OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

Implementierung einer Komponente zur Visualisierung und automatischen Überprüfung von Algorithmen aus der Lehre unter Verwendung eines bestehenden 3D-Frameworks

An der Fakultät für Informatik und Mathematik der
Ostbayerischen Technischen Hochschule Regensburg
im Studiengang
Medizinische Informatik

eingereichte

Bachelorarbeit

zur Erlangung des akademischen Grades des
Bachelor of Science (B.Sc.)

Vorgelegt von: André Helgert
Matrikelnummer: 3014888

Erstgutachter: Prof. Dr. Carsten Kern
Zweitgutachter: Prof. Dr. Daniel Jobst

Abgabedatum: 01.09.2018

Kurzfassung

Die Intention dieser Arbeit ist es, eine bestehende Voxel-Bibliothek mit neuen Methoden und Funktionalitäten auszustatten, um Studenten für die Vorlesung „Algorithmen & Datenstrukturen“ begeistern zu können.

Um dieses Ziel zu erreichen, wurde eine Komponente implementiert, welche es Studenten erlaubt, den Algorithmus ihrer Wahl in eine Benutzerschnittstelle einzutippen, um diesen daraufhin visualisieren zu können und auf Richtigkeit überprüfen zu lassen. Dadurch erweitern Studenten ihr erlerntes Wissen durch eine optische Komponente und können so die vorgestellten Konzepte aus der Vorlesung besser verstehen.

Inhaltsverzeichnis

I Abbildungsverzeichnis

II Abkürzungsverzeichnis

BA Bachelorarbeit

BST Binärer Suchbaum

LWJGL Lightweight Java Game Library

MST Minimaler Spannbaum

OOP Objektorientierte Programmierung

UI User Interface

UML Unified Modeling Language

1 Einleitung

1.1 Blocklib

Die *Blocklib* ist ein 3D-Framework, welche prozedural generierte Welten aus Würfeln generieren kann. Sie wurde 2016 von Tobias Zink [Zin16] im Rahmen einer Bachelorarbeit erschaffen. Seitdem wurde die Bibliothek durch viele Projekte und Abschlussarbeiten erweitert. Die Blocklib soll in erster Linie in der Lehre eingesetzt werden.

1.2 Motivation und Zielsetzung

Die Einführung in zahllose *Algorithmen* ist für Studenten oft eine Herausforderung. Schnell verliert man den Überblick über vorgestellte Konzepte aufgrund der Anzahl und Varianz der Algorithmen und die Motivation an dem Lehrmodul lässt nach. Zwar gibt es zahlreiche Visualisierungen für Algorithmen, sei es in Lehrbüchern oder im Internet, dennoch können Studenten aufgrund der fehlenden Nachvollziehbarkeit aus diesen statischen Vorgaben meist nur einen begrenzten Nutzen ziehen.

Die Erweiterung der Bibliothek um Komponenten zur Visualisierung und automatischen Überprüfung von Algorithmen soll dem Benutzer somit die Kontrolle über die Implementierung des Codes geben. Der Benutzer sollte sich z.B. nicht mehr an statische Variablendefinitionen halten müssen, sondern den Algorithmus seiner Wahl implementieren, wie es gewünscht ist. Dies erlaubt dem Benutzer, den vorgegebenen Programmcode zu manipulieren und verschiedene Eingaben zu testen, aber auch seine eigene Ausführung eines Algorithmus zu implementieren. Schließlich soll der Student nicht den Code eines Algorithmus auswendig lernen, sondern das Prinzip hinter dem vorgestellten Konzept verstehen und dieses, unabhängig von der Implementation, wiedergeben können.

Damit dies möglich ist, wird der eingegebene Code auf Korrektheit geprüft. Es hilft dem Benutzer nicht, wenn er seinen Code visualisieren lässt, dieser aber seinen Zweck nicht erfüllt und z.B. im Falle eines Sortieralgorithmus nicht korrekt sortiert. Wenn der eingegebene Algorithmus zulässig und somit korrekt terminiert, wird dieser mithilfe der bereits vorhandenen 3D-Bibliothek Blocklib visualisiert.

Anforderungen an die automatische Überprüfung auf Korrektheit sind:

- Korrekte Überprüfung des Codes über semantische Definitionen hinweg.
- Korrekte Überprüfung des Codes über verschiedene syntaktische Implementierungen hinweg.

Anforderungen an die Visualisierung sind:

- Flüssige Darstellung der Algorithmen.
- Nachvollziehbarkeit der Arbeitsschritte.

Anforderungen an die Bachelorarbeit sind:

- Den Quellcode dokumentieren, um eine einfache Benutzung zu garantieren.
- Schnittstellen und Klassen implementieren, um unnötige Komplexität zu vermeiden.
- Dem Benutzer eine möglichst einfache Bedienung und Implementierung der Algorithmen gewährleisten.
- Möglichst modulare Benutzerschnittstellen generieren, um eine einfache Erweiterung der Bibliothek um neue Algorithmen zu gewährleisten.

1.3 Gliederung

Zu Beginn dieser schriftlichen Ausarbeitung werden die implementierten Lehralgorithmen beschrieben. Der Sinn hinter diesen Algorithmen wird erläutert, ebenso deren Funktionalitäten und Abläufe. Außerdem wird auf die grafischen Grundlagen der Blocklib eingegangen, die im Rahmen dieser Bachelorarbeit verwendet wurden. Wenn die Grundlagen geklärt sind, geht es weiter mit der implementierten Architektur. Dieses Kapitel beinhaltet eine Komplettübersicht über die neu implementierten Klassen und Funktionalitäten, sowie das Zusammenspiel dieser neuen Klassen mit bereits vorhandenen Klassen. Nach diesem Überblick folgen die Details. Es wird erklärt, wie die Algorithmen validiert werden und somit visualisiert werden können. Um letzteres geht es im nächsten Kapitel. Detailliert wird beschrieben, wie eine Visualisierung zustande kommt, und welche Modalitäten sich Algorithmen teilen. Dass Tests sehr wichtig sind, wird auf den nächsten Seiten erläutert. Es wurden verschiedenartige visuelle- und Modultests durchgeführt, um die generierten Komponenten möglichst fehlerfrei zu halten. Im vorletzten Kapitel wird nun auf die Anwendung der implementierten Komponenten eingegangen. Dabei wird erklärt, wie der Student ohne Vorwissen zu dieser Bibliothek, einfach und sicher seine Algorithmen visualisieren lassen kann. Schließlich folgt noch ein Ausblick. Mögliche Erweiterungen dieser Komponente werden genannt und es wird ein Fazit über die vorliegende Bachelorarbeit gezogen.

2 Grundlagen

In diesem Kapitel werden die implementierten Algorithmen, deren Funktionen und Abläufe, sowie der Grund, warum genau diese Algorithmen in diese Arbeit aufgenommen wurden, beschrieben. Jeder dieser verwendeten Algorithmen wird durch eine Abbildung und/oder einen Pseudocode genauestens beschrieben und veranschaulicht. Außerdem beinhaltet dieses Kapitel eine Beschreibung der verwendeten Visualisierungsgrundlagen der Blocklib.

2.1 Sortieralgorithmen

Sortieralgorithmen sortieren Datensätze. Diese elementare Anwendung wird oft als Teilschritt anderer Algorithmen wie z.B. des Kruskal Algorithmus eingesetzt. Es gibt eine große Anzahl solcher Sortieralgorithmen, die sich anhand ihrer Vorgehensweise, Elemente zu sortieren, unterscheiden. Diese Arbeit beschränkt sich auf zwei solcher Algorithmen. Zum einen der *Insertionsort*, welcher ein einfacher Einstieg in die Welt der Algorithmen ist. Er ist nicht komplex und intuitiv leicht zu verstehen. Auf der anderen Seite wurde der *Quicksort* implementiert, der eine größere Herausforderung darstellt. Der Suchvorgang mit einem Quicksort wird durch das Aufteilen der Sortierliste in kleinere Teillisten und das rekursive Zusammensetzen dieser Teillisten zu dem sortierten Endergebnis sehr komplex. Beide Algorithmen bekommen als Eingabe eine Folge von n Zahlen $[a_1, a_2, \ldots, a_n]$. Diese Reihenfolge wird sortiert, sodass $[a_1' \leq a_2' \leq \cdots \leq a_n']$ gilt.

2.1.1 Insertionsort-Algorithmus

Bei den Insertionsort [Cor+13, S. 17] handelt es sich um ein Sortierverfahren, welches bei einer kleinen Anzahl von Elementen sehr effizient arbeitet. Der Ablauf dieser Sortierung gestaltet sich wie folgt: Eine unsortierte Folge von n Zahlen $[a_1, a_2, \ldots, a_n]$ wird schrittweise durch Einfügen sortiert. Diese Folge wird als Array dargestellt, welches komplett von vorne bis hinten durchlaufen wird. Man fängt mit dem Arrayindex $j = 1$, also dem zweiten Element, an. Dieses Element wird als *Schlüssel* bezeichnet, wie im Listing 1 (Vgl. [Cor+13, S. 18]) zu erkennen ist. In der Abbildung 1 wird dieser Schlüssel durch einen roten Zahlenwert gekennzeichnet. Dieser wird nun mit den Elementen links vom Schlüssel verglichen. In dem ersten Iterationsschritt der Abbildung 1 wäre das der Wert 3. Ist nun der Schlüssel kleiner als der Wert 3, werden die beiden Arrayindizes, und somit die Werte, vertauscht. Diese Vertauschungen werden solange durchgeführt, bis sich der Index des Schlüssels in der richtigen Position befindet. Ist dies der Fall, wird die Position *j + 1* zum neuen Schlüssel. Der Algorithmus terminiert, wenn das Element *j = A.länge* korrekt sortiert wurde.

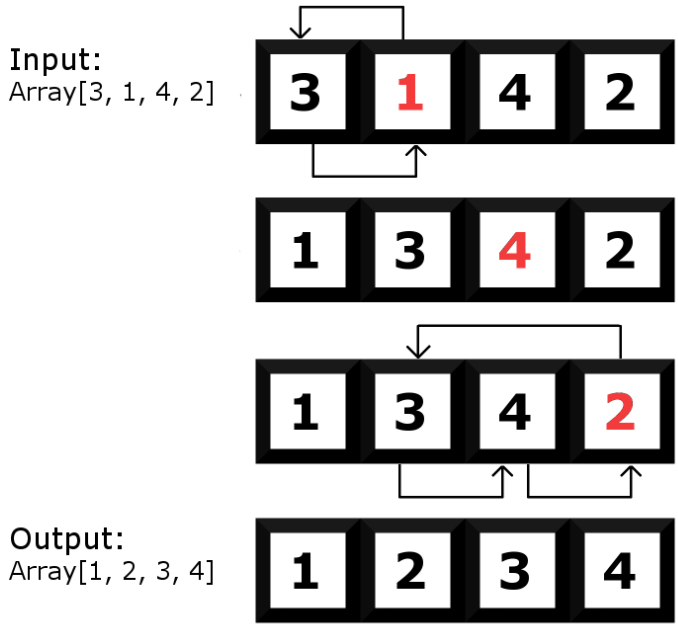

Abbildung 1: Ablauf des Insertionsorts-Algorithmus

```
1  INSERTIONSORT(A)
2       for j = 1 to A.länge
3               schlüssel = A[j]
4               // Füge A[j] in die
5               //sortierte Sequenz A[1..j-1] ein.
6               i = j - 1
7               while i > 0 und A[i] > schlüssel
8                   A[i+1] = A[i]
9                   i = i - 1
10              A[i + 1] = schlüssel
```

Listing 1: Pseudocode des Insertionsort-Algorithmus

2.1.2 Quicksort-Algorithmus

Der Quicksort-Algorithmus [Cor+13, S. 171] sortiert, genauso wie der bereits beschriebene Insertionsort, Datensätze. Er wendet dabei das *Teile-und-Beherrsche Paradigma* [Cor+13, S. 31] an, welches wie folgt abläuft: Das Gesamtproblem wird solange rekursiv in mehrere kleine Teilprobleme aufgespaltet, bis diese Teilprobleme klein genug sind, um sie mit geringem Aufwand lösen zu können. Die Teilprobleme werden anschließend wieder zu einer Lösung für das Gesamtproblem rekonstruiert. Im Falle eines Sortieralgorithmus ist das Gesamtproblem die unsortierte Zahlenfolge, die sortiert werden soll. Das Teile-und-Beherrsche Paradigma bei dem Quicksort-Algorithmus läuft folgendermaßen ab: Die unsortierte Zahlenfolge *A[p...r]* wird durch das *Pivotelement A[r]* in Teilfolgen zerlegt. Wie in Listing 2 zu sehen ist, sortiert die Methode `PARTITION` anschließend die Teilfelder. In Abbildung 2 wird eine Aufteilung einer Beispielfolge in Teilfolgen gezeigt. Hierbei ist der Bereich *A[p..i]* grün gekennzeichnet. Für diesen

Bereich gilt $A[p..i] \leq A[r]$. Die nächste Teilliste ist rot eingefärbt und es gilt *A[i+1..r-1] > A[r]*. Das Pivotelement selbst ist weiß und wird am Ende der Aufspaltung zwischen diesen Teilfolgen eingeordnet. Diese Prozedur wird so lange wiederholt, bis alle entstehenden Teillisten korrekt sortiert wurden.

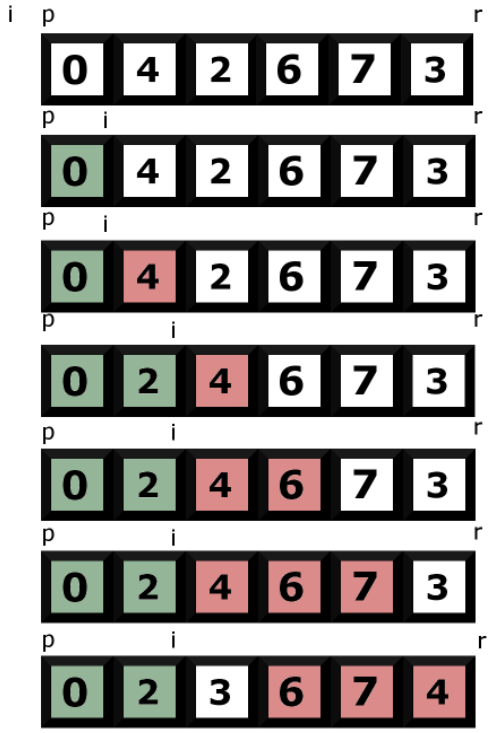

Abbildung 2: Ablaufs eines Durchgangs mit einem Pivotelement

```
1  QUICKSORT(A,p,r)
2      if p < r
3      q = PARTITION(A,p,r)
4      QUICKSORT(A,p,q-1)
5      QUICKSORT(A,q+1,r)
6
7
8  PARTITION(A,p,r)
9      x = A[r]
10     i = p-1
11     for j = p to r-1
12         if A[j] <= x
13             i = i + 1
14             vertausche A[i] mit A[j]
15     vertausche A[i+1] mit A[r]
16     return i+1
```

Listing 2: Pseudocode des Quicksort-Algorithmus

2.2 Suchalgorithmen

Ein *Suchalgorithmus* hat die Aufgabe, nach Objekten zu suchen. Dieses Konzept wird verwendet, um bestimmte Daten in einem großen Datensatz zu finden. Im Folgenden werden zwei solcher Algorithmen vorgestellt. Zum einen ist es das *Hashing* [Cor+13, S. 256], welches mithilfe einer *Liste* sucht. In der Blocklib wurde dieser Suchalgorithmus implementiert, da es für diesen Algorithmus viele praktische Anwendungen gibt. Verwendet werden *Hashtabellen* z.B. bei Datenbankmanagementsystemen [Sed14]. Hierbei werden große Datenbestände mit einer *Hashfunktion* durchsucht. Zum anderen ist es der *AVL-Baum*, der mit einem *binären Suchbaum (BST)* sucht. Ein BST ist eine Datenstruktur und besteht aus *Knoten* und *Schlüsseln*. Dabei dürfen die Schlüssel des linken Teilbaums eines Knotens nur kleiner oder gleich und die des rechten Teilbaums nur größer oder gleich gegenüber der Schlüssel des Knotens selbst sein.

