neu gesetzt wird.

Die letzte ständig abgefragte Methode ist die „bewegen()“- Methode. Zunächst wird der „counter“ um eins erhöht und jedes Körperteil der Schlange erhält die Koordinaten des Vorherigen, bis auf das letzte Körperteil. Anschließend wird die Variable „richtung“ auf die vier Möglichkeiten 'R','L','U','O' geprüft und die neuen Koordinaten des Schlangenkopfes dementsprechend angepasst. Ist die Richtung zum Beispiel auf 'L' gesetzt, so wird die x-Koordinate des Schlangenkopfes um eine volle Kästchengröße vermindert, da sich der Kopf nur in horizontaler Richtung nach links bewegt.

Die letzte Methode ist die „paintComponent()“-Methode. Sie steht im Zusammenhang mit der „repaint()“-Methode und wird überschrieben, um im „JPanel“ das gewünschte Design zu erhalten. Diese Methode zeichnet nach jedem Aufruf der „repaint()“-Methode das aktuelle Spielfeld. Bezogen ist dies auf den Schachbrettartigen Hintergrund, die Spielelemente und die Leiste auf der rechten Seite, die Informationen für das Spiel liefert (Anhang – 5.2 *Bild 1)*. Anzumerken ist auch, dass bei einem Spielgewinn (alle Felder sind entweder mit dem Schlangenkörper oder einem vergifteten Apfel besetzt) alle Spielelemente entfernt werden und lediglich in grüner Schrift der Satz:

„Spiel vorbei, du hast gewonnen!!!“

auf einem blauen Hintergrund zu sehen ist.

3. Schlussbemerkungen

3.1. Mögliche Erweiterungen des Spiels

Der Anreiz des Spiels ist, einen immer höheren Highscore zu erreichen, da das Spiel schwerer ist, wenn die Schlange an Länge zunimmt. Eine mögliche Erweiterung wäre es, seine Bestleistungen mit seinen Freunden zu vergleichen und eventuell aus diesem Single-playergame ein Multi-playergame zu machen. Dies ließe sich entweder Online oder sogar auf demselben Computer umsetzten. Dazu müsste lediglich eine weitere Schlange initialisiert werden. Das bedeutet es müssten zwei weitere „Arrays“ erstellt werden, in denen die x- bzw. y-Koordinaten gespeichert werden. Des Weiteren müssten die Bewegungen der zweiten Schlange durch neue Tasten ersetzt und zugewiesen werden - sowie einige andere Attribute, wie z.B. „Score2“ oder „richtung2“, für die derzeitige Richtung der zweiten Schlange. Natürlich müssten auch die Methoden angepasst und entsprechend verändert werden. Um die Schwierigkeit des Spiels weiterhin zu erhöhen könnten außerdem Wände in dem Spiel platziert werden. Dafür müsste allerdings überprüft werden, ob die Wände einen bestimmten Bereich auf dem Spielbrett eingrenzen. Wenn dem so wäre, könnte sich die Schlange niemals auf diesem Bereich bewegen und so, wenn ein Apfel dort erscheinen würde, das Spiel nicht gewinnen.

Meine dritte und letzte Erweiterung des Spiels ist etwas schwieriger. Die Idee ist es, dass sich die Schlange auch schräg bewegen kann. Zu beachten ist hierbei die Anzahl der Kästchen auf dem Spielfeld. Diese müsste um ein Vielfaches erhöht werden, wodurch auch die Schlangenbreite nicht nur aus einem Kästchen bestehen würde. Notwendig ist dies, weil sich die Schlange immer nur von Koordinate zu Koordinate bzw. von Kästchen zu Kästchen bewegen kann. Gibt es nun sehr viele Kästchen, aus denen die Schlange besteht, könnte sie, wenn z.B. die Pfeiltasten „rechts“ und „oben“ gleichzeitig gedrückt werden, immer zwischen diesen hin und her wechseln. Das würde den Anschein erwecken, dass sich die Schlange nach rechts oben bewegt, auch wenn sie sich strenggenommen nur nach rechts und anschließend nach oben bewegt.

