OoSe Projekt: Conway's Game of Life

Paul Völkel und Dylan Stogianni

Rheinische-Friedrich Wilhelms Universität Bonn Informatik IV

1 Einführung

Im Rahmen des Praktikums Objektorientierte Softwareentwicklung unter der Leitung von Herrn Dr. Hassan Errami entwickeln wir ein Projekt, das mittels JavaFX Conway's Game of Life implementiert. Dieses stellt eine Simulation zellulärer Automaten dar, die auf Basis zuvor festgelegter Regeln zwei unterschiedliche Zustände annehmen können. Zellen bleiben lebendig, wenn sie zwischen zwei und drei lebende Nachbarn haben, ansonsten stirbt sie. Zudem ist es möglich eine tote Zelle wiederzubeleben, sollte sie genau drei lebendige Nachbarn haben.

2 Benutzerdokumentation

2.1 Zweck und Funktionalität

Conway's Game of Life besitzt vielfältige Anwendungs- und Interpretationsmöglichkeiten, wobei vor allem der biologische Faktor eine große Rolle spielt, da man mittels angepasster Regeln einen Mikrokosmos simulieren kann in dem unter Umständen neue Ärtenäuftreten können und sich deren Verhalten somit analysieren lässt. Hinzu kommt, dass das Game of Life ein System simulieren kann, das sich auf Angebot und Nachfrage stützt, womit es möglich ist, eine Situation zu simulieren, die einem Nutzer verdeutlicht, wann es am rentabelsten ist, beispielsweise Aktien zu verkaufen.
Letztenendes kommt noch die Turing-Vollständigkeit hinzu, was bedeutet, dass man jedes algorithmische Problem, das sich mittels eines Computers lösen lässt, ebenfalls im Game of Life lösen lässt, was seine besondere Relevanz für die Mathematik verdeutlicht. Mittels dieses Projektes soll aber nicht nur ein bestimmter Aspekt betrachtet werden, sondern nur der Grundstein für die genannten Anwendungsmöglichkeiten gelegt werden, also das normale Game of Life mit seinen ursprünglichen Regeln.

2.2 Installation

Um das Projekt zu starten, empfehlen wir die Intellij IDEA, da sich hiermit das Einbinden von Gradle und JavaFX, die für die Ausführung unerlässlich sind, stark vereinfachen lässt. In der eclipse IDEA wäre es nötig eine gesonderte main-Klasse zu schreiben, die die eigentliche Klasse ausführt, was in Intellij nicht nötig ist. Es gilt jedoch zu beachten, dass man zuvor Gradle und SceneBuilder

installieren muss, um alle nötigen Voraussetzungen für die Ausführung und weiterführende Bearbeitung des Projektes zu erfüllen.
Möchte man das Game of Life nun starten, so ist es nötig zunächst ein neues Gradle-Projekt zu erstellen. Damit einhergehend wird eine build.gradle-Datei erstellt, deren Inhalt man mit dem Inhalt der gleichnamigen Datei unseres Projektes ersetzen muss. Anschließend wird Intellij den Nutzer darauf hinweisen, dass man nun die build.gradle-Datei aktualisieren muss, um sämtliche Änderungen anzuwenden. Anschließend ist es nötig sämtliche .fxml-Dateien in den resources-Ordner des erstellten Projektes einzufügen, sowie alle .java-Dateien in src/main/java. Nun übergibt man Gradle den Task run und schon ist das Projekt bereit ausgeführt zu werden.