2.2.1 Hashtabellen

Das Hashing ist eine Methode zur dynamischen Datenverwaltung. Sie hat allgemein drei wichtige Funktionen: Das Einfügen, Löschen und das Suchen von Elementen. Die Grundidee des Hashings ist es, eine eindeutige Identifikation der Listenpositionen zu schaffen, um die genannten Funktionen effizient umsetzen zu können. Dies wird mit einem sogenannten *Schlüssel* erreicht. Die Menge der Schlüssel S ist eine Teilmenge des vorhandenen Wertebereichs D. Um nun Operationen ausführen zu können muss zu jedem Schlüssel ein *Hashwert* ausgerechnet werden. Dies geschieht mit einer *Hashfunktion*, welche die Datensätze dann in ihrer Position in der Hashtabelle einordnen. Da der Speicherplatz endlich ist, kann es zu Kollisionen kommen, wenn unterschiedliche Schlüssel auf dieselbe Hashadresse abgebildet werden. Abbildung 3 zeigt ein solches Szenario. Dabei werden 2 Schlüssel auf die selbe Position 0 in der Hashtabelle abgebildet. Um solche Kollisionen zu vermeiden, muss festgelegt werden, dass jede Position $i \in \{1, 2, ...n\}$ in der Hashtabelle maximal einen Datensatz aufnehmen kann. Des Weiteren sollte ein Schlüssel an jeder Stelle in der Hashtabelle vorkommen können. In der Blocklib wurden drei Varianten implementiert, die diese Festlegungen einhalten. Folgende Übersicht liefert einen Überblick über diese Verfahren und deren Hash-Funktionen:

Lineare Sondierung (Sequentielle Verschiebung des Intervalls):

- $h(s, i) = (\hat{h}(s) + i) \bmod m$

Quadratisches Sondieren (Quadrieren des Intervalls):

- $h(s, i) = (\hat{h}(s) + c_1 i + c_2 i^2) \bmod m$

Doppel-Hashing (Zusätzliche Hash-Funktion):

- $h(s, i) = (h_1(s) + i * h_2(s)) \bmod m$

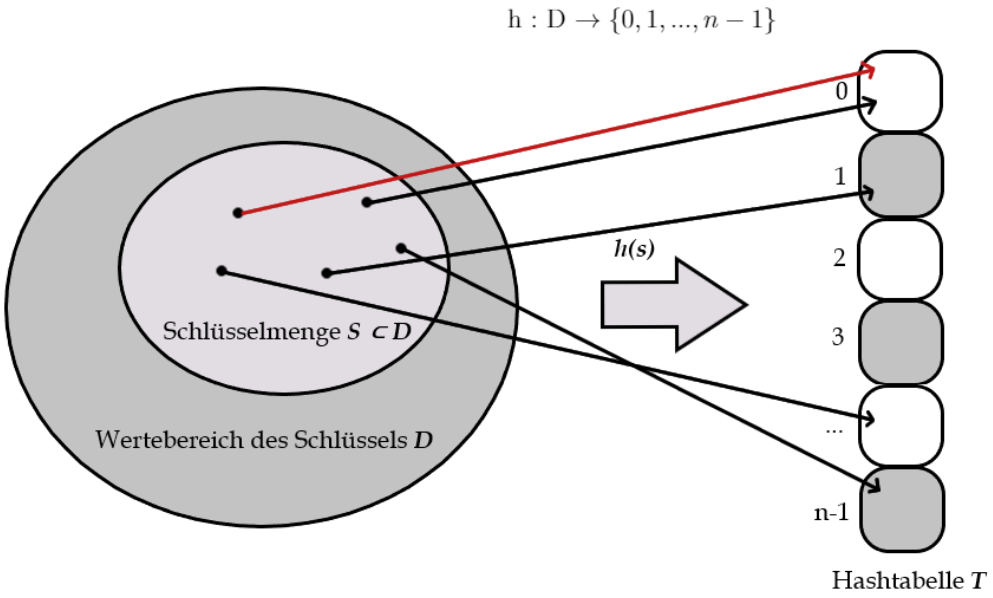

Abbildung 3: Verteilung der Schlüssel mittels Hashfunktion auf die Hashtabelle

2.2.2 AVL-Bäume

Ein AVL-Baum ist ein binär verketteter Suchbaum. Somit hat der Baum die Eigenschaften eines BST (siehe Kapitel 2.2). Der AVL-Baum hat außerdem die notwendige Eigenschaft, dass sich die Höhen der Knoten in den Teilbäumen nur um höchstens eins unterscheiden dürfen. Somit wird verhindert, dass der AVL-Baum nicht zu einem links- bzw. rechtsartigen Suchbaum entartet. Durch diese Eigenschaft wächst die Höhe des Baumes nur logarithmisch mit der Anzahl an Werten, die der Baum speichert. Operationen wie das Einfügen, Löschen und Suchen von Werten sind somit sehr effizient. Abbildung 4 zeigt links einen Baum, der alle Eigenschaften besitzt, die ein AVL-Baum haben muss. Dabei werden die Höhenunterschiede der einzelnen Knoten jeweils rechts neben dem Knoten angezeigt. Die Höhendifferenz beträgt hier höchstens 1. Rechts ist ein Baum zu sehen, der zwar die Eigenschaften eines BST erfüllt, nicht jedocht die eines AVL-Baumes. Durch das Einfügen und Löschen von Werten aus dem Baum wird diese balancierte Struktur aus der Bahn geworfen. Da nur ein maximaler Höhenunterschied von eins zwischen den Teilbäumen herrschen darf, müssen die Teilbäume der unbalancierten Suchbäume rotiert werden, um einen AVL-Baum zu generieren. Mit einer *Rotation* werden die Knoten des Baumes zusammen mit ihren Teilbäumen so bewegt, dass wiederum die AVL-Eigenschaften hergestellt sind. Man unterscheidet hierbei zwischen einer *Einfachrotation* und einer *Doppelrotation*. Abbildung 5 stellt ein Beispiel einer Einfachen Links-Rotation und einer Doppelten Rechts-/Links-Rotation dar.

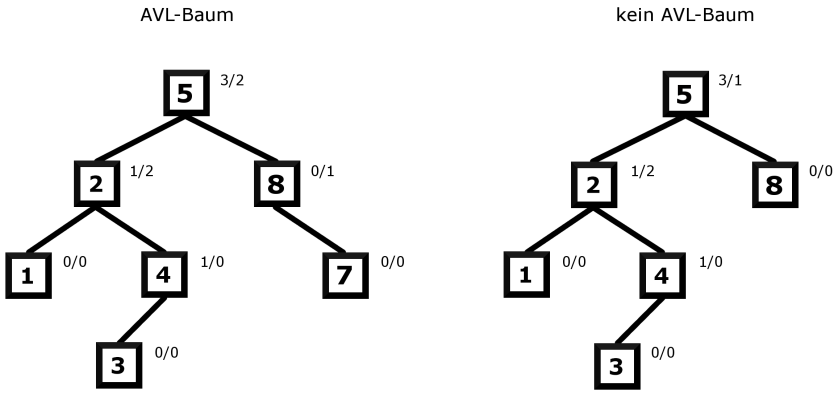

Abbildung 4: Beispielbäume mit den Höhenangaben der Teilbäume

Eine Einfachrotation ist notwendig, wenn jeweils der linke oder rechte äußere Teilbaum zu hoch ist. Im ersten Beispiel der Abbildung 5 ist der rechte, äußere Teilbaum zu hoch, deshalb erfolgt hier eine Linksrotation. Eine Doppelrotation hingegen wird benötigt, wenn jeweils der innere Teilbaum zu hoch ist. Im zweiten Beispiel in der Abbildung 5 ist dies der Fall. Durch Rotation ist die AVL-Eigenschaft des inneren Teilbaums wieder hergestellt, die des äußeren Teilbaums jedoch nicht. Mittels einer Linksrotation wurde nun wieder ein AVL-Baum gebildet.

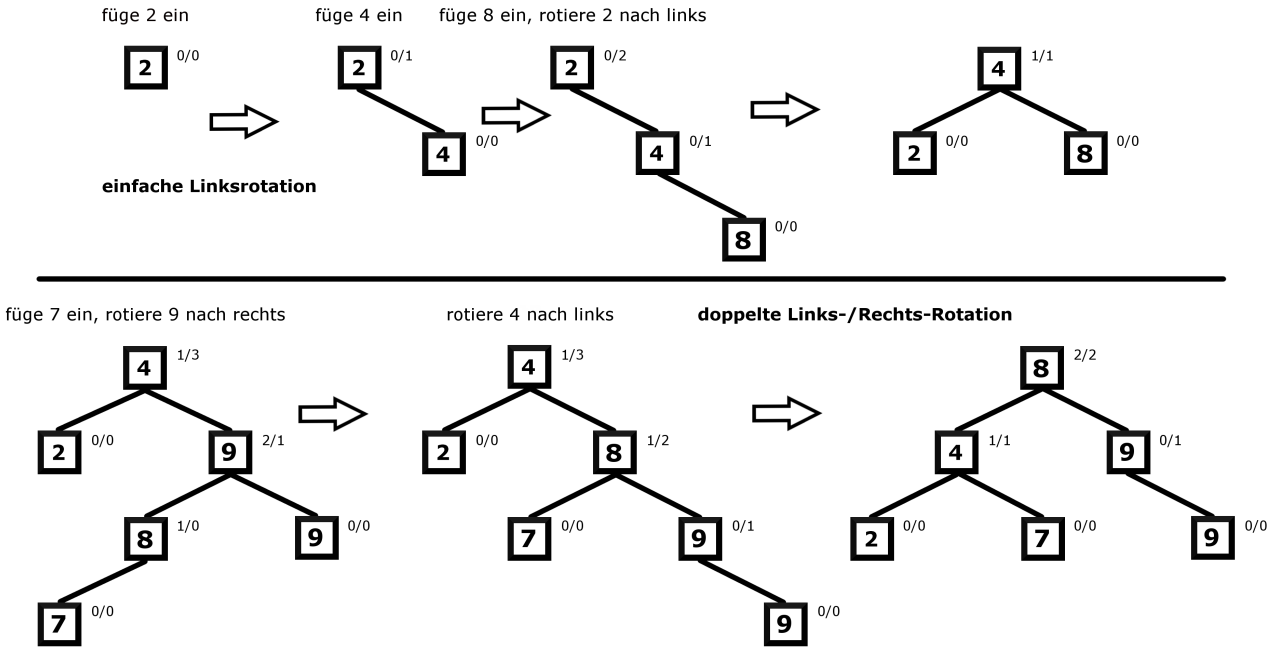

Abbildung 5: Beispiel einer einfachen- und doppelten Rotation

2.3 Graphalgorithmen

Graphalgorithmen beschreiben häufig strukturelle Zusammenhänge. Diese Graphen bestehen aus Knoten und sind untereinander mit Kanten verbunden, welche gewichtet sein können. Dabei können die Kanten auch eine Richtung aufweisen, also *gerichtet* sein, oder wiederum *ungerichtet* sein und somit keine Richtung besitzen. In Abbildung 6, wird ein Beispiel für einen gerichteten und einen ungerichteten Graphen vorgestellt:

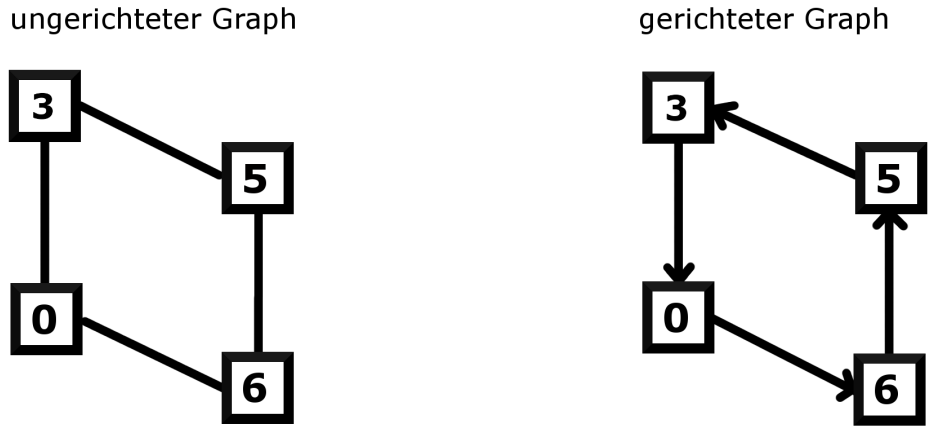

Abbildung 6: Links ein ungerichteter, rechts ein gerichteter Graph

Im folgenden Abschnitt wird der *Algorithmus von Kruskal* [Cor+13, S. 642] vorgestellt. Dieser hat zur Aufgabe, einen *Minimalen Spannbaum (MST)* eines vorgegebenen Graphen aufzustellen. Ziel hierbei ist es, alle vorhandenen Knoten des Graphens mit minimalen Kosten zu verbinden.

2.3.1 Algortihmus von Kruskal

Der Algorithmus von Kruskal bildet aus einem ungerichteten, kantengewichteten Graphen einen kreisfreien und minimalgewichtigen Spannbaum. Abbildung 7 zeigt einen Minimalen Spannbaum eines zufällig gewählten Graphs mit den optischen Gegebenheiten der Implementierung in der Blocklib. In Listing 3 (Vgl. [Cor+13, S. 642]) ist der Pseudocode des Algorithmus abgebildet. Der Algorithmus läuft wie folgt ab: Die Gewichte der einzelnen Kanten werden in nicht fallender Reihenfolge sortiert. Somit hat man eine Reihenfolge der Kanten, welche die geringsten Gewichtungen aufweisen. Anschließend werden die Gewichte in jener Reihenfolge ausgewählt und zu einem neuen Spannbaum hinzugefügt, vorausgesetzt, es entsehen keine Zyklen. Ein Zyklus zwischen Knoten ist nicht erlaubt, da bereits zwei dieser Knoten ausgewählt wurden und somit das Verbinden dieser zwei Knoten mit einem dritten keinen Minimalen Spannbaum bilden würde. Da jeder Knoten erreicht werden muss, terminiert der Algorithmus, wenn alle Knoten mindestens einmal erreicht wurden.