3.2. Zusammenfassung

Diese Facharbeit sollte dem Leser oder der Leserin nicht nur einen Einblick in das Spiel „Snake“, sondern auch einen allgemeinen Einblick in die Spieleprogrammierung liefern. Dies ist wichtig, da viele Aspekte des Programms auf andere Spiele übertragbar sind.

Zum Beispiel braucht jedes Spiel ein Fenster oder eine graphische Oberfläche, auf der das Spieldesign implementiert wird. Dafür bieten die „Swing“-Klassen, wie bereits erörtert, einen sehr guten und weiträumigen Gestaltungsspielraum. Anzumerken ist auch, dass die Facharbeit einen verständlichen Weg zur Findung der Spiellogik eines Programms liefert, denn dies ist ja schließlich das, was ein Programm ausmacht. Dies kann, wie in Abschnitt 1.3, mithilfe eines Struktogramms erreicht werden.

3.3. Ausblick auf die Spielezukunft

Mit Spielen wie „Snake“ oder „Tetris“ wurde Ende des 20. Jahrhunderts der erste Schritt zum heute riesigen Angebot von Videospielen auf der ganzen Welt gemacht. Über die Jahre hinweg entwickelten sich die Spiele natürlich weiter, ebenso die Programmiersprachen und Klassen mit denen sie programmiert wurden. Aus den 2D Computer-Spiele-Klassikern wurden 3D-Verwirklichungen, die auf so gut wie jeder Bildschirmoberfläche spielbar sind. Damals wie auch heute dienen Spiele der Unterhaltung und werden in der fortschreitenden Digitalisierung unserer Welt auch immer präsenter, wenn die Menschen in ihrer Freizeit eine Beschäftigung suchen. Je realer die Spiele sind, desto größer ist der Platz, den sie in unserem Leben einnehmen und desto eher verdrängen sie die Spiele, die in der „echten Welt“ ausführbar sind. Worauf ich hinaus will, ist, dass Videospiele immer Teil unseres Lebens sind und auch sein werden. Deshalb ist es für jeden sinnvoll, diese nicht nur zu spielen, sondern auch zu verstehen, um so den Weg in die eigene Spieleprogrammierung zu finden und dies anderen Menschen begreiflich zu machen.

4. QUELLENANGABEN

Quelle 1: http://www.snakesite.de/der-klassiker-snake/ , 19.03.2021, 13:02 Uhr

Quelle 2: https://www.spielbar.de/spiele/145443/snake-1997 , 19.03.2021, 13:06 Uhr

Quelle 3: https://de.wikipedia.org/wiki/Snake_(Computerspiel), 19.03.2021, 19:00 Uhr

Quelle 4: https://zetcode.com/javagames/snake, 01.03.2021, 04:31 Uhr

Quelle 5: https://www.inf-schule.de/programmierung/oopjava, 20.03.2021, 13:53 Uhr

Quelle 6: https://www.geeksforgeeks.org/extends-vs-implements-in-java/, 20.03.2021, 13:56 Uhr

Quelle 7: https://www.geeksforgeeks.org/interfaces-in-Java/ , 18.03.2021, 13:59 Uhr

Quelle 8: https://www.java-tutorial.org/swing.html, 18.03.2021, 14:12 Uhr

Quelle 9: https://www.java-tutorial.org/dialoge-und-fenster.html, 18.03.2021, 15:01 Uhr

Quelle 10: https://www.java-tutorial.org/menueleiste-und-popupmenue.html, 19.03.2021, 15:07

Quelle 11: https://www.java-tutorial.org/container.html, 20.03.2021, 17:45 Uhr

Quelle 12: https://www.java-tutorial.org/bedienelemente.html, 20.03.2021, 18:00 Uhr

Quelle 13: https://www.java-tutorial.org/event-handling.html, 21.03.2021, 11:00 Uhr

