Quadtrees

Bernhard Mallinger

0707663

TU Wien

29. März 2011

Betreut durch Univ.-Ass. Dipl.-Ing. Christian Schauer, BSc

Inhaltsverzeichnis

Teil I - Theorie

1 Einführung

Quadtrees stellen eine Datenstruktur dar, welche von Binärbäumen abgeleitet ist. Sie erweitern das Binärbaum-Prinzip der rekursiven, hierarchischen Aufteilung eines Raumes auf mehrere Dimensionen. Im Allgemeinen beschränkt man sich hier zur Vereinfachung auf zwei- bzw. dreidimensionale Daten, wobei die Verallgemeinerung auf k-dimensionale Daten trivial ist.[1]

Anstatt wie bei Binärbäumen den Raum in jedem Schritt in einen linken und einen rechten Teilbaum zu zerlegen, werden bei Quadtrees vier Kinder verwendet, um alle möglichen Richtungen (links, rechts, oben, unten) in zwei Dimensionen bezüglich des aktuellen Knotens abzudecken. Der Name „Quadtree“ leitet sich hiervon ab (lat. „quadri-“: „vier-“). Analog dazu bezeichnet man dreidimensionale Bäume als „Octtrees“. Es ist weiters einfach zu sehen, dass k-dimensionale Bäume 2^k Kinder pro Knoten aufweisen.[2]

Räumliche Strukturen wie Quadtrees sind generell dann sinnvoll, wenn die Entfernung zweier Punkte für die Anwendung relevant ist. Beispiele hierfür sind unter anderem die Repräsentation von Bildern (angrenzende Bereiche besitzen oftmals ähnliche Farben), Kollisionserkennung (wo das Interesse auf sich überschneidenden Strukturen liegt), oder Bereichsabfragen in Datenbanken.

Beim Zugriff auf die so in einem Quadtree gespeicherten Daten unterscheidet man zwischen verschiedenen Anfragetypen:[3]

Point Query. Mit dieser Anfrage versucht man herauszufinden, welche Daten sich an einem bestimmten Punkt befinden. Antworten auf diese Anfrage können entweder ein Datenobjekt oder *NULL* sein.

Range Query. Hier interessiert man sich für die Knoten, welche innerhalb von Bereichen liegen, die hinsichtlich mehrerer Dimensionen ausgedehnt sein können. Zurückgegeben wird eine

[1] Vgl [FB74], Seite 9
[2] Vgl. [BF79], Seite 16
[3] Basierend auf [Knu98], Seite 559 und der Erweiterung durch [Ben75a]

möglicherweise leere Liste von Datenobjekten.

Neighborhood Query. Dieser Anfragetyp fokussiert sich auf Nähebeziehungen. Konkret werden die Knoten gesucht, welche die geringste Entfernung zu einem bestimmten Knoten aufweisen oder sich innerhalb eines gewissen Radius davon befinden. Der Rückgabewert kann hier eine Liste oder ein einzelnes Datenobjekt sein.

1.1 Funktionsprinzip

Wie eingangs erwähnt, wird bei einem Quadtree der Raum in jedem Schritt in zwei Dimensionen geteilt. In diesem Kontext bezeichnet man die Abschnitte der Unterteilung als „Quadranten". Diese Aufteilung kann, wie Abbildung 1 demonstriert, direkt am Knoten stattfinden, je nach Quadtreetyp werden hier jedoch verschiedene Strategien eingesetzt.

Die eigentlichen Daten werden je nach Quadtreetyp in allen Knoten oder nur den Blättern gespeichert. Alle Knoten eines Baumes beinhalten zusätzlich Verweise auf vier Teilbäume, welche diejenigen Daten beinhalten, die sich jeweils vollständig in der entsprechenden Richtung des Elternknotens befinden. Die Koordinaten der zu speichernden Daten bestimmen demnach die Struktur des Quadtrees.

