UNIKASSEL
VERSITÄT

Parallele Räumliche 3D Datenstrukturen für Nachbarschafts-Abfragen

Abschlussarbeit zum Diplom II
an der Universität Kassel

Offiziel abgegeben: 30.1.2008

Vorgelegt von
Alexander Wirz

Betreuer:
Dipl.-Inf. Björn Knafla
Prof. Dr. Claudia Leopold
Prof. Dr. Gerd Stumme
Universität Kassel
Fachbereich 16 - Elektrotechnik/Informatik
Fachgebiet Programmiersprachen/-methodik
Wilhelmshöher Allee 73
34121 Kassel

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Kassel, den 30.01.2008

Alexander Wirz

Inhaltsverzeichnis

Abbildungsverzeichnis iii

Tabellenverzeichnis iv

1 Einführung 1
- 1.1 Überblick . . . 1
- 1.2 Struktur der Arbeit . . . 1

2 Grundlagen 2
- 2.1 OpenSteer . . . 2
- 2.2 Paralleles Rechnen . . . 3
  - 2.2.1 OpenMP . . . 4

3 Einsatz räumlicher Datenstrukturen in OpenSteer 5
- 3.1 Motivation für den Einsatz räumlicher Datenstrukturen . . . 5
- 3.2 Aufbau einer OpenSteer Simulation . . . 6
- 3.3 Schnittstelle der räumlichen Datenstrukturen . . . 6
- 3.4 Beispiel für die Verwendung einer räumlichen Datenstruktur . . . 8

4 Räumliche Datenstrukturen 10
- 4.1 k-d-Bäume . . . 11
  - 4.1.1 Prinzip . . . 11
  - 4.1.2 Einfügen neuer Daten . . . 12
  - 4.1.3 Löschen von Knoten . . . 13
  - 4.1.4 Nachbarschaftssuche . . . 15
  - 4.1.5 Updates . . . 17
- 4.2 Grid . . . 17
  - 4.2.1 Prinzip . . . 17
  - 4.2.2 Hashing . . . 18
  - 4.2.3 Modulo-Grid . . . 20
  - 4.2.4 Nachbarschaftssuche . . . 21
  - 4.2.5 Updates . . . 23
- 4.3 Cell array with binary search . . . 24
  - 4.3.1 Prinzip . . . 24
  - 4.3.2 Einfügen neuer Daten . . . 26
  - 4.3.3 Löschen von Daten . . . 26
  - 4.3.4 Nachbarschaftssuche . . . 26
  - 4.3.5 Updates . . . 28
- 4.4 Vergleich . . . 28

5 Implementierung 30
- 5.1 k-d-Baum . . . 30

5.1.1 kd_tree_node . . . 30
5.1.2 kd_tree . . . 31
5.2 Grid-Varianten . . . 32
5.2.1 Konfiguration der Grids . . . 33
5.2.2 Buckets . . . 33
5.2.3 Updates . . . 34
5.2.4 Nachbarschaftssuche . . . 35

6 Experimente **38**
6.1 Zeitmessungen . . . 38
6.1.1 Vorgehensweise . . . 38
6.1.2 Plug-ins . . . 38
6.2 Ergebnisse . . . 40
6.2.1 Nachbarschaftssuche . . . 40
6.2.2 Updatephase . . . 45
6.2.3 Einfluss der räumlichen Datenstrukturen auf die Gesamtperformance der Simulation . . . 46
6.3 Vergleich der Datenstrukturen . . . 48

7 Zusammenfassung **50**

8 Ausblick **51**

Literaturverzeichnis **52**

Abbildungsverzeichnis

2.1 Beispiele für Steuerungsverhalten (Quelle: http://www.steeringbehaviors.de). . 2
2.2 Das Pedestrian Plug-in der OpenSteerDemo Applikation. 3

4.1 Raumaufteilung durch einen k-d-Baum. 12
4.2 Einfügen eines Knotens in einen k-d-Baum. 13
4.3 Löschen aus einem k-d-Baum. 14
4.4 Suche in einem k-d-Baum. 16
4.5 Ein zweidimensionales Grid, der mit Hilfe von einer Hashfunktion auf ein Array mit sechs Buckets abgebildet wird. 18
4.6 Ein zweidimensionales Grid und acht Buckets, auf die die Gridzellen im rot-umrundeten Bereich direkt abgebildet werden. 21
4.7 Suche in einem zweidimensionalen Grid. 22
4.8 Cell array with binary search. 25
4.9 Suche in einem zweidimensionalen cell array with binary search. 27

5.1 Klassendiagramm der Klasse `kd_tree_node`. 30
5.2 Klassendiagramm der Klasse `kd_tree`. 31

6.1 Der Pfad, auf sich die Agenten bewegen. 39
6.2 Zusammenhang zwischen der Kantenlänge einer Zelle und der Performance der Nachbarschaftssuche. 44
6.3 Anteil der einzelnen Phasen an der Gesamtlaufzeit des Boids-Plug-ins. 46
6.4 Framerate des Pedestrian-Plug-ins (1000 Agenten). 47
6.5 Framerate des Boids-Plug-ins (1000 Agenten). 47

Tabellenverzeichnis

4.1 Laufzeitaufwand für die Suche nach einem bestimmten Agenten, das Einfügen eines neuen Agenten und das Löschen eines Agenten aus der Datenstruktur. . . 28
4.2 Laufzeitaufwand für den Aufbau einer Datenstruktur, die Such- und die Updatephase. 29

6.1 Laufzeit der Suchphase im Pedestrian-Plug-in mit 1000 Agenten (in Sekunden). 40
6.2 Laufzeit der Suchphase im Boids-Plug-in mit 1000 Agenten (in Sekunden). . . . 40
6.3 Konfiguration der Grid-Datenstrukturen für die Laufzeitmessungen. 41
6.4 Anzahl der Agenten, die durchschnittlich für die Bestimmung der Nachbarn eines Agenten überprüft werden (Simulation mit insgesamt 1000 Agenten). . . . 42
6.5 Optimale Werte für das Verhältnis von der Kantenlänge einer Zelle und dem Durchmesser der Suchregion. 44
6.6 Speedups, die die unterschiedlichen Datenstrukturen bei der Nachbarschaftssuche mit vier Threads erreichen. 45
6.7 Laufzeit der Updatephase im Boids-Plug-in (Sekunden). 45
6.8 Laufzeit der Updatephase im Pedestrian-Plug-in (Sekunden). 45
6.9 Maximale Anzahl der Agenten, die mit einer Framerate von ca. 30 fps simuliert werden können. 48

1 Einführung

1.1 Überblick

Ziel dieser Diplomarbeit war die Implementierung von räumlichen Datenstrukturen (spatial data structures) zur Organisation von Punkten in einem dreidimensionalen Raum. Die implementierten Datenstrukturen unterstützen parallele Nachbarschaftsabfragen und parallele Modifikationen von Einträgen. Bei der Implementierung lag der Schwerpunkt auf der Performance der Nachbarschaftsabfragen. Um die Performance der einzelnen Datenstrukturen zu testen, wurden diese in eine parallelisierte Version von OpenSteer eingebaut.

Bei OpenSteer handelt es sich um eine Bibliothek, die den Programmierer bei der Entwicklung von Steuerungsverhalten autonomer Agenten unterstützt. Für die Realisierung der Steuerungsverhalten werden Informationen über die Nachbarschaften eines Agenten benötigt. Durch den Einsatz von räumlichen Datenstrukturen wird die Ermittlung der Nachbarn eines Agenten (Nachbarschaftsabfragen) beschleunigt.

Die Performance der verschiedenen Datenstrukturen wurde asymptotisch und experimentell bestimmt und miteinander verglichen. Außerdem wurde geprüft, wie gut sich die Datenstrukturen für parallele Nachbarschaftsabfragen und Modifikationen von Einträgen eignen.

1.2 Struktur der Arbeit

In Kapitel 2 werden Grundbegriffe geklärt, die für das Verständnis der Arbeit hilfreich sind. Dieses Kapitel enthält eine kurze Beschreibung der OpenSteer-Bibliothek und der Techniken, die ich während der Implementierung der räumlichen Datenstrukturen verwendet habe. Desweiteren werden in diesem Kapitel Grundbegriffe der Parallelverarbeitung vorgestellt.

Die Gründe für die Verwendung von räumlichen Datenstrukturen in OpenSteer werden in Kapitel 3 erläutert. In diesem Kapitel wird außerdem beschrieben, wie die implementierten Datenstrukturen in die Bibliothek eingebunden wurden.

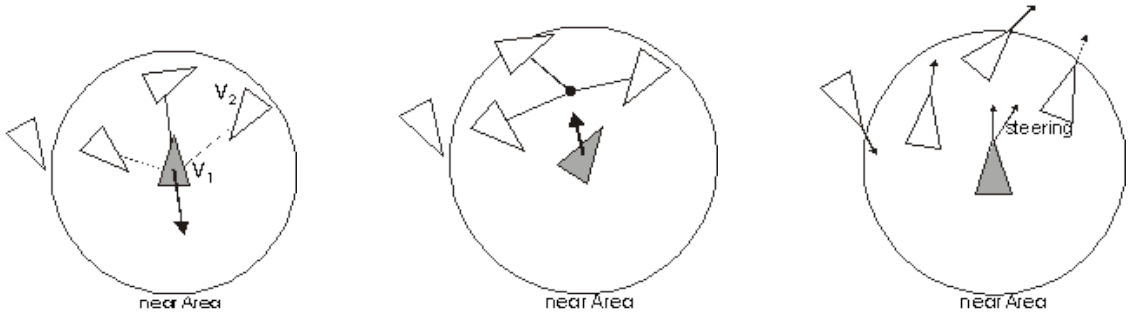
Die in dieser Arbeit untersuchten Datenstrukturen werden in Kapitel 4 detailliert beschrieben. Kapitel 5 enthält einige Details über die Implementierung dieser Datenstrukturen.

In Kapitel 6 wird die Experimentierumgebung, in der die Datenstrukturen getestet wurden, beschrieben sowie die Ergebnisse der Experimente präsentiert und analysiert. Kapitel 7 enthält eine Zusammenfassung der Ergebnisse dieser Arbeit und Kapitel 8 schließt die Arbeit mit einem Ausblick.

2 Grundlagen

2.1 OpenSteer

OpenSteer ist eine C++ Bibliothek, die den Programmierer bei der Entwicklung von *Steuerungsverhalten* (englisch: *steering behaviours*) von autonomen Agenten in Computerspielen unterstützt [10]. Der Begriff Agent bezieht sich dabei auf bestimmte Objekte der Spielewelt, wie z. B. Menschen, Tiere oder Fahrzeuge. Diese werden nicht vom Spieler, sondern durch die KI des Spiels gesteuert. OpenSteer stellt dem Entwickler mehrere Steuerungsverhalten zur Verfügung, die miteinander kombiniert werden können. Mittels dieser Steuerungsverhalten lassen sich Bewegungen erzeugen, die auf den Spieler als natürlich und intelligent wirken.



(a) Separaion (b) Cohesion (c) Alignment

Abbildung 2.1: Beispiele für Steuerungsverhalten (Quelle: http://www.steeringbehaviors.de).

Die Abbildung 2.1 demonstriert drei Beispiele für Steuerungsverhalten: Separation, Cohesion und Alignment. Das Separation Verhalten bewirkt, dass ein Agent immer einen gewissen Abstand zu anderen Agenten in seiner Nähe hält. Es wird verwendet um Kollisionen zwischen den Agenten zu vermeiden [13]. Das Cohesion Verhalten bewirkt dagegen, dass mehrere Agenten sich zu einer Gruppe zusammenschließen. Dabei wird für jeden Agenten die durchschnittliche Position seiner Nachbarn berechnet und der Agent auf diese Position hingesteuert [13]. Das Alignment Verhalten wird verwendet, um die Bewegungsrichtung und die Geschwindigkeit eines Agenten an die Bewegung seiner Nachbarn anzupassen. Dieses Steuerungsverhalten bewirkt, dass eine Gruppe von Agenten scheinbar als eine Einheit agiert [13]. Die Bewegung der Gruppe wird aber nicht zentral gesteuert. Sie ist das Ergebnis der Bewegungen der einzelnen Agenten in dieser Gruppe.

Die Kombination von mehreren Steuerungsverhalten lässt den Eindruck einer komplexen Verhaltensweise der Agenten entstehen. So lässt sich z. B. durch die Kombination der drei oben beschriebenen Steuerungsverhalten das Verhalten eines Schwarms nachahmen [13].

Zusätzlich enthält OpenSteer ein Framework, genannt OpenSteerDemo, das dem Programmierer eine schnelle Entwicklung von Prototypen ermöglicht. Diese Prototypen können als Plug-ins in OpenSteerDemo eingebunden und mithilfe dieses Frameworks visualisiert und getestet

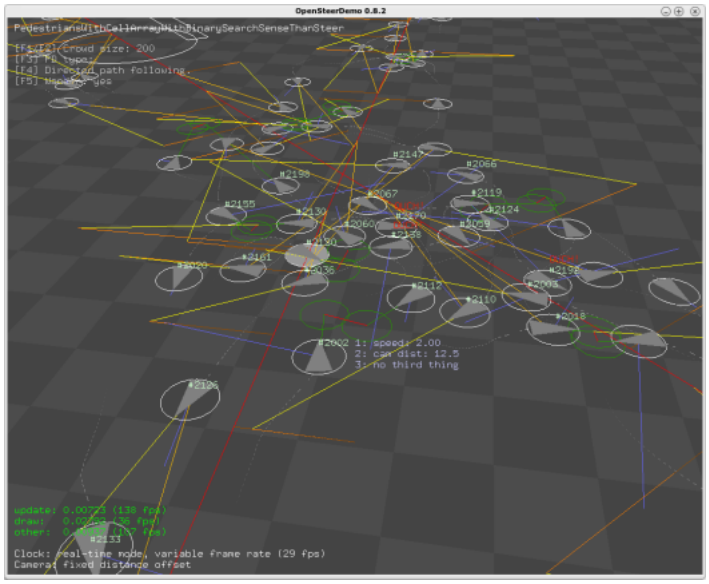


Abbildung 2.2: Das Pedestrian Plug-in der OpenSteerDemo Applikation.

werden. Die Abbildung 2.2 zeigt ein typisches Plug-in aus dem OpenSteerDemo-Framework. Es handelt sich dabei um das Plug-in Pedestrian. In diesem Plug-in werden mehrere Fußgänger simuliert, die entlang eines Pfades laufen und dabei versuchen, Kollisionen mit anderen Fußgängern zu vermeiden [10].

2.2 Paralleles Rechnen

Wie die meisten Programme wurden Computerspiele lange Zeit ausschließlich für Rechner, die über einen einzigen Prozessor verfügen, geschrieben. Die Entwickler der Computerspiele konnten dabei von der rasant steigenden Leistung der Mikroprozessoren profitieren [7]. Das Tempo der Leistungssteigerung bei Mikroprozessoren lässt sich mit dem Mooreschen Gesetz verdeutlichen. Das Gesetz besagt, dass sich die Anzahl der Transistoren alle 18 Monate verdoppelt [16]. Dank der ständig steigenden Leistung von Mikroprozessoren war es möglich immer komplexere und aufwändigere Spiele für sequentielle Rechensysteme zu entwickeln. Parallele Rechensysteme wurden bis vor kurzem nur für Anwendungen mit einem sehr hohen Rechenaufwand verwendet, d. h., sie wurden nur dann eingesetzt, wenn die Leistung eines einzelnen Prozessors nicht ausreichte, um eine Berechnung in einer angemessenen Zeit ausführen zu können.

Mit der Einführung der so genannten MultiCore-Prozessoren auf dem Massenmarkt begann sich die Situation zu ändern. Um die Leistung von mehreren Prozessorkernen zu nutzen, sind die Entwickler der Computerspiele nun gezwungen ihre Programme so zu ändern, dass sie parallel von mehreren Prozessoren ausgeführt werden können [7].

Bei der Entwicklung von Programmen für parallele Rechensysteme geht es vor allem darum, die Berechnungen auf die verschiedenen Prozessoren zu verteilen und somit die Ausführungszeit zu reduzieren. Dabei wird das Programm in mehrere Teilaufgaben zerlegt, die dann parallel zueinander von mehreren Prozessoren ausgeführt werden [9]. Der Geschwindigkeitsgewinn

eines parallelen Programms wird über den Begriff *Speedup* ausgedrückt. Der Speedup eines parallelen Programms mit der Laufzeit $T_p(n)$ wird definiert als [9]:

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

p bezeichnet dabei die Anzahl der Prozessoren zur Lösung des Problems der Größe n und $T^*(n)$ die Laufzeit einer sequentiellen Implementierung zur Lösung dieses Problems. Im Idealfall entspricht der Speedup der Anzahl der verwendeten Prozessoren (*linearer Speedup*). Um einen hohen Speedup zu erreichen ist es wichtig, die einzelnen Teilaufgaben möglichst gleichmäßig zwischen den Prozessoren zu verteilen, um eine gute Lastverteilung zu erreichen [9].

2.2.1 OpenMP

Die parallelisierte Version der OpenSteer-Bibliothek wurde mithilfe von OpenMP realisiert. OpenMP ist eine Spezifikation, die die Programmiersprachen C, C++ und FORTRAN um Konstrukte für parallele Programmierung von Rechnern mit gemeinsamem Speicher erweitert. In der OpenMP-Spezifikation sind eine Reihe von Compilerdirektiven, Bibliotheksfunktionen und Umgebungsvariablen definiert, die zur Aufteilung der Arbeit zwischen mehreren Threads sowie zur Synchronisation und Deklaration von gemeinsamen und privaten Variablen verwendet werden können [9].

OpenMP verwendet Threads, um eine parallele Ausführung zu realisieren. Der Begriff Thread bezeichnet dabei einen Kontrollfluss innerhalb eines Prozesses. Die Threads eines Programms werden nach einem fork-join-Prinzip erzeugt und beendet [9]. Ein OpenMP-Programm beginnt mit der Ausführung eines so genannten Master-Threads. Dieser Thread führt das Programm bis zum Auftreten eines parallelen Bereichs sequentiell aus. Ein paralleler Bereich lässt sich über eine spezielle Compilerdirektive deklarieren: `#pragma omp parallel`. Sobald der Master-Thread einen parallelen Bereich erreicht, erzeugt er ein Team von Threads. Zusammen mit diesen Threads führt der Master-Thread dann den Code des parallelen Bereichs aus.

Zusätzlich bietet OpenMP die Möglichkeit, die Berechnungen innerhalb eines parallelen Bereichs automatisch durch das OpenMP-Laufzeitsystem auf die Threads zu verteilen. So lassen sich `for`-Schleifen, bei denen die Anzahl der Iterationen im voraus bekannt ist, mit der Compilerdirektive `#pragma omp parallel for` parallelisieren. Der OpenMP-Compiler und das Laufzeitsystem kümmern sich darum, dass die Iterationen der Schleife auf die Threads verteilt werden, so dass jeder Thread nur einen Teil der Iterationen ausführen muss.

Für die Synchronisation des Zugriffs auf gemeinsamen Speicher bietet OpenMP neben diversen Compilerdirektiven Lockvariablen, die mithilfe der Bibliotheksfunktionen manipuliert werden können. Zusätzlich stellt OpenMP dem Programmierer weitere Konstrukte für die Parallelisierung eines Programms zur Verfügung, auf die ich an dieser Stelle nicht näher eingehen werde. Weitere Informationen über OpenMP findet man in der aktuellen Spezifikation dieses Programmiersystems [1].

3 Einsatz räumlicher Datenstrukturen in OpenSteer

In diesem Kapitel wird kurz dargestellt, wie und wofür räumliche Datenstrukturen in OpenSteer verwendet werden. Zunächst erläutere ich in Abschnitt 3.1 den Grund für den Einsatz von räumlichen Datenstrukturen in OpenSteer. Im Abschnitt 3.2 beschreibe ich den Ablauf einer typischen OpenSteer-Simulation. Der Abschnitt 3.3 enthält die Beschreibung der Schnittstelle einer räumlichen Datenstruktur. Schließlich zeige ich im Abschnitt 3.4, wie eine räumliche Datenstruktur in einem OpenSteer-Plugin verwendet werden kann.

3.1 Motivation für den Einsatz räumlicher Datenstrukturen

Die OpenSteer-Bibliothek enthält mehrere Steuerungsverhalten, mit denen das Verhalten von autonomen Agenten simuliert werden kann. Steuerungsverhalten wie Separation, Cohesion und Alignment basieren nicht auf einem langfristigen Plan, sondern resultieren aus einer direkten Reaktion des Agenten auf seine Umgebung. Solche Verhaltensmuster können auch bei Menschen beobachtet werden. Ein Mensch, der durch eine belebte Fußgängerzone läuft, verhält sich ähnlich, indem er anderen Menschen in seiner Umgebung instinktiv ausweicht [13]. Menschen oder Tiere bekommen die Information über ihre Umwelt durch ihre Sinnesorgane. Damit ein Agent auf das Geschehen um ihn herum reagieren kann, benötigt er ebenfalls Informationen über seine Umwelt. Für die Vermeidung von Kollisionen muss er z. B. die Positionen der Agenten in seiner Nähe kennen, damit seine Bewegungsrichtung und Geschwindigkeit so angepasst werden können, dass es nicht zu einem Zusammenstoß mit anderen Agenten kommt.

Für die Realisierung eines Steuerungsverhalten, wie Separation, sind nur die unmittelbaren Nachbarn eines Agenten interessant. Ein Agent kann nicht mit einem anderen Agenten, der weit von ihm entfernt ist, zusammenstoßen. Für die Steuerung eines Agenten A benötigt man also die Positionen der Agenten, die sich in einem bestimmten Radius r um den Agenten A befinden. Diese Agenten werden im weiteren Text *Nachbarn* des Agenten A genannt. Von allen Nachbarn eines Agenten A sind vor allem die interessant, die A am nächsten sind. OpenSteer-Module, die Steuerungsverhalten realisieren, müssen also in der Lage sein, die *k-nächsten Nachbarn* eines Agenten A, d. h. die Gruppe aus k Agenten, deren Entfernung von A höchstens r beträgt und die A am nächsten sind, zu bestimmen.

Für die Suche nach den k-nächsten Nachbarn kann eine Liste mit den Positionen aller Agenten verwendet werden. Die Lösung ist einfach zu implementieren, hat aber den Nachteil, dass bei der Suche nach den Nachbarn eines Agenten, jedes Element der Liste durchlaufen werden muss. Die meisten Elemente dieser Liste müssten jedoch gar nicht geprüft werden, weil sie Agenten speichern, die zu weit von dem Agenten entfernt sind, desen Nachbarn gesucht werden. Durch die Verwendung von speziellen Datenstrukturen, läßt sich die Anzahl der Agenten, die bei der Suche nach den k-nächsten Nachbarn eines Agenten überprüft werden

müssen, reduzieren. Diese speziellen Datenstrukturen werden auch räumliche Datenstrukturen genannt. Die in dieser Arbeit untersuchten Datenstrukturen werden im Kapitel 4 ausführlich beschrieben.