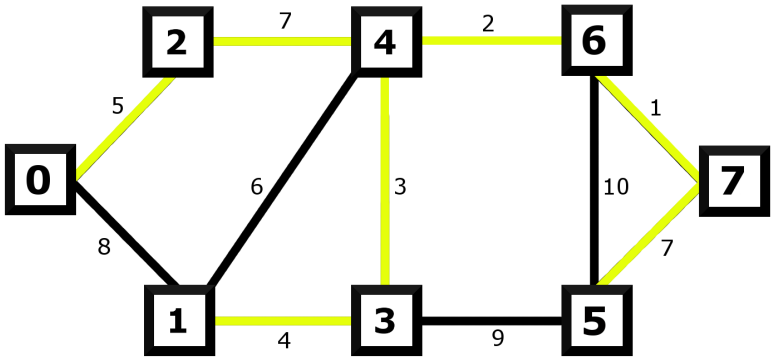

Abbildung 7: Minimaler Spannbaum eines Graphen

```
1  MST-KRUSKAL(G, w)
2      A = { }
3      for jeden Knoten v von G.V
4          MAKE-SET(v)
5      sortiere die Kanten aus G.E in nichtfallender Reihenfolge nach dem Gewicht w
6      for jede Kante (u,v) von G.E, in nichtfallender Reihenfolge nach ihren Gewichten
7          if kein Zyklus
8              A = A vereinigt mit {(u,v)}
9              Vereinige u & v
10 return A
```

Listing 3: Pseudocode des Algorithmus von Kruskal

2.4 Grundlagen der Visualisierung in der Blocklib

Für die Visualisierungsgrundlagen von Algorithmen wurde in der vorliegenden Ausarbeitung auf bereits implementierte Methoden zurückgegriffen. Es wurden jedoch auch eigene Funktionalitäten und Texturen hinzugefügt. Eine Übersicht über diese Grundlagen liefern die folgenden Zeilen.

Blöcke als Visualisierungsgrundlage:
Die Blocklib besteht, wie das Vorbild Minecraft [Min], aus Blöcken. Dabei können Blöcke unterschiedliche Texturen aufweisen, sei es eine Lava-Textur für Vulkanausbrüche, implementiert von Tobias Müller in Form einer früheren Masterarbeit [Mül17], oder seien es Gras-Texturen, welche für die Weltengenerierung von Daniel Beer [Bee17] eingeführt wurden. In dieser Komponente werden Blöcke zur Visualisierung der Algorithmen verwendet. Dabei wurden einige neue Texturen in die Blocklib hinzugefügt. In Abbildung 8 sieht diese neuen Blöcke zu sehen. Sie bestehen aus den Zahlen $0 - 10$, einmal in schwarz für die Sortierelemente bei Sortieralgorithmen und bei den Such- und Graphalgorithmen als Knoten, sowol einmal in rot, wobei diese

als Positionsindex einer Hashtabelle dienen. Außerdem wurde eine Pfeil-Textur als Sortierindex implementiert.

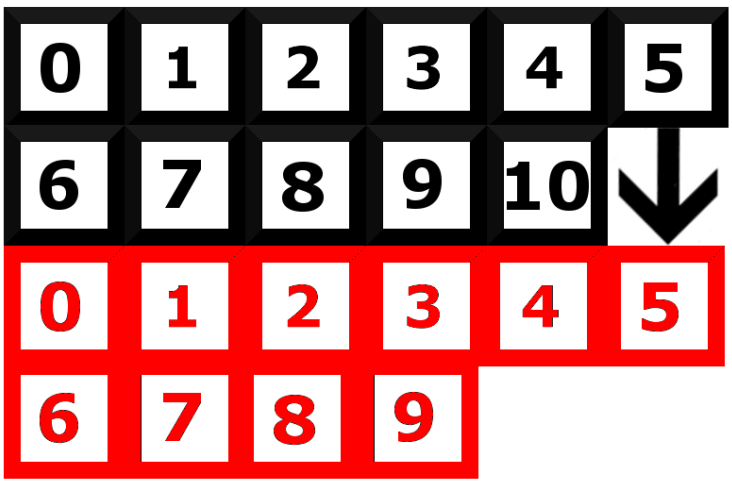

Abbildung 8: Die neu-hinzugefügten Texturen

Verbinden der Blöcke mit Linien:
Bei verschiedenen Algorithmen, wie z.B. den AVL-Bäumen oder dem Algorithmus von Kruskal, müssen diese Knoten bzw. Blöcke untereinander verbunden werden. Dies geschieht mithilfe der bereits implementierten Methode `addRenderable` der Klasse `MasterRenderer`. Diese Funktionalität wurde von Tobias Werner [Wer18] im Rahmen einer Masterarbeit implementiert. Die Methode benötigt als Übergabeparameter Start- und Zielkoordinaten, um eine Linie zu ziehen. Mithilfe der in den einzelnen Benutzerschnittstellen der Algorithmen global deklarierten Variable `currentPosition`, können diese Koordinaten dynamisch übergeben werden.

Der Pfeil als Zeiger:
Wie in Abbildung 8 zu sehen ist, wurde eine Pfeil-Textur in die Blocklib eingefügt. Dieser Block hat die Funktion, bei Sortiervorgängen auf bestimmte Arrayindizes zu zeigen. Durch diese einfache Erweiterung weiß der Benutzer immer, welcher Index gerade bearbeitet wird.

3 Architektur der Implementierung

In diesem Kapitel wird auf die Architektur der Implementierung und das Zusammenspiel der einzelnen Pakete eingegangen. Die detaillierte Beschreibung der einzelnen Klassen und Abläufe folgt in den nächsten Kapiteln. Wie in Abbildung 9 zu sehen ist, wurden jeweils drei Pakete für die unterschiedlichen Algorithmen-Typen (`SortAlgorithm`, `SearchAlgorithm` und `GraphAlgorithm`) implementiert.

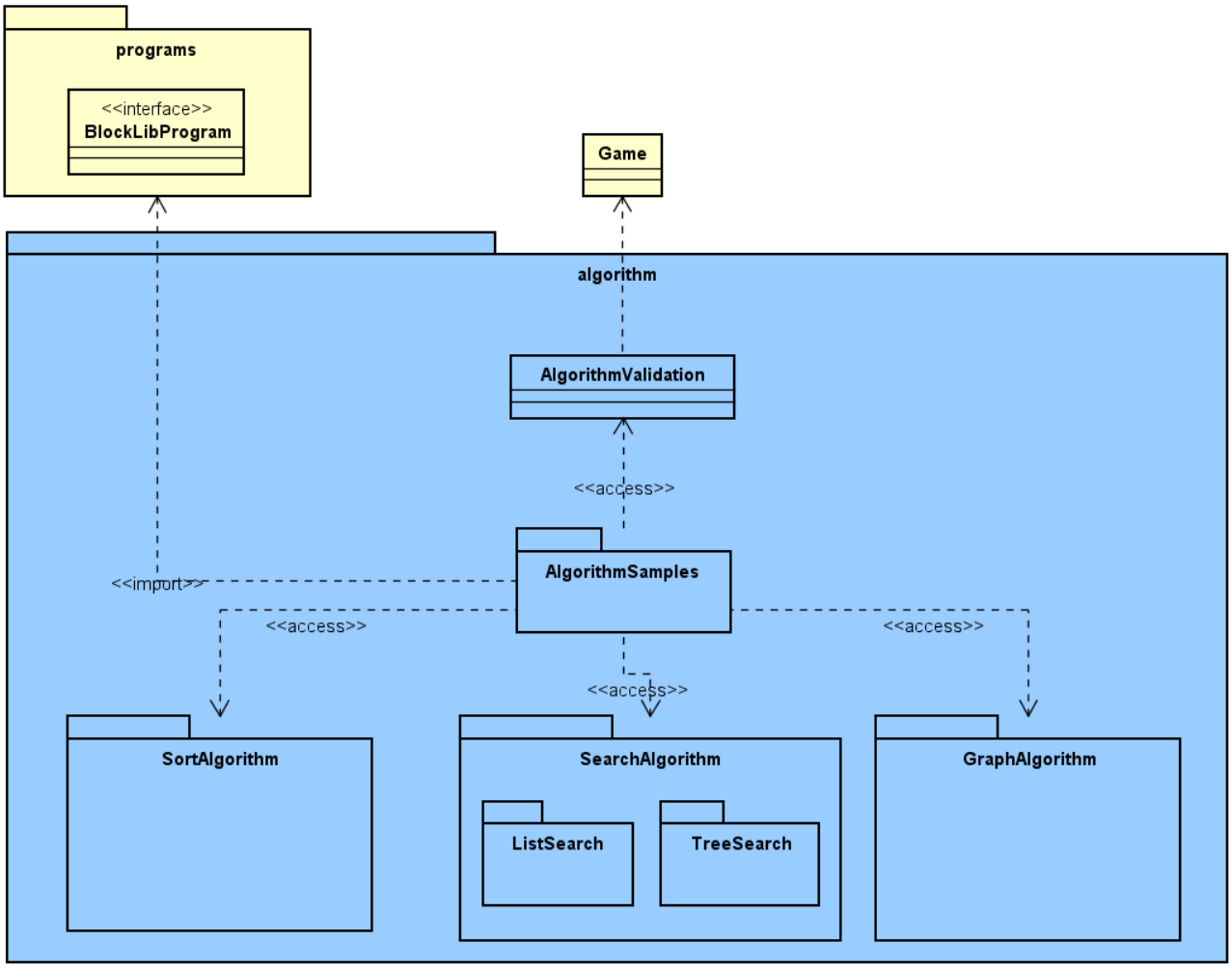

Abbildung 9: UML-Diagramm zur Architektur der Algorithmen-Komponente

Diese Pakete enthalten jeweils Klassen, welche gemeinsame Funktionalitäten der einzelnen Algorithmen in eigenen modularen Klassen zusammenfassen. Außerdem wurden algorithmenspezifische Klassen erstellt, deren Methoden nur zu dem jeweiligen speziellen Algorithmus passen. Im Kapitel 5 werden diese Klassen detailliert beschrieben. Das Paket `AlgorithmSamples` enthält die Benutzerschnittstellen der einzelnen Algorithmen, welche im Kapitel 7 vorgestellt werden. Diese Schnittstellen greifen auf die Klassen der Pakete `SortAlgorithm`, `SearchAlgorithm` und

`GraphAlgorithm` zu, um deren Methoden zu benutzen. Außerdem implementieren die Klassen des Pakets `AlgorithmSamples` die Klasse `BlockLibProgramm` des Pakets `Programs`, um deren Methoden zu erben. Mit diesen Methoden ist es nun simpel, die Visualisierung durchzuführen, da u.a. wichtige Funktionen wie `onGameStart`, welche das Spiel initialisiert und vorbereitet, und `keyPressed`, mit welcher man die Algorithmen schrittweise visualisieren kann, enthalten sind. Von der jeweiligen Benutzerschnittstelle im Paket `AlgorithmSamples` geht es weiter zu der Klasse `AlgorithmValidation`. Diese validiert die Eingaben in den Benutzerschnittstellen mit einigen Abfragen, welche im nächsten Kapitel vorgestellt werden. Sind alle Abfragen zulässig, wird mit der Klasse `game` aus dem Paket `Blocklib` das Spiel gestartet und die Visualisierung beginnt.

Die Aufteilung der Pakete wurde so gewählt, um dem Benutzer unnötige Komplexität zu verbergen. Dieser sieht nur die Benutzerschnittstellen im Paket `AlgorithmSamples` und kann diese somit, unabhängig von der Implementierung der Details, ausführen. Außerdem wurden die drei Algorithmen-Pakete `SortAlgorithm`, `SearchAlgorithm` und `GraphAlgorithm` voneinander abgegrenzt, um eine einheitliche und modulare Klassenstruktur zu generieren. Durch diese Aufteilung wurde die Erweiterbarkeit der Komponenten ermöglicht.

4 Automatische Validierung der Lehralgorithmen

Mithilfe der Klasse `AlgorithmValidation` wird die automatische Überprüfung der Implementierung des Benutzers durchgeführt. Es sind sowohl allgemeine, unterstützende Abfragen auf Korrektheit des Codes implementiert, alsr auch spezielle Überprüfungen, welche auf die einzelnen Lehralgorithmen abgestimmt sind. Grundlage der Untersuchung auf Korrektheit der studentischen Implementierung ist der jeweilige Mustercode, der in den einzelnen Unterklassen der Algorithmen hinterlegt ist. Dabei werden die Parameter abgefangen und verglichen. Durch das Abfragen wird sichergestellt, dass die Implementierung des Anwenders korrekt arbeitet und das gewünschte Ergebnis liefert. Auf diese Weise können auch verschiedene Implementierungen der Algorithmen zum Ziel führen. In Abbildung 10 ist das Klassendiagramm von `AlgorithmValidation` abgebildet. Folgende, allgemeine Untersuchungsmethoden wurden entwickelt:

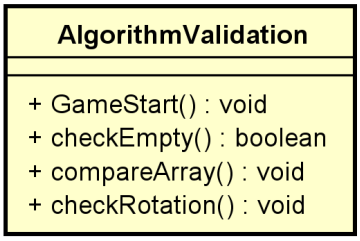

Abbildung 10: UML-Diagramm der Klasse `AlgorithmValidation`

Überprüfung der Eingabe:
Es werden statische Variablen deklariert, die von den Studenten definiert werden müssen. Zusammen mit diesen Variablen gibt der Student seine Ausführung des Algorithmus in die Schnittstelle ein. Diese Variablendefinitionen werden vor der Visualisierung mit den Werten der Mustervariablen verglichen. Die Methode `compareArray` wird von allen Schnittstellen aufgerufen. Sie bekommt als Übergabeparameter zwei Arrays, welche dann verglichen werden. Sind die Arrays identisch, ist diese Abfrage korrekt. Zusätzlich wird mit der Funktion `checkEmpty` noch abgefragt, ob die vorgegebenen, statischen Variablen überhaupt deklariert wurden. Diese beiden Abfragen müssen positiv verlaufen, da sonst keine Visualisierung des Algorithmus erfolgt.

Endergebnis:
Wenn der Algorithmus terminiert, werden jeweils das Ergebnis des Benutzercodes, sowie das Ergebnis des korrekt ablaufenden Mustercodes mit der Methode `compareArray` verglichen. Dies passiert ebenfalls vor der Ausführung der Visualisierung, damit Studenten bei einer falschen Implementierung schnell ihre Eingabe überarbeiten können. Außerdem macht es keinen Sinn, den Algorithmus bis zu einer kritischen Stelle zu visualisieren, wenn dieser falsch ist.

Spezielle Überprüfungen:
Diese Überprüfungen alleine reichen nicht aus, um dem Benutzer eine korrekte Ausführung des Algorithmus zu garantieren. Darum wurden neben jenen Abfragen auch noch spezielle, auf die jeweiligen Algorithmen zugeschnittenen Korrektheitsüberprüfungen implementiert:

- **Suchalgorithmische Überprüfungen:**
  - **Hashing**
    Die Implementierung des Hashings unterscheidet sich von dem Rest der Algorithmen in der Blocklib (siehe Kapitel 3.3.1). Hierbei ist eine Beispielberechnung der Werte mittels einer Hashfunktion in der Klasse `Hashing` hinterlegt. Anhand diesem Muster wird das Ergebnis der vom Benutzer eingegeben Hashfunktion überprüft und bei Korrektheit visualisiert.
  - **AVL-Bäume**
    AVL-Bäume rotieren ihre Knoten häufig. Eine einfache Überprüfung der Start- und Endwerte reicht hierbei nicht aus. Es muss zusätzlich überprüft werden, wann und wie rotiert wird. Dafür ist die Methode `checkRotation` zuständig. Sie wird im Falle einer Rotation aufgerufen und prüft, unabhängig von dem eingegebenen Code des Benutzers, ob die Rotationen richtig durchgeführt werden.
- **Graphalgorithmische Überprüfungen:**
  Bei den Graphalgorithmen werden Kantengewichte und deren Knotenpositionen abgegriffen. Diese Werte müssen validiert werden. Da diese Algorithmen ihre Werte in Arrays speichern, sind mit `compareArray` die meisten Überprüfungen durchführbar. Algorithmeninterne Abfragen wie z.B. das Vermeiden der Bildung von Zylken bei der Generierung eines Minimalen Spannbaumes, wurden bereits in den Graphalgorithmen-Klassen integriert.