Dieses Prinzip der rekursiven, hierarchischen Dekomposition erlaubt das effiziente Suchen in logarithmischer Zeit, sofern eine adäquate Balancierung sichergestellt ist.[4]

2 Punktbasierte Quadtrees

2.1 Point Quadtree

Dieser Quadtreetyp wurde erstmals 1974 von Finkel und Bentley in [FB74] eingeführt und stellt die Basis vieler weiterer Entwicklungen dar.[5] Point Quadtrees kann man als direkte Verallgemeinerung von binären Suchbäumen auf mehrdimensionale Räume auffassen.[6] Ähnlich zu diesen Bäumen erfolgt die Aufteilung direkt an den Koordinaten der Knoten, sowie an beiden Dimensionen gleichzeitig.

[4] Siehe Kapitel 2.1.1
[5] Vgl. [BKOS00], Seite 318
[6] Vgl. [Sam90], Seite 48

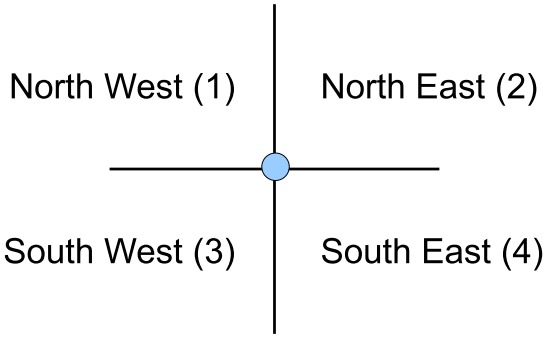


Abbildung 1: Alle Kinder eines Knotens repräsentieren Teilbäume, welche sich vollständig in NW-, NE-, SW- oder SE-Richtung des aktuellen Knotens befinden. Je nach Konvention können die Quadranten auch mittels ihrer Nummerierung referenziert werden. Diese Darstellungsform wird im Folgenden als Koordinatendarstellung bezeichnet.

Abbildung 2 zeigt eine Instanz eines Point Quadtrees in zwei üblichen Darstellungsvarianten: Baum- und Graphenform. In der Ersten wird nur die Struktur des Baumes betrachtet, die Lage der Datenknoten ist ausschließlich relativ zueinander sichtbar. Die Zweite konzentriert sich mehr auf die absolute Position der Punkte in einem Koordinatensystem und zeigt zusätzlich dazu die rekursive Dekompositionsstruktur ein. Die Bezeichnung der Knoten in der Grafik gibt hier die Reihenfolge des Einfügens an. Bereits hier wird deutlich, dass die Form des Baumes durch diese Reihenfolge determiniert wird.[7]

2.1.1 Suche

Der Algorithmus zur Suche von Knoten (Point Query) ergibt sich unmittelbar aus dem rekursiven Aufbauprinzip des Quadtrees.

Die Suche beginnt an der Wurzel. In jedem Schritt werden je ein Vergleich der x- und y-Koordinate des aktuellen Knotens mit dem gesuchten Knoten durchgeführt, wodurch der Quadrant bestimmt wird, in welchem sich der gesuchte Knoten befindet. Ist dieser Quadrant leer, d.h. ist kein Kindknoten bezüglich dieser Richtung vom aktuellen Knoten ausgehend vorhanden, wird die Suche erfolglos abgebrochen; andernfalls wird sie bei dem Kindknoten, welcher