3.2 Aufbau einer OpenSteer Simulation

Mithilfe der OpenSteer-Bibliothek lassen sich Bewegungen von autonomen Agenten simulieren. Eine solche Simulation verläuft in mehreren Schritten. Damit der Eindruck einer Bewegung entsteht, müssen die Positionen der Agenten in jedem Simulationsschritt neu berechnet werden.

Ein Simulationsschritt kann wiederum in mehrere Zwischenschritte unterteilt werden. Im ersten Zwischenschritt werden für jeden Agenten die Positionen seiner Nachbarn bestimmt. Dieser Zwischenschritt wird im Folgenden als *Suchphase* bezeichnet. Für die Suche nach den Nachbarn eines Agenten werden dabei räumliche Datenstrukturen verwendet.

Anschließend folgt die *Steeringphase*. In dieser Phase wird aus den bei der Suchphase ermittelten Positionen der Nachbarn unter Anwendung verschiedener Steuerungsverhalten die Geschwindigkeit und die Bewegungsrichtung eines Agenten neu bestimmt. Aus der aktuellen Position eines Agenten, seiner Geschwindigkeit und Bewegungsrichtung wird schließlich die neue Position des Agenten berechnet.

Der letzte Zwischenschritt ist die sogenannte *Updatephase*. Die alten Koordinaten der Agenten werden in dieser Phase durch die neu berechneten Positionen ersetzt. Zusätzlich wird die räumliche Datenstruktur, die für die Suche nach den Nachbarn eines Agenten benutzt wird, aktualisiert.

Die Such- und die Steeringphase können zusammengefasst werden: die neue Position eines Agenten kann sofort, nachdem die Positionen seiner Nachbarn bekannt sind, berechnet werden. Alternativ kann man zuerst die Nachbarn für alle Agenten bestimmen und erst dann mit der Steeringphase beginnen. Wichtig ist die Trennung zwischen der Steering- und der Updatephase. Da die alten Positionen der Agenten für die Steuerung der Agenten benötigt werden, kann die Updatephase erst dann beginnen, wenn die Steeringphase abgeschlossen ist.

In der parallelisierten Version von OpenSteer wird die Arbeit so zwischen den Threads verteilt, dass ein Thread das Verhalten von mehreren Agenten simuliert [5]. Die Trennung zwischen verschiedenen Phasen der Simulation erleichtert dabei die Parallelisierung. Da die Threads während der Such- und der Steeringphase nur lesend auf die Daten der Agenten zugreifen, müssen die Speicherzugriffe während dieser Simulationsphasen nicht synchronisiert werden. Nach der Steeringphase muss durch eine Barriere sichergestellt werden, dass ein Thread nur dann zur Updatephase übergehen kann, wenn alle anderen Threads die Steeringphase ebenfalls beendet haben. Damit während der Updatephase keine zeitkritischen Abläufe entstehen, muss die Datenstruktur, die für die Suche nach den Nachbarn eines Agenten verwendet wird, über thread-sichere Update-Operationen verfügen.

3.3 Schnittstelle der räumlichen Datenstrukturen

Damit eine räumliche Datenstruktur für die Nachbarschaftsabfragen in einem OpenSteer Plugin verwendet werden kann, muss sie eine bestimmte Funktionalität zur Verfügung stellen. In

C++ kann diese Funktionalität in einer Klasse gekapselt werden. Diese Klasse muss neben der Implementierung der Datenstruktur bestimmte Methoden bereitstellen, über die z. B. Positionen der Agenten aktualisiert oder die Nachbarn eines Agenten ermittelt werden können. Diese Methoden bilden eine Schnittstelle, die in der Aufgabenstellung vorgegeben wurde und für jede der untersuchten räumlichen Datenstrukturen implementiert werden muss.

```
1  template< typename Reference, class Position = math3d::vector3 >
2  class spatial_database
3  {
4  public:
5      bool add( Reference const& data, Position const& position );
6
7      bool remove( Reference const& data, Position const& position );
8
9      bool update( Reference const& data, Position const& new_position, Position const&
           old_position );
10
11     template< typename OutputIterator >
12     void find_neighbors_within_radius( Position const& position, typename Position::value_type
           max_radius, OutputIterator iter ) const;
13
14     template< typename OutputIterator >
15     void find_k_nearest_neighbors( Position const& position, size_type k, OutputIterator iter )
           const;
16
17     template< typename OutputIterator >
18     void find_k_nearest_neighbors_in_radius( Position const& position, size_type k, typename
           Position::value_type max_radius, OutputIterator iter ) const;
19 };
```

Listing 3.1: Schnittstelle einer räumlichen Datenstruktur.

Diese Schnittstelle ist im Listing 3.1 dargestellt. Die einzelnen Methoden dieser Schnittstelle stelle ich im Folgenden vor.

`add(Reference const& data, Position const& position)`
: fügt einen neuen Agenten in die Datenstruktur ein. Diese Methode erwartet als Parameter die Beschreibung und die Koordinaten des Agenten, der eingefügt werden soll. Die Beschreibung des Agenten, die über den Parameter `data` übergeben wird, ermöglicht eine eindeutige Identifizierung des Agenten. Meistens handelt es sich dabei um die Speicheradresse des Objekts, das die Daten des Agenten enthält. Die Koordinaten des Agenten werden in einem dreidimensionalen Vektor über den Parameter `position` übergeben.

`remove(Reference const& data, Position const& position)`
: löscht einen Agenten aus der Datenstruktur. Damit der Agent in der Datenstruktur gefunden werden kann, benötigt diese Methode die Beschreibung und die Koordinaten des Agenten.

`update(Reference const& data, Position const& new_position, Position const& old_position)`
: aktualisiert die Position eines Agenten in der Datenstruktur. Als Parameter erwartet diese Methode die Beschreibung und die neue Position des Agenten. Um das Auffinden des Agenten in der Datenstruktur zu beschleunigen, braucht diese Methode zusätzlich die alte Position des Agenten.

`find_neighbors_within_radius(Position const& position, typename Position::value_type max_radius, OutputIterator iter)`
: sucht nach allen Agenten, die sich innerhalb einer kreisförmigen Suchregion befinden.

Die Suchregion wird über den Punkt `position` und den Radius `max_radius` beschrieben. Die Methode sucht nach allen Agenten, die nicht weiter als dieser Radius von dem angegebenen Punkt entfernt sind. Agenten, die gefunden wurden, werden über den Iterator `iter` an der Aufrufer zurückgegeben.

`find_k_nearest_neighbors(Position const& position, size_type k, OutputIterator iter)`
sucht nach `k` Agenten, die dem Punkt mit den Koordinaten `position` am nähesten sind. Wie bei der Methode `find_neighbors_within_radius` werden die gefundenen Agenten über den Iterator `iter` zurückgegeben. Die Anzahl der Agenten, die zurückgegeben werden, wird über den Parameter `k` bestimmt.

`find_k_nearest_neighbors_in_radius(Position const& position, size_type k, typename Position::value_type max_radius, OutputIterator iter)`
sucht in einer kreisförmigen Suchregion nach k-nächsten Nachbarn eines Agenten mit der Position `position`. Die Suchregion wird wie bei `find_neighbors_within_radius` durch den Punkt `position` und den Radius `max_radius` beschrieben. Die Agenten werden über den Iterator `iter` zurückgegeben. Die Anzahl der Nachbarn, die diese Methode liefern soll, wird über den Parameter `k` bestimmt.

3.4 Beispiel für die Verwendung einer räumlichen Datenstruktur

In diesem Abschnitt möchte ich die Verwendung einer räumlicher Datenstruktur in einem OpenSteer-Plugin an einem Beispiel demonstrieren.

Damit eine räumliche Datenstruktur für die Nachbarschaftssuche verwendet werden kann, muss sie zunächst initialisiert werden. Dazu wird eine Instanz der räumlichen Datenstruktur erzeugt und mit den Positionen der Agenten, die an der Simulation teilnehmen, gefüllt.

```
1 // agents ist ein Container, der alle Agenten dieser Simulation enthält.
2 for (std::size_t i = 0; i < agents.size (); ++i) {
3       // spatial_database ist die Instanz der räumlichen Datenstruktur.
4       spatial_database.add (agents[i], agents[i]->position ());
5 }
```

Listing 3.2: Einfügen mehrerer Agenten in eine räumliche Datenstruktur

Um die Agenten in die räumliche Datenstruktur einzufügen wird die Methode `add()` verwendet (Listing 3.2). Die Datenstruktur speichert dabei nur die Position des Agenten. Das Objekt, das die kompletten Daten eines Agenten speichert, befindet sich in einem separaten Container.

Nachdem das Plugin und die räumliche Datenstruktur initialisiert sind, kann die Simulation beginnen. Die Simulation wird in Schritten ausgeführt. Jeder Simulationsschritt besteht aus mehreren Phasen (vergleiche Abschnitt 3.2). In diesem Beispiel werden die Such- und die Steeringphase getrennt. Das bedeutet, dass zuerst für alle Agenten die Nachbarn bestimmt werden müssen (Listing 3.3).

```
1 #pragma omp parallel for
2 for (int i = 0; i < agents.size (); ++i) {
3       // Suche nach den k-nächsten Nachbarn des Agenten mit dem Index i:
4       spatial_database.find_k_neareset_neighbors_in_radius (
5                                   agents[i]->position (),
6                                   k_nearest,
7                                   radius,
8                                   agents[i]->neighbors_iterator);
9 }
```

Listing 3.3: Nachbarschaftssuche über eine räumliche Datenstruktur

Die Nachbarschaftssuche im Listing 3.3 wird parallel von mehreren Threads durchgeführt. Die Arbeit wird so zwischen den Threads aufgeteilt, dass jeder Thread die k-nächsten Nachbarn für einen Teil der Agenten sucht. Die Aufteilung der Arbeit auf die einzelnen Threads erfolgt über das OpenMP-Laufzeitsystem. Die Direktive in Zeile 1 bewirkt, dass die Iterationen der `for`-Schleife gleichmäßig zwischen den Threads aufgeteilt werden.

Nachdem die k-nächsten Nachbarn der einzelnen Agenten ermittelt wurden, wird für jeden Agenten seine neue Position unter der Anwendung von verschiedenen Steuerungsverhalten berechnet. Zum Abschluss des Simulationsschrittes müssen die neuen Positionen der Agenten in der räumlichen Datenstruktur gespeichert werden.

```
1 #pragma omp parallel for
2 for (int i = 0; i < agents.size (); ++i) {
3       spatial_database.update (agents[i], agents[i]->position (), agents[i]->old_position);
4 }
```

Listing 3.4: Updatephase

Die einzelnen Update-Operationen können wie im Listing 3.4 parallel ausgeführt werden. Nach der Update-Phase ist ein Simulationsschritt zu Ende und der nächste Simulationsschritt beginnt wieder mit der Nachbarschaftssuche.

4 Räumliche Datenstrukturen

Räumliche Datenstrukturen verwalten so genannte multidimensionale oder räumliche Daten. Multidimensionale Daten werden häufig als eine Sammlung von Punkten in einem Raum mit mehr als einer Dimension definiert [12]. Das Einsatzgebiet dieser Datenstrukturen umfasst viele Bereiche der Informatik, wie z. B. Datenbankmanagementsysteme, Computergrafik, Spieleprogrammierung, geographische Informationssysteme (GIS) und viele mehr.

Neben einfachen Punkten können auch komplexere geometrische Strukturen wie Linien oder Polygone in räumlichen Datenstrukturen gespeichert werden. Einige dieser Datenstrukturen ermöglichen sogar die Verwaltung von hochdimensionalen Daten, wobei die Anzahl der Dimensionen deutlich größer als drei sein kann. Ziel dieser Arbeit ist es jedoch, verschiedene Datenstrukturen für die Verwaltung der Positionen von Agenten in einem dreidimensionalen Raum zu untersuchen. Bei der Beschreibung der Datenstrukturen werde ich mich daher nur auf den Einsatz von räumlichen Datenstrukturen für die Speicherung von dreidimensionalen Punktdaten beschränken. Komplexe und hochdimensionale Daten werden in dieser Arbeit nicht betrachtet.

Räumliche Datenstrukturen sind vergleichbar mit Indexstrukturen wie den Binärbäumen, B-Bäumen oder Hashtabellen. Diese speziellen Datenstrukturen beschleunigen den gezielten Zugriff auf Datensätze einer größeren Datenmenge. Bestimmte Anfragen können effizient durchgeführt werden. So kann z. B. schnell geprüft werden, ob ein bestimmter Datensatz in der Datenstruktur gespeichert ist. Außerdem können Bereichsanfragen, die alle Datensätze liefern, deren Werte in einem vorgegebenen Bereich liegen, effizient ausgewertet werden. Räumliche Datenstrukturen werden speziell zum Indizieren von multidimensionalen Daten eingesetzt. Eine der Schwierigkeiten beim Umgang mit multidimensionalen Daten besteht darin, dass es keine lineare Ordnung in einer Menge mit solchen Daten gibt [2]. Multidimensionale Daten lassen sich, anders als eindimensionale Daten, nicht in einer linearen Sequenz speichern, bei der die Punkte, die im Raum nah bei einander liegen, auch in dieser linearen Sequenz benachbart sind. Diese Eigenschaft erschwert die Verwendung von traditionellen Indexstrukutren wie Binärbäumen für die Verwaltung von multidimensionalen Daten. Um effiziente Suchoperationen auch für multidimensionale Daten zu ermöglichen, sind räumliche Datenstrukturen nötig.

Räumliche Datenstrukturen basieren auf einer Aufteilung des Raumes in kleinere Teilbereiche. Durch eine solche Aufteilung kann die Suche in multidimensionalen Daten auf einen Teil des Raumes beschränkt werden, d. h. bei der Auswertung einer Suchanfrage müssen nicht alle in der Datenstruktur gespeicherten Daten, sondern nur ein Teil davon ausgewertet werden. Es gibt mehrere Methoden den Raum aufzuteilen. Grundsätzlich unterscheidet man zwischen zwei Arten von räumlichen Datenstrukturen: flachen und hierarchischen Datenstrukturen [18].

Flache Datenstrukturen teilen den Raum in disjunkte Zellen auf, so dass die Zelle, in der ein Punkt liegt, in einer konstanten Zeit bestimmt werden kann. Ein typisches Beispiel für eine räumliche Datenstruktur, die den Raum auf diese Weise aufteilt, ist die Grid-Datenstruktur. Ein Grid wird wie ein gleichmäßiges Gitter über den Raum gelegt. Die Größe der Zellen ist dabei fest definiert. Dieses Kapitel behandelt drei Varianten des Grids. Die ersten beiden

Varianten unterscheiden sich nur in der Art, in der sie die Zellen des Grids auf den Speicher abbilden. Die erste Variante, die im Abschnitt 4.2.2 beschrieben wird, verwendet dazu eine Hash-Funktion, die ein beliebig großes Grid mit beliebig vielen Zellen auf ein Array fester Größe abbildet. Die im Abschnitt 4.2.3 beschriebene Variante bildet mit Hilfe der Modulo-Funktion alle Zellen auf einen Ausschnitt des Grids ab, der im Speicher gehalten wird. Die letzte Grid-Variante spannt das Grid für einen k-dimensionalen Raum in nur $k-1$-Dimensionen auf. Die Positionen der Objekte, die in einer Zelle gespeichert sind, werden dann nach der Koordinate der letzten Dimension sortiert. Diese Modifikation eines Grids wird im Abschnitt 4.3 ausführlich beschrieben.

Hierarchische Datenstrukturen teilen den Raum dagegen rekursiv mit Hilfe einer baumartigen Datenstruktur auf. Als Beispiel für eine hierarchische Datenstruktur beschreibe ich im Abschnitt 4.1 den so genannten k-d-Baum.

Am Ende des Kapitels werde ich im Abschnitt 4.4 die vorgestellten Datenstrukturen bezüglich des Laufzeitaufwandes der einzelnen Operationen vergleichen.

4.1 k-d-Bäume

Ein bekanntes Beispiel für eine räumliche Datenstruktur ist der k-d-Baum. Diese Datenstruktur wurde 1975 von J. L. Bentley vorgestellt [3]. Dabei steht k für die Dimensionalität des Raums, der durch die Datenstruktur dargestellt wird. Wie der Name andeutet, kann ein k-d-Baum für die Darstellung von hochdimensionalen Räumen verwendet werden, wobei k viel größer als drei sein kann. Im Bereich der Computergraphik werden k-d-Bäume häufig für das Raytracing, eine Methode zur Berechnung einer dreidimensionalen Szene, verwendet.

Dieser Abschnitt ist wie folgt aufgebaut: Zunächst erkläre im Abschnitt 4.1.1 ich, wie ein k-d-Baum den Raum aufteilt. Im Abschnitt 4.1.2 beschreibe ich, wie die Daten in diese Datenstruktur eingefügt werden. Der Abschnitt 4.1.3 befasst sich mit dem Verfahren zum Löschen von Knoten aus einem k-d-Baum. Das Vorgehen bei der Suche nach den Nachbarn eines Agenten wird im Abschnitt 4.1.4 beschrieben. Zum Schluss erläutere ich im Abschnitt 4.1.5, wie die Positionen der Agenten aktualisiert werden können.

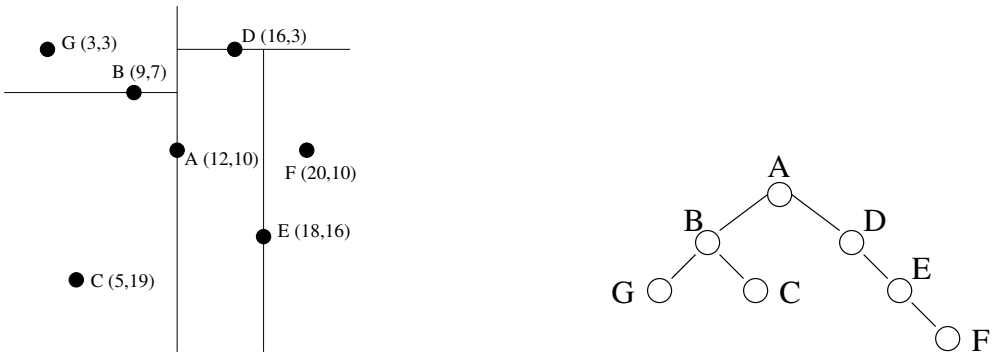
4.1.1 Prinzip

Ein k-d-Baum ist ein binärer Suchbaum, der auch für mehrdimensionale Daten verwendet werden kann. In der gebräuchlichsten Form (der so genannten homogenen Variante) eines k-d-Baums dienen sowohl die Blätter als auch die inneren Knoten als Speicher für Punkte. Jeder innerer Knoten speichert neben den Verweisen auf das linke und rechte Kind auch einen Punkt im k-dimensionalen Raum.

An jedem Knoten eines k-d-Baumes wird der Raum durch eine $(k-1)$-dimensionale Hyperebene in zwei Teile geteilt. Der Punkt, den ein Knoten speichert, repräsentiert dabei die Hyperebene, die den Raum teilt. Eine der Koordinaten des Punktes wird dabei als der so genannte Diskriminator (engl.: discriminator) verwendet. Der Diskriminator ist für alle Knoten auf einer Ebene des Baums gleich. An der Wurzel eines k-d-Baums dient die erste Koordinate (x) als Diskriminator. An jeder neuen Ebene des Baums wird abwechselnd eine andere Koordinate als ein neuer Diskriminator verwendet. So wird z. B. in einem zweidimensionalen Raum an jeder geraden Ebene des k-d-Baums die x-Koordinate und an jeder ungeraden Ebene die

y-Koordinate als Diskriminator benutzt. Wenn sich ein Knoten N auf einer Ebene mit x als Diskriminator befindet, und einen Punkt P mit den Koordinaten (x_P, y_P) speichert, wird der Raum an diesem Knoten wie folgt aufgeteilt: alle Punkte mit einer x-Koordinate kleiner als x_P werden im linken Teilbaum von N gespeichert und alle anderen Punkte, deren x-Koordinate größer oder gleich x_P ist, im rechten Teilbaum.

Die Abbildung 4.1 zeigt einen zweidimensionalen Raum mit dem entsprechenden k-d-Baum.



(a) Ein zweidimensionaler Raum aufgeteilt mit Hilfe eines k-d-Baums.

(b) Ein k-d-Baum mit allen Punkten aus der Abbildung 4.1(a).

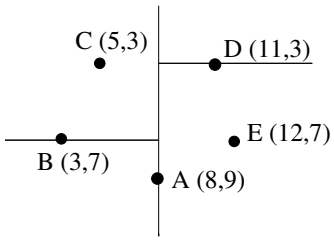
Abbildung 4.1: Raumaufteilung durch einen k-d-Baum.

Die Struktur eines k-d-Baums erleichtert die Suche nach einem bestimmten Punkt. Angefangen an der Wurzel prüft man an jedem Knoten entlang des Suchpfades, ob der Punkt, den dieser Knoten speichert, der gesuchte ist. Ist er es nicht, entscheidet man mit Hilfe der Diskriminator-Koordinate, welchen der beiden Kindknoten man als Nächstes untersuchen muss: ist der Wert der Diskriminator-Koordinate des gesuchten Punktes kleiner als der entsprechende Wert des aktuellen Knotens, folgt man dem Pfad zum linken Kindknoten, sonst besucht man als Nächstes den rechten Kindknoten. Existiert der Knoten, der als Nächstes besucht werden müsste, nicht, kann die Suche abgebrochen werden, weil der gesuchte Punkt nicht in der Datenstruktur enthalten ist.

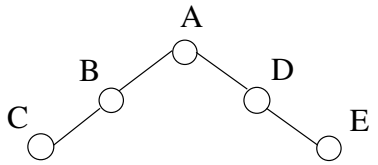
4.1.2 Einfügen neuer Daten

Das Einfügen eines Punktes in einen k-d-Baum ist sehr einfach. Ist der Baum leer, wird ein neuer Knoten angelegt und der Punkt, der eingefügt werden soll, in diesem Knoten gespeichert. Der neue Knoten ist nun die Wurzel des k-d-Baums. Ist der Baum nicht leer, steigt man beginnend bei der Wurzel den Baum hinab, bis man eine geeignete Stelle gefunden hat, an der man den Punkt einfügen kann. Wie bei der im Abschnitt 4.1.1 beschriebenen Suche entscheidet man mit Hilfe der Diskriminator-Koordinate an jedem Knoten, ob der Punkt im linken oder im rechten Teilbaum eingefügt werden soll. Erreicht man die Stelle, an der der Knoten, der als Nächstes besucht werden soll, nicht existiert, wird ein neuer Knoten mit dem Punkt, der eingefügt werden soll, erzeugt und an den aktuellen Knoten angehängt [12].