5 Visualisierung der Algorithmen

In diesem Kapitel wird auf die Erstellung und Integrierung der Komponenten für eine Visualisierung von Algorithmen aus der Lehre eingegangen. Für dieses Vorhaben wurde die Blocklib um einige neue Klassen und Funktionalitäten erweitert. Diese neuen Komponenten arbeiten mit bereits vorhandenen Methoden zusammen. Gleiche Funktionalitäten wurden innerhalb dieser Pakete in eigenen Klassen zusammengefasst, um die Implementierung so modular und erweiterbar wie nur möglich zu halten. Dieses Kapitel fängt mit der Beschreibung des Pakets `SortAlgorithm` an und behandelt darauf noch das Paket `SearchAlgorithm` und `GraphAlgorithm`.

5.1 Das Paket `SortAlgorithm`

Die Sortieralgorithmen enthalten zu dem jetzigen Zeitpunkt drei Klassen: `SortManager`, `InsertionSort` und `QuickSort`. Diese Klassen wurden so konzipiert, dass verschiedene Sortieralgorithmen auf die Klasse `SortManager` zugreifen können, um Methoden zu nutzten, die jeder Sortieralgorithmus in dieser Implementierung benötigt. Abbildung 11 stellt das Assoziationsverhältnis der Klassen dar. Diese drei Entitäten werden in den nächsten Unterkapiteln vorgestellt.

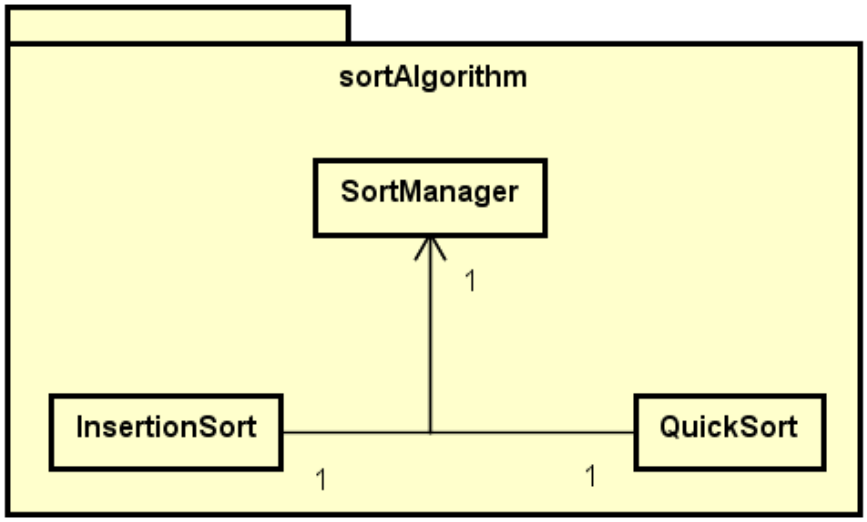

Abbildung 11: UML-Diagramm der Sortieralgorithmen

5.1.1 Die Visualisierung eines Sortieralgorithmus

Die Klasse `SortManager` stellt Funkionalitäten bereit, welche Sortieralgorithmen zur korrekten Visualisierung benötigen. Diese sind so konzipiert, dass sie ohne Probleme von jedem Sortieralgorithmus übernommen werden können. Abbildung 12 zeigt das *UML-Diagramm* der Klasse `SortManager` und deren Methoden zur Visualisierung der Sortierreihenfolgen und eines Pfeils:

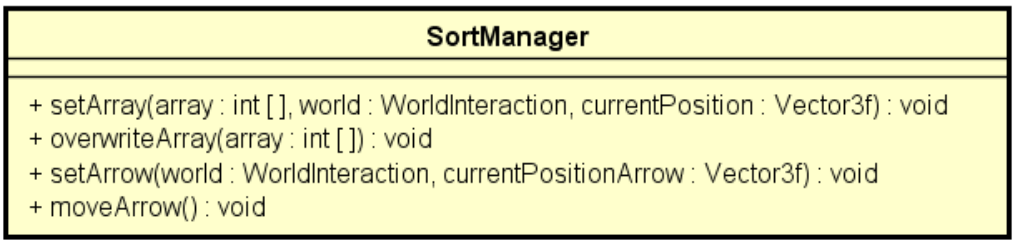

Abbildung 12: UML-Diagramm der Klasse `SortManager`

Das Visualisieren eines Arrays:
Die Methode `setArray` bekommt als Übergabeparameter ein statisch definiertes Array, dass in der Benutzerschnittstelle für Sortieralgorithmen definiert wird. Dieses Feld wird bis zum Ende durchlaufen und die Werte der Indizes abgegriffen. Diese Werte werden wiederum mit den dazugehörigen Blöcken in der Blocklib visualisiert. Die Variable `currentPosition`, welche auch für das Setzen der Linien zwischen zwei Blöcken genutzt wird (siehe Kapitel 2.4), wird dabei bei jedem Einsetzen eines Blockes inkrementiert, um eine aufeinanderfolgende Abbildung der Blöcke zu gewährleisten. Würde man die `currentPosition` nicht inkrementieren, würden alle Blöcke an derselben Position abgebildet werden. Folglich entstünde der Anschein, nur ein Block wäre in der Welt visualisiert worden. In Abbildung 13 ist eine solche Visualisierung eines Arrays, welches bereits sortiert ist, zu sehen.

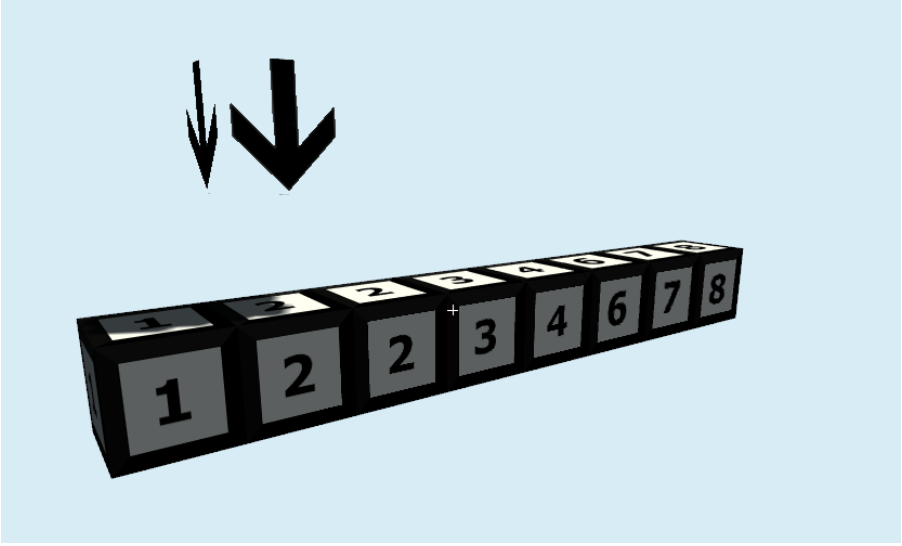

Abbildung 13: Beispielvisualisierung einer sortierten Zahlenfolge

Das Darstellen eines neuen Arrays:
Um eine fehlerfreie und richtig sortierte Darstellung der Elemente zu gewährleisten, wurde die Methode `removeArray` implementiert. Diese bekommt als Übergabeparameter ebenfalls ein Array. Mithilfe einer Schleife und der bereits festgelegten Variable `currentPositiont` werden nun alle Blöcke des Arrays überschrieben. Somit wird eine bisherige Visualisierung durch eine neue Reihenfolge der Elemente ersetzt. Dies wird verwendet, um die einzelnen Zwischenschritte der Sortierung zu visualisieren. Dieses Vorgehen, alle Elemente auf einmal neu zu besetzen, hat sich als effizienter und einfacher herausgestellt, als dass man einzelne Elemente untereinander tauscht.

Das Setzen des Pfeils:
Die Methode `setArrow` setzt einen Pfeil an die gewünschte Position. Dabei wird die bereits festgelegte Variable `currentPositionArrow` in der Benutzerschnittstelle (siehe Kapitel 7.1) um jeweils die x- und y-Achse translatiert. Der Pfeil zeigt den aktuellen Index in der Sortierreihenfolge beim Insertionsort an. Bei dem Quicksort hingegen, wird damit das Pivotelement angezeigt.

Das Bewegen des Pfeils:
Der Pfeil muss bewegt werden können und darf nicht statisch auf ein Element zeigen, da sich die Zahlenfolgen und die zu sortierenden Elemente beim Ablauf eines Sortieralgorithmus ständig ändern. Im Falle des Insertionsorts z.B. soll der Pfeil nach jedem Iterationsschritt auf das nächste Element im Array zeigen. Hierbei wird einfach die Variable `currentPositionArrow` um jeweils 1 in x-Richtung inkrementiert.

5.1.2 Spezielle Implementierungen des InsertionSorts-Algorithmus

Diese Klasse enthält zwei Funktionen, die speziell für den Insertionsort implementiert wurden. `patternInsertionSort` dient als Validierungsvorlage für die Klasse `AlgorithmValidation` und die Methode `visualizeInsertionSort` steuert den Visualisierungsvorgang speziell für den Insertionsort-Algorithmus.

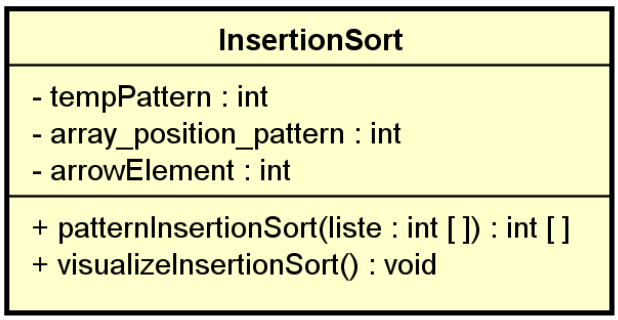

Abbildung 14: UML-Diagramm der Klasse `InsertionSort`

Der Mustercode:
Die Methode `patternInsertionSort` enthält den Mustercode für den Insertion Sort und ist somit die Validierungsgrundlage der Eingabe des Benutzers. Sie wird den Methoden `checkEmpty` und `compareArray` als Übergabeparameter übergeben und daraufhin auf Korrektheit überprüft. Der prinzipielle Ablauf der Methode `patternInsertionSort` wird in Kapitel 2.1.1 erklärt.

Der Visualisierungsvorgang des Insertionsorts-Algorithmus:
Neben diesem Muster wird auch die Funktion `visualizeInsertionSort` bereitgestellt, mit welcher es möglich ist, den Ablauf des Insertionsorts schrittweise darzustellen. Dieser Visualisierungsvorgang wird durch den Tastendruck *U* mithilfe der Programmbibliothek *LWJGL* [Lib] ausgelöst (Siehe Kapitel 7). Um an die einzelnen Zwischenschritte des Algorithmus zu kommen, wurde der Mustercode des Insertionsorts modifiziert. Als Grundlage wird der Pseudocode von Listing 1 verwendet. Von diesem Pseudocode wurde die erste for-Schleife entfernt und eine global definierte Variable `arrowElement` eingeführt, welche inkrementiert wird, sobald die Methode `visualizeInsertionSort` aufgerufen und abgearbeitet wurde. Durch dieses Vorgehen kann man die einzelnen Iterationsschritte des Algorithmus durch mehrmaligen Aufruf der Methode `visualizeInsertionSort` abgreifen. Dies wird realisiert durch den Tastendruck *J*. Die lokalen Variablen `temp`, `firstElement` und `secondElement` dienen zum Vertauschen der Arraypositionen, wobei `temp` eine Hilfsvariable darstellt und nur zum Tausch der Variablen `firstElement` und `secondElement` dient. Die Methode `visualizeInsertionSort` bedient sich auch an den Funktionen der Klassen `SortManager` (Vgl. Kapitel 5.1.1). Es werden `removeArray` und `setArray` aufgerufen, um die Visualisierung das geänderten Sortierarrays zu ändern. Die Variable `currentPosition` wird in x-Richtung wieder auf 0 gesetzt, um eine Visualisierung an der gleichen Stelle zu garantieren. Am Ende der Methode wird noch `moveArrow` der Klasse `SortManager` aufgerufen, um den Pfeil immer auf das, zu sortierende Element, zu zeigen. `arrowElement` wird inkrementiert, um mit dem nächsten Tastendruck *J* einen Schritt weiter in der Sortierung des Arrays zu kommen. In Listing 4 ist der ein Ausschnitt der Klasse `patternInsertionSort` zu sehen.

```
1        public static void visualizeInsertionSort() {
2        while ((arrowElement> 0) && (Insertion_Sample.unsortedList[arrowElement-1]>
             Insertion_Sample.unsortedList[arrowElement])) {
3
4           int temp = 0;
5           int firstElement = Insertion_Sample.unsortedList[arrowElement];
6           int secondElement = Insertion_Sample.unsortedList[arrowElement-1];
7
8           //swapping elements
9           temp = secondElement;
10          secondElement = firstElement;
11          Insertion_Sample.unsortedList[arrowElement-1] = firstElement;
12          firstElement = temp;
13          Insertion_Sample.unsortedList[arrowElement] = temp;
14
15          //overwrite the old visualization
16          SortManager.overwriteArray(Insertion_Sample.unsortedList);
17          SortManager.setArray(Insertion_Sample.unsortedList);
```

```
18            Insertion_Sample.currentPosition.x  = 0;
19
20            arrowElement--;
21        }
22        //set the Arrow and go on to the next element
23        SortManager.moveArrow();
24        arrowElement++;
25    }
```

Listing 4: Der Modifizierte Code von `patternInsertionSort`

5.1.3 Spezielle Implementierungen des QuickSort-Algorithmus

Die Klasse `QuickSort` beinhaltet, ähnlich wie der Insertionsort-Algorithmus, eine Validierungsvorlage und Methoden, die die Visualisierung eines Quicksort-Algorithmus steuern. Das UML-Diagramm in Abbildung 15 zeigt die implementierten Methoden dieser Klasse.

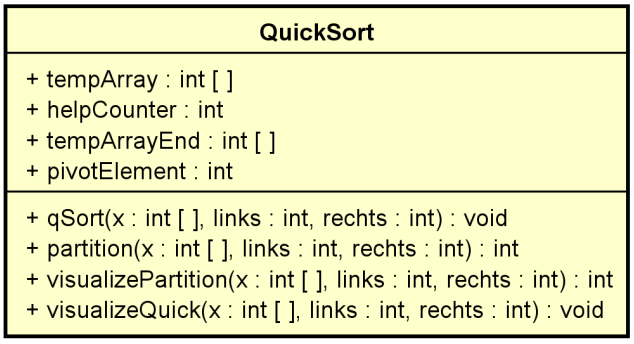

Abbildung 15: UML-Diagramm der Klasse `QuickSort`

Der Mustercode:
Die Musterberechnung einer Sortierung nach dem Quicksort-Algorithmus wird mit den beiden Methoden `qSort` und `partition` durchgeführt. Diese beiden Funktionen halten sich an die Vorgehensweisen des Algorithmus, welche im Kapitel 2.1.2 vorgestellt wurden, wobei `qSort` die Methode `QUICKSORT` des Pseudocodes in Listing 2 repräsentiert, und `partition` die Methode `PARTITION`.