[7] Vgl. Kapitel 2.1.2

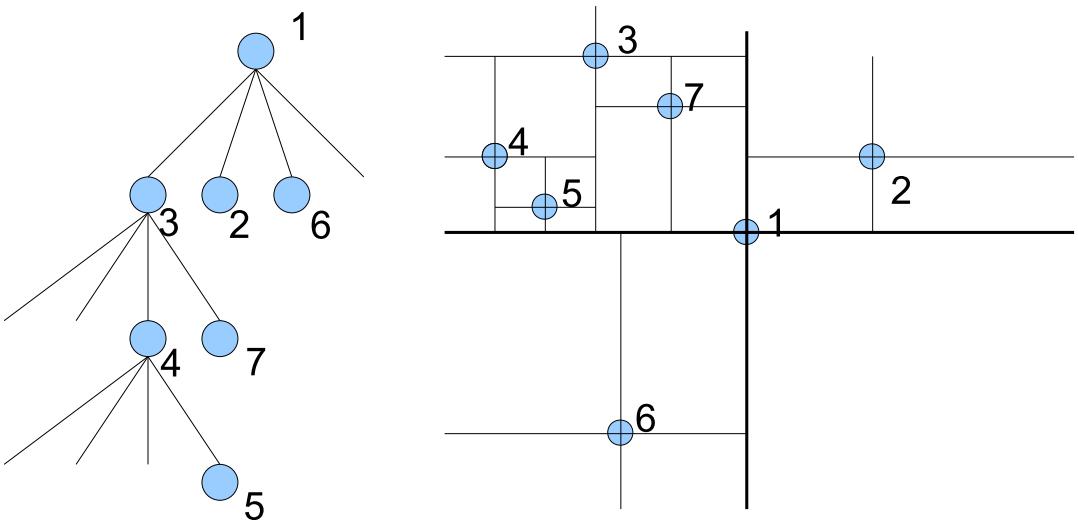


Abbildung 2: Gegenüberstellung: ein Point Quadtree in Baum- bzw. Koordinatendarstellung

diesen Quadranten abdeckt, rekursiv fortgesetzt, bis die Koordinaten des gesuchten Knotens mit jenen des aktuellen Knoten übereinstimmen.

Durch die rekursive Aufteilung kann bei N Knoten nach dem Erstellen des Quadtrees mit einer Gesamtpfadlänge[8] von $O(N \log_4 N)$ gerechnet werden, was in empirischen Untersuchungen mit Einfügungen in zufälliger Reihenfolge bestätigt wurde.[9] Dies führt in der Folge bei der Suche nach einzelnen Punkten zu einer Laufzeit von $O(\log_4 N)$. Allerdings hängt diese Größe von der Balancierung des Baumes ab, wofür es beim Point Quadtree keine Garantien gibt,[10] was im Falle einer Entartung (Worst Case) zu einer linearen Laufzeit von $O(N)$ führt.[11]

Die Bereichssuche gestaltet sich ähnlich der Punktabfrage. Der essentielle Unterschied besteht darin, dass hier die Suche möglicherweise in mehreren Quadranten eines Knotens fortgesetzt werden muss. Die zu durchsuchenden Quadranten werden bestimmt, indem überprüft wird, ob der Bereich dieses Quadranten mit dem gesuchten Bereich überlappt. Weiters wird für jeden Knoten, welcher besucht wird, festgestellt, ob dieser sich in dem spezifizierten Bereich befindet; dieser wird gegebenenfalls zurückgegeben.[12] In dieser Formulierung wird ebenso sofort deutlich, dass die gesuchte Region nicht unbedingt ein Rechteck bilden muss.

[8] engl. TPL: Total path length, d.h. die Summe der Längen aller Pfade im Baum
[9] Vgl. [FB74], Seite 4
[10] Vgl. Kapitel 2.1.2
[11] Vgl. [Sam90], Seite 52
[12] Vgl. [FB74], Seite 5f

2.1.2 Einfügen

Das Verfahren zum Einfügen neuer Daten baut direkt auf die Suche auf. Das heißt, man sucht nach dem einzufügenden Objekt und findet dieses nicht. Anstatt nun aber zu terminieren, wird an der potentiell freien Stelle das entsprechende Objekt hinzugefügt. Konkret geschieht dies durch Setzen des diesem Quadranten zugeordneten Zeigers im Elternknoten. Der asymptotische Aufwand dieser Operation ist natürlich identisch mit jenem der Suche.

Der neu eingefügte Knoten teilt den Raum wiederum an seinen x- bzw. y-Koordinaten und erweitert so die Struktur des Baumes. Es ist schnell ersichtlich, dass hier, genau wie bei einem Binärbaum, Entartungen auftreten können, da sich die Struktur dynamisch mit jedem eingefügten Element herausbildet.

Ein besonders Problem stellen hier etwa geordnete Daten da. Werden diese in der gegebenen Reihenfolge eingefügt, degeneriert der Baum zu einer linearen Liste, wodurch ein linearer Such- und somit gleichzeitig auch Einfügeaufwand entsteht, wie Abbildung 3 demonstriert.