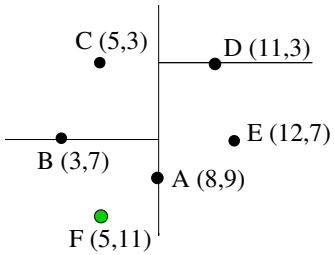
Die Abbildung 4.2 demonstriert das Einfügen eines Punktes in einen k-d-Baum der bereits mehrere Knoten enthält. Obwohl der Baum aus dieser Abbildung relativ gleichmäßig aussieht, sind k-d-Bäume nicht höhenbalanciert. Im schlimmsten Fall kann ein k-d-Baum sogar zu einer



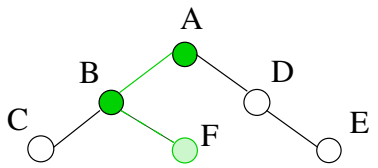
(a) Zweidimensionaler Raum ohne den Punkt F.



(b) Ein k-d-Baum mit allen Punkten aus der Abbildung 4.2(a).



(c) Zweidimensionaler Raum nach dem Einfügen von F.



(d) Der k-d-Baum aus der Abbildung 4.2(b) nach dem Einfügen von F.

Abbildung 4.2: Einfügen eines Knotens in einen k-d-Baum.

Liste mutieren. Die Struktur eines k-d-Baums hängt stark von der Reihenfolge ab, in der die Punkte eingefügt werden.

Das Einfügen eines Punktes in einen k-d-Baum ähnelt sehr dem Einfügen in einen binären Suchbaum [12]. Der einzige Unterschied besteht darin, dass bei k-d-Bäumen an jeder Ebene des Baums jeweils eine andere Koordinate verwendet wird, um die Richtung, in der man den Baum hinabsteigt, zu bestimmen. Bentley [3] hat gezeigt, dass sowohl die Komplexität für die Suche nach einem bestimmten Punkt, als auch die Komplexität für das Einfügen eines neuen Punktes in einen k-d-Baum mit N Knoten im Durchschnitt $O(\log_2 N)$ beträgt. Im schlimmsten Fall, wenn der Baum zu einer Liste mutiert ist, beträgt der Laufzeitaufwand für die Suche und das Einfügen $O(N)$.

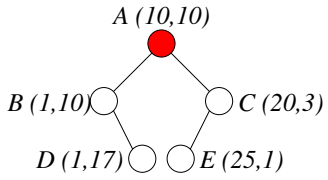
4.1.3 Löschen von Knoten

Anders als das Einfügen neuer Punkte, ist das Löschen von Knoten aus einem k-d-Baum wesentlich komplexer als das Löschen aus einem binären Suchbaum. Falls der Knoten, der gelöscht werden muss, ein Blattknoten ist, ist das Löschen, wie bei binären Suchbäumen trivial. Wenn der Knoten allerdings Kinder hat, muss man zunächst einen geeigneten Ersatzknoten finden, was bei k-d-Bäumen nicht einfach ist [12].

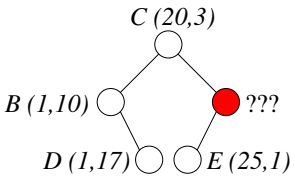
Um in einem binären Suchbaum einen Knoten mit zwei Kindknoten zu entfernen, reicht es aus, wenn man den Knoten entweder durch den am weitesten links stehenden Knoten des rechten Teilbaums, oder durch den am weitesten rechts stehenden Knoten des linken Teilbaums ersetzt. Bei k-d-Bäumen funktioniert dieses Verfahren nicht. Der Grund dafür liegt darin, dass die Diskriminator-Koordinate sich auf jeder Baumebene ändert. Dies hat zu Folge, dass wenn man einen Knoten in eine Ebene mit einer anderen Diskriminator-Koordinate verschiebt,

die Beziehung des Knotens zu seinen neuen Kindknoten möglicherweise die Regeln für den Aufbau eines k-d-Baums verletzt [12].

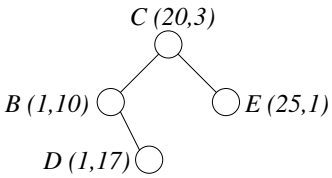
Wenn man z. B. aus dem k-d-Baum in der Abbildung 4.3(a) den Knoten A entfernen möchte, muss man einen geeignet Ersatzknoten für A finden. In einem binären Suchbaum wären sowohl



(a) Ein k-d-Baum, in dem der Knoten A gelöscht werden muss.



(b) Der k-d-Baum nach dem Ersetzen des Knotens A durch den Knoten C.



(c) Der k-d-Baum nach dem Abschluss der Löschoperation.

Abbildung 4.3: Löschen aus einem k-d-Baum.

D, als auch E geeignete Kandidaten. In einem k-d-Baum dagegen kann keiner dieser Knoten als Ersatz für A verwendet werden. Die Diskriminator-Koordinate auf der Ebene von A ist x. Der x-Wert von E wäre aber größer als der x-Wert von seinem neuen rechten Kindknoten C. Würde man A durch D ersetzen, dann befände sich der Knoten B nach der Ersetzung im linken Teilbaum von D. Die x-Werte von B und D sind gleich. Im linken Teilbaum von D dürfen aber nur Punkte gespeichert werden, deren x-Werte kleiner sind als der x-Wert von D (vergleiche 4.1.1). Somit kann auch D kann geeigneter Ersatzknoten für A sein.

Ein geeigneter Ersatzknoten muss also einen x-Wert haben, der größer ist als die x-Werte aller Knoten im linken Teilbaum, darf aber gleichzeitig nicht größer sein, als die x-Werte aller Knoten im rechten Teilbaum. Das bedeutet, dass es entweder der Knoten mit dem größten x-Wert im linken Teilbaum, oder der Knoten mit dem kleinsten x-Wert im rechten Teilbaum sein muss. Falls man einen Ersatzknoten aus dem linken Teilbaum wählt, kann es aber vorkommen, dass der linke Teilbaum einen anderen Knoten, mit demselben x-Wert enthält. So haben z. B. beide Knoten des linken Teilbaums in der Abbildung 4.3(a) denselben x-Wert. Egal für welchen der beiden Knoten man sich entscheidet, es würde immer ein Knoten im linken Teilbaum bleiben, dessen x-Wert gleich dem x-Wert der Wurzel ist. Nach der Definition eines k-d-Baums darf der linke Teilbaum aber solche Knoten nicht enthalten. Der Ersatzknoten muss also immer aus dem rechten Teilbaum gewählt werden [12]. In dem Beispiel aus der Abbildung 4.3(a) ist es der Knoten C.

Beim Löschen muss weiterhin berücksichtigt werden, dass der Ersatzknoten nicht notwendigerweise ein Blattknoten sein muss. Falls der Ersatzknoten ein innerer Knoten ist, muss der Ersatzknoten selbst erst rekursiv mit demselben Verfahren aus dem Baum gelöscht werden, bevor er an einer anderen Stelle eingefügt werden kann. Im Beispiel aus der Abbildung 4.3 ist C ein innerer Knoten. Wie für den Knoten A muss auch für C ein geeigneter Ersatzknoten gefunden werden, um die Lücke in der Baumstruktur zu füllen (Abbildung 4.3(b)). Dieser Ersatzknoten muss ein Knoten im rechten Teilbaum von C sein. Der rechte Teilbaum des Knotens C ist aber leer. Da der Ersatzknoten aber immer aus dem rechten Teilbaum gewählt werden muss, vertauscht man in so einem Fall zuerst die beiden Teilbäume. Nach dem Vertauschen enthält der rechte Teilbaum wieder Knoten, aus denen einer als Ersatz für C gewählt wird. In dem Beispiel aus der Abbildung 4.3 ist es der einzige Kindknoten von C: der Knoten E.

Der durchschnittliche Aufwand für das Löschen eines zufällig gewählten Knotens aus einem k-d-Baum mit insgesamt N Knoten beträgt $O(\log_2 N)$ [12]. Dieser relativ niedrige Wert kommt zustande, weil die meisten Knoten in einem k-d-Baum Blattknoten sind. Ein Blattknoten kann, sobald er gefunden wurde, in konstanter Zeit entfernt werden. Das Löschen von inneren Knoten und insbesondere der Wurzel ist wesentlich aufwändiger. Da ein k-d-Baum nicht höhenbalanciert ist, kann er zu einer Liste mutieren. In so einem Fall beträgt der Aufwand für das Löschen der Wurzel aus einem Baum mit N Knoten $O(N)$ [12].

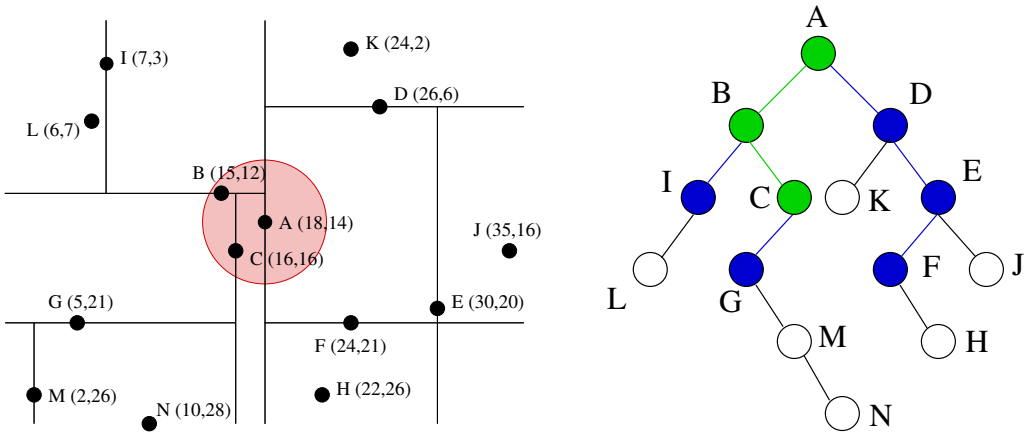
4.1.4 Nachbarschaftssuche

Wie alle räumlichen Datenstrukturen eignen sich k-d-Bäume für die Suche nach Punkten, die sich in einer bestimmten Region befinden, und somit auch für die Nachbarschaftssuche. In diesem Abschnitt wird das Verfahren, das für eine solche Suche in k-d-Bäumen verwendet werden kann, beschrieben.

Ein k-d-Baum teilt den Raum rekursiv an jedem Knoten durch eine Hyperebene in zwei Halbräume auf (vergleiche Abschnitt 4.1.1). Diese Halbräume entsprechen dann den Teilbäumen, die an diesem Knoten hängen. Dank dieser Raumaufteilung, kann bei der Suche nach allen Punkten innerhalb einer Region auf wenige Teilbäume beschränkt werden: auf jeder Baumebene wird ein Teilbaum nur dann untersucht, wenn sich der entsprechende Halbraum mit der Suchregion überschneidet. Überschneidet sich der Halbraum nicht mit der Suchregion, kann der entsprechende Teilbaum ignoriert werden.

Bei der Suche nach allen Punkten im Radius r um den Punkt $P_{center} = (a, b, c)$ beginnt man an der Wurzel des k-d-Baums und steigt dann rekursiv bis zu den Blattknoten des Baums bis alle Punkte in der Suchregion gefunden wurden. Für jeden Knoten auf dem Suchpfad wird geprüft, ob der Punkt, den dieser Knoten speichert, in der Suchregion liegt oder nicht. Im Fall einer kreis- oder kugelförmigen Suchregion, kann dazu die Entfernung des Punktes vom Mittelpunkt der Suchregion berechnet werden. Ist diese kleiner als der Radius, liegt der Punkt in der Suchregion [12].

Falls der Punkt sich in der Suchregion befindet, wird er in die Menge der gefundenen Punkte eingefügt und seine beiden Kindknoten untersucht. Wenn der Punkt, den dieser Knoten speichert, aber nicht in der Suchregion liegt, kann mithilfe der Diskriminator-Koordinate geprüft werden, ob man auf die Untersuchung eines der beiden Teilbäume, die von diesem Knoten ausgehen, verzichten kann. Dazu benötigt man für jede Dimension den minimalen und den maximalen Wert, den die Koordinate eines Punktes innerhalb der Suchregion haben kann. Für eine Suchregion mit dem Mittelpunkt $P_{center} = (a, b, c)$ und dem Radius r werden die minimalen Werte der drei Dimensionen wie folgt berechnet: $x_{min} = a - r$, $y_{min} = b - r$ und $z_{min} = c - r$. Analog dazu ergeben die maximalen Werte für einzelne Dimensionen: $x_{max} = a + r$, $y_{max} = b + r$ und $z_{max} = c + r$. Wenn der aktuelle Knoten den Raum in der Dimension x teilt, vergleicht man den x-Wert des Punktes, den dieser Knoten speichert, mit x_{min} und x_{max}. Ist der x-Wert des Punktes kleiner als x_{min}, muss der linke Teilbaum dieses Knotens nicht weiter untersucht werden, weil alle Punkte in diesem Teilbaum weiter als r von dem Mittelpunkt der Suchregion entfernt sind und somit außerhalb der Suchregion liegen. Ist der x-Wert des Punktes größer als x_{max}, kann auf die Suche im rechten Teilbaum des Knotens verzichtet werden, weil alle x-Werte der Punkte im rechten Teilbaum größer als x_{max} sein müssen. Dieses Verfahren lässt sich genauso auf die restlichen Dimensionen anwenden.



(a) Ein zweidimensionaler Raum, in dem alle Nachbarn des Punktes A gesucht werden.

(b) Ein k-d-Baum mit allen Punkten aus der Abbildung 4.4(a).

Abbildung 4.4: Suche in einem k-d-Baum.

Abbildung 4.4 veranschaulicht die Suche in einem k-d-Baum: in einem zweidimensionalen Raum (Abbildung 4.4(a)) werden alle Nachbarn des Agenten A gesucht. In diesem Beispiel soll ein Agent als Nachbar des Agenten A gelten, wenn er nicht weiter als $r = 4$ von A entfernt ist. Das bedeutet, dass die gesuchten Agenten sich im Radius $r = 4$ um den Agenten A befinden müssen. Die gestrichelten Linien stellten die Raumaufteilung, die durch den in der Abbildung 4.4(b) abgebildeten k-d-Baum entsteht, dar. Alle in der Abbildung 4.4(b) blau dargestellten Knoten müssen während der Suche untersucht werden. Die grün dargestellten Knoten entsprechen den Agenten, die sich in der Suchregion befinden.

Die Suche startet an der Wurzel des k-d-Baums. In diesem Beispiel speichert die Wurzel die Koordinaten von A. Dieser Punkt ist gleichzeitig der Mittelpunkt der Suchregion. Da A sich natürlich in der Suchregion befindet, werden alle seine Kindknoten untersucht: B und D (dabei spielt es keine Rolle, welcher dieser beiden Knoten zuerst besucht wird). Weil B, wie A, innerhalb der Suchregion liegt, werden auch alle Kindknoten von B untersucht. Der Agent D befindet sich dagegen außerhalb der Suchregion. Der Knoten, der die Koordinaten von D speichert, befindet sich in der zweiten Ebene des k-d-Baums und teilt den Raum somit in der y-Dimension mit der Gerade $y = 6$. Da die y-Koordinate von D kleiner ist als y_{min} $(6 < 10)$, muss der linke Teilbaum von D nicht weiter untersucht werden.

Die Suche wird fortgesetzt, bis es keine Knoten mehr gibt, die untersucht werden müssen. Insgesamt müssen in diesem Beispiel 8 von 14 Knoten untersucht werden, um beide Nachbarn von A zu finden.

Der Aufwand für die Suche nach allen Punkten innerhalb einer Region in einem k-d-Baum mit N Knoten beträgt im schlimmsten Fall (worst case) $O(d * N^{1-1/d} + F)$ [12], wobei d die Anzahl der Dimensionen des Raums und F die Anzahl der Punkte, die in der Suchregion liegen, ist. Für die Suche in einem dreidimensionalen Raum beträgt der Aufwand also $O(N^{\frac{2}{3}} + F)$. Im durchschnittlichen Fall beträgt der Laufzeitaufwand $O(\log N + F)$ [8]. F ist dabei die durchschnittliche Anzahl der Punkte im Suchraum.

4.1.5 Updates

In einer Schwarmsimulation sind die Agenten ständig in Bewegung und ändern dauernd ihre Koordinaten. Diese Änderungen müssen auch auf die Datenstruktur, die die Positionen der Agenten speichert, übertragen werden, damit die Suchanfragen richtige Ergebnisse liefern. Ein k-d-Baum ist aber eine statische Datenstruktur. Das bedeutet, dass mit den Positionen der Objekte sich auch die Struktur des k-d-Baums entsprechend ändert. Falls sich die Position eines Objekts ändert, reicht es nicht einfach den entsprechenden Knoten zu finden und die alten Koordinaten des Objekts durch neue zu ersetzen. Abhängig von der Positionsänderung und der Struktur des Baums müssen eventuell mehrere Knoten in dem k-d-Baum verschoben werden.

Um sicherzustellen, dass die Regeln für den Aufbau eines k-d-Baums durch die Aktualisierung der Koordinaten eines Objekts nicht verletzt werden, könnte man den entsprechenden Knoten aus dem Baum entfernen und mit den neuen Koordinaten wieder einfügen. Bei einer Simulation, in der alle Objekte in Bewegung sind, muss dieses Verfahren für jedes einzelne Objekt in jedem Simulationsschritt durchgeführt werden. Wie im Abschnitt 4.1.3 erwähnt, ist aber das Löschen von Knoten aus einem k-d-Baum eine aufwändige Operation. Es ist daher sinnvoll auf das Löschen zu verzichten und den Baum stattdessen komplett neu aufzubauen. Diese Annahme wird auch durch Zeitmessungen bestätigt, demnach ist der Neuaufbau des k-d-Baums doppelt so schnell.

4.2 Grid

In diesem Abschnitt beschreibe ich die Grid-Datenstruktur. Zuerst erkläre ich in Abschnitt 4.2.1, wie der Raum durch ein Grid aufgeteilt wird. Die Abschnitte 4.2.2 und 4.2.3 befassen sich mit jeweils einer möglichen Implementierung dieser Datenstruktur. In Abschnitt 4.2.4 beschreibe ich das Verfahren, das die Raumaufteilung eines Grids für die Nachbarschaftssuche ausnutzt. Der Abschnitt 4.2.5 enthält die Beschreibung der Update-Operationen für die Grid-Datenstruktur.

4.2.1 Prinzip

Ein Grid benutzt eine sehr einfache, aber gleichzeitig effektive Methode für die Aufteilung des Raums. Der Raum wird wie durch ein Gitter in mehrere Zellen aufgeteilt. Einzelne Zellen überschneiden sich nicht und keine der Zellen ist in einer anderen Zelle enthalten. Jeder Agent im Raum befindet sich somit in genau einer Gridzelle. Gridzellen werden auf die so genannten *Buckets* abgebildet. Ein Bucket ist ein Container, der alle Agenten, die sich in einer Zelle befinden, speichert. Je nach der Realisierung des Grids können Buckets auch Inhalte von mehreren Zellen enthalten. Es gibt auch Grid-Varianten, die zulassen, dass eine Zelle mehreren Buckets zugeordnet werden kann [12].

In diesem Abschnitt wird ein so genanntes *gleichmäßiges Grid* (engl.: fixed oder uniform grid) beschrieben. Dieses teilt den Raum so auf, dass alle Zellen gleich groß sind. In einem dreidimensionalen Raum hat eine Zelle die Form eines Quaders. Die Größe einer Zelle wird durch die Kantenlänge des Quaders bestimmt.

Die Zuordnung zwischen den Zellen und den Buckets ist statisch und ermöglicht ein schnelles Auffinden des Buckets, in dem ein Agent gespeichert ist, anhand der Koordinaten dieses

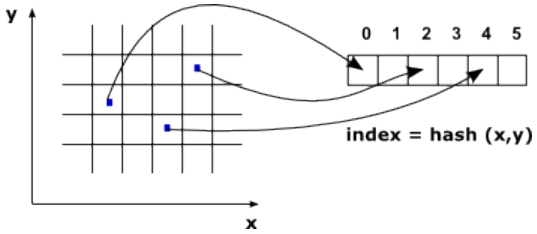


Abbildung 4.5: Ein zweidimensionales Grid, der mit Hilfe von einer Hashfunktion auf ein Array mit sechs Buckets abgebildet wird.

Agenten. Es ist möglich mehrere Zellen in einem Bucket zu speichern, eine Zelle darf aber nicht mehreren Buckets zugeordnet werden. Die Adresse des Buckets lässt sich über die *Zellkoordinaten* bestimmen. Die Zellkoordinaten werden berechnet, indem die Koordinaten eines Agenten in jeder Dimension ohne Rest durch die Kantenlänge einer Zelle geteilt werden [4]: wenn z. B. ein Grid den Raum in Zellen der Größe 10 unterteilt, dann befindet sich ein Agent mit den Koordinaten $(23, 7, 12)$ in der Gridzelle mit den Koordinaten $(2, 0, 1)$.

Ist der Raum, der durch das Grid verwaltet werden soll, begrenzt, so kann die Abbildung der Zellen auf die Buckets einfach realisiert werden. Da der Raum begrenzt ist, gibt es nur eine konstante Anzahl von Zellen. Man kann in so einem Fall genauso viele Buckets, wie es Zellen gibt, allozieren und jede Zelle genau einem Bucket zuordnen. Diese Realisierung schränkt jedoch die Anwendungsmöglichkeiten der Datenstruktur ein: falls ein Objekt den Raum, der durch das Grid aufgeteilt wurde, verlässt, kann es nicht mehr durch dieses Grid verwaltet werden. Die Anwendung muss also sicherstellen, dass die Objekte sich nur innerhalb des begrenzten Raums bewegen und die Größe des Raums muss vor der Initialisierung des Grids bekannt sein, damit der Raum entsprechend aufgeteilt werden kann. Diese Einschränkung kann beseitigt werden, indem man zulässt, dass mehrere Zellen auf ein einziges Bucket abgebildet werden können. Für diese Abbildung benötigt man eine Funktion, die aus den Koordinaten einer Zelle, die Adresse des Buckets berechnet. Das im Abschnitt 4.2.2 beschriebene Verfahren, verwendet dazu eine Hashfunktion. Ein anderes Verfahren, das ich im Abschnitt 4.2.3 beschreiben werde, wählt eine Region des Grids aus und weist jeder Zelle in dieser Region ein Bucket zu. Eine Zelle, die außerhalb dieser Region liegt, wird dann mit Hilfe des Modulo-Operators auf eine der Zellen in der ausgewählten Region projiziert.

4.2.2 Hashing

Um beliebig viele Gridzellen auf eine konstante Anzahl von Buckets abzubilden, kann eine Hashfunktion verwendet werden [4]. Eine Hashfunktion bildet jedes Element einer Menge auf ein Element einer kleineren Menge ab. Für die Abbildung von Gridzellen auf Buckets wird eine Hashfunktion benötigt, die aus den Koordinaten einer Zelle die Adresse des Buckets berechnet, in dem der Inhalt dieser Zelle gespeichert wird. Die Adresse des Buckets kann in konstanter Zeit berechnet werden. Die Abbildung 4.5 zeigt ein Grid der mit Hilfe einer Hashfunktion auf ein Array mit sechs Buckets abgebildet wird.