Der Visualisierungsvorgang des Quicksort-Algorithmus:
Um diesen Algorithmus nun vollständig visualisieren zu können, braucht man die Methoden `visualizePartition` und `visualizeQuick`. Die beiden Methoden fangen die Zwischenschritte des Quicksort-Algorithmus ab und visualisieren diese. `visualizeQuick` lehnt sich stark an die Methode `qSort` an. Sie läuft im Prinzip gleich ab, jedoch in drei Einzelschritten, welche mit dem Tastendruck J abgerufen werden können. Zuerst wird die unsortierte Reihenfolge mit Aufruf der Methode `visualizePartition` in die Teillisten aufgespaltet. Die dabei anfallenden Vertauschungen der Elemente werden einzeln Visualisiert. Ist dies passiert, werden die linken

Teillisten nach dem gleichen Prinzip bearbeitet. Als letzter Schritt, werden die rechten Teillisten sortiert. Durch die Hilfsvariable `helpCounter`, kann man jeden dieser Schritte einzeln abrufen, indem nach jedem Aufruf von `visualizeQuick` die Variable `helpCounter` inkrementiert wird. Die Methode `visualizePartition` visualisiert nach jedem Tauschen der Elemente und nach jedem Verschieben des Pivotelements die komplette Sortierreihenfolge. Bei dem Quicksort-Algorithmus werden die einzelnen Arrays untereinander visualisiert. Da bei diesem Algorithmus sehr viele kleine Einzelschritte erfolgen können, und somit der Überblick und die Nachvollziehbarkeit der Arbeitsschritte erschwert werden wenn das Array immer an der gleichen Stelle visualisiert wird, wurde dies anders gestaltet als beim Insertionsort-Algorithmus. Zusätzlich ist der Wechsel des Pivotelements durch einen Roten Strich zwischen den Sortierreihenfolgen gekennzeichnet. So kann man leicht erkennen, wann Elemente getauscht werden und wann ein neues Pivotelement ausgesucht wird. Das Pivotelement selber ist stets durch einen roten Block gekennzeichnet. Abbildung 16 zeigt einen Ausschnitt einer Beispiel-Visualisierung des Quicksort-Algorithmus.

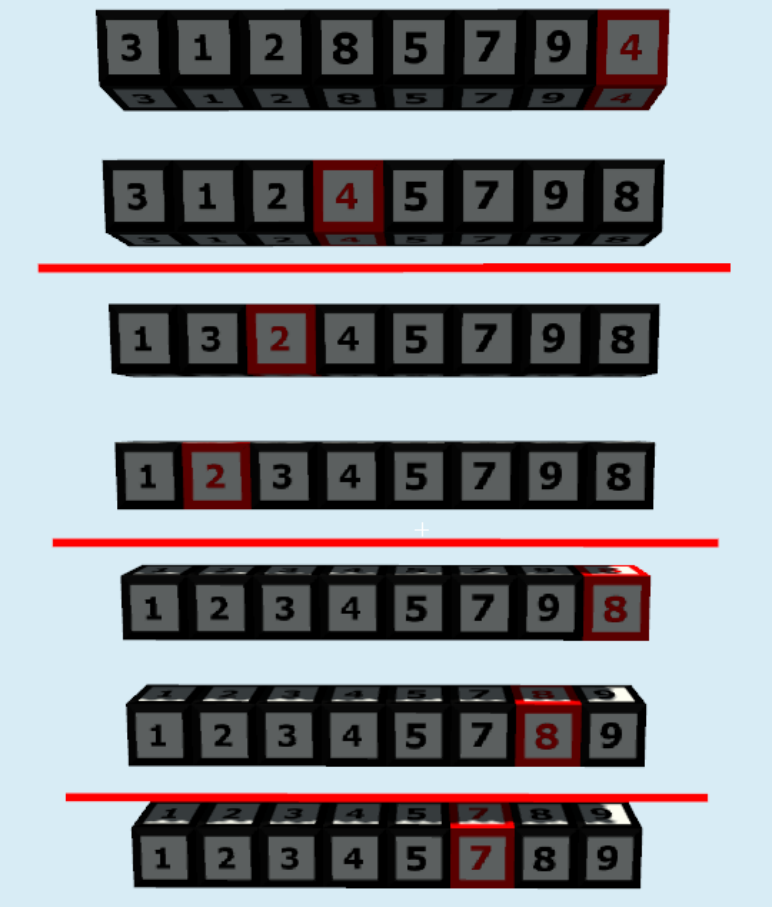

Abbildung 16: Ausschnitt der Visualisiserung des Quicksort-Algorithmus

5.2 Das Paket `SearchAlgorithm`

Das Paket `SearchAlgorithm` besteht wiederum aus zwei Paketen (`ListSearch` und `TreeSearch`). Diese Aufteilung wurde festgelegt, da sich das Suchen in Listen und Bäumen zu sehr voneinander unterscheidet und es somit keinen Sinn gemacht hätte, die jeweiligen Klassen in einem Paket zu implementieren. In diesem Unterkapitel wird zuerst auf die Visualisierung einer Hashtabelle unter Verwendung einer Hashfunktion eingegangen, um daraufhin zu dem Aufbau der Baum- und AVL-Klassen zu kommen.

5.2.1 Die Visualisierung einer Hashtabelle

Das Paket `ListSearch` besteht aus einer Klasse `ListSearchVisualization`. Diese beinhaltet fünf Methoden, mit denen dynamisch Hashtabellen, Positionsindizes und die Eingliederung der Elemente in diese Indizes durch Pfeile, generiert und visualisiert werden können. In Abbildung 17 ist das UML-Diagramm der Klasse `ListSearchVisualization` gezeigt.

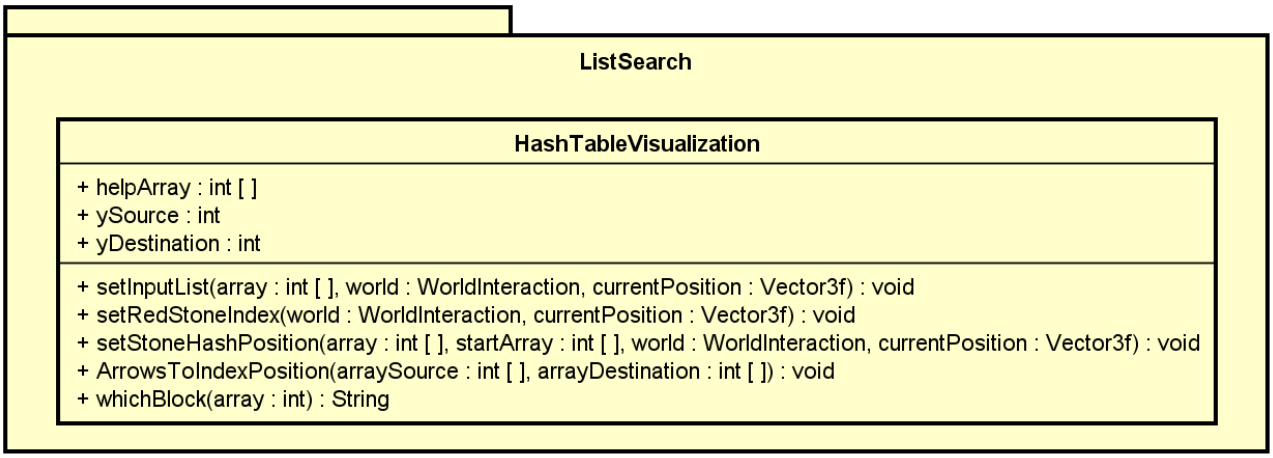

Abbildung 17: UML-Diagramm des Pakets `ListSearch`

Das Visualisieren der Eingabewerte:
Für die Visualisierung benötigt man als erstens die Werte, welche durch eine Hashfunktion in die Hashtabelle eingegliedert werden sollen. Diese Werte werden mit der Methode `setInputList` visualisiert, welche als Übergabeparameter ein Array dieser einzugliedernden Werte bekommt. Außerdem wird eine Variable `world` vom Typ `WorldInteraction` und eine `currentPosition` vom Typ `Vector3f` übergeben. Dies ist notwendig, um die Methode modula zu halten und um eine Erweiterbarkeit um neue Algorithmen zu gewährleisten, da diese Methoden auf diese Weise von jedem beliebigen Algorithmus aufgerufen werden können. Das übergebene Array wird visualisiert, indem man das Array mit einer Schleife durchläuft und nach jedem Setzten eines Blockes die $y - Koordinate$ von `currentPosition` um eins inkrementiert. Somit werden die Blöcke übereinander gesetzt. Ein Beispiel des Hashing mit der Hashfunktion $h(s, 3) = (3 * \hat{h}(s) + 3)\ mod\ 8$ wird in Abbildung 18 visualisiert.

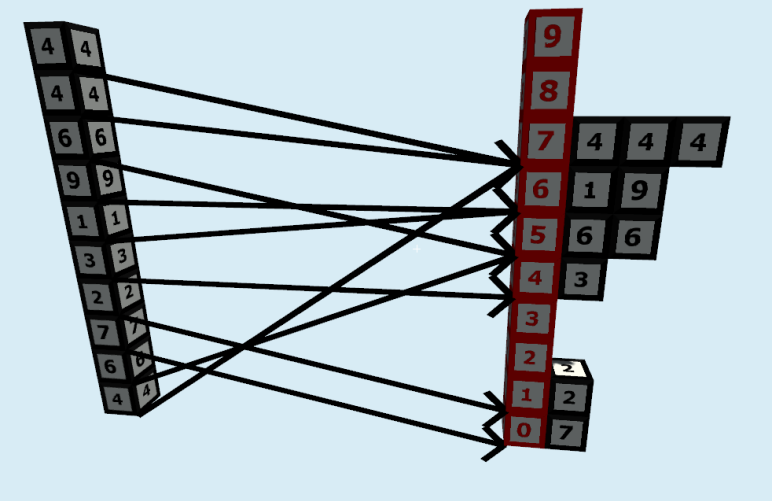

Abbildung 18: Beispiel einer Visualisierung des Hashings in der Blocklib

Das Visualisieren des Positionsindex:
Die Roten Blöcke dienen als Positionsindex. Dieses ist statisch und lässt sich analog zu der Visualisierung der Eingabewerte mit der Methode `setRedStoneIndex` generieren.

Das Visualisieren der gefüllten Hashtabelle:
Mit der, vom Studenten eingegebenen Hashfunktion, werden nun die Elemente der Methode `setInputList` berechnet und an die richtige Position neben dem Index gesetzt. Dafür ist die Methode `setStoneHashPosition` zuständig. Diese bekommt als Übergabeparameter das selbe Array, was auch der Methode `setInputList` übergeben worden ist. Zusätzlich wird noch ein Array übergeben, welche die berechneten Positionsindizes des ursprünglichen Arrays beinhaltet. Die Methode `whichBlock` wird hierbei verwendet, um die richtigen Blöcke visualisieren zu können. Dies ist notwendig, da die berechneten Indexpositionen bei jeder Visualisierung unterschiedlich sind, und somit nicht statisch definiert werden können. Da eine Indexposition mehrere Elemente aufnehmen kann, muss man vorher abfragen, ob an dieser Stelle bereits ein Element hinterlegt ist. Dies passiert mit einem Hilfsarray `helpArray`, wobei alle Arrayindizes am Anfang mit den Wert 0 definiert sind. Dabei hat jede Indexposition in der Visualisierung einen zugewiesenen Arrayindex der Variable `helpArray`. Wird nun ein Element an einer leeren Position eingegliedert, wird diese Arrayposition inkrementiert. Auf diese Weise kann man abfragen, ob und wie viel Elemente sich bereits an dieser Position befinden. Wenn sich bereits Elemente an dieser Indexposition befinden, wird das neue Element einfach mit der x-Koordinate nach rechts verschoben.

Das Visualisieren der Pfeile:
Wenn diese Struktur steht, werden noch Pfeile visualisiert. Diese Pfeile dienen zur Übersicht, wo welches Element durch die Berechnung der Hashfunktion eingesetzt wurde. Die Methode `ArrowToIndexPosition` bekommt als Übergabeparameter das Array mit den berechneten

Indexposition und sie legt statische, nicht veränderbare $x-$ und z-Koordinaten fest. Die y-Koordinaten müssen dynamisch der Berechnung angepasst werden. Die Variable `ySource` und `yDestination` setzten hierbei die Start- und Ziel Koordinaten bei dem ursprünglichen Array und bei dem roten Indexarray. Die Visualisierung der Pfeile beginnt beim ersten Element. Dementsprechend wird `ySource` nach jedem Iterationsschritt inkrementiert, um von den richtigen Startpositionen zu starten. `ySource` wird ebenfalls nach jedem Schritt inkrementiert, und dem Übergabearray übergeben. Somit werden den Zielkoordinaten die berechneten Positionsindexe des Übergabearrays übergeben und die Pfeile finden sicher ihr Ziel.

5.2.2 Suchbäume

Jeder Suchbaum hat eine Anzahl an gleicher Funktionen und Eigenschaften. Diese unterscheiden sich von den Suchalgorithmen, welche mithilfe einer Liste suchen. Darum wurde das Paket `TreeSearch` von dem Paket `ListSearch` abgespaltet. Das Interface `Tree` ist dabei die Basis jeden Suchbaumes. Es stellt alle elementaren Funktionen bereit, die jeder Baum besitzen muss. Diese werden von der Klasse `AVLTree` aufgerufen und implementiert. Folgende Methoden werden bereitgestellt:

- `insert`
- `delete`
- `print`
- `isEmpty`

Mithilfe dieses Interfaces werden Methodendeklarationen bereitgestellt, die einen Suchbaum definieren. Somit wird sichergestellt, dass jeder Suchbaum diese vier Methoden enthält. Neben der Klasse `Tree` gibt es noch eine zweite Klasse `Node`, welche von Suchbäumen verwendet wird. Die Klasse bietet eine Auswahl an `get-` und `set-`Methoden, die eine einfache Datenverarbeitung garantieren. Mit `Node` wurden u.a. diese Methoden definiert: `getData, setData, getLeftChild, setLeftChild, getRightChild, setRightChild, getHeight, setHeight`. Das UML-Diagramm in Abbildung 19 zeigt die implementierten Klassen für Suchbäume.