Quelle 14: https://www.javatpoint.com/java-actionlistener#:~:text=The%20Java%20ActionListener%20is%20notified%20whenever%20you%20click,automatically%20whenever%20you%20click%20on%20the%20registered%20component., 21.03.2021, 12:00 Uhr

Quelle 15: https://www.tutorialspoint.com/awt/awt_keyadapter.htm,

5. Anhang

5.1. Code

5.1.1 Klasse „Spiel“

```
package Game;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.time.Instant;
import java.util.Random;
import javax.swing.*;
public class Spiel extends JPanel implements ActionListener {
	private int counter = 0;
	private final int breite = 600;
	private final int hoehe = 600;
	private final int kaestchengroesse = 25;
	private final int allePunkte = (hoehe*breite)/ kaestchengroesse * kaestchengroesse;
	private final int geschwindigkeit;
	private final Timer timer;
	private int score, bestscore;
	private final int[] x = new int[allePunkte];
	private final int[] y = new int[allePunkte];
	private final int[] mx = new int[allePunkte/2];
	private final int[] my = new int[allePunkte/2];
	private int vApfel = 0;
	private int koerperteile = 1;
	private int apfel_x;
	private int apfel_y;
	private final Random r = new Random();
	char richtung = 'R';
	public static boolean imSpiel = true;
	public Spiel(int geschwindigkeit) {
		this.geschwindigkeit = geschwindigkeit;
		addKeyListener(new Tastatur());
		setBackground(new Color(0,0,200));
		setFocusable(true);
		setPreferredSize(new Dimension(breite + 300, hoehe));
		x[0] = 2* kaestchengroesse;
		y[0] = 2* kaestchengroesse;
		platziereApfel();
		timer = new Timer(this.geschwindigkeit, this);
		timer.start();
	}
	@Override
	public void paintComponent(Graphics g) {
```

```
            super.paintComponent(g);
            if (imSpiel) {
                for(int i = 0; i < hoehe/ kaestchengroesse; i++){
                    g.drawLine(i * kaestchengroesse, 0, i * kaestchengroesse, hoehe);
                    g.drawLine(0, i* kaestchengroesse, breite, i * kaestchengroesse);
                }
                g.setColor(Color.GRAY);
                g.fillRect(breite, 0, 900- breite, hoehe);
                g.setColor(Color.red);v
                g.fillOval(apfel_x, apfel_y, kaestchengroesse, kaestchengroesse);
                g.setColor(new Color(0,100,0));
                if(x[0] != breite) {
                    g.fillRect(x[0], y[0], kaestchengroesse, kaestchengroesse);
                }
                for (int i = 1; i < koerperteile; i++) {
                    g.setColor(new Color(34, 139, 34));
                    g.fillRect(x[i], y[i], kaestchengroesse, kaestchengroesse);
                }
                g.setColor(Color.yellow);
                for (int i = 0; i < vApfel; i++) {
                    g.fillOval(mx[i], my[i], kaestchengroesse, kaestchengroesse);
                }
                g.setColor(Color.BLACK);
                Toolkit.getDefaultToolkit().sync();
                Font schrift = new Font("Calibri", 1, 40);
                g.setFont(schrift);
                g.drawString("Snake Game ", breite + 60, 40);
                Font s2 = new Font("Verdana", 2, 20);
                g.setFont(s2);
                g.drawString("Dein Score: " + score, breite + 40, 100);
                g.drawString("Dein Highscore: " + bestscore, breite + 40, 130);
                g.drawString("Bewegungen: " + counter + ", ", breite + 40, 160);
                g.drawString("Geschwindigkeit: " , breite + 40, 190);
                g.drawString( 1000/geschwindigkeit + " Kästchen/Sekunde" , breite + 40, 220);
                g.drawString("Bewegungen pro Apfel: ", breite + 40, 250);
                g.drawString("Zeit im Spiel: " + (counter*geschwindigkeit/1000) + "Sek", breite + 40,
310);
                if(score > 0) {
                    g.drawString(String.valueOf(counter / score), breite + 40, 280);
                } else {
                    g.drawString(String.valueOf(counter), breite + 40, 280);
                }
                g.drawString("Beenden: Enter drücken", breite + 40, 400);