Bei dem Hashing-Ansatz belegt das Grid selbst keinen Speicher, es muss nur ein Array fester Größe alloziert werden, das die Buckets enthält [4]. Die Anzahl der Buckets in diesem Array ist unabhängig von der Größe des Raums, der durch das Grid verwaltet wird. Der Raum muss daher nicht auf eine bestimmte Größe beschränkt werden.

Buckets sind als Objekte der STL-Klasse[1] `vector` realisiert. Diese Klasse implementiert ein dynamisches Array. `vector` ermöglicht einen direkten wahlfreien Zugriff auf die einzelnen Elemente, aber anders als bei herkömmlichen Arrays, kann die Größe des Containers dynamisch geändert werden. Ich habe mich für diesen Container-Typ vor allem deswegen entschieden, weil das Iterieren über einen Vektor schneller ist als das Iterieren über eine Liste [15]. Die Geschwindigkeit der Iteration über die Elemente des Buckets spielt besonders bei der Nachbarschaftsuche eine wichtige Rolle. Außerdem sagt die STL-Dokumentation [14], dass `vector` einer Liste oder einem ähnlichen Container in den meisten Fällen vorzuziehen ist.

Das Auffinden eines Agenten mittels seiner Koordinaten ist wie bei k-d-Bäumen (vergleiche Abschnitt 4.1.1) sehr einfach. Zuerst berechnet man die Koordinaten der Zelle in der sich der Agent befindet, indem man die Koordinaten des Agenten durch die Größe einer Zelle teilt. Die Hashfunktion berechnet dann aus den Koordinaten der Zelle den Index im Array mit den Buckets. Danach prüft man jeden in dem Bucket gespeicherten Agenten, bis man den gesuchten Agenten gefunden hat. Der Aufwand für die Suche nach dem richtigen Bucket hängt von der Dimensionalität des Raums ab und ist daher bei Räumen mit nur wenigen Dimensionen sehr schnell. Das Bucket selbst muss linear durchsucht werden. Die Komplexität für Suche nach einem bestimmten Agenten in einem Bucket beträgt daher sowohl im schlimmsten (worst case) als auch im durchschnittlichen Fall (average case) $O(B)$ [6], wobei B die Anzahl der im Bucket gespeicherten Agenten ist.

Hashfunktion

Die Anzahl der Gridzellen kann viel größer sein als die Anzahl der Buckets. Es lässt sich daher nicht vermeiden, dass ein Bucket Objekte aus mehr als einer Zelle speichern muss. Wenn in einem Bucket jedoch zu viele Objekte abgelegt werden, verlangsamt sich die Suche nach einem bestimmten Objekt in diesem Bucket: der Aufwand für die Suche steigt linear mit der Anzahl der Objekte in diesem Bucket. Die Hashfunktion muss daher sicherstellen, dass die Zellen einigermaßen gleichmäßig auf die Buckets verteilt werden. In dieser Arbeit habe ich eine Hashfunktion verwendet, die sehr effizient ausgewertet werden kann und die Daten dabei relativ gleichmäßig auf die Buckets verteilt [17]:

$$hash(x, y, z) = (x * p_1 \textbf{ xor } y * p_2 \textbf{ xor } z * p_3) \textbf{ mod } n$$

p_1, p_2 und p_3 sind dabei Primzahlen mit folgenden Werten: 73856093, 19349663, 83492791. n ist die Anzahl der Buckets, auf die die Gridzellen abgebildet werden.

Einfügen neuer Daten

Um einen Agenten in die Grid-Datenstruktur einzufügen, muss zunächst das Bucket bestimmt werden, in dem die Koordinaten dieses Agenten abgelegt werden sollen. Dazu werden zuerst die Koordinaten der Gridzelle bestimmt, in der sich der Agent befindet. Diese Koordinaten werden an die Hashfunktion übergeben, die aus diesen Koordinaten den Index im Array mit

[1] STL (*Standard Template Library*) ist eine generische C++-Bibliothek, die dem Programmierer viele einfache Algorithmen und Datenstrukturen zur Verfügung stellt.

Buckets berechnet. Das Einfügen des Agenten in einen Bucket kann meistens[2] in konstanter Zeit durchgeführt werden. Der Aufwand für die Berechnung des Hashwertes beträgt $O(d)$, wobei d die Anzahl der Dimensionen ist. Da die Anzahl der Dimensionen aber mit $d = 3$ konstant ist, ergibt der Gesamtaufwand für das Einfügen eines Objekts in ein Grid im durchschnittlichen Fall $O(1)$.

Löschen von Daten

Beim Löschen eines Agenten aus der Grid-Datenstruktur, muss wie beim Einfügen, zuerst der richtige Bucket bestimmt werden. Das Bucket wird wie beim Einfügen (vergleiche 4.2.2), anhand der Koordinaten des Agenten bestimmt. Bevor ein Agent entfernt werden kann, muss zunächst seine Position innerhalb des Buckets bestimmt werden. Wurde der Agent in dem Bucket gefunden, kann er aus dem Bucket gelöscht werden.

Das Bucket, aus dem ein Element gelöscht werden soll, kann in konstanter Zeit bestimmt werden. Falls in dem Bucket insgesamt B Agenten gespeichert sind und der Agenten mit dem Index i gelöscht werden muss, benötigt man i Schritte um den Agenten mit linearer Suche zu finden. Dazu kommt der Aufwand für das Verschieben von $B - i$ Agenten, um die entstandene Lücke zu füllen. Der Aufwand für das Entfernen eines Agenten beträgt also $O(B)$.

4.2.3 Modulo-Grid

Ein anderes Verfahren für die Abbildung von beliebig großen Räumen auf ein Array mit Buckets, wählt einen Teilbereich des Grids und weist jeder Gridzelle in diesem Teilbereich genau ein Bucket zu. Zellen, die außerhalb des ausgewählten Grid-Abschnitts liegen, werden mittels Modulo-Operators auf diesen Grid-Ausschnitt projiziert.

Die Abbildung 4.6 zeigt ein zweidimensionales Grid und die Buckets, auf die die Zellen dieses Grids abgebildet werden. Die Zellen aus dem rot umrandeten Bereich des Grids werden direkt auf die Buckets abgebildet. Der ausgewählte Bereich enthält n Zellen in jeder Dimension. In diesem Beispiel mit $n = 3$ sind es insgesamt $n * n = 9$ Zellen. Die Anzahl der Buckets ist gleich der Anzahl der Zellen in diesem Bereich. Der Inhalt einer Gridzelle mit den Koordinaten (x, y) wird in dem Bucket mit dem Index $y * n + x$ gespeichert. Beispielsweise werden alle Objekte, die sich in der Zelle $(1, 2)$ befinden, in dem Bucket $2 * 3 + 1 = 7$ abgelegt. In einem dreidimensionalen Raum wird der Index des Buckets analog über $z * n^2 + y * n + x$ berechnet.

Gridzellen, die außerhalb der rot-umrundeten Bereichs liegen werden wie folgt auf die Zellen innerhalb des ausgewählten Bereichs abgebildet: jede Koordinate der Zelle wird durch die Anzahl der Zellen in einer Dimension geteilt. Der Rest, der bei der Ganzzahlendivision entsteht, ergibt die Koordinate einer Zelle im ausgewählten Bereich. Auf diese Weise wird beispielsweise die Gridzelle mit den Koordinaten $(5, 1)$ auf die Zelle mit den Koordinaten $(5 \textbf{ mod } 3 = 2,\ 1 \textbf{ mod } 3 = 1)$ projiziert. Die Inhalte beider Zellen werden im selben Bucket gespeichert: im Bucket mit dem Index 7.

[2] Es kann vorkommen, dass beim Einfügen eines neuen Elements festgestellt wird, dass der Vektor bereits voll belegt ist und vergrößert werden muss, um ein neues Element aufnehmen zu können. In so einem Fall muss zusätzlich der Aufwand für die Erweiterung des Arrays berücksichtigt werden.

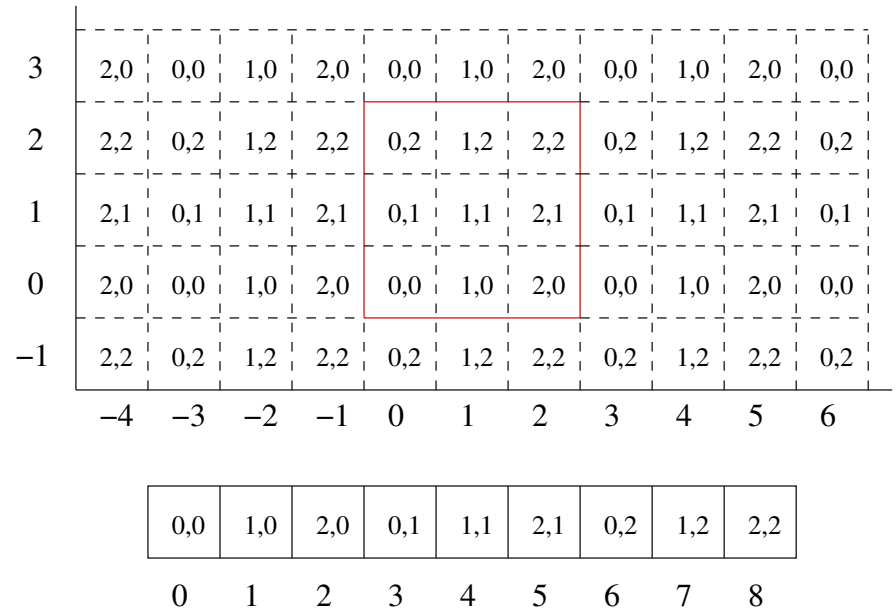


Abbildung 4.6: Ein zweidimensionales Grid und acht Buckets, auf die die Gridzellen im rot-umrundeten Bereich direkt abgebildet werden.

Falls eine Gridzelle links oder unterhalb des ausgewählten Bereichs liegt, hat sie negative Koordinaten. Um diese Gridzelle auf den ausgewählten Gridbereich abzubilden, muss zusätzlich nach der Anwendung des Modulo-Operators n zu der negativen Koordinate addiert werden. Hat eine Gridzelle z. B. die Koordinaten $(-4, 1)$ ergibt sich für die x-Koordinate mit x **mod** $n = -4$ **mod** 3 der Wert -1. Addiert man n zu diesem Wert, bekommt man die x-Koordinate der Gridzelle im ausgewählten Bereich: $-1 + 3 = 2$. Die Gridzelle mit den Koordinaten $(-4, 1)$ entspricht also der Zelle $(2, 1)$ im rot-markierten Grid-Bereich.

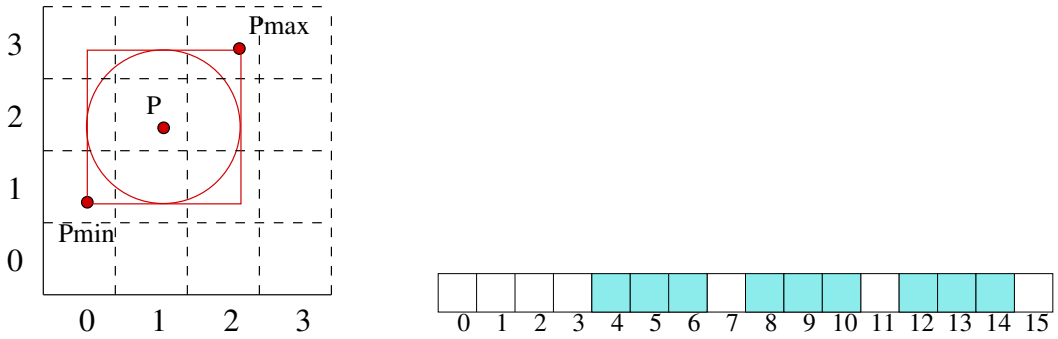
Um einen Agenten im Grid anhand seiner Koordinaten zu finden, müssen zunächst die Koordinaten der Zelle, in der sich dieser Agent befindet, bestimmt werden. Danach wird mithilfe des oben beschriebenen Verfahrens der Index des Buckets, in dem sich dieser Agent befinden muss, bestimmt. Dieses Bucket wird dann durchsucht, bis der gesuchte Agent gefunden wird. Wenn der komplette Inhalt des Buckets durchsucht wurde, der gesuchte Agent aber nicht gefunden wurde, bedeutet es, dass dieser Agent nicht im Grid enthalten ist.

Analog geht man beim Einfügen und Entfernen eines Agenten vor. Man bestimmt zuerst mithilfe der Koordinaten des Objekts das richtige Bucket. Danach kann der Agent in das Bucket eingefügt oder aus dem Bucket gelöscht werden. Die Komplexität dieser Operationen stimmt mit der Komplexität der entsprechenden Operationen beim Hashing-Ansatz (vergleiche Abschnitt 4.2.2) überein.

4.2.4 Nachbarschaftssuche

Durch die Aufteilung des Raums in gleichgroße Zellen muss bei Nachbarschaftsabfragen nur ein Teil der in der Grid-Datenstruktur gespeicherten Agenten untersucht werden. Wenn sich eine Gridzelle mit dem Suchraum überschneidet werden alle Punkte, die in dieser Zelle liegen auf ihre Entfernung zu dem Mittelpunkt des Suchraums untersucht. Befindet sich eine

Gridzelle dagegen außerhalb des Suchraums, kann sie bei der Suche nach allen Agenten, die sich in diesem Suchraum befinden, ignoriert werden.



(a) Grid mit der Suchregion

(b) Buckets (farbig markierte Buckets speichern die Gridzellen in der Suchregion)

Abbildung 4.7: Suche in einem zweidimensionalen Grid.

Um zu bestimmen, welche Gridzellen durchsucht werden müssen, wird für jede Dimension der minimale und der maximale Wert, den die entsprechende Koordinate eines Agenten innerhalb des Suchraums haben kann, berechnet. Die Entfernung zwischen dem Mittelpunkt des Suchraums und einem Agenten, das sich in dem Suchraum befindet, darf höchstens so groß wie der Radius r sein. Die minimalen und die maximalen Werte für jede Dimension berechnet man, indem man für jede Dimension den Radius r von der entsprechenden Koordinate des Mittelpunkts der Suchregion abzieht, bzw. den Radius zu dieser Koordinate addiert. Für die x-Koordinate ergeben sich auf diese Weise folgende Werte: $x_{min} = x - r$, $x_{max} = x + r$. Ein Agent dessen x-Koordinate kleiner als x_{min} bzw. großer als x_{max} ist, befindet sich eindeutig außerhalb der Suchregion und kann bei der Nachbarschaftssuche ignoriert werden.

Die minimalen Werte für jede Dimension bilden zusammen den Punkt P_{min}. Die maximalen Werte bilden den Punkt P_{max}. Wie in der Abbildung 4.7(a) zu sehen ist, spannen die Gridzellen, in denen sich diese beiden Punkte befinden, einen Sektor auf dem Grid auf, der die Suchregion komplett einschließt. Für die Auswertung der Nachbarschaftsabfrage sind also lediglich die Gridzellen innerhalb dieses Sektors von Bedeutung. Dementsprechend muss nur ein Teil der Buckets untersucht werden. In dem Beispiel aus der Abbildung 4.7 müssen daher nur neun von sechzehn Buckets durchsucht werden. Die Gridzellen außerhalb dieses Sektors können ignoriert werden, da sie sich nicht mit dem Suchraum überschneiden und somit nur Agenten enthalten können, die weiter als r von dem Mittelpunkt der Suchregion entfernt sind.

Für jede Gridzelle, die sich mit dem Suchraum überschneidet, muss der Index des entsprechenden Buckets berechnet werden. Dieser Index kann entweder über die Hashfunktion (Vergleiche Abschnitt 4.2.2) oder mithilfe des Modulo-Operators bestimmt werden. Für jeden Agenten in so einem Bucket muss die Entfernung zum Mittelpunkt des Suchraums berechnet werden. Diese Prüfung muss aus zwei Gründen gemacht werden. Zum einen ist der Suchraum kreisförmig. Der Sektor auf dem Grid, der den Suchraum einschließt, ist aber rechteckig. Wie die Abbildung 4.7 verdeutlicht können daher einige Zellen am Rand dieses Sektors sowohl Agenten, die sich innerhalb des Suchraums befinden, als auch Agenten, die weiter als r vom Mittelpunkt entfernt sind, enthalten. So wird z. B. die Gridzelle mit den Koordinaten $(0, 1)$ nur zum Teil von der Suchregion überdeckt und kann daher auch Objekte enthalten, die sich nicht in der Suchregion befinden. Das entsprechende Bucket muss aber dennoch durchsucht werden, weil es möglicherweise Objekte speichert, deren Entfernung vom Mittelpunkt der Suchregion kleiner

als r ist. Zum anderen können Buckets mehrere Gridzellen enthalten. Es ist daher möglich das ein Bucket neben Agenten aus einer Zelle, die sich mit dem Suchraum überschneiden, auch Agenten aus Zellen, die außerhalb der Suchregion liegen, speichert.

Da während einer Nachbarschaftssuche nicht die komplette Datenstruktur durchsucht wird, sondern nur die Gridzellen, die sich mit der Suchregion überschneiden, hängt der Laufzeitaufwand für die Nachbarschaftssuche in erster Linie nicht von der Gesamtzahl der Agenten in der Datenstruktur, sondern von der Anzahl der Zellen, die untersucht werden und der Anzahl der Agenten, die in den entsprechenden Buckets gespeichert sind. Sean Mauch gibt in seiner Arbeit [8] den Laufzeitaufwand für die Suche nach den Nachbarn eines Agenten wie folgt an: $O(J + I)$. J ist dabei die Anzahl der Gridzellen, die sich mit der Suchregion überschneiden und I die Anzahl der Agenten, die in den entsprechenden Buckets gespeichert sind und bei der Nachbarschaftssuche untersucht werden.

4.2.5 Updates

Ein gleichmäßiges Grid hat die Eigenschaft, dass seine Struktur, die Position und die Größe der Gridzellen immer gleich bleibt, unabhängig davon, ob die im Grid gespeicherten Objekte ihre Position ändern oder nicht. Anders als bei k-d-Bäumen muss die Struktur des Grids nicht ständig an neue Positionen der gespeicherten Objekte angepasst werden. Dank dieser Eigenschaft können effiziente Update-Operationen für die Grid-Datenstruktur einfach realisiert werden.

Um die Position eines gespeicherten Agenten zu aktualisieren, muss zunächst dieser Agent in der Datenstruktur gefunden werden. Wie im Abschnitt 4.2.3 beschrieben, bestimmt man dazu mittels der alten Position des Agenten das Bucket, in dem dieser Agent gespeichert ist. Hat man den Agenten in diesem Bucket gefunden, werden seine alten Koordinaten einfach durch neue ersetzt. An der Struktur des Grids muss nichts geändert werden. Es muss aber natürlich berücksichtigt werden, dass ein Agent aus einer Gridzelle in die andere wechseln kann. In diesem Fall muss der Agent nur in ein anderes Bucket verschoben werden. Der Index des neuen Buckets lässt sich über die neuen Koordinaten des Agenten berechnen.

Parallelisierung der Update-Phase

Die Tatsache das die Struktur eines gleichmäßigen Grids durch die Update-Operationen nicht geändert wird, erleichtert die Parallelisierung der Update-Phase. Während der Update-Phase ändern sich nur die einzelnen Elemente der Buckets. Die Änderungen in unterschiedlichen Buckets sind unabhängig voneinander, so dass man einzelne Buckets auf die vorhandenen Threads gleichmäßig verteilen kann. Jeder Thread aktualisiert dann die Koordinaten der Agenten aus seinen Buckets. Diese Art der Parallelität wird *Datenparallelität* genannt und ist bestens für die Realisierung mit OpenMP geeignet.

Bei der Parallelisierung muss jedoch beachtet werden, dass Agenten sich über die Grenzen der Gridzellen bewegen können. In einem solchen Fall muss ein Agent von einem Bucket in ein anderes verschoben werden. In einem sequenziellen Programm stellt es kein Problem dar. Werden die Buckets aber gleichzeitig von verschiedenen Threads bearbeitet, kann dies zu zeitkritischen Abläufen führen.

Eine Möglichkeit, dieses Problem zu lösen, ist die Verwendung von Lock-Variablen, die den Zugriff auf die Buckets synchronisieren. Durch die Synchronisation der Zugriffe auf ein Bucket über Lock-Variablen entsteht aber ein zusätzlicher Mehraufwand. Ich habe mich daher für eine andere Lösung dieses Problems entschieden: Agenten, die während einer Update-Phase in andere Buckets verschoben werden müssen, werden in speziellen Listen gesammelt. Jeder Thread verfügt über so eine Liste. Falls ein Thread während der Ausführung einer Update-Operation feststellt, dass ein Agent in ein anderes Bucket verschoben werden muss, fügt er diesen Agenten in seine Liste ein. Am Ende der Update-Phase werden die Agenten aus den Listen aller Threads sequentiell in die richtigen Buckets kopiert. Obwhol das Kopieren der Agenten in die richrigen Bucket auch parallel erfolgen kann, habe ich auf die Parallelisierung dieses Teils der Update-Phase verzichtet. Die Anzahl der Agenten, die während einer Update-Phase die Buckets wechseln, ist gering: durchschnittlich wechseln in jedem Simulationsschritt 0,5 bis 2 Prozent der Agenten die Buckets. Der sequentielle Anteil ist somit im Vergleich zum Rest der Update-Phase recht klein, so dass sich die Parallelisierung dieses Teils der Update-Phase nicht lohnt.

4.3 Cell array with binary search

Die in diesem Abschnitt beschriebene Datenstruktur kombiniert zwei verschiedene Techniken, um räumliche Such-Operationen zu beschleunigen [8]. Wie bei den, im Abschnitt 4.2 beschriebenen Grids, wird der Raum in Zellen gleicher Größe aufgeteilt. Anders als beim gleichmäßigen Grid wird der Raum jedoch nicht in drei sondern nur in zwei von insgesamt drei Dimensionen aufgeteilt. Die letzte Dimension wird verwendet, um die Inhalte der Buckets zu sortieren.

Der Abschnitt 4.3.1 befasst sich mit der Raumaufteilung durch cell array with binary search. Das Einfügen neuer Daten in diese Datenstruktur wird im Abschnitt 4.3.2 beschrieben. Wie die Daten aus der Datenstruktur gezielt gelöscht werden können, beschreibe ich im Abschnitt 4.3.3. Im Abschnitt 4.3.4 beschreibe ich, wie diese Datenstruktur für die räumliche Suche verwendet werden kann. Der Abschnitt 4.3.5 befasst sich mit den Update-Operationen.