2.3 Notwendige Ressourcen

Wie für jedes Computerprogramm, gibt es auch für Intellij gewisse Systemvoraussetzungen, die für die Installation benötigt werden. Dazu sollte der Computer mindestens Windows 7 in seiner 64-bit Version unterstützen und über 2GB RAM, sowie 5GB ROM verfügen. Zudem wird eine Auflösung von mindestens 1024x768 Pixeln vorausgesetzt, ebenso eine Installation der Java JDK. Von uns empfohlen wird eine Ausführung mittels der Java-Version 11, da es mit neueren Versionen zu unerwünschten Kompatibilitätsproblemen kommen kann, die sich damit umgehen lassen.
Mit dem im vorherigen Absatz beschriebenen Kopieren der einzelnen .java und .fxml Dateien und Einfügen der build.gradle ist man sofort in Besitz sämtlicher notwendigen Bibliotheken, die man zum Ausführen benötigt, weshalb kein selbstständiges Hinzufügen oder Anpassen dieser notwendig ist. Dabei stützt sich das Projekt vor allem auf die JavaFX-Bibliotheken Scene und Application, ebenso auf die Datei-Bibliotheken nio.file.Path und nio.file.Files und die JSON - Konvertierungsbibliothek GSON.
Um für eine ansprechende Visualisierung zu sorgen, haben wir uns dazu entschlossen, fxml zu nutzen, da man mit dem SceneBuilder ein passendes Fenster erzeugen kann, das sämtliche Buttons und Textfelder darstellt und mit den entsprechenden Funktionen verknüpft. Dahingehend ist auch die Bibliothek jfoenix relevant, da sie eine alternative Visualisierung der Buttons ermöglicht.

3 Systemdokumentation

Die folgenden Abschnitte beschäftigen sich mit dem formalen Aufbau des Programmes und den einzelnen darin enthaltenen Methoden mit ihren jeweiligen Funktionalitäten. Zudem wird auf die wichtigsten Bestandteile gesondert eingegangen, sowie die hauptsächlich verwendeten Bilbliotheken erläutert. Dabei sei angemerkt, dass sich die Methoden innerhalb mehrerer Controller befinden, die mit dem SceneBuilder verknüpft sind, sodass sich die Methoden direkt auf die visuellen Elemente anwenden lassen.

3.1 Struktur und Aufbau

Regelumsetzung Zu Beginn betrachten wir dem umfangreichen Window - Controller, der mit einer Vielzahl von Initialisierungen beginnt, die sich auf FXML-Elemente beziehen, aber auch die Größe des Feldes bestimmen, auf dem sich letztendlich das Game of Life abspielen wird.
Bei der ersten implementierten Methode handelt es sich um die draw()-Methode, die die Zellen anhand ihres Zustandes entsprechend einfärbt und somit für die Visualisierung des Ergebnisses jeder Methode zuständig ist. Dementsprechend findet man sie auch am Ende der meisten Methoden in diesem Controller. Es geht weiter mit der Methode init(), die beim Öffnen des Programmes ein Feld erzeugt, bei dem zufällig ausgewählte Zellen lebendig sind, wofür man ein Objekt des Random-Typs nutzt, das jede Zelle mit einem zufälligen Wert zwischen 0 und 1 ausfüllt.
Die darauf folgende Methode countNeighbors(int, int) stellt mit der Bestimmung der Anzahl der Nachbarn einer Zelle, eine der wichtigsten Methoden dar. Auf Basis dieser Zahl wird nämlich jede neue Generation an Zellen generiert. Zu Beginn nutzen wir den ternary operator ?, um die Randfälle zu betrachten. Dabei bedeutet die Zeile int iEnd=i==grid.length-1 ? 0:1, dass sobald i gleich der Länge des Gatters-1 ist, wird i auf Null gesetzt. Andernfalls gibt i=1. Die anderen Zeilen verhalten sich auf dieselbe Art und Weise, wobei immer ein anderer Rand betrachtet wird. Danach werden sämtliche Zellen betrachtet, die die gerade betrachtete Zelle umgeben, also ihre Nachbarn. Demnach müssen für jede Zelle maximal acht Nachbarn ausgewertet werden, deren Wert aufsummiert wird, woraufhin man die Anzahl der lebenden Zellen erhält. Da der Wert der gerade betrachteten Zelle für dieses Ergebnis irrelevant ist, wird außerhalb der beiden for-Schleifen dieser Wert vom Gesamtergebnis abgezogen. Damit wäre die Umsetzung der Regeln abgeschlossen. Nun fehlt noch die Methode tick(), die dafür da ist auf Basis der gegebenen Regeln die nächste Generation an Zellen zu generieren. Dafür wird mittels einer doppelten for-Schleife von jeder einzelnen Zelle die Anzahl der Nachbarn in grid betrachtet und wenn die Bedingungen der Regeln erfüllt sind, wird der entsprechende Eintrag in next auf eins gesetzt. Abschließend kommt erneut die zuvor erwähnte draw()-Methode zum Einsatz.