                if (koerperteile == allePunkte - vApfel) {
                    imSpiel = false;
                }
            } else {
                Font schrift = new Font("Calibri", 1, 40);
                g.setColor(Color.green);
```

```
                g.setFont(schrift);
                g.drawString("Spiel vorbei, du hast gewonnen!!!", 20, 40);
        }
    }
    private void pruefeApfel() {
        if ((x[0] == apfel_x) && (y[0] == apfel_y)) {
            koerperteile++;
            score++;
            if(score > bestscore) bestscore = score;
            platziereVApfel();
            platziereApfel();
        }
    }
    private void pruefeVApfel()  {
        for(int i = 0; i < vApfel; i++){
            if(x[0] == mx[i] && y[0] == my[i]){
                koerperteile--;
                score--;
                vApfelEntfernen(i);
                break;
            }
        }
    }
    private void platziereApfel() {
        apfel_x = r.nextInt((breite/ kaestchengroesse)) * kaestchengroesse;
        apfel_y = r.nextInt((breite/ kaestchengroesse)) * kaestchengroesse;
        for(int i = 0; i < koerperteile; i++){
            if(x[i] == apfel_x && y[i] == apfel_y ||mx[i] == apfel_x && my[i] == apfel_y){
                platziereApfel();
            }
        }
    }
    private void platziereVApfel(){
        mx[vApfel] = r.nextInt((breite/ kaestchengroesse)) * kaestchengroesse;
        my[vApfel] = r.nextInt((breite/ kaestchengroesse)) * kaestchengroesse;
        for(int i = 0; i < koerperteile; i++){
            if(x[i] == mx[vApfel] && y[i] == my[vApfel]){
                platziereVApfel();
            }
        }
        vApfel++;
    }
    private void bewegen() {
        counter++;
        for (int i = koerperteile; i > 0; i--) {
            x[i] = x[i - 1];
            y[i] = y[i - 1];
        }
        switch(richtung){
```

```
                case 'R': x[0] = x[0] + kaestchengroesse;break;
                case 'L': x[0] = x[0] - kaestchengroesse;break;
                case 'U': y[0] = y[0] - kaestchengroesse;break;
                case 'D': y[0] = y[0] + kaestchengroesse;break;
                default:break;
        }
}
private void pruefeUnfall() {
        for (int i = koerperteile; i > 3; i--) {
                if ((x[0] == x[i]) && (y[0] == y[i])) {
                        koerperteile = 1;
                        counter = 0;
                        vApfel = 0;
                        score = 0;
                }
        }
        if(y[0] >= hoehe || x[0] >= breite ||y[0] < 0|| x[0] < 0){
                counter = 0;
                koerperteile = 1;
                vApfel = 0;
                x[0] = 2* kaestchengroesse;
                y[0] = 2* kaestchengroesse;
                richtung = 'R';
                score = 0;
        }
}
private void vApfelEntfernen(int x){
        mx[x] = 1000;
}
@Override
public void actionPerformed(ActionEvent e) {//Code 10
        if (imSpiel) {//Code 10
                pruefeApfel();//Code 10
                pruefeVApfel();//Code 10
                pruefeUnfall();//Code 10
                bewegen();//Code 10
        }//Code 10
        repaint();//Code 10
}//Code 10
private class Tastatur extends KeyAdapter {
        @Override
        public void keyPressed(KeyEvent e) {
                switch (e.getKeyCode()){
                        case KeyEvent.VK_LEFT:
                                if(richtung != 'R')richtung = 'L';break;
                        case KeyEvent.VK_RIGHT:
                                if(richtung != 'L')richtung = 'R';break;
                        case KeyEvent.VK_DOWN:
                                if(richtung != 'U')richtung = 'D';break;
```

```
                        case KeyEvent.VK_UP:
                                if(richtung != 'D')richtung = 'U';break;
                        default:break;
                }
                if(e.getKeyCode() == KeyEvent.VK_ENTER){
                        System.exit(0);
                }
        }
    }
}
```