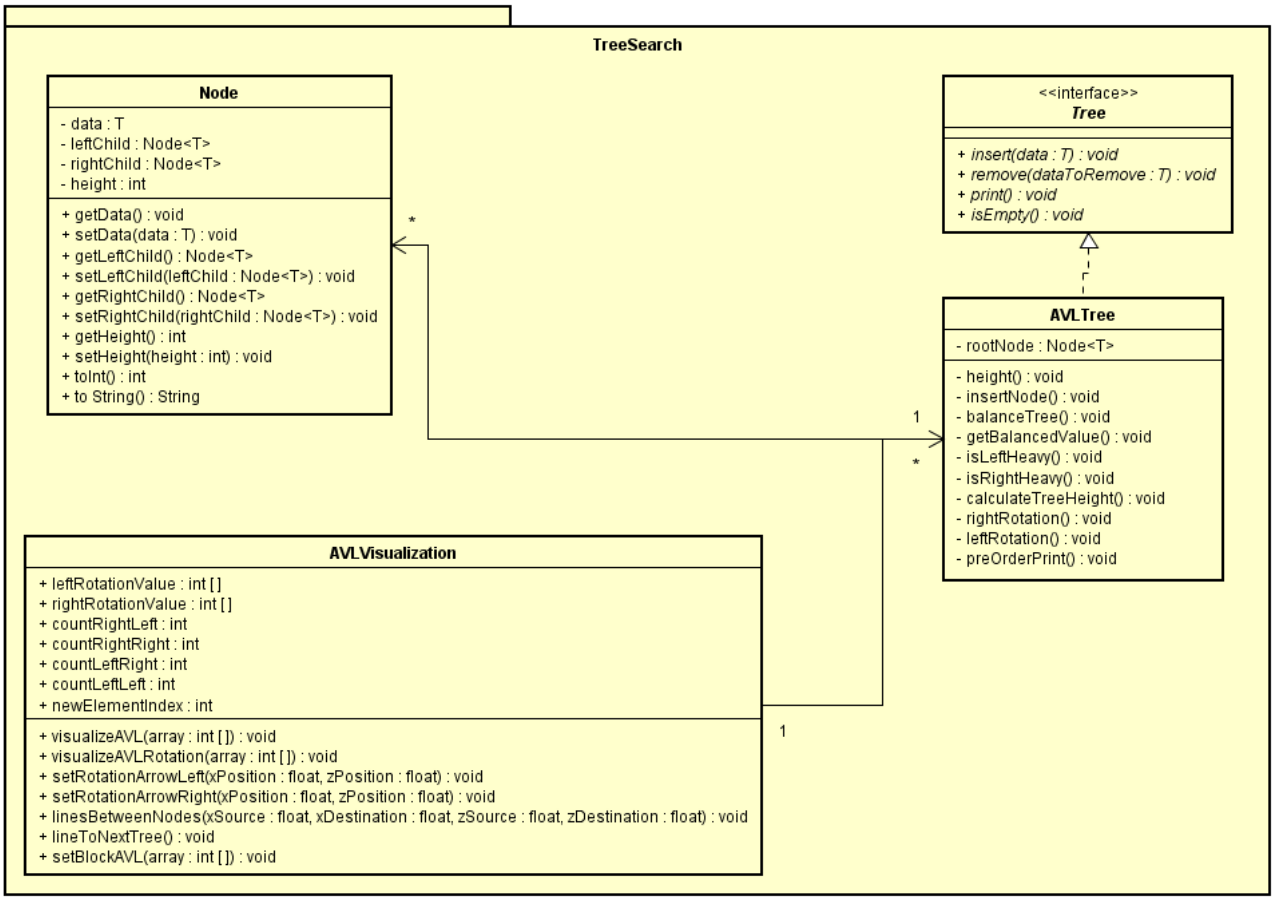

Abbildung 19: UML-Diagramm des Pakets `TreeSearch`

5.2.2.1 Das Visualisieren eines AVL-Baumes

In dieser Klasse sind alle Algorithmen-spezifischen Methoden und Deklarationen untergebracht, die essentiell für einen AVL-Baum sind. Für das Visualisieren eines AVL-Baumes werden die beiden Klassen `AVLTree` und `AVLVisualization` benötigt. Dabei haben die Methoden der `AVLTree`-Klasse die Aufgabe, komplett unabhängig von der Visualisation, den AVL-Baum zu generieren, Rotationen zu erkennen und durchzuführen und die erforderlichen AVL-Baum-Eigenschaften zu halten. Da dieses Vorgehen bereits im Kapitel Grundlagen geklärt wurde, werden die Methoden der Klasse `AVLTree` nur in ihrer Funktionalität beschrieben:

- `height:`
  Dieser Methode wird ein Knoten übergeben und mithilfe von `getHeight` der Klasse `Node` wird die Höhe berechnet.

- `insertNode:`
  Um neue Knoten in einen Suchbaum einfügen zu können, wird `insertNode` benötigt. Diese Methode prüft, an welcher Stelle sich das neue Element einfügen muss. Außerdem

wird die Methode `balanceTree` aufgerufen, um die erforderlichen Eigenschaften des AVL-Baumes zu halten. Die Höhe der neu eingefügten Knoten werden noch mit `setHeight` und `calculateTreeHeight` aktualisiert.

- `getBalanceValue`:
  Die Methode gibt einen Wert zurück, welcher in der Methode `balanceTree` benötigt wird, um zu entscheiden ob und welche Rotation erfolgen muss. Es wird die Höhe des rechten Teilbaums von der Höhe des linken Teilbaums subtrahiert.

- `isLeftHeavy:`
  Hierbei wird überprüft, ob der Baum mehr Elemente in dem linken Teilbaum aufweist, als im rechten. Dies ist der Fall, wenn `getBalanceValue` einen Wert größer als 1 zurückgibt.

- `isRightHeavy:`
  Im Gegensatz zu `isLeftHeavy`, hat der Baum mehr Elemente im rechten Teilbaum als im linken, wenn der Wert von `getBalanceValue` kleiner als -1 ist.

- `balanceTree:`
  `balanceTree` überprüft nun, ob und welche Rotation erfolgen muss. Dabei bedient sie sich bei den Methoden `getBalanceValue`, `isLeftHeavy` und `isRightHeavy`. Wird eine Rotation benötigt, wird entweder `rightRotation` oder `leftRotation` aufgerufen.

- `rightRotation:`
  Mithilfe von `rightRotation` wird nun eine Rotation nach rechts an einem bestimmten Knoten durchgeführt.

- `leftRotation:`
  Diese Methode funktioniert genauso wie `rightRotation`, nur für eine Linksrotation.

- `calculateTreeHeight:`
  Die Methode gibt die Höhe des Knotens an, der als Übergabeparameter übergeben worden ist. Diese Daten werden jeweils mit `getLeftChild` und `getRightChild` ausgegeben. Zusätzlich müssen diese Werte um eins inkrementiert werden, um die tatsächliche Höhe des Baumes zu erhalten. Dies ist notwendig, da `getLeftChild` und `getRightChild` den ersten Knoten nicht mitzählen und nur die Teilbäume berücksichtigen.

- `preOrderTraversal:`
  Um den Studenten neben der Visualisierung noch eine andere Darstellung der Knoten zu liefern, wurde `preOrderTraversal` implementiert. Diese Methode gibt zuerst den Anfangsknoten aus, gefolgt von dem linken und rechten Teilbaum.

Diese Methoden dienen als Grundlage der Visualisierung eines AVL-Baumes. In den nächsten Zeilen wird auf die Klasse `AVLVisualization` eingegangen:

Das Visualisieren der Blöcke:
Die Blöcke werden mit der Methode `setBlockAVL` visualisiert. Sie bekommt als Übergabeparameter ein Array, welches in der Benutzerschnittstelle für die Suchbäume von dem Studenten definiert wurde. Dieses Array visualisiert anschließend den Wert der gesuchten Arrayindizes mit der Funktion `setBlock` der Klasse `WorldInteraction`. Die Methode `setBlockAVL` wird von `visualizeAVL` benutzt, um die Blöcke in der Baumstruktur zu visualisieren.

Die Baumstruktur:
Die Methode `visualizeAVL` baut die Baumstruktur auf und gibt dem Benutzer noch weitere Visuelle Informationen in Form von Rotationspfeilen, welche in den Methoden `setRotationArrowLeft` und `setRotationArrowRight` generiert werden, sowie Verbindungsstücke zwischen den Knoten in Form von Linien, die mit der Methode `LinesBetweenNodes` visualisiert werden. Die Baumstruktur hat dabei immer den gleichen Aufbau. Abbildung 20 visualisiert diesen Aufbau:

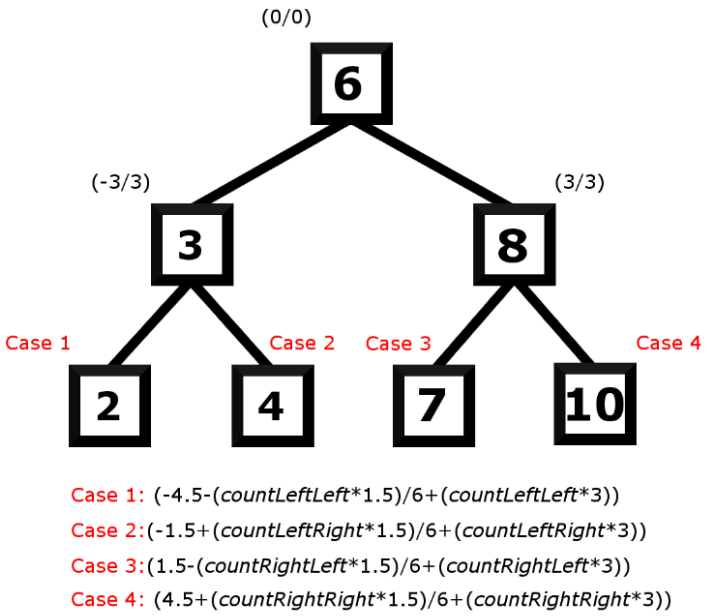

Abbildung 20: Aufbau der Baumstruktur in der Blocklib

Der Anfangsblock und die beiden ersten Blöcke des Linken- und Rechten Teilbaumes sind statisch definiert. Diese Blöcke haben immer die gleiche Position in der Baumstruktur und somit auch in der Spielwelt. Um nun eine dynamische Anzahl an Knoten in einen AVL-Baum einfügen zu können, wurden die Positionen der darauf folgenden Blöcke nicht statisch definiert. Diese suchen sich ihre Position in der Spielewelt mithilfe der global definierten Hilfsvariablen `countLeftLeft`, `countLeftRight`, `countRightRight`, und `countRightLeft`. Diese vier Variablen speichern die Anzahl der bereits eingefügten Knoten in dem jeweiligen Teilbaum. Diese Variablen werden inkrementiert, sobald ein Element nach dem statisch definierten Knotens des linken/rechten Teilbaumes eingefügt wurde. Somit hat man die Anzahl

der Knoten abgefangen, welche bereits eingefügt worden sind, um somit die richtigen Koordinaten für den neuen Block zu finden. Wurde der Knoten eingefügt, prüft die Methode `visualizeAVL` außerdem, ob an diesem Knoten eine Rotierung erfolgen muss. Dies wird mit den Arrays `leftRotationValue` und `rightRotationValue` bewerkstelligt. Diese greifen jeweils den Wert der Methoden `leftRotation` und `rightRotation` der Klasse `AVLTree` ab, an dem rotiert werden muss. Dieser Wert wird in dem ersten Arrayindex gespeichert. Ist dieser Wert dann gleich des Elements, welches gerade visualisiert wurde, wird mithilfe der Methoden `setRotationArrowLeft` und `setRotationArrowRight` ein Rotationspfeil an dieser Position eingefügt. Da es vorkommen kann, dass mehrere Links- oder Rechtsrotationen benötigt werden, würden `leftRotationValue` und `rightRotationValue` als Arrays definiert. Nach jedem Visualisierungsschritt wird eine lokale Hilfsvariable inkremenitert, um den nächsten Arrayindex zu überprüfen und mehrere Rotationen zu ermöglichen.

Die Rotationspfeile:
Falls rotiert werden muss, wird entweder die Methode `setRotationArrowLeft` für eine Linksrotation, oder die Methode `setRotationArrowRight` für eine Rechtsrotation aufgerufen. Diese bekommen als Übergabeparameter die aktuelle x- und z-Position des zu rotierenden Blockes. Daraufhin wird entweder ein Rotationspfeil, der nach links deutet visualisiert, oder ein Pfeil, der nach rechts zeigt. In Abbildung 21 ist dieser rot gekennzeichnet.

Die Verbindungslinien:
In einem Baum müssen die Knoten untereinander verbunden werden. Dies geschieht mit der Methode `linesBetweenNodes`. Diese Methode wird bei jedem Visualisieren eines Blockes aufgerufen. Als Parameter bekommt `linesBetweenNodes` vier Werte, jeweils eine x- und z- Koordinate für Start- und Zielpositionen, die zwei Blöcke verbinden. Durch die Methode `addRenderable` der Klasse `MasterRenderer`, implementiert von Tobias Werner [Wer18], können diese Linien visualisiert werden.

Der Übergangspfeil zum nächsten Iterationsschritt:
Die Methode `lineToNextTree` dient dazu, einzelne Zwischenschritte abzugrenzen. Durch das Anzeigen aller Iterationsschritte, lassen sich die Suchbäume vergleichbar halten und man kann das Ergebnis besser nachvollziehen. Wie auch schon bei der Methode `linesBetweenNodes`, wird `addRenderable` benutzt, um diesen Pfeil zu generieren. Dieser Pfeil wird ebenfalls in Abbildung 21 gezeigt.

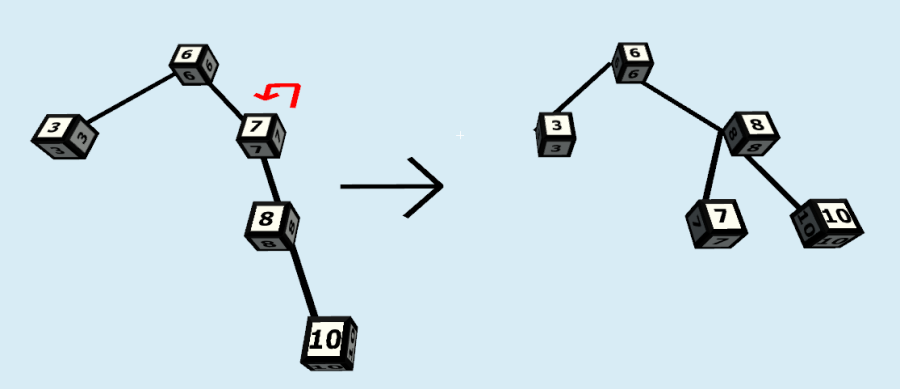

Abbildung 21: Visualisierungsbeispiel eines AVL-Baumes in der Blocklib

5.3 Das Paket `GraphAlgorithm`

Das Paket `GraphAlgorithm` enthält drei Klassen: Die Klassen `Graphs` und `Edge` besitzen Methoden und Funktionen, welche Graphalgorithmen sich teilen. Diese werden von den Algorithmen-Klassen übernommen, zu welchen die `Kruskal` Klasse gehört. Mithilfe der Funktionen in diesem Kapitel kann man jeden erdenklichen MST oder Kürzeste-Wege-Algorithmus implementieren. Zuerst wird die Klasse `Edge` erklärt, gefolgt von `Graphs`. Am Ende dieses Kapitels wird noch auf die Visualisierung des Algrotihmus von Kruskal eingegangen. Eine Übersicht des Pakets `GraphAlgorithm` ist in Abbildung 22 gezeigt.

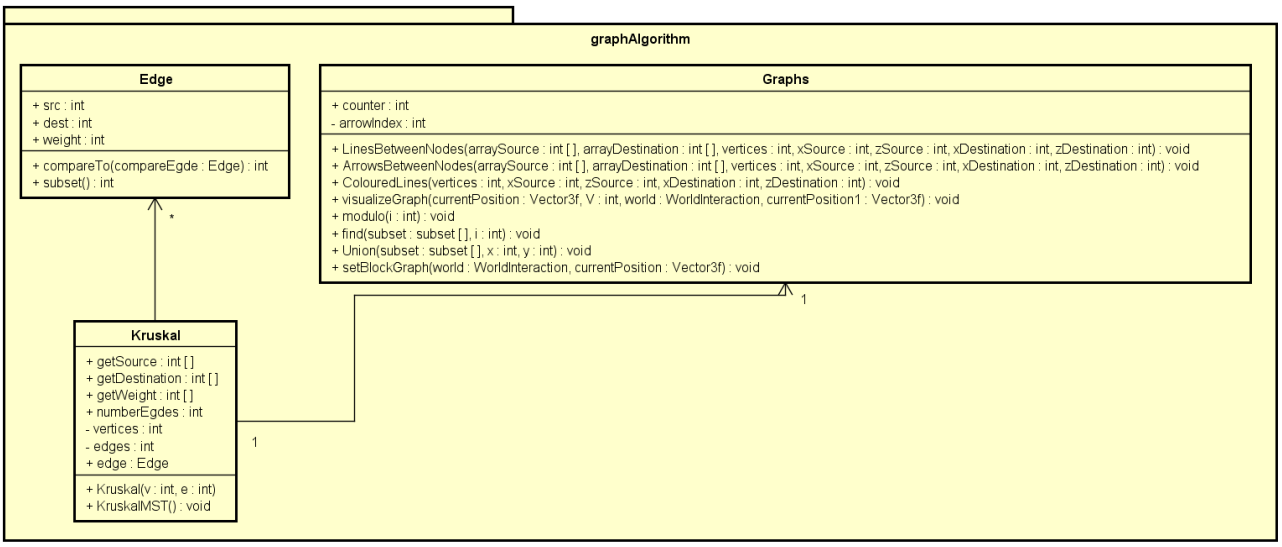

Abbildung 22: UML-Diagramm zum Paket `GraphAlgorithm`

5.3.1 Das Visualisieren des Graphs

Die Klasse `Edge`:
Die `Edge` Methode liefert den Graphalgorithmen eine einheitliche Struktur zur Deklaration von Start- und Zielwerten und den Gewichtungen der Kanten. Diese Klasse wurde mit dem Interface `Comparable` erweitert, um ein einfaches Vergleichen der Kantengewichtungen zu ermöglichen. `Edge` wird von jedem implementierten Graphalgorithmus aufgerufen um ihre Knoten- und Kanteninformationen zu verarbeiten. `Edge` enthält noch die Methode `subset`, welche als Variablen `parent` und `rank` enthält. Diese Methode ist für das Finden eines Minimalen Spannbaums wichtig und wird auch in den Methoden `find` und `Union` der Klasse `Graphs` verwendet.