4.3.1 Prinzip

Wie ein gleichmäßiges Grid teilt auch diese Grid-Variante den Raum in Zellen gleicher Größe auf. Anders als bei gleichmäßigen Grids wird das Grid jedoch nicht in allen k Dimensionen des Raums, sondern nur in $k-1$ Dimensionen aufgespannt. Wie bei den vorher beschriebenen Grid-Varianten werden die Inhalte der Gridzellen in Buckets gespeichert. Die letzte Dimension des Raums wird verwendet, um die Inhalte der Buckets zu sortieren. In einem dreidimensionalen Raum wird das Grid also nur in den ersten zwei Dimensionen aufgespannt: x und y. In einem Bucket werden die Agenten nach ihrer z-Koordinate sortiert.

Die Abbildung 4.8 zeigt einen zweidimensionalen Raum der auf diese Weise aufgeteilt wurde. Der Raum wird in der Dimension x in Zellen mit der Kantenlänge $l = 10$ aufgeteilt. Das Grid erstreckt sich auf den gesamten Raum, es wird jedoch, wie bei der im Abschnitt 4.2.3 beschriebenen Grid-Variante, nur ein bestimmter Bereich des Grids direkt auf die Buckets abgebildet. In diesem Beispiel entspricht dieser Bereich dem Intervall $[0, 30]$ auf der x-Achse. Es befinden sich also drei Gridzellen in diesem Beireich. Jede dieser drei Gridzellen wird direkt auf ein Bucket abgebildet. Wie bei gleichmäßigen Grids wird der Index des Buckets, in dem

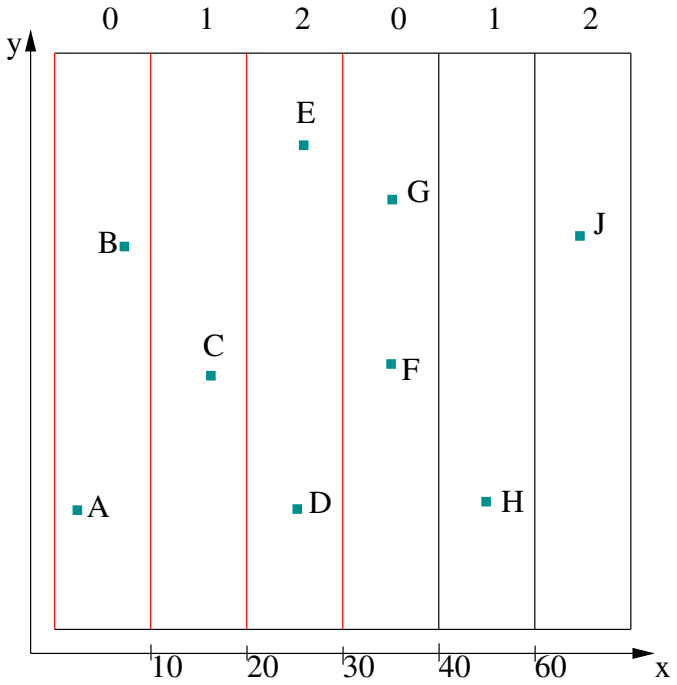


Abbildung 4.8: Cell array with binary search.

die Agenten aus einer Gridzelle gespeichert werden sollen, aus den Koordinaten dieser Zelle berechnet. Der Punkt A aus der Abbildung 4.8 hat eine x-Koordinate, die größer als 0 und kleiner als 10 ist. Teilt man die x-Koordinate von A durch die Kantenlänge einer Gridzelle $l = 10$, hat man die Koordinate der Zelle in der A liegt bestimmt: A liegt in der Zelle 0 und muss daher im Bucket mit dem Index 0 gespeichert werden.

Gridzellen außerhalb des Bereichs werden mit Hilfe des Modulo-Operators auf die Zellen innerhalb des ausgewählten Bereichs projiziert: Die x-Koordinate des Punktes H aus der Abbildung 4.8 hat einen Wert zwischen 40 und 50 und liegt daher in der Zelle 4. Teilt man die Koordinate der Gridzelle, in der H liegt, durch die Anzahl der Zellen in jeder Dimension (in diesem Beispiel 3), ergibt sich ein Rest von 1. Der Punkt H muss also im selben Bucket gespeichert werden, wie die Punkte aus der Zelle mit der Koordinate 1.

Die Tatsache, dass der Inhalt eines Buckets nach der letzten Koordinate sortiert wird, erleichtert die Suche nach einem bestimmten Agenten innerhalb eines Buckets. Ein unsortierter Bucket muss im schlimmsten Fall komplett durchsucht werden, bevor ein Agent gefunden wird. Sind die im Bucket gespeicherten Agenten aber sortiert kann binäre Suche angewandt werden. Binäre Suche ist ein Algorithmus das ein Array rekursiv nach dem Schema "Teile und Herrsche" durchsucht. Zuerst wird das mittlere Element des Arrays überprüft. Ist das Element kleiner als das gesuchte, wird in der hinteren Hälfte des Arrays weitergesucht. Ist das mittlere Element größer, muss der gesuchte Wert in der vorderen Hälfte des Arrays gespeichert sein. Die Suche wird rekursiv nach dem selben Muster fortgesetzt, bis das gesuchte Element entweder gefunden wurde oder sein Fehlen festgestellt wurde. Die Komplexität einer binären Suche beträgt $O(\log n)$.

4.3.2 Einfügen neuer Daten

Da Agenten, die in einem Bucket gespeichert werden, nach ihrer letzten Koordinate sortiert sind, muss beim Einfügen eines neuen Agenten in ein Bucket darauf geachtet werden, dass die Ordnung der Agenten im Bucket erhalten bleibt. Bevor ein neuer Agent a eingefügt werden kann, muss in diesem Bucket der erste Agent c gefunden werden, dessen letzte Koordinate einen Wert hat, der größer oder gleich dem entsprechenden Wert von a ist. a wird dann vor c in das Bucket eingefügt. Da der Inhalt eines Buckets sortiert ist kann bei der Suche nach c binäre Suche angewandt werden. Insgesamt ist aber das Einfügen eines neuen Agenten bei dieser Datenstruktur aufwändiger als bei gleichmäßigen Grids. Das Bucket, in den ein Element eingefügt werden muss, kann zwar wie bei gleichmäßigen Grids in einer konstanten Zeit gefunden werden. Hinzu kommt aber die Suche nach einer geeigneten Position innerhalb des Buckets. Der Aufwand für diese Suche beträgt $O(\log B)$. B ist dabei die Anzahl der Agenten, die bereits in dem Buckets gespeichert sind. Zusätzlich müssen c und alle Elemente, die nach c kommen, um eine Stelle nach hinten verschoben werden, um Platz für das neue Element, das vor c eingefügt werden muss, zu schaffen. Wieviele Elemente verschoben werden müssen, hängt von der Position von c innerhalb des Buckets ab. Im schlimmsten Fall, wenn der Agent ganz vorne in das Bucket eingefügt werden muss, müssen alle B Agenten, die bereits im Bucket gespeichert sind, um jeweils eine Stelle nach hinten verschoben werden. Der Aufwand für das Einfügen des Agenten beträgt in diesem Fall $O(B)$.

4.3.3 Löschen von Daten

Das Löschen eines Agenten aus den Buckets in einem cell array with binary search verläuft nach dem gleichen Muster, wie bei Grids. Zuerst wird das richtige Bucket mithilfe der Koordinaten des Agenten gefunden und dann der Agent aus dem Bucket gelöscht. Bei gleichmäßigen Grids kann ein bestimmter Agent innerhalb eines Buckets nur mittels linearer Suche gefunden werden. Bei cell array with binary search kann dafür die binäre Suche verwendet werden. Im schlimmsten Fall beträgt der Aufwand für das Löschen eines Agenten aus der Datenstruktur wie bei gleichmäßigen Grids $O(B)$.

4.3.4 Nachbarschaftssuche

Auch diese Datenstruktur beschleunigt die Suche nach allen Nachbarn eines Agenten, indem die Anzahl der Agenten, die bei dieser Suche überprüft werden müssen, verringert wird. Wie bei allen Grids müssen auch hier nur die Gridzellen untersucht werden, die sich mit der Suchregion überschneiden. Das Verfahren für die Bestimmung dieser Gridzellen ähnelt dem im Abschnitt 4.2.4 beschriebenen Verfahren, das bei gleichmäßigen Grids angewandt wird. Auch hier werden für jede Dimension die minimalen und maximalen Werte der Koordinaten, die ein Agent im Suchraum haben kann, bestimmt. Da das Grid aber nur in $k-1$ von insgesamt k Dimensionen aufgespannt wird, werden für die Bestimmung der Zellen, die überprüft werden müssen, nur die minimalen bzw. maximalen Werte der ersten $k-1$ Dimensionen verwendet. Über die Koordinaten dieser Zellen werden die Buckets bestimmt, die durchsucht werden müssen.

Da die Inhalte der Buckets sortiert sind, müssen, anders als bei den im Abschnitt 4.2 beschriebenen Grid-Varianten, nicht alle Agenten, die in einem Bucket gespeichert sind, sondern nur ein Teil davon überprüft werden. Der minimale und der maximale Wert der k-ten Dimension

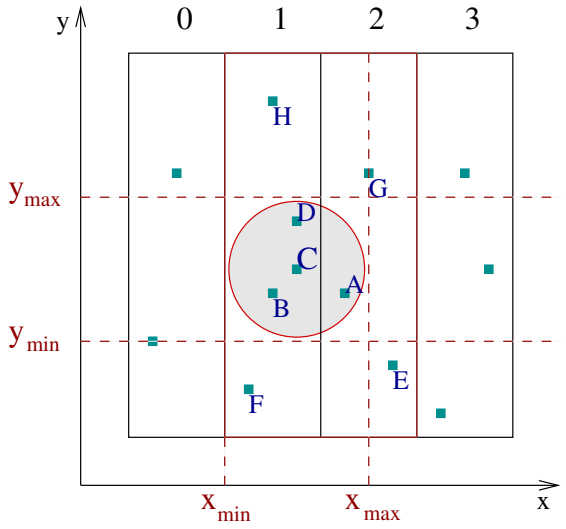


Abbildung 4.9: Suche in einem zweidimensionalen cell array with binary search.

wird verwendet, um zu bestimmen, welcher Abschnitt eines Buckets für die Suche relevant ist.

Die Abbildung 4.9 zeigt einen zweidimensionalen Raum, in dem alle Punkte, die im Radius r um den Punkt C liegen, gesucht werden. Der Raum ist durch ein Grid in mehrere separate Zellen aufgeteilt. Da der Raum zweidimensional ist, wird das Grid nur in einer Dimension aufgespannt. In der Abbildung sind vier Zellen dieses Grids dargestellt. Jede dieser Gridzellen wird in jeweils einem Bucket mit dem Index zwischen 0 und 3 gespeichert. Gesucht werden alle Agenten, die sich in dem rot umrundeten Bereich befinden. Da die Suchregion sich mit den Gridzellen 1 und 2 überschneidet, müssen nur die Buckets 1 und 2 untersucht werden. Die restlichen Buckets können ignoriert werden.

Das Bucket mit dem Index 1 enthält die Agenten F, B, C, D und H. Es müssen aber nicht alle fünf Agenten überprüft werden, um alle Nachbarn von C zu finden. Die y-Koordinate eines Agenten, der in der Suchregion liegt, muss im Wertebereich zwischen y_{min} und y_{max} liegen. Man sucht daher den ersten Agenten im Bucket mit einer y-Koordinate die größer oder gleich y_{min} ist. In diesem Beispiel ist das der Agent B. Alle Agenten, die im Bucket 1 vor B liegen, wie z. B. F, können bei der Suche übersprungen werden. Angefangen mit dem Agenten B, überprüft man jeden weiteren Agenten in diesem Bucket auf seine Entfernung zum Mittelpunkt der Suchregion (C). Sobald man auf einen Agenten stößt, dessen y-Koordinate größer als y_{max} ist, kann die Suche abgebrochen werden. In diesem Beispiel ist es der Agent H. Da die Agenten in einem Bucket nach der y-Koordinate sortiert sind, haben alle Agenten die nach H kommen, eine y-Koordinate, die größer ist als die y-Koordinate von H und liegen somit offensichtlich nicht in der Suchregion.

Wie bei einem gleichmäßigen Grid hängt die Laufzeit der Nachbarschaftssuche auch bei dieser Datenstruktur von der Anzahl der Zellen in der Suchregion. Mauch [8] gibt den Aufwand für die Laufzeit mit $O(N(J * \log(N/M^{2/3}) + I'))$ an. $M^{2/3}$ ist dabei die Anzahl der Buckets die untersucht werden müssen. J ist die Anzahl der Zellen, die sich mit der Suchregion überschneiden und I' die Anzahl der Agenten in den entsprechenden Buckets.

4.3.5 Updates

Auch beim cell array with binary search wird die Struktur des Grids durch die Bewegungen der Agenten nicht beeinflusst. Die Aktualisierung der Positionen von Agenten erfolgt im Wesentlichen nach dem selben Muster wie bei gleichmäßigen Grids (vergleiche 4.2.5). Beim cell array with binary search muss allerdings zusätzlich darauf geachtet werden, dass die Ordnung innerhalb einzelner Buckets erhalten bleibt. Weil sich nicht alle Agenten immer in dieselbe Richtung bewegen, kann es durchaus vorkommen, dass zwei Agenten, die in ein und demselben Bucket gespeichert sind, nach einem Simulationsschritt ihre Plätze im Bucket tauschen müssen. Wenn z. B. der Agent C aus der Abbildung 4.9 sich in Richtung des Agenten H bewegt und der Agent D sich gleichzeitig in die entgegengesetzte Richtung bewegt, dann ist die y-Koordinate von C irgendwann größer als die von D. Da C aber im Bucket 1 vor D gespeichert ist, wird die Ordnung im Bucket 1 durch diese Positionsänderung verletzt.

Um die Ordnung wiederherzustellen, müssen die Bucket, nachdem die Aktualisierung der Agenten-Positionen abgeschlossen ist, neu sortiert werden. Das Sortieren der Buckets kann parallel von mehreren Threads durchgeführt werden. Dabei werden die Buckets unter den Threads so aufgeteilt, das ein Bucket komplett von einem einzigen Thread sortiert wird. Einzelne Agenten müssen beim Sortieren nicht zwischen verschiedenen Buckets verschoben werden, so dass beim parallelen Sortieren der Buckets keine Synchronisation notwendig ist.

4.4 Vergleich

In diesem Abschnitt werden die vorgestellten Datenstrukturen bezüglich des Laufzeitaufwandes der einzelnen Operationen verglichen. Zusätzlich wird die Komplexität der entsprechenden Operationen für den Brute-Force-Ansatz, bei dem die Daten der Agenten in einer Liste gespeichert werden, angegeben.

In der Tabelle 4.1 sind einige Operationen, wie das Einfügen oder Löschen von Daten aus den Datenstrukturen aufgelistet. Die Tabelle enthält für jede dieser Operationen den durchschnittlichen Laufzeitaufwand in der O-Notation (average case). N bezeichnet dabei die Anzahl der Agenten, die in der Datenstruktur gespeichert sind und B ist die Anzahl der Agenten in einem Bucket.

Datenstruktur	Suchen	Einfügen	Löschen
Brute-Force	$O(N)$	$O(1)$	$O(N)$
k-d-Baum	$O(\log N)$	$O(\log N)$	$O(\log N)$
Grid	$O(B)$	$O(1)$	$O(B)$
cell array	$O(\log B)$	$O(B)$	$O(B)$

Tabelle 4.1: Laufzeitaufwand für die Suche nach einem bestimmten Agenten, das Einfügen eines neuen Agenten und das Löschen eines Agenten aus der Datenstruktur.

Jede der drei aufgelisteten Operationen hat bei k-d-Bäumen eine logarithmische Komplexität. Bei Grids hängt der Laufzeitaufwand dieser Operationen von der Anzahl der Buckets und der Verteilung der Agenten im Raum. Nimmt man an, dass ein Grid die Agenten gleichmäßig auf M Buckets verteilt, kann B im durchschnittlichen Fall als $\frac{N}{M}$ ausgedrückt werden. Daran erkennt

man bereits, dass die Performance eines Grids mit der Anzahl der Buckets zusammenhängt. Es fällt auch auf, dass ein gleichmäßiges Grid beim Einfügen von Daten schneller sein müsste als cell array with binary search. Das liegt daran, dass die Buckets der letzteren Datenstruktur sortiert werden müssen, was einen zusätzlichen Aufwand bedeutet. Dieser Aufwand macht sich aber bei der Suche nach einem bestimmten Agenten bezahlt.

Datenstruktur	Aufbau	Nachbarschaftssuche	Update
Brute-Force	$O(N)$	$O(N^2)$	$O(N^2)$
k-d-Baum	$O(N(\log N))$	$O(\log N + F)$	$O(N(\log N))$
Grid	$O(M + N)$	$O(N(J + I))$	$O(N)$
cell array	$O(M + N^2)$	$O(N(J * \log(N/M) + I))$	$O(N * \log N)$

Tabelle 4.2: Laufzeitaufwand für den Aufbau einer Datenstruktur, die Such- und die Updatephase.

Die Tabelle 4.2 gibt für jede Datenstruktur einen Überblick über den durchschnittlichen Laufzeitaufwand für den Aufbau, die Such- und die Updatephase. Bei den Angaben für den Laufzeitaufwand werden folgenden Größen verwendet:

- N: Anzahl der Agenten in der Datenstruktur
- M: Anzahl der Buckets
- F: durchschnittliche Anzahl der gefundenen Agenten
- J: durchschnittliche Anzahl der Gridzellen in der Suchregion
- I: Anzahl der Agenten in den Gridzellen, die sich mit der Suchregion überschneiden

Aus der Tabelle 4.2 erkennt man, dass der Brute-Force-Ansatz beim Aufbau der Datenstruktur der schnellste sein müsste. Auch der Laufzeitaufwand für die Updatephase ist bei diesem Ansatz linear. Andere Ansätze, mit Ausnahme der Grid-Datenstruktur, haben eine höhere Komplexität für die Updatephase. Wie ich im Kapitel 6 zeigen werde, hat jedoch die Nachbarschaftsuche den meisten Anteil an der Gesamtlaufzeit einer Simulation. Die Performance bei der Nachbarschaftssuche spielt daher für die Gesamtgeschwindigkeit der Simulation eine wesentlich größere Rolle als die Performance bei der Updatephase oder beim Aufbau der Datenstruktur. Der Aufwand für die Nachbarschaftssuche ist bei dem Brute-Force-Ansatz am größten.

Obwohl die Laufzeit bei der Nachbarschaftssuche bei anderen Datenstrukturen asymptotisch nicht wesentlich kleiner ist als beim Brute-Force-Ansatz, werden die Laufzeitmessungen im Kapitel 6 zeigen, dass einige Datenstrukturen bei der Nachbarschaftssuche deutlich besser abschneiden, als der Brute-Force-Ansatz.

5 Implementierung

Um die im Kapitel 4 vorgestellten Datenstrukturen für Nachbarschaftsabfragen in OpenSteer-Plug-ins zu testen, wurde jede dieser Datenstrukturen als eine C++-Klasse implementiert. In diesem Kapitel werde ich einige Details über die Implementierung der Datenstrukturen vorstellen.

5.1 k-d-Baum

In der Umsetzung des k-d-Baums wurden zwei Klassen implementiert: `kd_tree` und `kd_tree_node`.

5.1.1 kd_tree_node

Die Klasse `kd_tree_node` implementiert einen Knoten des k-d-Baums. Das UML-Diagramm in der Abbildung 5.1 zeigt die wichtigsten Methoden und Attribute dieser Klasse.

Jedes Objekt dieser Klasse speichert neben der Position (`position_`) und der Beschreibung (`reference_`) eines Agenten jeweils einen Verweis auf das linke und rechte Kindknoten. Die Verweise auf die beiden Kinder des aktuellen Knotens werden in dem Array `children_` gespeichert. Das Array enthält zwei Zeiger auf Objekte vom Typ `kd_tree_node`. Zusätzlich wird im Attribut `axis_` die Diskriminator-Koordinate des Knotens gespeichert.

Die Methode `route` bekommt als Parameter Koordinaten eines Agenten und entscheidet anhand dieser Koordinaten und des Diskriminators (`axis_`) in welchen der beiden Teilbäume dieses Knotens ein Agent mit diesen Koordinaten gehört.

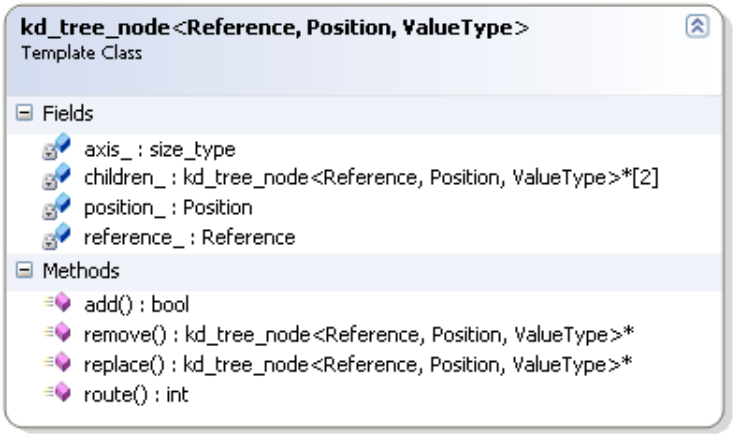


Abbildung 5.1: Klassendiagramm der Klasse `kd_tree_node`.

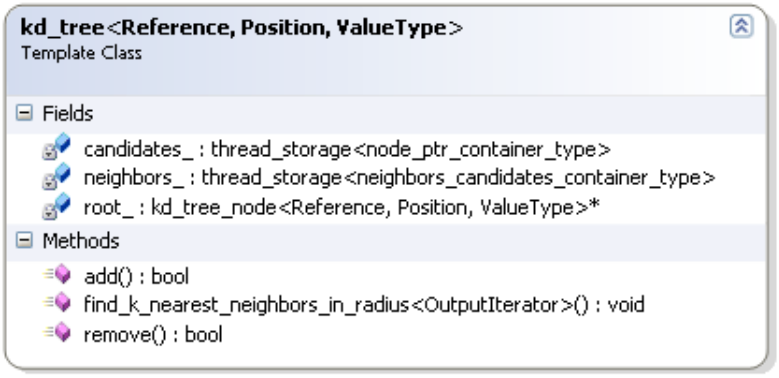


Abbildung 5.2: Klassendiagramm der Klasse `kd_tree`.

Die Methode `add` fügt einen Agenten in einen der Teilbäume des aktuellen Knotens ein. Dazu ruft diese Methode zunächst `route` mit den Koordinaten des neuen Agenten auf, um zu entscheiden, in welchem der beiden Teilbäume der Agent gespeichert werden soll. Falls dieser Teilbaum leer ist, wird ein neues `kd_tree_node` Objekt mit den Daten des neuen Agenten erzeugt und ein Zeiger auf dieses Objekt im Array `children_` gespeichert. Ist der Teilbaum dagegen nicht leer, wird die Methode `add` rekursiv auf diesem Teilbaum aufgerufen.