5.1.2 Klasse „Snake“

```
package Game;
import javax.swing.*;
public class Snake extends JFrame {
    ImageIcon bild;
    public Snake(int gesch) {
        add(new Spiel(gesch));// Code 9
        bild = new ImageIcon("C:\\Users\\paul-\\IdeaProjects\\SnakeGame\\src\\re-
cources\\kopf.png");// Code 9
        setIconImage(bild.getImage());// Code 9
        setResizable(false);// Code 9
        pack();// Code 9
        setTitle("Snake");// Code 9
        setLocationRelativeTo(null);// Code 9
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);// Code 9
        setVisible(true);// Code 9
    }
}
```

5.1.3 Klasse „Menü“

```
package Game;
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class Menu extends JFrame implements ActionListener{// code 1
    public  static Menu m;
    private JButton beenden;
    private JButton einfach;
    private JButton mittel;
    private JButton schwer;
    private JButton sehrSchwer;
    private JButton regeln;
    public static void main(String[] args) {
        erstellen();
    }
    public static void erstellen(){
```

```
            m = new Menu("Menü");//Code 3
            ImageIcon kopfBild = new ImageIcon("C:\\Users\\paul-\\IdeaProjects\\SnakeGame\\src\\re-
sources\\kopf.png");
            m.setIconImage(kopfBild.getImage());
            m.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//Code 3
            m.setSize(600, 600);//Code 3
            m.setLocationRelativeTo(null);//Code 3
            m.setLayout(null);//Code 3
            m.setResizable(false);//Code 3
            m.setVisible(true);//Code 3
        }
        public Menu(String titel) {
            super(titel);
            beenden = new JButton("Beenden");//code 2
            beenden.setBounds(150, 480, 300, 60);//code 2
            beenden.addActionListener(this);//code 2
            add(beenden);//code 2
            regeln = new JButton("Spielregeln");
            regeln.setBounds(150,400 ,300, 60);
            regeln.addActionListener(this);
            add(regeln);
            einfach = new JButton("Einfach");
            einfach.setBounds(150, 20, 300, 60);
            einfach.addActionListener(this);
            add(einfach);
            mittel = new JButton("Mittel");
            mittel.setBounds(150, 100, 300, 60);
            mittel.addActionListener(this);
            add(mittel);
            schwer = new JButton("Schwer");
            schwer.setBounds(150, 180, 300, 60);
            schwer.addActionListener(this);
            add(schwer);
            sehrSchwer = new JButton("Sehr Schwer");
            sehrSchwer.setBounds(150, 260, 300, 60);
            sehrSchwer.addActionListener(this);
            add(sehrSchwer);
        }
        @Override
        public void actionPerformed(ActionEvent e) {//Code 4
            if(e.getSource() == beenden){//Code 4
                System.exit(0);//Code 4
            }//Code 4
            if(e.getSource() == regeln){
                m.setVisible(false);
                new Spielregeln();
            }
            if(e.getSource() == einfach){//Code 4
                dispose();//Code 4
```

```
                    new Snake(200);//Code 4
                }
                if(e.getSource() == mittel){
                    dispose();
                    new Snake(170);
                }
                if(e.getSource() == schwer){
                    dispose();
                    new Snake(140);
                }
                  if(e.getSource() == sehrSchwer) {
                      dispose();
                      new Snake(110);
                  }
        }//Code 4
}
```