Das Visualisieren der Knoten:
Um einen beliebigen Graphen darzustellen, wird die Methode `visualizeGraph` der Klasse `Graphs` benötigt. Mit dieser Arbeit wurden einige Konventionen von Visualisierungen von Graphalgorithmen in der Blocklib eingeführt. Zum einen werden als Knoten die Zahlenblöcke in aufsteigender Reihenfolge verwendet. Da Graphalgorithmen keine Werte in den Knoten speichern, und eine willkürliche Auswahl an Knotentexturen zu verwirrend wäre, wurde auf ein nachvollziehbares Konzept gesetzt. Zum anderen kann die Struktur von Graphalgorithmen sehr verschieden sein. In dieser Implementierung wurde die Visualisierung auf zwei Strukturen beschränkt. In der Abbildung 23 sieht man links einen Graphen mit einer geraden Anzahl von Knoten und der daraus resultierende Graph. Rechts ist ein Graph mit einer ungeraden Anzahl an Knoten. Die Graphen lassen sich mit bis zu zehn Knoten gleichzeitig darstellen.

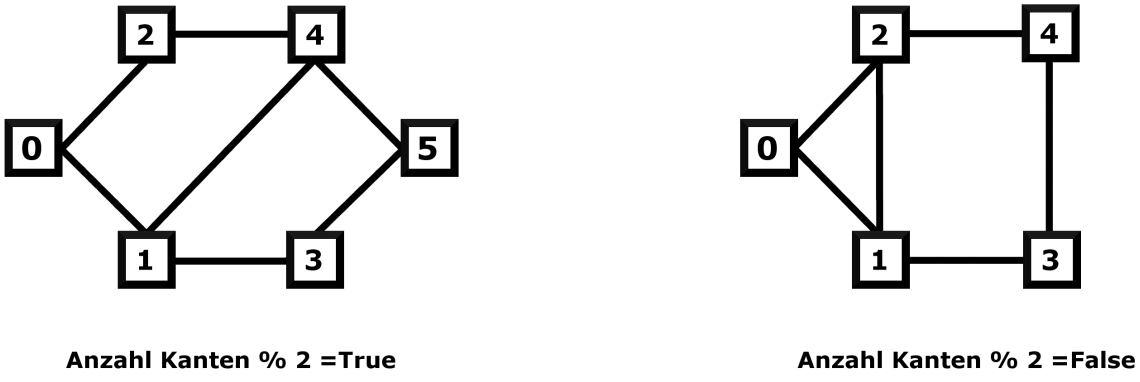

Abbildung 23: Die vorgegebenen Graph-Strukturen

Diese Einschränkung wurden getroffen, um den Studenten ein möglichst einfaches und nachvollziehbares Ergebnis zu liefern. In der Methode `visualizeGraph` werden die Knoten aufgebaut. Da die Struktur statisch ist, kann man mithilfe von verschiedenen Abfragen dieses Gitter mit vordefinierten Knotenabständen generieren. In Abbildung 24 werden diese Abstände zwischen den Knoten erläutert.

Die `currentPosition` ist bei Start an der Position (0.0/0.0). Dies markiert die linke, untere Ecke eines Knotens bzw. des Würfels. Von dort aus werden die weiteren Knoten mit der Methode `setStone` gesetzt. Mithilfe der Methode `modulo` wird entschieden, ob der Wert des Knotens gerade ist, und somit die vorherige z-Koordinate der `currentPosition` um 3 verschoben wird, oder bei ungeraden um -3. Dies ist nötig, da eine dynamische Anzahl an Knoten implementiert werden können. Mithilfe dieser Prozedur kann man beliebige Graphalgorithmen, mit den oben genannten Einschränkungen, darstellen und verarbeiten.

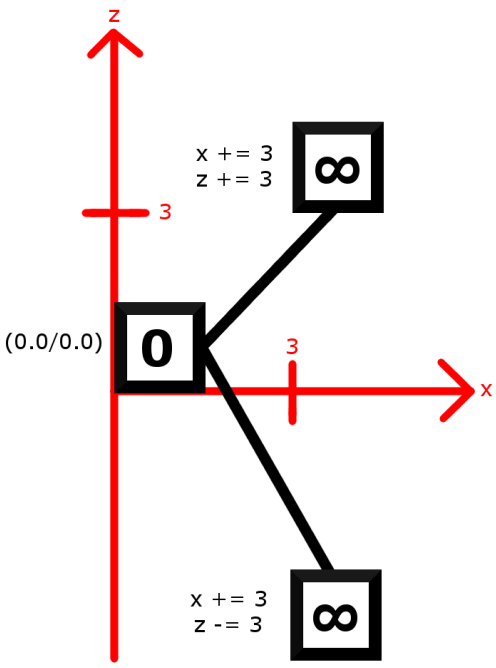

Abbildung 24: Die Berechnung der Knotenabstände

Das Setzen der Kanten:
Der Visualisierte Graph besteht nun aus Knoten. Um diese untereinander zu verbinden benötigt man Kanten. Zum einen wurde die Funktion `LinesBetweenNodes` implementiert, mit der es möglich ist, die Knoten dynamisch untereinander zu verbinden. Zum anderen wurde auch die Methode `ColouredLines` geschrieben, welche anstatt schwarze Kanten, gelbe ausgibt. Diese Kanten sollen den Minimalen Spannbaum bzw. die kürzesten Wege in der Blocklib markieren. Durch diesen Farbunterschied erkennt man klar das Vorgehen des hinterlegten Algorithmus und die Arbeitsschritte sind nachvollziehbar. Beide Methoden funktionieren homogen. Für das generieren von Kanten braucht man vier Positionsvariablen: `xSource, zSource, xDestination und zDestination`. Diese markieren jeweils den Start- und Zielpunkt. Da man ein statisches Knotennetz hat, kann man als Übergabeparameter jeweils zwei Arrays mit den

Start- und Zielknoten der Kanten übergeben. Somit ist jede erdenkliche Kantenbildung innerhalb diesen Graphens möglich. Die vier Positionsvariablen werden der Methode `addRenderable` aus der Klasse `MasterRenderer` übergeben. Abbildung 25 zeigt einen visualisierten Beispielgraphen In der Blocklib mit dem Zusammenspiel zwischen den Methoden `visualizeGraph`, `LinesBetweenNodes` und `ColouredLines`.

Weitere Methoden der Klasse `Graphs`:
Neben der Visualisierung der Knoten und Kanten bietet die Klasse `Graphs` noch andere Funktionalitäten. Die Methode `find` wird benutzt, um die Werte einer bestimmten Teilliste zu finden. Mit `Union` kann man zwei solcher Teillisten zusammenfügen.

5.3.2 Die Visualisierung des Algorithmus von Kruskal

`Kruskal` ist eine Klasse, welche Funktionalitäten von `Graphs` und `Edge` übernimmt. Der Konstruktor initialisiert die Variablen `vertices`, `edges` und `edge`. Die Methode `KruskalMST` bearbeitet den Graphen und speichert die Start-, Ziel- und Gewichtungswerte in den Variablen `getSource`, `getDestination` und `getWeight` ab. Diese Prozedur fängt mit der Sortierung der Gewichte in absteigender Reihenfolge an. Diese Liste wird nun dynamisch in Teillisten mit einem Element aufgespaltet, um jeweils die Start-, Ziel- und Gewichtungswerte abgreifen zu können. Die Kante mit der geringsten Gewichtung wird ausgesucht, vorausgesetzt diese Kante bildet mit bereits vorher ausgewählten Kanten keinen Zyklus, und mit der Funktion `Union` der Klasse `Graphs` abgespeichert. Dies wird so oft wiederholt, bis alle Kanten erreicht sind. Mithilfe der Start-, Ziel- und Gewichtungswerten kann nun `Graphs` die Knoten und Kanten visualisieren. Der Minimale Spannbaum wird dabei mithilfe der Methode `ColouredLines` der Klasse `Graphs` mit gelben Kanten versehen. Wie so ein MST aussehen kann, wird in Abbildung 25 gezeigt.

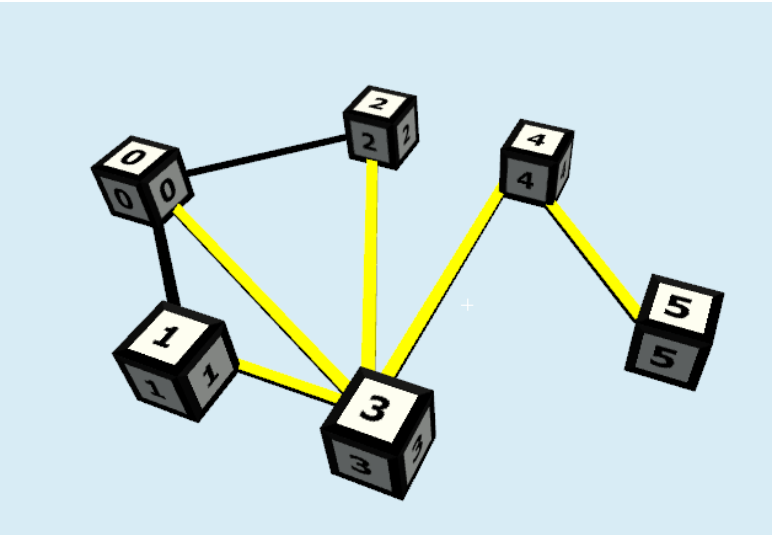

Abbildung 25: Der MST eines Graphens in der Blocklib

6 Tests

Um eine korrekte Validierung der Algorithmen zu gewährleisten, wurden einige Tests der Implementierung durchgeführt. Zum einen wären das *Modultests*, mit denen die Funktionalität der Komponenten getestet wurde. Zum anderen sind es visuelle Tests, um die optische Komponente zu beurteilen.

6.1 Modultests

Modultests überprüfen einzelne Funktionalitäten des Systems, isoliert von anderen Modulen, auf deren Korrektheit. Realisiert wurden diese Tests mit dem *JUnit Framework* [JUn]. Die Unit-Tests wurden für die Validation des Benutzercodes durchgeführt. So wurde z.B. getestet, ob eine andere Implementierung des gleichen Algorithmus sicher validiert werden kann. Dies ist wichtig, damit sich der Benutzer nicht an etliche Konventionen und Definitionen innerhalb der Implementierung der Algorithmen halten muss. Im Codeabschnitt 5 ist dieses Beispiel dargestellt. Dabei kann man mithilfe der Methode `assertArrayEquals` aus dem Framework `JUnit` zwei Arrays vergleichen. Es wird jeweils das Array nach dem Durchlauf einer Benutzerdefinierten Implementation des Insertionsort-Algrotihmus mit dem Ergebnis des Durchlaufs der `patternInsertionSort` Methode verglichen.

```
1     @Test
2     public void test() {
3        int [] array = {3,4,2,7,4,2,1,6,7};
4        int [] studentCodeResult = studentsExampleCode(array);
5        int [] patternCodeResult = InsertionSort.patternInsertionSort(array);
6        assertArrayEquals(studentCodeResult, patternCodeResult);
7     }
8
9     public int[] studentsExampleCode(int liste[]) {
10    int i, key, j;
11       for (i = 1; i < liste.length; i++)
12       {
13           key = liste[i];
14           j = i-1;
15              while (j >= 0 && liste[j] > key)
16              {
17                liste[j+1] = liste[j];
18                  j = j-1;
19              }
20              liste[j+1] = key;
21       }
22    return liste;
23    }
```

Listing 5: Unit-Test für das Validieren der Gleichheit von Arrays

6.2 Visuelle Tests

Durch Modultests lässt sich die Visualisierung von Lehralgorithmen nicht komplett testen. Da diese Funktion vom optischen Geschehen auf dem Bildschirm lebt, muss manuell überprüft werden, ob die Visualisierung richtig angezeigt wird.

- **Sortieralgorithmen**
  Bei den Sortieralgorithmen ist es wichtig, dass die Blöcke nach den Zwischenschritten immer an den selben Position gesetzt werden. Ist dies nicht gegeben, entsteht eine unnötige Komplexität und Verwirrung.
- **Suchbäume und Graphalgorithmen** Die Struktur der Blöcke muss zu jederzeit gehalten werden. Von dieser Struktur hängt nicht nur die Logik hinter den Algorithmen ab, sondern z.B. auch das Verbinden dieser Blöcke mit Linien oder Pfeilen.

7 Anwendung der Visualisierung und Validierung von Algorithmen

In diesem Kapitel geht es um die Sicht des Benutzers auf die Kompenenten zur Visualisierung und der automatischen Überprüfung von Algorithmen aus der Lehre. Es wird erläutert, wie eine Benutzerschnittstelle aufgebaut ist und welche Funktionalitäten sie bietet. Dies wird durch einen Codeausschnitt der Benutzerschnittstelle des Insertionsorts unterstützt. Am Ende dieses Kapitels wird noch erklärt, wie der Benutzer nun diese Schnittstellen anwenden können.