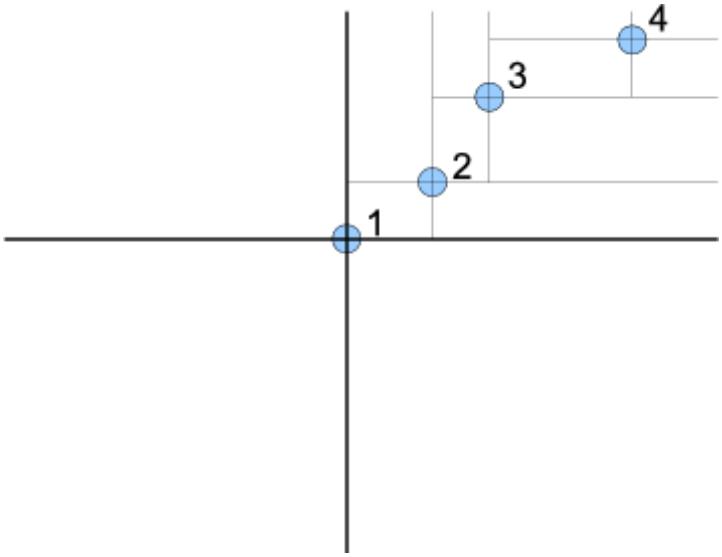


Abbildung 3: Ein Entarteter Point Quadtree

Sind andererseits beim Aufbau des Baumes bereits alle Daten verfügbar, kann durch folgende Strategie ein optimale Balancierung erreicht werden: Der Median der sortierten Liste von Daten wird als Wurzel verwendet, die restlichen Elemente werden in 4 Gruppen eingeteilt, welche den Quadranten dieser Wurzel zugeordnet werden. Auf jede dieser Unterteilungen wird der

Prozess rekursiv angewendet. Dadurch kann erreicht werden, dass für jeden Knoten kein Kind mehr als die Hälfte aller Subknoten von diesem enthält, was empirisch zu 15% geringeren Gesamtpfadlängen führt.[13]

2.1.3 Löschen

Das Entfernen eines Knotens aus einem Quadtree ist im Allgemeinen eine komplexe Operation, da die Knoten alleine die Struktur des Baumes bilden, und diese auch nach dem Löschen weiterhin gewisse Eigenschaften innehaben muss. Daher ist es notwendig, eventuelle Kinder eines entfernten Knoten gemäß dem Baumaufbau neu zu platzieren.[14] Bei Binärbäumen stellt dies kein allzu großes Problem dar, da in einem eindimensionalen Raum immer ein geeigneter Ersatz in Form des nächsten Knotens für einen zu entfernenden Knoten zur Verfügung steht.[15] Im Fall von Quadtrees muss eine Substitution hingegen Bedingungen hinsichtlich zweier Dimensionen genügen, wodurch es ein Näheverhältnis wie im eindimensionalen Fall nicht gibt.[16] Hier ist ein Ersatz nur dann möglich, wenn ein Blattknoten existiert, für den sich im Bereich zwischen den Achsen, welche durch die Koordinaten des Blattknoten bzw. deren des zu entfernenden Knoten verlaufen, kein anderer Knoten befindet.[17]

Ursprünglich wurde vorgeschlagen, alle Kinder des zu entfernenden Knotens neu einzufügen, was einen sehr aufwändigen Prozess darstellen kann.[18] Sollte kein Knoten das oben genannte Kriterium erfüllen, kann nur durch günstige Wahl des Ersatzknotens die Anzahl der neu einzufügenden Knoten minimiert werden, wie etwa in [Sam80] gezeigt wird.

Eine weitere Optimierung kann dadurch erreicht werden, dass gelöschte Knoten nicht aus dem Baum entfernt, sondern nur als gelöscht markiert werden, um so die Struktur zu erhalten. Die markierten Knoten dürfen bei zukünftigen Suchen nicht mehr zurückgegeben werden. Nach jeder Entfernung erhöht sich hierdurch der Anteil der „toten“ Knoten, was sich negativ auf die Performance aller Operationen auswirkt, und periodische Restrukturierungsmaßnahmen des Baumes erfordert. Durch dieses Verfahren kann folglich der Aufwand einzelner Löschoperationen

[13] Vgl. [FB74], Seite 8f
[14] Vgl. [FB74], Seite 8
[15] Vgl. [Sam80], Seite 2
[16] Vgl. [Sam80], Seite 2
[17] Vgl. [Sam80], Seite 3
[18] Vgl. [FB74], Seite 8

abgefedert werden, auf Kosten von regelmäßigen Wartungskosten.