5.1.4 Klasse „Spielregeln“

```
package Game;
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class Spielregeln extends JFrame implements ActionListener {
        private JPanel p;
        private JLabel l1, l2, l3, l4, l5,l6;
        private JButton zurueck;
        ImageIcon bild;
        public Spielregeln(){
                setPreferredSize(new Dimension(600, 600));//Code 6
                setResizable(false);//Code 6
                pack();//Code 6
                setLocationRelativeTo(null);//Code 6
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//Code 6
                setTitle("Spielregeln");//Code 6
                bild = new ImageIcon("C:\\Users\\paul-\\IdeaProjects\\SnakeGame\\src\\re-
cources\\kopf.png");
                setIconImage(bild.getImage());//Code 6
                zurueck = new JButton("<-- Zurück");//Code 7
                zurueck.setBounds(190, 300, 220, 60);//Code 7
                zurueck.addActionListener(this);//Code 7
                add(zurueck);//Code 7
                p = new JPanel(null);
                l1 = new JLabel("In dem Spiel Snake, geht es um das Erlangen eines möglichst großen Kör-
pers, indem die");//Code 8
                l1.setBounds(5,0, 600, 40);//Code 8
                l2 = new JLabel("Schlange Äpfel isst. Dies geschieht wenn sich die Schlange mit ihrem Kopf
über einen Apfel bewegt.");
```

```
                l2.setBounds(5,20, 600, 40);
                l3 = new JLabel("Gesteuert werden kann der Kopf mittels der Pfeiltasten, die sich an der un-
teren rechten Ecke der ");
                l3.setBounds(5,40, 600, 40);
                l4 = new JLabel("Tastatur befinden. Die Schlangenbewegungen entsprechen dabei jeweils
der Pfeilrichtung auf");
                l4.setBounds(5,60, 600, 40);
                l5 = new JLabel("den Pfeiltasten. Berührt die Schlange die Wand oder sich selbst an irgend
einer Stelle, stirbt");
                l5.setBounds(5,80, 600, 40);
                l6 = new JLabel("sie und das Spiel startet neu. Viel Spaß!!!");
                l6.setBounds(5,100, 600, 40);
                p.add(l6);
                p.add(l5);
                p.add(l4);
                p.add(l3);
                p.add(l2);
                p.add(l1);
                add(p);
                setVisible(true);
        }
        @Override
        public void actionPerformed(ActionEvent e) {
                if(e.getSource() == zurueck){
                        dispose();
                        Menu.m.setVisible(true);
                }
        }
}
```

5.2. BILDER

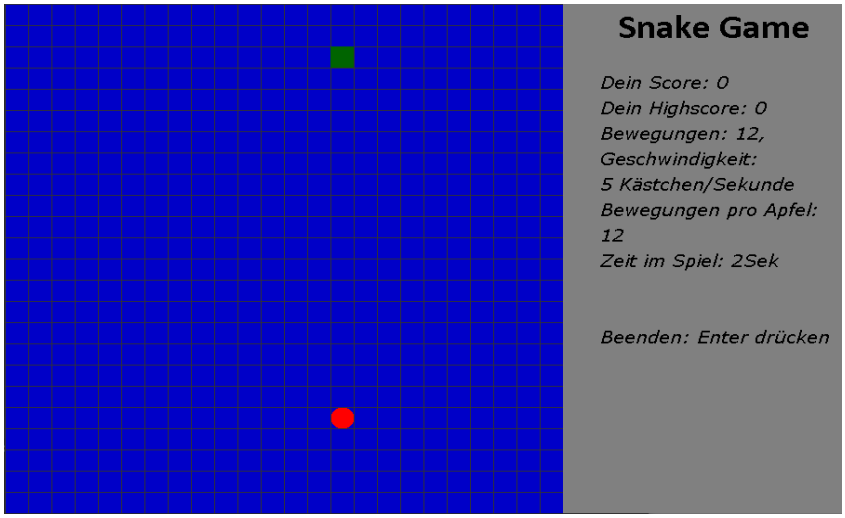


Bild 1: Spielfeld mit Seitenfeld für Zwischeninformationen