Speichern und Laden Dieser Abschnitt geht auf die zwei Methoden save() und load() ein, die den Nutzer eine Game of Life-Instanz abspeichern und später wieder neu laden lässt. Dafür wird ein JsonArray genutzt, das an einen vorbestimmten Speicherort die .json Datei speichert, die das zur Zeit der Speicherung dargestellte Gatter von lebenden Zellen, beinhaltet. Mittels StringBuilder ist es dem Nutzer möglich der .json-Datei einen Namen zu geben, so dass man mehrere Instanzen abspeichern kann.
Bei der load()-Methode handelt es sich eigentlich um eine genau umgekehrt implementierte Variante der save()-Methode, denn auch hier wird zunächst wieder mittels StringBuilder der Pfad des Speicherortes zusammengesetzt, wobei anschließend nicht die Daten des Gatters in ein JsonArray gespeichert werden,

sondern Daten aus diesem ausgelesen und in das aktuell dargestellte Gatter geladen werden.
Neben dem Speichern und Lades des dargestellten Gatters, ist es auch noch möglich benutzerdefinierte Regeln zu speichern und zu laden, wobei sich beide Verfahren mit den vorherigen decken. Beim Speichern und Laden gilt es zu beachten, dass man den Speicherpfad manuell im Programm selbst abändern muss, wobei es meist ausreichen sollte, wenn man den Nutzernamen entsprechend abändert, wenn man nicht einen gänzlich anderen Pfad bevorzugt.

Invertieren und Reset Die Methode reverse() dreht den Zustand jeder einzelnen Zelle um, setzt also lebende auf tot und tote auf lebend. clear() dagegen ermöglicht es dem Nutzer sämtliche Zellen des Feldes auf tot zu setzen, sodass man ein weißes Feld erhält auf dem man nun beliebige Zellen lebendig machen kann.

Zoomen Mit den Methoden zoom-in() und zoom-out() ist es möglich den dargestellten Bereich zu verringern oder zu vergrößeren, wobei dies immer in vierer Schritten geschieht. Dafür werden die Werte der Randbereiche der x- und y-Achsen im zweidimensionalen Raum um den gegebenen Faktor angepasst.

3.2 Mausinteraktionen

Beleben und töten Um das Game of Life möglichst interaktiv zu gestalten, ist es nötig das Programm beziehungsweise das dargestellte Gatter mittels Mausoperationen veränderbar zu gestalten. Dafür wird der Mauszeiger lokalisiert, sobald der Nutzer die linke Maustaste drückt. Befindet sich dieser auf einer Zelle, auf der die zuvor gewählte Interaktion ausführbar ist, so wird diese auf die entsprechende Zelle angewendet. Für beide Interaktionen befindet sich auf der linken Seite des Programmes eine Schaltfläche Add/Delete, die die gerade angezeigte Funktion ermöglicht.
Damit man nicht jede Zelle einzeln auswählen muss, bietet das Programm zusätzlich die Möglichkeit per gedrückt gehaltener Maustaste die oben beschriebenen Aktionen durchzuführen.

3.3 Wichtige Bestandteile

Die Klasse initialize() kümmert sich im Großen und Ganzen um das Laden der einzelnen Steuerelemente und FXML Dateien, damit diese bei einer Nutzeraktion miteinander interagieren un die gewünschten Aktionen ausführen können. Zudem werden Einstellungen, wie die initiale Slidergeschwindigkeit verwaltet, auf die der Nutzer nicht Einfluss nehmen kann, wobei es natürlich möglich ist, diese Werte während der Ausführung des Programmes dynamisch anzupassen, wofür initialize() spezielle Listener auf die unterschiedlichen Interaktionsobjekte, wie Eingabefelder, verwendet, die die durch den Nutzer gegebenen Informationen sofort an die entsprechenden Funktionen weiterleiten, damit die Änderungen

wirksam werden. Hinzu kommt die Initialisierung des Pfades, der gespeicherte Regeln und Gatter beinhalten kann, sollte sich der Nutzer dazu entscheiden diese Funktion zu nutzen.