Für das Entfernen eines Agenten aus einem der Teilbäume stellt die Klasse `kd_tree_node` die Methode `remove` bereit. In dieser Methode wird zunächst mithilfe von `route` entschieden, in welchem der beiden Teilbäume der Agent, der gelöscht werden soll, gespeichert ist. Auf diesem Teilbaum wird `remove` rekursiv aufgerufen. Die Rekursion endet, wenn der aktuelle Knoten, den Agenten, der gelöscht werden soll, enthält. In diesem Fall wird `replace` aufgerufen, um einen Ersatzknoten zu finden. Zum Schluss wird ein Zeiger auf den Ersatzknoten zurückgegeben, damit der Aufrufer den Speicher für diesen Knoten freigeben kann.

Die Methode `replace` findet einen Ersatz für den Knoten, auf dem sie aufgerufen wurde. Der Ersatzknoten wird immer im rechten Teilbaum des aktuellen Knotens gesucht (vergleiche 4.1.3). Die Daten, die der aktuelle Knoten speichert (Beschreibung und die Position des Agenten), werden mit den Daten des Ersatzknotens überschrieben. Ist der Ersatzknoten kein Blattknoten, wird `replace` rekursiv auf dem Ersatzknoten aufgerufen, um einen Ersatz für den Ersatzknoten zu finden. Ist der Knoten, auf dem `replace` aufgerufen wurde, ein Blattknoten, muss er nicht ersetzt werden. In diesem Fall gibt `replace` einen Zeiger auf diesen Knoten zurück.

5.1.2 kd_tree

Das Klassendiagramm aus der Abbildung 5.2 zeigt die wichtigsten Methoden der Klasse `kd_tree`. Diese Klasse implementiert einen k-d-Baum. Jeder Knoten des k-d-Baums ist ein Objekt vom Typ `kd_tree_node`. Das Attribut `root_` speichert den Zeiger auf die Wurzel des Baums.

`candidates_` und `neighbors_` sind temporäre Container, die bei der Suche nach den k-nächsten Nachbarn verwendet werden. Jeder Thread verfügt über seine eigene Kopie dieser Container. Alternativ zu dieser Lösung wäre es möglich diese Container nicht global als Attribute der Klasse, sondern lokal in der Methode `find_k_nearest_neighbors_in_radius` zu deklarieren. In so einem Fall würde der Speicher für diese Container jedesmal beim Aufruf von `find_k_nearest_neighbors_in_radius` reserviert werden. Die Methode

`find_k_nearest_neighbors_in_radius` wird jedoch aus einer parallelen Region aufgerufen und häufige Reservierungen von Speicherblöcken aus einer parallelen Region würden sich negativ auf die Performance des Programms auswirken. Um die vielen Systemaufrufe für das Reservieren vom Hauptspeicher in einer parallelen Region zu vermeiden, wird der Speicher für die temporären Container `neighbors_` und `candidates_` nur einmal bei der Initialisierung des `kd_tree`-Objekts für jeden Thread, der an der Nachbarschaftssuche beteiligt ist, reserviert.

Die Methode `add` fügt einen neuen Agenten in die Datenstruktur ein, indem sie die gleichnamige Methode der Klasse `kd_tree_node` auf der Wurzel des Baums aufruft: `root_.add (data, pos)`.

Um einen Agenten aus der Datenstruktur zu entfernen, stellt die Klasse `kd_tree` die Methode `remove()` zur Verfügung. Diese ruft wiederum die gleichnamige Methode der Klasse `kd_tree_node` auf der Wurzel des Baums auf.

Die k-nächsten Nachbarn eines Agenten in einem bestimmten Radius können über die Methode `find_k_nearest_neighbors_in_radius` bestimmt werden. Diese Methode implementiert den im Abschnitt 4.1.4 beschriebenen Algorithmus um alle Agenten in einem bestimmten Radius zu finden. Der Container `candidates_` wird als eine Warteschlange für die Knoten, die noch besucht werden müssen, verwendet (`candidates` verwaltet für jeden Thread jeweils eine private Warteschlange). Vor der Suche wird zuerst die Wurzel des aktuellen Baums in die Warteschlange eingefügt. Während der Suche wird in jedem Schritt ein Knoten aus der Warteschlange entnommen und die Position des Agenten, den dieser Knoten speichert, überprüft. Die Kinder dieses Knotens, die besucht werden müssen, weil sie möglicherweise Agenten speichern, die in der Suchregion liegen (vergleiche: 4.1.4), werden in die Warteschlange eingefügt. Die Suche endet, wenn die Warteschlange leer ist. Während der Suche werden aus allen Nachbarn, die gefunden werden, die k-nächsten mithilfe des Containers `neighbors_` ausgewählt. Befindet sich der Agent in der Suchregion, wird er in den Container `neighbors_` eingefügt. Falls `neighbors_` bereits so viele Agenten enthält, wie diese Methode zurückgeben soll und ein neuer Nachbar gefunden wurde wird geprüft, ob dieser Nachbar weiter vom Mittelpunkt der Suchregion entfernt ist, als alle Agenten im Container `neighbors_`. Falls ja, wird der neue Nachbar ignoriert, weil man bereits genügend Nachbarn gefunden hat, die näher zum Mittelpunkt der Suchregion sind. Wenn nicht, wird der Agent aus dem Container `neighbors_` entfernt, der am weitesten von dem Mittelpunkt der Suchregion entfernt ist, und der zuletzt gefundene Nachbar nimmt seinen Platz ein. Auf diese Weise enthält der Container `neighbors_` am Ende der Suche die k-nächsten Nachbarn, die dann über einen Iterator an den Aufrufer der Methode zurückgegeben werden.

5.2 Grid-Varianten

Da beide Varianten des gleichmäßigen Grids und cell array with binary search viele Gemeinsamkeiten haben und auf dem gleichen Prinzip basieren, sind die Klassen, die sie implementieren, sehr ähnlich aufgebaut. Alle drei Klassen speichern die Agenten intern auf dieselbe Art und Weise und bieten dem Benutzer dieselbe Schnittstelle. Daher werde ich in diesem Kapitel nicht jede einzelne Klasse beschreiben, um unnötige Wiederholungen zu vermeiden. Außerdem werde ich auf die Beschreibung der Implementierung der Methoden für das Einfügen bzw. Entfernen von Agenten an dieser Stelle verzichten. Diese Operationen sind bei Grids sehr einfach zu implementieren und wurden im Kapitel 4 ausführlich beschrieben. Unterschiede in

der Implementierung der drei Datenstrukturen gibt es im Wesentlichen nur bei der Nachbarschaftssuche. Die Umsetzung der Nachbarschaftssuche werde ich daher im Abschnitt 5.2.4 für jede der drei Klassen einzeln beschreiben.

5.2.1 Konfiguration der Grids

Wie die Ergebnisse der Laufzeitmessungen in Kapitel 6 zeigen werden, haben die Größe der Gridzellen und die Anzahl der Buckets, in denen diese Zellen gespeichert werden, einen großen Einfluss auf die Performance der Grid-Datenstrukturen bei der Nachbarschaftssuche. Die Klassen, die diese Datenstrukturen implementieren, bieten daher die Möglichkeit diese Parameter über die Konstruktoren dieser Klassen anzupassen. Für cell array with binary search und die Modulo-Variante des gleichmäßigen Grids ist es außerdem möglich, die Position des Gridausschnitts, der direkt auf die Buckets abgebildet wird, im Raum festzulegen.

5.2.2 Buckets

Wie ich bereits im Kapitel 4 erwähnt habe, werden bei den Grid-Datenstrukturen die Agenten in so genannten Buckets gespeichert. Jedes Bucket ist ein Objekt vom Typ `std::vector< std::pair<Reference, Position> >`. In jedem Element eines Buckets werden die Beschreibung und die Position eines Agenten, zusammengefasst in einem `std::pair`-Objekt, gespeichert. Alle Buckets eines Grids befinden sich wiederum in einem Vektor.

Lastenverteilung über die Buckets

Bei der Parallelisierung eines Simulationsschrittes (vergleiche 3.2) wird die Arbeit so zwischen den Threads aufgeteilt, dass jeder Thread mehrere Agenten simuliert. Wenn ein Grid für die Nachbarschaftssuche verwendet wird, können anstelle der einzelnen Agenten, die Buckets des Grids auf die Threads verteilt werden, so dass jeder Thread die Agenten aus den Buckets, die ihm zugewiesen wurden, bearbeitet.

```
#pragma omp parallel for
for (int i = 0; i < grid.num_buckets(); ++i) {
       // Nachbarn für alle Agenten aus einem Bucket nacheinander bestimmen:
       grid::bucket_type& actual_bucket = grid.get_bucket (i);
       for (int j = 0; j < actual_bucket.size(); ++j) {
              // Referenz auf den j-ten Agenten im Bucket mit dem Index i holen:
              agent_type& agent = static_cast<agent_type*>(actual_bucket[j].first);
              // Suche nach den Nachbarn dieses Agenten:
              grid.find_k_neareset_neighbors_in_radius (
                            agent->position (),
                            k_nearest,
                            radius,
                            agent->neighbors_iterator);
       }
}
```

Listing 5.1: Nachbarschaftssuche (Lastenverteilung über die Buckets)

Wenn z. B. bei einer Simulation eine Grid-Datenstruktur verwendet wird, die die Agenten in zehn Buckets speichert, und die Simulation mit zwei Threads ausgeführt wird, so bearbeitet der erste Thread alle Agenten aus den ersten fünf Buckets und der zweite Thread die Agenten aus den restlichen fünf Buckets. Das Listing 5.1 zeigt ein Beispiel für die Lastenverteilung über die Buckets des Grids.

Diese Art der Lastenverteilung hat allerdings einen Nachteil. Bei der Parallelisierung ist es wichtig die Arbeit möglichst gleichmäßig zwischen den Threads zu verteilen. Einzelne Buckets können jedoch unterschiedlich viele Agenten enthalten. Im schlimmsten Fall befinden sich die meisten Agenten in den Buckets eines Threads, während der andere Thread nur wenige Agenten bearbeiten muss. Die Last wäre in so einem Fall sehr ungleichmäßig auf die Threads verteilt, was sich negativ auf die Performance auswirken würde. Die Performance hängt also bei dieser Art der Lastenverteilung auch davon ab, wie gleichmäßig die Agenten auf die Buckets verteilt sind.

Andererseits ermöglicht diese Art der Lastenverteilung eine bessere zeitliche Lokalität der Speicherzugriffe. Das Grid teilt den Raum so auf, dass alle Agenten aus einer Gridzelle in einem Bucket liegen. Die Nachbarn aller Agenten aus einer Gridzelle befinden sich zum Teil in derselben Gridzelle und zum Teil in den umliegenden Zellen. Das heißt, die Regionen, die bei der Auswertung von Nachbarschaftsabfragen für die Agenten aus ein und derselben Gridzelle durchsucht werden müssen, überschneiden sich häufig. Dementsprechend muss bei der Durchführung von Nachbarschaftsabfragen für alle Agenten aus einem Bucket häufig dieselbe Menge von Buckets durchsucht werden. Führt ein Thread die Nachbarschaftsabfragen erst nacheinander für alle Agenten eines Buckets bevor er sich mit Agenten aus anderen Buckets beschäftigt, so ist die Wahrscheinlichkeit hoch, dass er mehrfach hintereinander auf dieselben Speicherbereiche zugreifen muss. Das bedeutet, dass der Prozessor, auf dem der Thread ausgeführt wird, mehrfach auf den Inhalt seines Caches zugreifen kann, bevor der Cache aktualisiert werden muss. Durch die Lastenverteilung über die Buckets sollten demzufolge weniger Cache-Fehlzugriffe auftreten. Häufiges Nachladen von Daten aus dem Arbeitsspeicher in den Cache wird vermieden und somit die Performance der Nachbarschaftssuche verbessert. Die Laufzeitmessungen haben diese Vermutung bestätigt: die Suche nach den k-nächsten Nachbarn eines Agenten ist ca. 35 Prozent schneller, wenn die Lastenverteilung über die Buckets erfolgt.

5.2.3 Updates

Die Update-Operationen sind bei Grid-Datenstrukturen relativ einfach. Das Einzige was dabei getan werden muss, ist, den Agenten im der Datenstruktur zu finden und seine alten Koordinaten durch neue zu ersetzen. Wenn man in der Updatephase nicht über die Liste mit Agenten, sondern über die Buckets iteriert, so wie im Abschnitt 5.2.2 beschrieben, wird auch die Suche nach der Position eines Agenten in einem Bucket überflüssig. In so einem Fall kann wie im Listing 5.2 der Index des Buckets und der Index des Agenten in diesem Bucket zusammen mit den neuen Koordinaten des Agenten direkt an die `update()` Methode übergeben werden.

```
#pragma omp parallel for
for (int i = 0; i < grid.num_buckets (); ++i) {
        // Alle Agenten aus dem Bucket i aktualisieren:
        grid::bucket_type& actual_bucket = grid.get_bucket (i);
        for (int j = 0; j < actual_bucket.size (); ++j) {
                grid.update (i, j, b[j].first->position ());
        }
}
grid.update_finish ();
```

Listing 5.2: Aktualisieren der Positionen von Agenten in einem Grid.

Bei den Updates muss zusätzlich darauf geachtet werden, dass ein Agent sich aus einer Gridzelle in einer andere bewegen kann. Wenn die Updates von mehreren Threads ausgeführt werden, kann das Kopieren von Agenten aus einem Bucket in ein anderes zu zeitkritischen Abläufen führen. In der Umsetzung der Grid-Datenstrukturen habe ich das Problem so gelöst, dass jeder Thread über eine Liste verfügt, in der er sich die Agenten, die in andere Buckets verschoben werden müssen, merkt (vergleiche Abschnitt 4.2.5). In dieser Liste werden für jeden Agenten, der in ein anderes Bucket verschoben werden muss, folgende Informationen gespeichert:

- Index des Buckets, in dem der Agent derzeit gespeichert ist.
- Index des Buckets, in das der Agent verschoben werden muss.
- Beschreibung des Agenten.
- Neue Position des Agenten.

Am Ende der Updatephase, müssen die Agenten aus diesen Listen sequentiell in die richtigen Buckets verschoben werden. Diese Funktionalität ist in der Methode `update_finish()` implementiert. `update_finish()` muss nach Abschluss der parallelen Updatephase von einem Thread aufgerufen werden.

Bei cell array with binary search erfüllt die Methode `update_finish()` eine weitere Aufgabe: sie sortiert die in den Buckets gespeicherten Agenten nach ihrer z-Koordinate. Dieser Schritt wird parallel ausgeführt. Die Buckets werden dabei auf die Threads verteilt, so dass jeder Thread einen Teil der Buckets sortiert. Anfangs wurde vermutet, dass die Anzahl der Agenten, die bei der Sortierung ihre Plätze tauschen müssen, eher klein wäre und InsertionSort sich daher am besten für das Sortieren eignen würde. Die Laufzeitmessungen haben diese Annahme jedoch wiederlegt. Der Sortieralgorithmus der STL erwies sich als die schnellere Sortiermethode.

5.2.4 Nachbarschaftssuche

Obwohl die Algorithmen für die Suche nach den k-nächsten Nachbarn eines Agenten bei allen drei Grid-Varianten sehr ähnlich sind, gibt es bei der Nachbarschaftssuche einige Unterschiede in der Implementierung. Ich werde daher im Folgenden die Umsetzung der Nachbarschaftssuche für alle drei Grid-Varianten einzeln beschreiben.

Hashing-Variante

Wie im Abschnitt 4.2.4 beschrieben, werden bei der Nachbarschaftssuche in einem Grid zunächst die Zellen bestimmt, die auf dem Grid einen Sektor aufspannen, der die Suchregion komplett einschließt. Anschließend wird für jede Zelle in diesem Bereich der Index des entsprechenden Buckets mithilfe der Hashfunktion bestimmt. Für jeden Agenten aus diesem Bucket wird die Entfernung zum Mittelpunkt der Suchregion berechnet. Falls diese Entfernung kleiner als der Suchradius ist, wird der Agent in den Container `neighbors_` eingetragen. Wie bei k-d-Bäumen (vergleiche Abschnitt 5.1.2), werden mithilfe dieses Containers die k-nächsten Nachbarn bestimmt.

Da die Hashfunktion die Gridzellen zufällig auf die Buckets verteilt, kann es vorkommen, dass mehrere Zellen aus der Suchregion auf ein und dasselbe Bucket abgebildet werden. Bei der oben beschriebenen Implementierung würden in so einem Fall einzelne Buckets mehrfach durchsucht werden. Um das zu vermeiden, verfügt jeder Thread über ein so genanntes Bitset, in dem bereits besuchte Buckets markiert werden. Ein Bitset ist eine STL-Datenstruktur, die eine Gruppe von Bits repräsentiert. Die Anzahl der Bits in diesem Bitset ist gleich der Anzahl der Buckets. Wird ein Bucket bei der Suche nach den Nachbarn eines Agenten durchsucht, markiert der Thread das entsprechende Bit in diesem Bitset. Über dieses Bitset kann später geprüft werden, ob ein Bucket bereits besucht wurde, um auf das erneute Durchsuchen dieses Buckets zu verzichten. Nach Abschluss einer Nachbarschaftsabfrage werden die Markierungen im Bitset des Threads wieder entfernt.

Modulo-Grid

Der Unterschied zwischen der Modulo-Variante und der Hashing-Variante des Grids ist im Wesentlichen die Art der Abbildung von Gridzellen auf die Buckets. Bei der Abbildung von Gridzellen auf die Buckets, die in der Modulo-Variante des Grids verwendet wird, wird die Nachbarschaftsbeziehung zwischen einzelnen Gridzellen auch auf die Buckets übertragen. Das bedeutet, dass aus dem Index eines Buckets die Indizes der Buckets, die benachbarte Zellen enthalten, bestimmt werden können. Diese Eigenschaft der Modulo-Variante des Grids ermöglicht eine zusätzlich Optimierung der Nachbarschaftssuche. Wie bei der Hashing-Variante, wird auch hier für jede Zelle in der Suchregion das entsprechende Bucket nach Nachbarn eines Agenten durchsucht. Anders als bei der Hashing-Variante muss hier jedoch nicht für jede einzelne Zelle der Index des Buckets, in der die Zelle gespeichert ist, komplett neu berechnet werden. Dieser Index lässt sich aus dem Index des vorher besuchten Buckets herleiten. Dadurch lassen sich viele Aufrufe der Modulo-Operation durch wesentlich schnellere Additionen ersetzen. Wie die Laufzeitmessungen im Kapitel 6 zeigen werden, hat diese Optimierung eine spürbare Auswirkung auf die Performance.

Cell Array With Binary Search

Die Tatsache, dass die Inhalte der Buckets bei dieser Datenstruktur sortiert werden, ermöglicht eine weitere Optimierung der Nachbarschaftssuche. Bei einer Suche nach den Nachbarn eines Agenten mit den Koordinaten (a, b, c) im Radius r, müssen die Buckets nicht komplett durchsucht werden, sondern nur die Teile der Buckets, die Agenten enthalten, deren z-Koordinate Werte zwischen $c - r$ und $c + r$ hat. Da die Buckets nach der z-Koordinate der Agenten sortiert sind, lässt sich für jeden Bucket mit binärer Suche der Index i des ersten Agenten im Bucket mit

einer z-Koordinate, die größer oder gleich $c - r$ ist, bestimmen. Analog wird der Index j des letzten Agenten, mit einer z-Koordinate, die kleiner oder gleich $c + r$ ist, ermittelt. Anschließend wird nur der Bereich des Buckets zwischen dem Agenten mit dem Index i und dem Agenten mit dem Index j durchsucht. Falls das Bucket die gesuchten Nachbarn enthält, können sie nur in diesem Teil des Buckets gespeichert sein.

6 Experimente

In diesem Kapitel werden die Laufzeitexperimente, in denen die Datenstrukturen getestet wurden, beschrieben, sowie die Ergebnisse dieser Experimente präsentiert und analysiert.

6.1 Zeitmessungen

Um die Performance der im Kapitel 4 vorgestellten Datenstrukturen zu testen, wurden diese Datenstrukturen in zwei verschiedenen Plug-ins der OpenSteerDemo-Applikation für die Nachbarschaftssuche eingesetzt. Für jede dieser Datenstrukturen wurden die Zeiten für die Such- und Updatephase gemessen. In Abschnitt 6.1.1 beschreibe ich die Vorgehensweise bei den Laufzeitmessungen. Die Plug-ins, die für die Performancetests verwendet wurden, sind im Abschnitt 6.1.2 beschrieben.

6.1.1 Vorgehensweise

Die Applikation OpenSteerDemo bietet die Möglichkeit, die Zeit zu messen, die die Simulation für eine bestimmte Anzahl an Simulationsschritten benötigt. Dank der Trennung der verschiedenen Phasen eines Simulationsschrittes, ist es außerdem möglich die Zeit für jede einzelne dieser Phasen separat zu messen. Bei den Laufzeitexperimenten wurde die Zeit gemessen, die ein Plug-in der Applikation OpenSteerDemo benötigt, um 7200 Simulationsschritte zu berechnen. Die angegebenen Zeiten sind Durchschnittswerte aus drei Zeitmessungen.

Für die Zeitmessungen wurde ein Rechner mit zwei AMD Dual-Core Opteron 270 Prozessoren und 2GB Arbeitsspeicher verwendet. Das Programm OpenSteerDemo wurde mit dem Intel Compiler 10.0.23 mit folgenden Optionen übersetzt: `-std=c99 -O2 -inline-level=2 -openmp -openmp-report1 -fp-model fast -xW`.

6.1.2 Plug-ins

Die Performance der räumlichen Datenstrukturen wurde mittels der OpenSteerDemo-Applikation gemessen. Dazu wurden zwei unterschiedliche Plug-ins verwendet: Pedestrian und Boids.

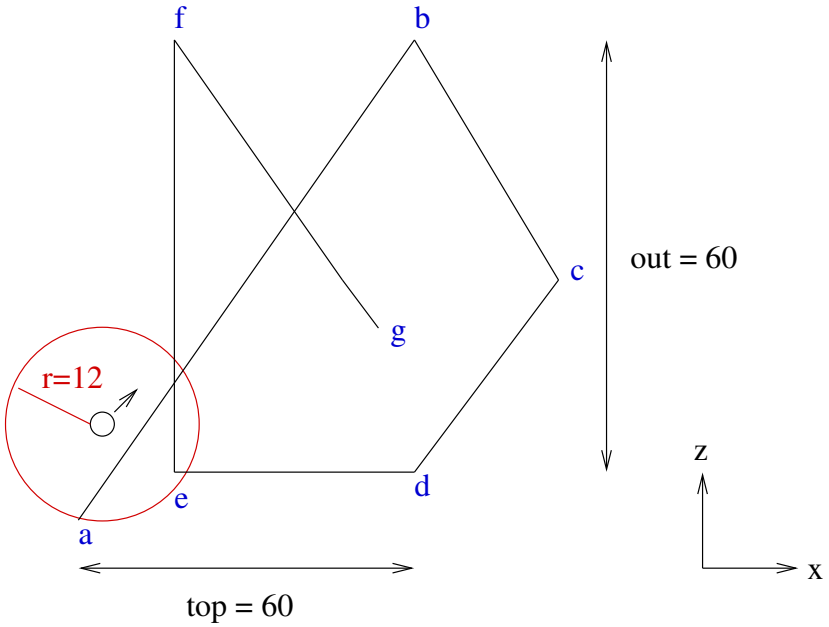


Abbildung 6.1: Der Pfad, auf sich die Agenten bewegen.