7.1 Aufbau der Benutzerschnittstellen

Die Benutzerschnittstellen sind die Klassen, die von einem Studenten ausgeführt und bearbeitet werden. Sie sind so abgestimmt, dass unnötige Komplexität dem Benutzer verborgen bleibt und eine einfache Bedingung gewährleistet ist. Diese Klassen beinhalten nur die nötigsten Methoden und Variablen. Man findet sie in dem Package `de/oth/programs/algorithmVisualization`. Einen Überblick über so ein *User Interface (UI)* liefert das Listing 6. Dieses Listing ist ein Ausschnitt der Klasse `InsertionSample`.

```
1  /**
2       * Main function.
3       * Validates the code.
4       * Starts the game.
5       */
6      public static void main(String[] args) {
7
8         AlgorithmValidation.compareArray(insertionSort(), InsertionSort.patternInsertionSort(
               unsortedList));
9         AlgorithmValidation.checkEmpty(unsortedList);
10        Insertion_Sample start = new Insertion_Sample();
11        Insertion_Sample.game = new Game(start);
12         AlgorithmValidation.GameStart(game);
13     }
14
15     /**
16      * Student s code
17      * Used for the validation.
18      * Student needs to define his own code in this method.
19      */
20     public static int[] insertionSort() {
21        int temp = 0;
22        int position = 0;
23           for(int i =1; i < liste.length;i++) {
24              position = i;
25              while ((position>1)&&(liste[position]>liste[position])) {
26                 temp = liste[position-1];
27                 liste[position-1] = liste[position];
28                 liste[position] = temp;
29              }
30           }
31           return liste;
32     }
33
```

```
34      /**
35       * KeyPressed Method.
36       * Triggers an Event in the Game.
37       * Key U: Visualize the unsorted Array and the arrow.
38       * Key J: Visualize every step to the end of the Algorithm.
39       * @param key Triggers an event on the screen.
40       */
41      @Override
42      public void keyPressed(int key) {
43
44            if(key == GLFW_KEY_U){
45               Insertion_Sample.currentPositionArrow.x += 2;
46               Insertion_Sample.currentPositionArrow.y += 2;
47               SortManager.setArrow(world, currentPositionArrow);
48               Insertion_Sample.currentPositionArrow.x -= 2;
49               Insertion_Sample.currentPositionArrow.y -= 2;
50               SortManager.setArray(unsortedList, world, currentPosition);
51            }
52            if(key == GLFW_KEY_J) {
53               InsertionSort.visualizeInsertionSort();
54            }
55      }
56
57      /**
58       * creates the Game
59       */
60      @Override
61      public void onGameStart() {
62         game.createEmptyWorld();
63         world = game.getWorld();
64         Context.getInstance().getCamera().setPosition(new Vector3f(1, 1, 1));
65      }
```

Listing 6: Aufbau der Benutzerschnittstelle anhand des Beispiels des Insertionsorts

Im Prinzip besteht jedes UI aus sechs Methoden. In der `main` Methode wird der dynamisch eingegebene Code des Benutzers validiert. Am Beispiel des Insertionsorts sind es nur die Abragen `compareArray` und `checkEmpty`(siehe Kapitel 3.1.1). Algorithmen-spezifische Validationen wie z.B. das Überprüfen der Rotationen der AVL-Bäume werden in diesem Block ebenfalls mit aufgenommen. Gleichzeitig wird in der `main`-Methode noch das Spiel initiiert und gestartet mithilfe der Methode `GameStart` aus der Klasse `AlgorithmValidation` (siehe ebenfalls Kapitel 3.1.1). Neben der `main`-Methode muss die Benutzerschnittstelle eine bereits vordeklarierte Methode liefern, die der Student mit seinem Algorithmencode erweitert, ergänzt und bearbeitet. Im Beispiel ist das die Funktion `insertionSort`. Diese Methode enthält den, vom Benutzer eingegeben Code, der in der `main`-Funktion validiert wird. Außer der Methodendefinition wird dem Studenten bei der Eingabe des Codes jegliche Freiheit gelassen. Dies ist auch die einzige Methode, welche vom Benutzer bearbeitet werden muss. Da jedes UI von der Klasse `BlockLibProgram` erbt, können diese Methoden in dem jeweiligen UI überschrieben werden. Von diesen Vererbungen sind für unsere Schnittstellen nur zwei Methoden wichtig: Zum einen wäre es die Funktionalität, eine bestimmte Aktion im Spiel bei einem bestimmten Tastendruck vorzunehmen. Dies wird realisiert mit der Methode `keyPressed`. Sie bekommt als Übergabeparameter einen String von

der Taste, welche am Eingabemedium gedrückt wurde. Dies passiert mit der Unterstützung der Lightweight Java Game Library [LWJGL]. Jedes UI hat dabei zwei unterschiedliche Tastenbelegungen. Mit der Taste *U* lässt sich das Anfangsszenario der Algorithmen visualisieren. Mit der Taste *J* hingegen werden die Zwischenschritte, bis hin zum Endergebnis visualisiert. Dies ist für alle Algorithmen und Benutzerschnittstellen gleich. Die letzte, relevante Methode ist die `onGameStart` Methode. Diese muss zwingend immer gleich deklariert werden, da sonst keine Daten in das Spiel geladen werden können.

7.2 Anwendung der Benutzerschnittstellen

Ein Ziel dieser Bachelorarbeit war, dem Benutzer eine möglichst einfache und sinnvolle Benutzung der implementierten Benutzerschnittstellen zu liefern. Um dies zu erreichen, wurde auf jegliche Komplexität innerhalb der Benutzerschnittstellen verzichtet. Die Klassen wurden so modular gehalten, wie nur möglich. Durch Kommentare wird dem Studenten erklärt, was er zu tun hat, um seinen Algorithmus visualisieren zu lassen. Schließlich geht es ja nicht darum, dass der Student die Ausarbeitung dieser Arbeit komplett versteht. Er soll ohne großen Aufwand seine Implementierung wie gewohnt vornehmen und ein gutes Ergebnis erhalten. Aus diesem Grund wurden das UI so gehalten, dass der Benutzer nur die jeweilige Methode bearbeiten muss, in der er seinen eigenen Code eingibt. Im Falle des Insertionsorts wäre dies die Methode `insertionSort`. Nachdem diese Methode bearbeitet wurde, wird der neue Code validiert. Entweder, der eingegebene Benutzercode entspricht den Erwartungen des jeweiligen Algorithmus und er wird visualisiert, oder die modifizierte Benutzermethode ist fehlerhaft. Bei letzterem wird der Algorithmus nicht visualisiert und es wird eine Fehlermeldung ausgegeben. In folgenden Listing Beispielen 7 und 8 wird jeweils ein zulässiger Code vorgestellt, der sich von dem hinterlegten Mustercode unterscheidet. Er terminiert korrekt trotz differierter Implementation. Außerdem wird ein Code gezeigt, der nicht zulässig ist, da er Fehler in den Schleifendeklarationen aufweist und somit nicht korrekt sortiert.

```
1  public static int[] insertionSort() {
2        int i, key, j;
3           for (i = 1; i < liste.length; i++)
4           {
5               key = liste[i];
6               j = i-1;
7                  while (j >= 0 && liste[j] > key)
8                  {
9                    liste[j+1] = liste[j];
10                     j = j-1;
11                 }
12                 liste[j+1] = key;
13          }
14       return liste;
15    }
```

Listing 7: Erfolgreiche Validierung des Codes

```
1     public static int[] insertionSort() {
2        int temp = 0;
3        int position = 0;
4           for(int i =1; i < liste.length;i++) {
5              position = i;
6              while ((position>1)&&(liste[position]>liste[position])) {
7                 temp = liste[position-1];
8                 liste[position-1] = liste[position];
9                 liste[position] = temp;
10             }
11          }
12          return liste;
13    }
```

Listing 8: Fehlgeschlagene Validierung des Codes

7.2.1 Spezialfall: Anwendung des Hashings

Die Anwendung des Hashings unterscheidet sich von dem Rest der implementierten Algorithmen. Bei dem Hashing muss der Benutzer keinen Code eingeben, sondern muss nur eine Hashfunktion definieren. Diese Entscheidung wurde getroffen, weil das Hashing in Java bereits mit der Klasse `HashMap` aus dem Paket `java.util` [jav] bereitgestellt wird. Außerdem ist das Einfügen der Elemente an den richtigen Positionsindizes die größere Herausforderungen für Studenten, als die Implementierung einer Hashtabelle. Durch dynamische Anpassungsmöglichkeiten, der einzufügenden Werte und der Hashfunktion, ist dem Benutzer jegliche Freiheit gelassen, dass Konzept des Hashings komplett zu verstehen. Einen Ausschnitt der Benutzerschnittstelle `HashSample` ist in Listing 9 zu sehen. Dabei wird die Hashfunktion in der Methode `Hashing` definiert.

```
1  public static void main(String[] args)
2         {
3
4            Hashing();
5             Hash_Sample start = new Hash_Sample();
6           Hash_Sample.game = new Game(start);
7           AlgorithmValidation.GameStart(game);
8         }
9
10        /**
11        * Students place
12        * HashFunction need to be defined
13        * Example: functionList[i] = ((3*HashArray[i] + 3)%8);
14        */
15        public static void Hashing() {
16          for (int i = 0; i < HashArray.length; i++)
17             functionList[i] = ((3*HashArray[i] + 3)%8);
18        }
```

Listing 9: Ausschnitt der Benutzerschnittstelle `HashSample`

8 Fazit

Diese Bachelorarbeit hatte das Ziel, den Studenten eine Komponente zu liefern, mit der sie ohne Vorwissen bzgl. der Blocklib, Algorithmen implementieren und testen können. Frühere Arbeiten in diesem Framework hatten es zur Aufgabe, den Studenten die objektorientierte Programmierung (OOP) näherzubringen. Mithilfe dieser Komponente ist der erste Schritt getan, dies auch für das oft nachfolgende Lehrmodul nach der OOP im Lehrplan, den Algorithmen und Datenstrukturen, zu ermöglichen.

Die Blocklib wurde um mehrere neue Komponenten erweitert, um verschiedene Algorithmen-Typen zu visualisieren. Mit den Insertionsort-Algrorithmus und dem Quicksort Algrotihmus hat man zwei Sortierverfahren, die eine Folge von Elementen jeweils unterschiedlich sortieren. Dabei geht der Insertionsort-Algrotihmus iterativ durch Vergleich der Werte untereinander vor. Die Visualisierung wird als eine Folge von Blöcken dargestellt und sortiert. Der Quicksort-Algorithmus teilt seine Folge von Elementen rekursiv in kleine Teillisten auf, um das Problem, die Folge zu sortieren, in mehrere kleine Probleme aufzuspalten. Daraufhin werden diese kleinen Teillisten wieder zu einer gesamten, sortierten Folge rekonstruiert. Er besteht aus mehreren Zwischenschritten, da gegeben falls die Aufteilung der großen Folge sehr viele Schritte benötigt.

Suchalgorithmen wurden ebenfalls implementiert. Eine Grundlage zur Visualisierung von Suchlisten und Suchbäumen wurde geschaffen. Zu diesen Komponenten wurde jeweils ein Algorithmus eingeführt. Das Hashing funktioniert auf der Basis einer Suchliste, um eine eindeutige Identifikation von Elementenindizes in einer Liste zu erschaffen. Diese Indizes werden über eine Hashfunktion berechnet. Mithilfe der Blocklib kann man das Zuweisen von Elementen mit einer Hashfunktion zu einer Listenposition visualisieren. Als Beispielimplementation für Suchbäume, wurde der AVL-Baum realisiert. Dieser Baum hat, neben den Eigenschaften eines BST, noch die Eigenschaft, dass sich die Höhen der Teilknoten um jeweils 1 unterscheiden dürfen. Da es beim Einfügen und Löschen von Elementen immer zu verschiedenen Baumstrukturen kommt, und somit ein Höhenunterschied größer als 1 wahrscheinlich ist, sind Rotation zur Wiederherstellung der AVL-Eigenschaft notwendig. Diese Rotationen wurden in der Blocklib implementiert und stehen zur Visualisierung bereit.

Als letzter Algorithmen-Typ wurde eine Komponente zur Visualisierung von Graphalgorithmen entwickelt. Diese Graphen bestehen aus Knoten und Kanten und weisen verschiedene Eigenschaften auf. Es wurde ein Algorithmus zum Finden eines Minimalen Spannbaumes implementiert. Der Algorithmus von Kruskal spannt aus einem Graphen einen MST auf. Da die Kantengewichte bekannt sind, sucht sich dieser Algorithmus die am wenigsten gewichtete Kante aus, um sie dem MST hinzuzufügen. Wichtig dabei ist, das keine Zyklen entstehen und ale Knoten besucht werden. In Der Blocklib wurde dieser MST durch eine gelbe Färbung der Kanten realisiert.

8.1 Erweiterungsmöglichkeiten und Ausblick

Diese Arbeit lässt sich um einige, sinnvolle Erweiterungen ausbauen. So könnte man die Auswahl der zu implementierenden Algorithmen vergrößern. Zu jedem der drei Algorithmentypen wurden spezielle Klassen und Methoden entwickelt, um eine Erweiterung um neue Algorithmen zu ermöglichen. Durch diesen potentiellen Wachstum der Bibliothek um weitere visualisierbare Algorithmen, kann man diese Komponente zu einer Übungsplattform für Studenten aufbauen.

Die Visualisierung ist, für den Stand dieser Arbeit, in Ordnung und Zweckmäßig. Diese kann aber in Form kommender Abschlussarbeiten verbessert werden, indem die Visualisierung dynamischer gestaltet wird. Mit einem Menü, mit welchen man z.B. Werte in einen AVL-Baum Einfügen und Löschen kann, könnte eine noch einfachere Benutzung gewährleistet werden.

Die Visualisierung eines Quicksorts zu dem jetzigen Zeitpunkt ist nachvollziehbar. Nichtsdestotrotz ist die optische Komponente verbesserbar. So könnten die Teillisten visuell unterteilt werden, indem man mehr Zahlenblöcke mit verschiedenen Farben einführt. Durch diese Unterscheidung kann sich der Student noch besser die Berechnung der Sortierung vorstellen.

Graphalgorithmen können nur zwei statische Knotenstrukturen in dieser Ausarbeitung annehmen. Für die Lehre und das Verständnis des Algorithmus ist dies ausreichend. Eine dynamische Anpassung des Gitters ermöglicht trotzdem eine noch dynamischere Visualisierungskomponente. Hierbei könnte man, ähnlich wie bei den AVL-Bäumen, ein Menü zur Hilfe ziehen. Desweiteren werden in der Visualisierung noch kene Kantengewichte angezeigt, diese sind jedoch in der Konsolenausgabe vorhanden.

Es wäre wünschenswert, diese Komponente zur Visualisierung und automatischen Überprüfung von Algorithmen durch kommende Abschluss- und Projektarbeiten zu erweitern und zu verbessern. Auf diese Weise könnte die vorliegende Ausarbeitung der Grundstein für eine Übungsplattform sein und in der Lehre eine hilfreiche Unterstützung für Studenten darstellen.

Anhang

Inhalt des beigefügten Datenträgers:

- Digitale Version dieser Bachelorarbeit im .pdf Format
- Kompilierbare LaTeX Dateien
- Verwendete UML-Diagramme im .asta Format
- Source-Code des aktuellen Projekts
- Source-Code-Dokumentation mit Javadoc
- Videos der Visualisationen der implementierten Lehralgorithmen

A Literaturverzeichnis

[Bee17] Daniel Beer. “Prozedurale Weltengenerierung und Inhaltsverwaltung für ein bestehendes 3D-Framework”. Bachelorarbeit. Ostbayerische Technische Hochschule Regensburg, 2017.

[Cor+13] Thomas H. Cormen u. a. *Algorithmen - Eine Einführung*. Oldenbourg Verlag, 2013.

[Mül17] Tobias Müller. “Dynamic changes of an environment through natural disasters in an existing 3d framework”. Masterarbeit. Ostbayerische Technische Hochschule Regensburg, 2017.

[Sed14] Robert Sedgewick. *Algorithmen*. Pearson, 2014.

[Wer18] Tobias Werner. “Evaluierung und Umsetzung moderner Rendertechniken in einem voxelbasierten 3D-Framework”. Masterarbeit. Ostbayerische Technische Hochschule Regensburg, 2018.

[Zin16] Tobias Zink. “Erstellung einer würfelbasierten 3D-Grafikbibliothek mit Java & OpenGL.” Bachelorarbeit. Ostbayerische Technische Hochschule Regensburg, 2016.

B Quellenverzeichnis

[jav] Package java.util. URL: https://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html.

[JUn] JUnit. URL: https://junit.org/junit5/.

[Lib] Lightweight Java Game Library. URL: https://www.lwjgl.org/.

[Min] Minecraft. URL: https://minecraft.net/de-de/.

C Quellcodeverzeichnis

Erklärung zur Bachelorarbeit

1. Mir ist bekannt, dass dieses Exemplar der Bachelorarbeit als Prüfungsleistung in das Eigentum der Ostbayerischen Technischen Hochschule Regensburg übergeht.
2. Ich erkläre hiermit, dass ich diese Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Regensburg, den 22. August 2018

André Helgert