3.4 Andere Controller

Da nun die Beschreibung des Window-Controller abgeschlossen ist, der sämtliche Funktionen beinhaltet, widmen wir uns nun den anderen Controllern, die auf die Methoden, die im Windown-Controller deklariert sind, zugreifen können.

Save-Controller Da wäre zum einen der Save-Controller, der sich mit der Verwaltung des save-Fensters beschäftigt, das dem Nutzer erlaubt, selbsterstellte Regeln und Gatter abzuspeichern. Dafür werden die verschiedenen Eingabemöglichkeiten aus der zugehörigen FXML geladen und der jeweilige Initialwert zugewiesen.

Load-Controller Hiermit wird das Load-Feld im Programm festgelegt und weist den dort gegebenen Funktionen Initialwerte zu, sollte es zu keiner direkten Eingabe (Festlegen des Namens der zu ladenden Datei) durch den Nutzer kommen. Andernfalls wird der durch den Nutzer gegebene Name übergeben.

Setting-Controller Der Setting-Controller initialisiert sämtliche RadioButtons über die der Nutzer eigene Regeln für das Game of Life festlegen kann. Anschließend werden die jeweiligen Auswirkungen, die das Anklicken eines RadioButtons auf die zugrundeliegenden Spielregeln haben, festgehalten. Wichtig bei dieser Funktion zu beachten ist, dass die abgeänderten Spielregeln erst dann gültig werden, wenn man auf den Button "Confirm Set of Rules"klickt.

Game-Setting-Controller Dieser Controller initialisiert die im Window - Controller beschriebenen Methoden zu und beschreibt demnach, wie sich die unterschiedlichen Eingabemöglichkeiten bei Aktionen verhalten sollen.

3.5 Die FXML Dateien

Die FXML Dateien selbst wurden mit Hilfe des SceneBuilders erstellt, beinhalten dementsprechend also sämtliche Namen der Buttons und sonstiger Interaktionsgegenstände, sowie deren zugehörige ID und Größe im Fenster. Außerdem ist ersichtlich, welche Methode womit verknüpft ist, sodass der Aufbau des gesamten Programmes leicht ersichtlich und verständlich ist.

4 Projektdokumentation

Da das gesamte Projekt in enger Zusammenarbeit angefertigt wurde, ist eine genaue Trennung des erarbeiteten Fortschrittes kaum möglich. In den folgenden Abschnitten haben wir dennoch versucht jeden Aspekt genauestens wie möglich aufzuzeigen.

4.1 Woche 1: 04.06. - 11.06.

In der ersten Woche haben wir uns zunächst damit beschäftigt einen groben Überblick über das Projekt zu erhalten und eine erste Implementation der Umsetzung der Regeln angefertigt. Dieser erste Entwurf konnte die ursprünglichen Regeln des Game of Life umsetzen und beinhaltete auch schon erste Möglichkeiten die Animation zu stoppen und schrittweise auszuführen. Jedoch waren andere, verlangte Funktionen, wie das Auswählen lebender Zellen noch nicht möglich, ebenso wenig wie das Abspeichern des gerade dargestellten Gatters. Darum beschränkten wir uns zunächst auf ein zufällig befülltes Gatter, das mittels eines Buttons eine neue zufällige Verteilung erzeugen konnte. Diese Funktion findet sich auch noch im finalen Projekt wieder. Zudem blieb die grundlegende Struktur des Programmes bis zur dritten Woche erhalten, wurde also nur um weitere Funktionen ergänzt. Auch in der finalen Version des Projektes finden sich noch einige Methoden, die noch aus dieser ersten Version stammen.
In dieser Version haben wir uns nur auf JavaFX und die für Java typischen Klassen gestützt, was zwar in einem funktionierenden Programm resultierte, jedoch ohne ansprechende Benutzeroberfläche.