Pedestrian

In dem Plug-in Pedestrian stellen die Agenten Fußgänger dar, die alle auf einem festgelegten Pfad wandern. Der Pfad, auf dem sich die Agenten bewegen ist in der Abbildung 6.1 dargestellt. Ein Agent läuft dabei zwischen den Punkten a und g. Erreicht er den Punkt g, kehrt er wieder um und läuft in die Richtung von a. Die Agenten bewegen sich dabei immer in der x-z-Ebene, d.h. in einem zweidimensionalen Raum. Die y-Koordinate der Agenten bleibt immer Null.

Während die Agenten auf dem Pfad wandern, versuchen sie, anderen Agenten oder Hindernissen auszuweichen. Um Kollisionen zu vermeiden, durchsucht ein Agent in jedem Simulationsschritt die Umgebung nach seinen Nachbarn. Die Suchregion (Umgebung) ist kreisförmig und hat einen Radius von zwölf Einheiten (In der Abbildung 6.1 ist diese Suchregion als ein roter Kreis dargestellt). Die Position des Agenten ist der Mittelpunkt der Suchregion. Aus allen Agenten in der Suchregion werden sieben ausgewählt, die dem Agenten am nächsten sind. Für die Nachbarschaftssuche wird dabei eine räumliche Datenstruktur verwendet. Die Positionen der Nachbarn werden benutzt, um die Bewegungsrichtung und die Geschwindigkeit den Agenten so anzupassen, dass Kollisionen mit den Nachbarn vermieden werden. Am Ende eines Simulationsschrittes werden die Positionen der Agenten in der räumlichen Datenstruktur aktualisiert.

Boids

Das Plug-in Boids geht auf eine 1986 von Craig Raynolds [11] entwickelte Simulation zurück. In dieser Simulation wurde zum ersten mal das Verhalten von Schwärmen modelliert. Boids steht dabei für *birds like objects*. Wie der Name andeutet, werden in diesem Plug-in Vögel simuliert, die sich in einem Schwarm bewegen. Das Schwarmverhalten entsteht durch die Kombination von drei Steuerungsverhalten: Separation, Alignment und Cohesion (vergleiche Abschnitt 2.1).

Obwohl jeder Agent eigenständig ist, bilden alle Agenten zusammen eine Gesamtstruktur, die an einen Vogel- oder Fischwarm erinnert.

Die Agenten bewegen sich in einem drei-dimensionalen Raum innerhalb einer Kugel. Diese Kugel hat einen Radius von 50 Einheiten. Bei der Bestimmung der Nachbarn eines Agenten, wird eine kugelförmige Region mit einem Radius von neun Einheiten durchsucht. Wie im Pedestrian Plug-in werden aus allen Agenten in der Suchregion sieben ausgewählt, die dem Agenten, dessen Nachbarn gesucht werden, am nächsten sind.

6.2 Ergebnisse

6.2.1 Nachbarschaftssuche

Die Tabelle 6.1 enthält die Laufzeiten der Nachbarschaftssuche im Pedestrian-Plug-in mit verschiedenen Datenstrukturen. Die Laufzeiten der Nachbarschaftssuche im Boids-Plug-in sind in der Tabelle 6.2 aufgelistet. Für die Zeitmessungen wurde jede Simulation mit 1000 Agenten ausgeführt. In jedem Simulationsschritt wurden die Nachbarn für alle Agenten der Simulation bestimmt. Die Werte in diesen Tabellen entsprechen somit den Zeiten, die die Datenstrukturen für die Auswertung von 7200000 Anfragen nach den k-nächsten Nachbarn eines Agenten in einem bestimmten Radius benötigen.

Datenstruktur	1 Thread	2 Threads	3 Threads	4 Threads
Brute-Force	66,94	34,54	23,91	18,39
k-d-Baum	59,22	31,39	21,07	16,3
Grid (Hashing)	41,38	20,97	17,53	11,94
Grid (Modulo)	28,48	14,75	10,01	7,67
cell array	24,1	13,32	8,56	6,77

Tabelle 6.1: Laufzeit der Suchphase im Pedestrian-Plug-in mit 1000 Agenten (in Sekunden).

Datenstruktur	1 Thread	2 Threads	3 Threads	4 Threads
Brute-Force	40,6	21,72	15,14	11,71
k-d-Baum	24,73	13,91	9,67	7,63
Grid (Hashing)	15,66	8,53	6,14	4,89
Grid (Modulo)	10,62	6,02	4,43	3,4
cell array	10,93	5,84	4,26	3,32

Tabelle 6.2: Laufzeit der Suchphase im Boids-Plug-in mit 1000 Agenten (in Sekunden).

Bei den Laufzeitmessungen wurden für die Grid-Datenstrukturen die Größe der Gridzellen und die Anzahl der Buckets speziell angepasst, um optimale Performance bei den Simulationen zu erzielen. Die Tabelle 6.3 enthält für jede Datenstruktur die Werte mit denen die Laufzeitmessungen durchgeführt wurden.

Wenn man die Laufzeiten aus beiden Tabellen vergleicht, fällt auf, dass die Nachbarschaftssuche im Pedestrian-Szenario mit allen Datenstrukturen länger dauert als im Boids-Szenario. Der

	Kantenlänge einer Gridzelle		Anzahl der Buckets	
	Pedestrian	**Boids**	**Pedestrian**	**Boids**
Grid (Hashing)	15	18	80	500
Grid (Modulo)	15	15	343	343
cell array	10	10	100	100

Tabelle 6.3: Konfiguration der Grid-Datenstrukturen für die Laufzeitmessungen.

Grund dafür ist die größere Suchregion im Pedestrian-Plug-in. Da der Suchradius im Pedestrian-Plug-in um ein Drittel größer ist als beim Boids-Plug-in, befinden sind mehr Agenten in der Suchregion. Das führt dazu, dass bei der Nachbarschaftssuche mit räumlichen Datenstrukturen mehr Agenten untersucht werden müssen. Die größere Suchregion bewirkt außerdem, dass die Auswahl der k-nächsten Nachbarn aus allen Agenten in der Suchregion beim Pedestrian-plug-in aufwändiger ist, weil sich insgesamt mehr Agenten in der Suchregion aufhalten. Damit beeinflusst die größere Suchregion auch die Performance der Nachbarschaftssuche mit dem Brute-Force-Ansatz, weil die Auswahl der k-nächsten Nachbarn aus allen Agenten in der Suchregion beim Brute-Force-Ansatz auf dieselbe Weise implementiert ist, wie bei räumlichen Datenstrukturen (die Realisierung der Auswahl der k-nächsten Nachbarn wurde im Abschnitt 5.1.2 beschrieben).

Räumliche Datenstrukturen können zusätzlich von der gleichmäßigeren Verteilung der Agenten im Raum beim Boids-Plug-in profitieren. Während die Agenten in der Pedestrian-Simulation immer in der Nähe des Pfades bleiben, können sich die Agenten beim Boids-Plug-in frei bewegen und verteilen sich gleichmäßiger im Raum. Dadurch, dass die Agenten besser auf den gesamten Raum verteilt sind, befinden sich in einer Suchregion durchschnittlich weniger Agenten, als in der Pedestrian-Simulation, was die Auswahl der k-nächsten Nachbarn zusätzlich beschleunigt.

Trotz der Unterschiede zwischen beiden Plug-ins konnte die Nachbarschaftssuche in beiden Fällen durch den Einsatz der räumlichen Datenstrukturen beschleunigt werden. Die Laufzeiten aus den Tabellen 6.1 und 6.2 zeigen, dass alle untersuchten räumlichen Datenstrukturen bei der Nachbarschaftssuche schneller sind als der Brute-Force-Ansatz. Die Zahlen aus den Tabellen zeigen außerdem, dass es deutliche Unterschiede in der Performance der einzelnen Datenstrukturen gibt. Vergleicht man die Performance der unterschiedlichen Datenstrukturen, lassen sich folgende Ergebnisse festhalten:

- flache Datenstrukturen schneiden bei der Nachbarschaftssuche deutlich besser ab, als der k-d-Baum
- die Modulo-Variante des gleichmäßigeren Grids ist schneller als die Hashing-Variante
- von allen Datenstrukturen erreicht cell array with binary search in den meisten Fällen die beste Performance

Um die unterschiedliche Performance der untersuchten Datenstrukturen zu erklären, muss die Anzahl der Agenten betrachtet werden, die eine Datenstruktur während einer Nachbarschaftsabfrage überprüfen muss. Die Performance der Nachbarschaftssuche hängt in erster Linie von diesem Faktor ab.

Die Tabelle 6.4 zeigt für jede Datenstruktur die ungefähre Anzahl der Agenten, die während der Nachbarschaftssuche untersucht werden. Diese Werte kamen zustande, indem die überprüfen

Datenstruktur	Brute-Force	k-d-Baum	Grid (Modulo)	Grid (Hashing)	cell array
Pedestrian	1000	200	268	336	164
Boids	1000	81	73	97	34

Tabelle 6.4: Anzahl der Agenten, die durchschnittlich für die Bestimmung der Nachbarn eines Agenten überprüft werden (Simulation mit insgesamt 1000 Agenten).

Agenten während der Nachbarschaftssuche mitgezählt wurden. Wie bei den Zeitmessungen wurde nach einer bestimmten Anzahl der Simulationsschritte der Durchschnitt aus den gesammelten Informationen berechnet. Wie aus der Tabelle 6.4 zu entnehmen ist, lässt sich die Anzahl der Agenten, die bei der Suche nach den Nachbarn eines Agenten überprüft werden müssen, durch die Verwendung von räumlichen Datenstrukturen deutlich reduzieren. Die Tatsache, dass bei der Nachbarschaftssuche nur ein Teil der Agenten überprüft wird, wirkt sich merkbar auf die Performance der Suche aus.

Vergleicht man die Zahlen aus der Tabelle, so fällt auf, dass die Anzahl der untersuchten Agenten bei der Nachbarschaftssuche im Boids-Plug-in für alle räumlichen Datenstrukturen deutlich niedriger ist als beim Pedestrian-Plug-in. Diese Tatsache bekräftigt auch die Vermutung, dass durch kleinere Suchregionen und eine bessere Verteilung der Agenten im Boids-Plug-in die Anzahl der Agenten, die überprüft müssen, sinkt und die Performance der Nachbarschaftssuche sich dementsprechend verbessert.

Die Zahlen aus der Tabelle 6.4 zeigen außerdem, dass bei der Modulo-Variante im Durchschnitt etwa ein viertel weniger Agenten während der Auswertung einer Nachbarschaftsabfrage überprüft werden müssen, als bei der Hashing-Variante. Entsprechend ist die Performance der Modulo-Variante bei der Nachbarschaftssuche besser als die Performance der Hashing-Variante. Der Grund dafür ist, dass die Modulo-Variante so konfiguriert wurde, dass das Grid den gesamten Raum, in dem sich die Agenten bewegen abdeckt, d.h. jedes Bucket speichert genau eine Gridzelle. Bei der Hashing-Variante erfolgt die Abbildung der Gridzellen auf die Buckets über eine Hashfunktion. Die Hashfunktion garantiert jedoch nicht, dass die Zellen, wie bei der Modulo-Variante, optimal auf die Buckets verteilt werden. So kommt es vor, dass ein Bucket mehr als eine Gridzelle enthält. Da die Nachbarschaftssuche aber so implementiert ist, dass ein Bucket immer komplett durchsucht werden muss, werden bei der Hashing-Variante auch Gridzellen durchsucht, die außerhalb der Suchregion liegen. Somit müssen bei der Hashing-Variante insgesamt mehr Agenten überprüft werden.

Die Anzahl der Agenten, die überprüft werden müssen, ist jedoch nicht der einzige Faktor, der für die Performance der Nachbarschaftssuche von Bedeutung ist: obwohl bei k-d-Bäumen nur wenige Agenten während der Nachbarschaftssuche überprüft werden, ist die Suche dennoch nicht wesentlich schneller als beim Brute-Force-Ansatz. Eine mögliche Ursache hierfür ist das langsame Traversieren des Baums: bei der Suche in einem k-d-Baum muss an jedem Knoten, der nicht in der Suchregion liegt, entschieden werden, ob die Suche im linken oder im rechten Teilbaum des aktuellen Knotens weitergeführt wird. Im Gegensatz dazu steht bei flachen Datenstrukturen am Anfang der Suche fest, welche Buckets durchsucht werden müssen. Während der eigentlichen Suche wird kontinuierlich über die Elemente eines Vektors iteriert. Im Vergleich zum Traversieren der Knoten eines k-d-Baums ist das Iterieren über die Elemente eines Vektors wesentlich effizienter. Diese Vermutung wird durch die Untersuchungen des Programms mit dem VTune Performance Analyzer Werkzeug von Intel bestätigt. Die Analyse mit diesem Profiler-Werkzeug hat ergeben, dass der Teil des Programms, in dem entschieden wird,

welcher der beiden Teilbäume untersucht werden muss, etwa sechs Prozent der Gesamtlaufzeit pro Thread in Anspruch nimmt. Im Vergleich dazu fallen auf die Bestimmung der k-nächsten Nachbarn insgesamt 30 - 32 Prozent der Gesamtlaufzeit. Hinzu kommt, dass beim Iterieren über einen Vektor die Caches der Prozessoren besser ausgenutzt werden als beim Iterieren über einen k-d-Baum, dessen Knoten nicht zusammenhängende Speicherblöcke sind, die über Zeiger verlinkt werden.

Die Anzahl der Agenten, die während der Suche überprüft werden müssen, ist bei cell array with binary search am kleinsten. Da die Elemente eines Buckets bei dieser Datenstruktur nach einer Raum-Koordinate sortiert sind, müssen, anders als bei gleichmäßigen Grids, nicht alle Elemente eines Buckets überprüft werden, sondern nur die, die möglicherweise in der Suchregion liegen. Diese Eigenschaft von cell array with binary search spiegelt sich auch an der Performance dieser Datenstruktur wieder.

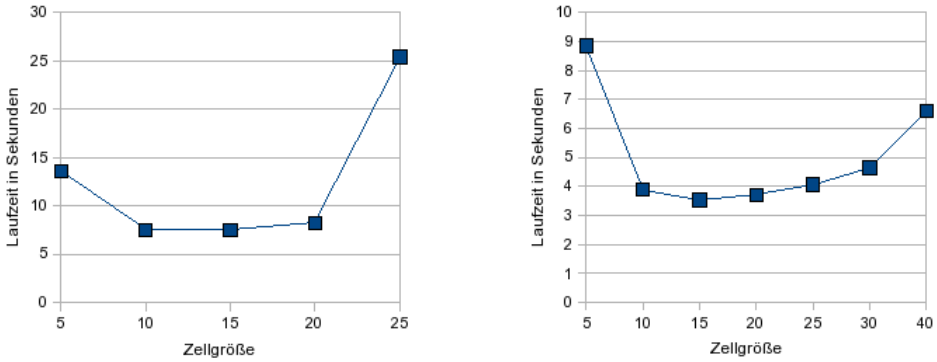
Der Performance-Unterschied zwischen der Modulo-Variante des gleichmäßigen Grids und cell array with binary search ist jedoch nicht so groß wie beim Pedestrian-Plug-in. Beide Datenstrukturen sind bei der Nachbarschaftssuche im Boids-Plug-in annähernd gleich schnell. Sequentiell ist das Modulo-Grid sogar etwas schneller als cell array with binary search, obwohl die letztere Datenstruktur bei der Nachbarschaftssuche weniger Agenten überprüft. Das liegt daran, dass die Reduzierung der Zahl an Agenten, die während einer Nachbarschaftsabfrage überprüft werden, bei cell array with binary search mit einem zusätzlichen Aufwand verbunden ist. Die Anzahl der untersuchten Agenten wird bei cell array with binary search dadurch reduziert, dass ein Bucket nicht komplett durchsucht wird, sondern nur der Abschnitt des Buckets, der die Agenten in der Suchregion enthält. Für jeden Bucket muss also zunächst dieser Abschnitt bestimmt werden, was im Vergleich zu den gleichmäßigen Grids einen zusätzlichen Aufwand bedeutet.

Dieser Mehraufwand lohnt sich aber nur dann, wenn die Anzahl der Agenten, die bei der Nachbarschaftssuche überprüft werden müssen, deutlich reduziert wird, wie es bei der Pedestrian-Simulation der Fall ist (siehe Tabelle 6.4). Bei der Boids-Simulation ist die Suchregion kleiner und die Agenten sind besser im Raum verteilt. Dadurch enthalten die Buckets, die während einer Nachbarschaftsabfrage durchsucht werden müssen, insgesamt weniger Agenten als bei der Pedestrian-Simulation. Dies führt dazu, dass die Nachbarschaftssuche bei allen flachen Datenstrukturen schneller wird, weil es weniger potenzielle Nachbarn gibt. Wenn die Nachbarschaftssuche insgesamt schneller wird, fällt der Mehraufwand bei cell array with binary search stärker ins Gewicht. Zum anderen lässt sich beim Boids-Plug-in die Anzahl der Agenten, die überprüft werden müssen, nicht so stark reduzieren wie beim Pedestrian-Plug-in: im Pedestrian-Plug-in konnte die Anzahl der Agenten, die überprüft werden müssen, durch die Verwendung von cell array with binary search im Vergleich zum Modulo-Grid um 104 Agenten reduziert werden, beim Boids-Plug-in konnte diese Zahl nur um 40 Agenten reduziert werden.

Konfiguration der Grids

Experimente mit den Grid-Datenstrukturen haben ergeben, dass die Kantenlänge einer Gridzelle einen großen Einfluss auf die Performance der Nachbarschaftssuche hat.

Die Abbildung 6.2 demonstriert den Zusammenhang zwischen der Zellgröße und der Performance der Modulo-Variante des gleichmäßigen Grids bei der Nachbarschaftssuche in beiden Plug-ins. Um bei der Nachbarschaftssuche eine gute Performance zu erreichen, muss die Größe



(a) Pedestrian (b) Boids

Abbildung 6.2: Zusammenhang zwischen der Kantenlänge einer Zelle und der Performance der Nachbarschaftssuche.

einer Gridzelle an das spezielle Problem angepasst werden: eine Zelle darf werder zu groß, noch zu klein sein.

Ein wichtiger Faktor, der bei der Wahl einer geeigneten Zellgröße berücksichtigt werden muss, ist die Größe der Suchregion. Wenn die Gridzellen sehr viel kleiner sind als die Suchregion, ist die Anzahl der Gridzellen, die sich mit der Suchregion überschneiden, entsprechend größer, d.h. es müssen viele Buckets durchsucht werden. Sind die Gridzellen dagegen zu groß, gibt es zwar nur wenige Gridzellen, die sich mit der Suchregion überschneiden, diese enthalten aber viele Agenten, die alle überprüft werden müssen.

Die Nachbarschaftssuche mit Grids erreicht also die beste Performance, wenn das Verhältnis R von der Kantenlänge einer Zelle und dem Durchmesser der Suchregion in einem bestimmten Wertebereich liegt. Dieser Zusammenhang lässt sich bei allen flachen Datenstrukturen beobachten. Die während der Laufzeitexperimente ermittelten optimalen Werte für R sind in der Tabelle 6.5 aufgeführt.

Datenstruktur	**Pedestrian**	**Boids**
Grid (Hashing)	0,4 – 0,8	0,7 – 1,3
Grid (Modulo)	0,4 – 0,8	0,5 – 1
cell array	0,2 – 0,6	0,3 – 1

Tabelle 6.5: Optimale Werte für das Verhältnis von der Kantenlänge einer Zelle und dem Durchmesser der Suchregion.

So erreicht z. B. die Modulo Variante des gleichmäßigen Grids bei der Nachbarschaftssuche im Pedestrian-Plug-in die beste Performance, wenn R zwischen 0,4 und 0,8 liegt. Für eine gute Performance beim Boids-Plug-in muss dieser Wert zwischen 0,5 und 1 liegen.

Eine andere Größe, die für die Performance der Grid-Datenstrukturen eine Rolle spielt, ist die Anzahl der Buckets. Bei den Laufzeitexperimenten hat es sich herausgestellt, dass für eine optimale Performance der Abschnitt des Grids, der direkt auf die Buckets abgebildet wird, bei der Modulo-Variante des gleichmäßigen Grids und cell array with binary search den gesamten Raum, in dem sich die Agenten bewegen abdecken muss. In diesem Fall speichert ein Bucket

genau eine Gridzelle. Bei der Hashing-Variante des gleichmäßigen Grids muss die Anzahl der Buckets groß genug sein, damit die Wahrscheinlichkeit dafür, dass mehrere Gridzellen auf ein und dasselbe Bucket abgebildet werden, minimiert wird.

Skalierbarkeit

Ein anderes Kriterium für die Beurteilung der Performance der einzelnen Datenstrukturen bei der parallelen Nachbarschaftssuche ist die Skalierbarkeit.

Datenstruktur:	Brute-Force	k-d-Baum	Grid (Hashing)	Grid (Modulo)	cell array
Pedestrian:	3,64	3,63	3,47	3,71	3,56
Boids:	3,47	3,24	3,2	3,12	3,29

Tabelle 6.6: Speedups, die die unterschiedlichen Datenstrukturen bei der Nachbarschaftssuche mit vier Threads erreichen.

Die Tabelle 6.6 zeigt die Speedups, die die unterschiedlichen Datenstrukturen bei der Nachbarschaftssuche mit vier Threads erreichen. Die Zahlen aus der Tabelle 6.6 zeigen, dass alle untersuchten Datenstrukturen bei der Nachbarschaftssuche in etwa gleich gut skalieren.

6.2.2 Updatephase

Wie die Zeiten in den Tabellen 6.7 und 6.8 zeigen, sind die im Kapitel 4 beschriebenen Datenstrukturen auch bei der Updatepahse schneller als der Brute-Force-Ansatz. Die einzige Ausnahme bilden die k-d-Bäume.

Datenstruktur	1 Thread	2 Threads	3 Threads	4 Threads
Brute-Force	3,85	2,98	2,37	1,96
k-d-Baum	2,24	3,2	3,46	3,38
Grid (Hashing)	1,52	1,16	0,94	0,78
Grid (Modulo)	1,4	1,09	1,45	0,97
cell array	1,08	0,95	0,87	0,72

Tabelle 6.7: Laufzeit der Updatephase im Boids-Plug-in (Sekunden).