4.2 Woche 2: 11.06. - 18.06.

Die zweite Woche war geprägt vom Implementieren neuer Funktionen, wozu vor allem das Speichern und Laden eines dargestellten Gatters gehörte. Dafür nutzten wir JsonArrays in Verbindung mit StringBuilder, den wir auch schon für eine der vorherigen Übungsaufgaben genutzt haben, um den Speicherpfad zu konstruieren. Ebenfalls in das Projekt gefunden, hat der Slider, der die Geschwindigkeit der Animation anpassen kann, sowie das Invertieren der Zustände. Dabei wird für jede Zelle der umgekehrte Zustand ermittelt und dargestellt, was nicht sonderlich schwierig zu implementieren ist.
Erste Versuche Zellen per Mausklick zu beleben, mündeten in zahlreichen Problemen, die zunächst weder durch Internetrecherche, noch durch reines Probieren gelöst werden konnten, weshalb sich die Implementierung dieser Funktion bis in die darauffolgende Woche hinauszog. Dies erlaubte uns eine Vielzahl von gescheiterten Möglichkeiten zu vergleichen und die Probleme langsam aber sicher zu beseitigen.
Zusätzlich konnten wir in dieser Woche die restlichen Buttons mitsamt zugehörigen Funktionen, wie Clear und Close hinzufügen.

4.3 Woche 3: 18.06. - 25.06.

In der vorletzten Woche unseres Projektes konnten wir die größten Fortschritte erzielen, da wir nun in der Lage waren, den Mausklick korrekt an die zugrundeliegende Funktion zu übertragen, sodass es von nun an möglich war, Zellen per Klick zu beleben. Die umgekehrte Funktion, also das Töten per Mausklick war auf Grundlage der neu hinzugewonnenen Erkenntnisse nicht mehr schwer. Nun konnten wir uns an die Umsetzung der Zoom-Funktion machen, die es

ermöglichen sollte, in das Gatter zu zoomen und sich mit geklickter rechter Maustaste im Fenster zu bewegen.
Zum Ende dieser Woche hin, hatten wir bereits ein vollständig funktionierendes Projekt, das sämtliche Funktionen unterstützte, jedoch waren wir von der Visualisierung nicht überzeugt, weshalb wir uns entschlossen haben, das gesamte Projekt mittels SceneBuilder umzugestalten, also auf FXML Dateien umzusteigen, da diese mehr Möglichkeiten zur Individualisierung bietet.

4.4 Woche 4: 25.06. - 02.07.

Mit der genannten Umwandlung in FXML beschäftigte sich vor allem Paul, wobei es beim Einfügen der Funktionen zu einigen Exceptions kam, die aus einer falschen Reihenfolge beim Aufrufen der Funktionen resultierte. Das Finden dieses Fehlers resultierte in einer langwierigen Suche, die letztenendes durch sukzessives Neu-Einfügen jeder Funktion, mit einem positiven Ergebnis beendet werden konnte. Hinzu kam das Problem, dass die einzelnen Fenster untereinander kommunizieren mussten, was durch ein Übergeben der Window-Controller-Instanz an die Unter-Controller, eine initialize()-Methode und dem einmaligen Erstellen aller benötigten Controller-Instanzen im Window-Controller gelöst wurde.
In diesem letzten Arbeitsschritt verwirklichten wir nun auch die Sidebar, die die unterschiedlichen Menüpunkte beinhaltet, sowie das Fenster zur Initialisierung der benutzerdefinierten Regeln. Zuletzt kam auch noch die Funktion hinzu, dass die Generationen gezählt werden, was bedeutet, dass sich die Anzahl der bisher durchgeführten Schritte ablesen lässt. Eine zwar nicht gewollte, aber sinnvolle Funktion stellt dabei die Möglichkeit des Invertieren der Regeln dar, so dass der Nutzer leicht unterschiedliche Regelwerke miteinander vergleichen kann.
Als kleines Gimmick und visuelles Schmankerl baute Paul noch eine farbliche Note in das Projekt ein, die Zellen auf Basis ihrer Lebenszeit in unterschiedliche Helligkeitsstufen der gewählten Farbe unterteilt. Umso dunkler die Zelle eingefärbt ist, desto länger lebt sie bereits.