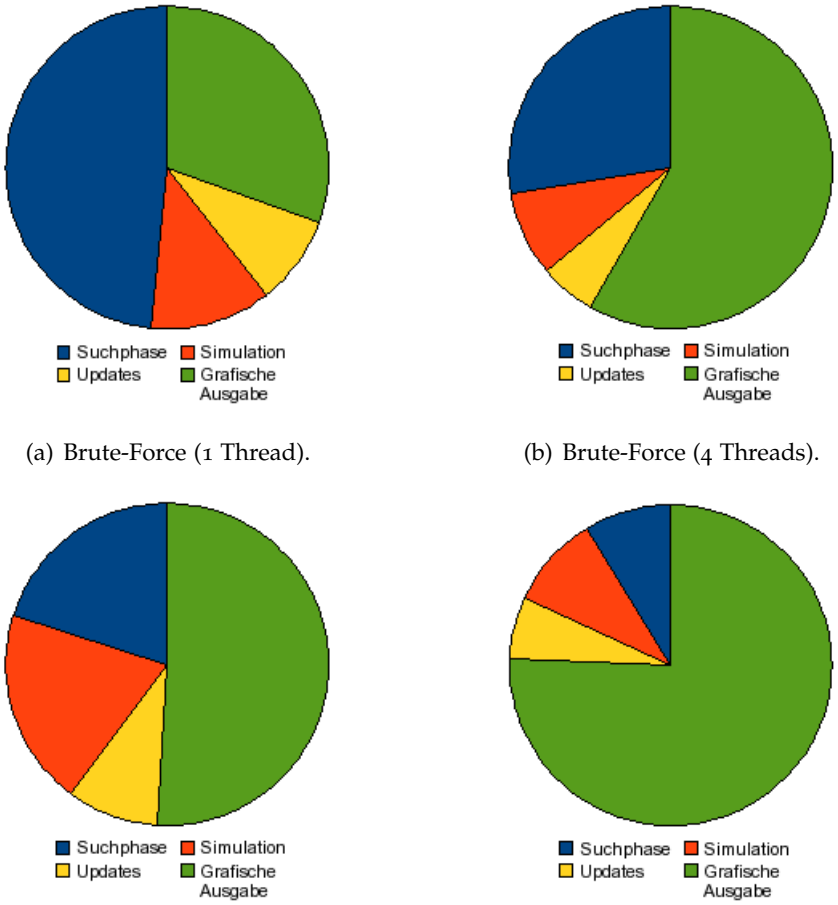
Datenstruktur	1 Thread	2 Threads	3 Threads	4 Threads
Brute-Force	3,85	2,99	2,37	1,96
k-d-Baum	3,19	4,4	4,63	4,52
Grid (Hashing)	1,51	1,06	0,97	0,7
Grid (Modulo)	1,42	1,47	1,17	1,09
cell array	1,13	0,96	0,81	0,69

Tabelle 6.8: Laufzeit der Updatephase im Pedestrian-Plug-in (Sekunden).

Die flachen räumlichen Datenstrukturen sind deswegen schneller als der Brute-Force-Ansatz, weil der Agent, dessen Koordinaten aktualisiert werden müssen, nicht erst in der Datenstruktur gefunden werden muss (vergleiche 5.2.3), wie es beim Brute-Force-Ansatz der Fall ist.

Bei k-d-Bäumen dauert die Updatephase länger als bei anderen Datenstrukturen, weil ein k-d-Baum während einer Updatephase neu aufgebaut werden muss. Allerdings ist der ständige Neuaufbau des k-d-Baums doppelt so schnell wie die alternative Update-Methode, bei der ein Agent erst aus dem Baum gelöscht und danach mit neuen Koordinaten eingefügt wird.

6.2.3 Einfluss der räumlichen Datenstrukturen auf die Gesamtperformance der Simulation



(a) Brute-Force (1 Thread).

(b) Brute-Force (4 Threads).

(c) Cell Array (1 Thread).

(d) Cell Array (4 Threads).

Abbildung 6.3: Anteil der einzelnen Phasen an der Gesamtlaufzeit des Boids-Plug-ins.

Die Abbildung 6.3 zeigt den Anteil der einzelnen Phasen an der Gesamtlaufzeit der Boids-Simulation. Wie aus dem Diagramm 6.3(a) zu entnehmen ist, hat die Suchphase den größten Anteil an der Gesamtlaufzeit der Simulation, wenn der Brute-Force-Ansatz für die Nachbarschaftssuche verwendet wird. Durch die Parallelisierung der Nachbarschaftssuche sinkt dieser Anteil von 48 auf 27 Prozent (Abbildung 6.3(b). Verwendet man für die Nachbarschaftssuche eine spezielle räumliche Datenstruktur, wie cell array with binary search, sinkt der Anteil

der Suchphase an der Gesamtlaufzeit beim Ausführen der Simulation mit vier Threads auf ca. 9 Prozent (Abbildung 6.3(d)). Beim Pedestrian-Plug-in wird die Laufzeit ähnlich auf die einzelnen Phasen der Simulation verteilt. In beiden Plug-ins nimmt die Nachbarschaftssuche sehr viel Zeit in Anspruch, wenn der Brute-Force-Ansatz verwendet wird. Die Beschleunigung der Nachbarschaftssuche durch den Einsatz von räumlichen Datenstrukturen wirkt sich somit spürbar auf die Gesamtperformance der Simulation.

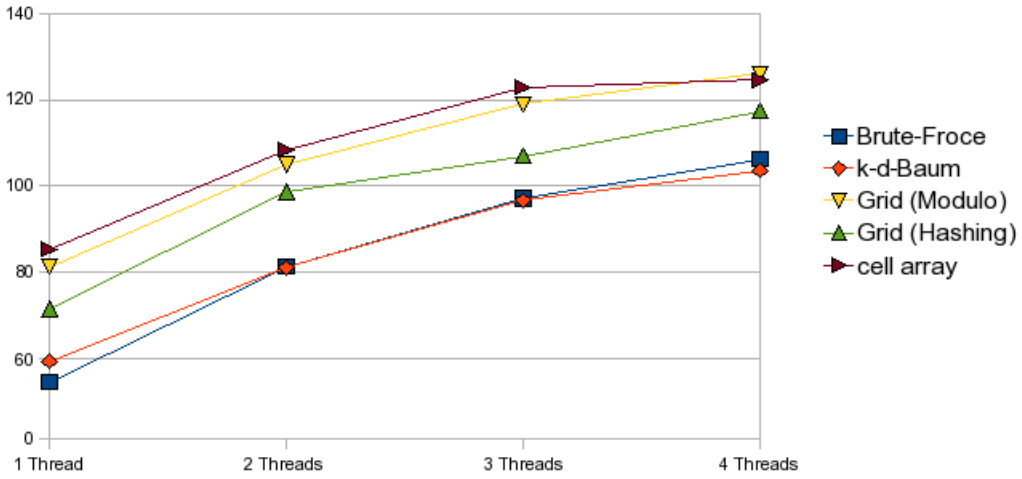


Abbildung 6.4: Framerate des Pedestrian-Plug-ins (1000 Agenten).

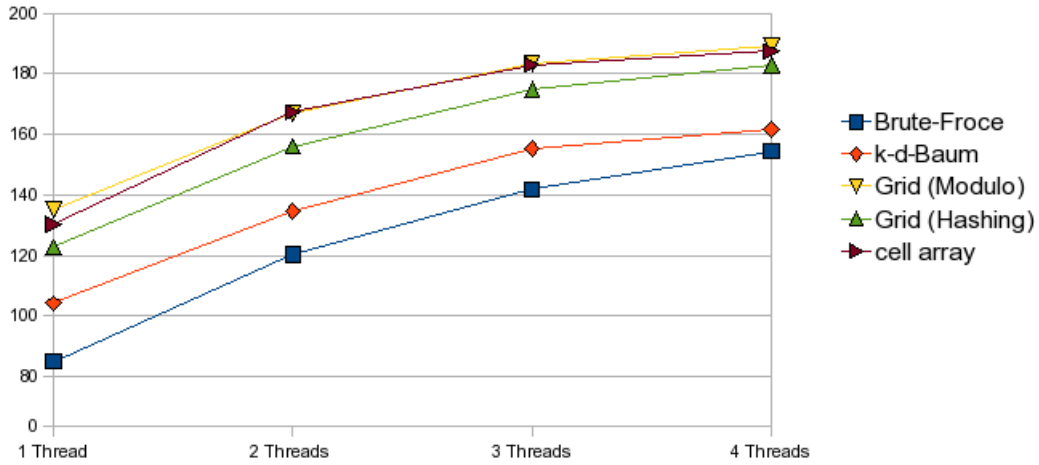


Abbildung 6.5: Framerate des Boids-Plug-ins (1000 Agenten).

Die Abbildung 6.4 zeigt die Änderung der Gesamtframerate im Pedestrian-Plug-in mit steigender Anzahl der Threads. Die Änderung der Gesamtframerate im Boids-Plug-in ist in der Abbildung 6.5 dargestellt. Aus beiden Diagrammen geht hervor, dass die Verwendung der Datenstrukturen, die bei der Nachbarschaftssuche schnell sind, die Framerate deutlich erhöht.

Mithilfe der räumlichen Datenstrukturen lässt sich aber nicht nur die Framerate des Plug-ins erhöhen, sondern auch die Anzahl der Agenten, die mit einer bestimmten Framerate simuliert werden können.

Plug-in	Brute-Force	k-d-Baum	Grid (Hashing)	Grid (Modulo)	cell array
Pedestrian	2850	3100	3800	4400	4700
Boids	3300	5000	7100	7800	8200

Tabelle 6.9: Maximale Anzahl der Agenten, die mit einer Framerate von ca. 30 fps simuliert werden können.

Die Tabelle 6.9 zeigt, wieviele Agenten in den beiden Plug-ins bei einer Framerate von 30 fps mit vier Threads simuliert werden können. Die Auswirkung der schnelleren Nachbarschaftssuche durch den Einsatz von räumlichen Datenstrukturen ist hier sogar größer als bei der Framerate. Während die Framerate der beiden Plug-ins durch den Einsatz von räumlichen Datenstrukturen um ca. 20 Prozent erhöht werden konnte, stieg die maximale Anzahl der Agenten, die bei 30 fps simuliert werden können, um 65 Prozent beim Pedestrian-Plug-in und um 148 Prozent beim Boids-Plug-in.

6.3 Vergleich der Datenstrukturen

Die Laufzeitexperimente mit unterschiedliche Datenstrukturen haben gezeigt, dass alle untersuchten Datenstrukturen sowohl Vor- als auch Nachteile haben. Vergleicht man die getesteten räumlichen Datenstrukturen anhand der Performance bei der Nachbarschaftssuche, so gehen die Grids eindeutig als Gewinner aus diesem Vergleich hervor: die Performance der Grids bei der Nachbarschaftssuche ist bis zu 2,5 mal besser als die Performance des k-d-Baums. Die Framerate beider Plug-ins, die für die Performancetests verwendet wurden, ließ sich durch den Einsatz von allen drei Grid-Datenstrukturen deutlich erhöhen. Durch den Einsatz des k-d-Baums für die Nachbarschaftssuche konnte nur die Framerate des Boids-Plug-ins merkbar gesteigert werden. Beim Pedestrian-Plug-in ließ sich keine Erhöhung der Framerate im Vergleich zum Brute-Force-Ansatz feststellen.

Ein anderer Nachteil des k-d-Baums besteht darin, dass diese Datenstruktur sehr statisch ist. Diese Eigenschaft erschwert die Modifikationen der im k-d-Baum gespeicherten Daten: um einen gespeicherten Datensatz zu modifizieren, muss eventuell die Struktur des Baums geändert werden. Der Aufbau eines k-d-Baums ist dagegen recht performant, so dass es sinnvoll ist, den k-d-Baum während der Updatephase neu aufzubauen, anstatt die im Baum gespeicherten Datensätze einzeln zu ändern. Trotzdem sind die Update-Operationen der Grid-Datenstrukturen im Vergleich zum ständigen Neuaufbau des k-d-Baums wesentlich effizienter und können außerdem einfach parallelisiert werden.

Was die Skalierbarkeit der Nachbarschaftssuche mit den unterschiedlichen Datenstrukturen bezüglich der Anzahl der Threads angeht, erreichen alle Datenstrukturen gute Ergebnisse. Die Unterschiede zwischen den einzelnen Datenstrukturen sind minimal. Bei der Skalierbarkeit bezüglich der maximalen Anzahl der Agenten, die simuliert werden können, erzielen die Grid-Datenstrukturen und insbesondere cell array with binary search die besseren Ergebnisse.

Aber auch die Grid-Datenstrukturen haben einen Nachteil. Um eine gute Performance zu erzielen, muss die Konfiguration eines Grids (die Anzahl der Buckets und vor allem die Kantenlänge einer Gridzelle) an ein gegebenes Szenario (insbesondere an den Suchradius) angepasst werden. Ein k-d-Baum muss dagegen nicht für einen bestimmten Suchradius oder andere Größen konfiguriert werden. Aus der Tatsache, dass die Performance eines Grids stark

von seiner Konfiguration abhängt, ergibt sich auch ein weiterer Nachteil: um gute Performance mit Grids zu erzielen, darf sich der Suchradius während einer Simulation nicht drastisch ändern. Wird der Suchradius zu klein oder zu groß verlangsamt sich die Nachbarschaftssuche. Im Gegensatz zu den Grids ist der k-d-Baum unempfindlich gegenüber häufigen Änderungen des Suchradius.

Aus dem Vergleich der beiden Varianten des gleichmäßigeren Grids (Modulo- und Hashing-Variante) geht hervor, dass die Modulo-Variante die Gridzellen gleichmäßiger auf die Buckets verteilt und somit eine bessere Performance bei der Nachbarschaftssuche erreicht.

Die Datenstruktur cell array with binary search hat der Vorteil, dass durch die Sortierung der Bucketinhalte die Anzahl der Agenten, die bei der Nachbarschaftssuche überprüft werden müssen, im Vergleich zu der Modulo-Variante des gleichmäßigen Grids reduziert werden kann. Dieser Vorteil wird aber durch einen zusätzlichen Aufwand erreicht. Der Geschwindigkeitsgewinn, der durch eine geringere Anzahl von Agenten, die als Nachbarn in Frage kommen, entsteht muss groß genug sein, um den Mehraufwand aufzuwiegen (vergleiche Abschnitt 6.2.1).

Als Fazit lassen sich folgende Punkte festhalten:

- In Simulationen, in denen der Suchradius gleichbleibt oder zumindest nicht stark variiert, sind Grids bei der Nachbarschaftssuche den k-d-Bäumen deutlich überlegen.
- Um eine gute Performance bei der Nachbarschaftssuche zu erreichen, muss ein Grid entsprechend konfiguriert werden.
- Wenn der Raum mit Agenten dicht bevölkert ist und die Gridzellen dementsprechend voll sind, erreicht cell array with binary search eine bessere Performance als die Modulo-Variante des gleichmäßigen Grids.
- Wenn die Agenten dagegen weit über den gesamten Raum verstreut sind und die Gridzellen wenige Agenten enthalten, ist das Modulo-Grid möglicherweise die bessere Wahl.

7 Zusammenfassung

Diese Arbeit hat mehrere räumliche Datenstrukturen zur Organisation von Punkten im dreidimensionalen Raum vorgestellt: zwei Varianten des gleichmäßigen Grids, den k-d-Baum und das cell array with binary search. Die Datenstrukturen wurden in C++ implementiert und für parallele Nachbarschaftsabfragen und Modifikationen der gespeicherten Datensätze optimiert. Um die Performance dieser Datenstrukturen zu testen, wurden sie in eine parallelisierte Version der OpenSteer-Bibliothek eingebunden. Die Performance der untersuchten Datenstrukturen bei der Nachbarschaftssuche und der Modifikation der gespeicherten Einträge wurde sowohl experimentell durch Laufzeitmessungen mit zwei Plug-ins der OpenSteerDemo-Applikation als auch asymptotisch bestimmt.

Bei den Laufzeitexperimenten hat sich herausgestellt, dass alle untersuchten Datenstrukturen in der Lage sind, die Nachbarschaftsabfragen im Vergleich zum Brute-Force-Ansatz, bei dem alle Agenten in einer Liste gespeichert werden, sowohl sequentiell als auch parallel schneller auszuwerten. Bei der parallelen Nachbarschaftssuche erreichen alle untersuchten Datenstrukturen hohe Speedups. Die Speedups bei der Nachbarschaftsuche mit räumlichen Datenstrukturen sind vergleichbar mit den Speedups, die bei der Nachbarschaftssuche mit dem Brute-Force-Ansatz erreicht werden.

Desweiteren konnten deutliche Unterschiede in der Performance der einzelnen Datenstrukturen festgestellt werden. So haben die Laufzeitmessungen ergeben, dass die flachen Datenstrukturen, zu denen gleichmäßige Grids und cell array with binary search gehören, sowohl bei der Nachbarschaftssuche als auch bei der Modifikation der gespeicherten Datensätze eine wesentlich bessere Performance erreichen als der k-d-Baum. Der Vergleich der Laufzeiten beider Varianten des gleichmäßigen Grids hat ergeben, dass die Modulo-Variante der Hashing-Variante bei der Nachbarschaftssuche deutlich überlegen ist, weil sie die Gridzellen optimal auf die zur Verfügung stehenden Buckets verteilt. Neben der Modulo-Variante des gleichmäßigen Grids hat sich cell array with binary search als sehr schnell erwiesen. Vor allem in Situationen, in denen der Raum mit Agenten dicht bevölkert ist, kann diese Datenstruktur bei der Nachbarschaftssuche bessere Ergebnisse erzielen als das Modulo-Grid.

Ein weiteres Ergebnis der Laufzeitexperimente war die Beobachtung, dass die Performance der flachen Datenstrukturen stark von ihrer Konfiguration abhängt. Um eine gute Performance zu erreichen, müssen die Kantenlänge einer Gridzelle und die Anzahl der Buckets, in denen die Gridzellen gespeichert werden, an die Größe der Suchregion angepasst werden.

Die Laufzeitexperimente haben außerdem gezeigt, dass die Performance der Nachbarschaftssuche einen wesentlichen Einfluss auf die Gesamtperformance der OpenSteer-Plug-ins hat. Mit der Beschleunigung der Nachbarschaftssuche durch den Einsatz der räumlichen Datenstrukturen konnte in den meisten Fällen auch eine merkliche Erhöhung der Framerate beobachtet werden. Neben der Framerate konnte auch die maximale Anzahl der Agenten, die mit einer bestimmten Framerate simuliert werden können, deutlich erhöht werden.

8 Ausblick

Es existiert eine Vielzahl von räumlichen Datenstrukturen, die für unterschiedliche Zwecke entwickelt wurden. Für viele dieser Datenstrukturen gibt es zudem unterschiedliche Varianten und Implementierungsmöglichkeiten. In dieser Arbeit wurden nur wenige räumliche Datenstrukturen untersucht, darunter nur eine hierarchische Datenstruktur. Obwohl die Variante des k-d-Baums, die in dieser Arbeit untersucht wurde, sich als die ineffizienteste von allen untersuchten Datenstrukturen erwiesen hat, lohnt es sich möglicherweise trotzdem andere Varianten des k-d-Baums zu untersuchen, die sich für die Szenarios wie Pedestrian oder Boids eventuell besser eignen als ein homogener k-d-Baum. Neben k-d-Bäumen könnten auch andere hierarchische Datenstrukturen wie z. B. Range Trees [12] oder Octrees [12] auf ihre Performance bei der Nachbarschaftssuche untersucht werden. Möglicherweise erlauben einige dieser Datenstrukturen im Gegensatz zum k-d-Baum eine schnelle Nachbarschaftssuche und effiziente Modifikationen der gespeicherten Datensätze. So ist z. B. der Aufwand die Nachbarschaftssuche bei den sogenannten Range Trees asymptotisch niedriger als bei k-d-Bäumen [12]. Bestimmte Octree-Varianten, die die Daten in den Blattknoten speichern, erlauben effiziente Modifikationen, weil die Daten aktualisiert werden können, ohne dass die Struktur des Baums verändert werden muss.

Ein anderer Aspekt, der untersucht werden könnte, betrifft die Simulationen, in denen die Datenstrukturen eingesetzt werden. In dieser Arbeit wurden die untersuchten Datenstrukturen nur mit zwei unterschiedlichen Plug-ins getestet. Möglicherweise würden andere Szenarien andere Ergebnisse bei der Untersuchung der Performance räumlicher Datenstrukturen liefern.

Die in dieser Arbeit vorgestellten Datenstrukturen wurden so implementiert, dass die Größe des Raums, in dem sich die Agenten befinden, nicht eingeschränkt werden muss. Bei den Grids war dies nur durch eine zusätzliche Komplexität der Implementierung möglich: ein Grid, der nur Agenten aus einem abgeschlossenen Raum verwalten muss, lässt sich einfacher implementieren. Es wäre interessant zu untersuchen, ob sich dieser Aufwand tatsächlich lohnt. Dazu müsste man erstens prüfen, welche Performance der vorgestellten Datenstrukturen in Simulationen erreichen, in denen der Raum tatsächlich unbegrenzt ist. Zweitens müsste man die Performance der vorgestellten Grids und Grids, die nur Daten aus einem begrenzten Raum, dessen Größe von Anfang an festegelegt ist, verwalten können, vergleichen.

Literaturverzeichnis

[1] Openmp application programm interface, version 2.5. http://www.openmp.org/mp-documents/spec25.pdf, May 2005.

[2] H. K. Ahn, N. Mamoulis, and H. M. Wong. A survey on multidimensional access methods. Technical report, Institute of Information and Computing Sciences, Utrecht University, 2001.

[3] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[5] Björn Knafla and Claudia Leopold. Parallelizing a real-time steering simulation for computer games with openmp. Technical report, University of Kassel, Research Group Programming Languages / Methodologies, 2007.

[6] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[7] Tom Krazit. Game makers adapt to multi-core chips. http://news.cnet.co.uk/ gamesgear/0,39029682,49289056,00.htm, 2007.

[8] Sean Mauch. *Efficent Algorithms for Solving Static Hamilton-Jacobi Equations*. PhD thesis, California Institute of Technology, April 2003.

[9] Thomas Rauber and Gudula Rünger. *Parallele und verteilte Programmierung*. Springer Verlag, 2000.

[10] C. W. Reynolds. Webseite des projekts opensteer. http://opensteer.sourceforge.net/, 2004.

[11] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[12] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, August 2006.

[13] Christian Schnellhammer and Thomas Feilkas. Steering behaviors. http://www.steeringbehaviors.de/, 2001.

[14] Inc. Silicon Graphics. Standard template library programmer's guide. http://www.sgi.com/tech/stl/Sequence.html, 2006.

[15] Bjarne Stroustrup. Container performance. Beitrag in der Newsgroup comp.lang.c++.moderated, 2003.

[16] Anrew S. Tanenbaum and James Goodman. *Computerarchitektur*. Prentice Hall, 2001.

[17] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects, 2003.

[18] Luiz Velho, Jonas Gomes, and Luiz Henrique Figueiredo. *Implicit Objects in Computer Graphics*. Springer Verlag, June 2002.