2.2 Pseudo Quadtree

Der Pseudo Quadtree ist eine Abwandlung des Point Quadtrees, welche Löschen in logarithmischer Zeit sowie einfachere Rebalancierung bei Deformation ermöglicht. Diese Form des Quadtrees wurde 1982 von Overmars und van Leeuwen in [OL82] entwickelt.

Dem Pseudo Quadtree liegt die Analyse zu Grunde, dass die Probleme beim Entfernen von Knoten und bei der Balancierung bei Point Quadtrees daher stammen, dass die Datenpunkte selbst die Struktur des Baumes bilden, und so nur äußerst wenig Flexibilität vorhanden ist.[19] Der Pseudo Quadtree unterscheidet sich davon insofern, als dass die Dekomposition des Raum an theoretisch beliebigen Punkten stattfinden kann.

Konkret funktioniert das Einfügen beim Pseudo Quadtree wie folgt: Der Quadrant eines Datenpunktes wird gesucht. Ist dieser leer, wird der Knoten hier eingefügt, ohne den Raum an dessen Koordinaten weiter zu unterteilen. Befindet sich hier hingegen bereits ein anderer Knoten, muss eine neue Untergliederung eingeführt werden. Hier können verschiedene Strategien verwendet werden, um einen Teilungspunkt zu bestimmen, wodurch ein der Anwendung angepasster, gut balancierter Baum entstehen kann. Als einzige Postcondition dieser Strategien muss garantiert werden, dass die Knoten in verschiedene Quadranten gespeichert werden.[20]

Abbildung 4 zeigt einen Pseudo Quadtree, welcher dieselben Daten wie der Point Quadtree in Abbildung 2 beinhaltet.

Sollte trotz einer angepassten Strategie ein Ungleichgewicht entstehen, kann der Subbaum, dessen Wurzel der höchste Knoten ist, bei welchem eine unbalancierte Verteilung seiner Kindknoten festgestellt wird, neu strukturiert werden, indem eine neue Aufteilung der vorhandenen Knoten berechnet wird.[21] Es kann gezeigt werden, dass im allgemeinen Fall eine Teilung möglich ist, so dass sich in keinem der vier Quadranten mehr als ein Drittel der Knoten befindet.[22] Mit dieser Methode kann weiters garantiert werden, dass ein Pseudo Quadtree aufgebaut werden

[19] Vgl. [OL82], Seite 6
[20] Vgl. [OL82], Seite 6
[21] Vgl. [OL82], Seite 4f
[22] Vgl. [OL82], Seite 7

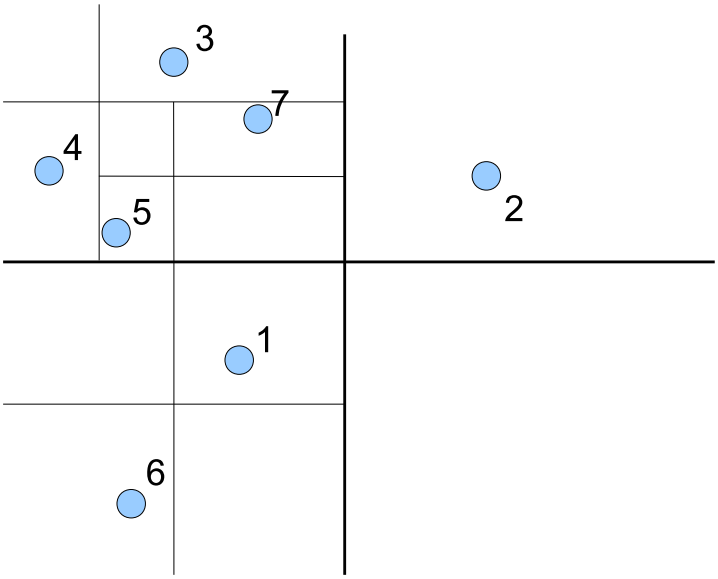


Abbildung 4: Pseudo Quadtree

kann, dessen Höhe $\lceil \log_{d+1} N \rceil$ nicht überschreitet, wobei N die Anzahl der Knoten darstellt.[23]

Durch Raumaufteilung der Punkte befinden sich Daten schließlich nur noch an den Blättern. Dies wiederum ist der optimale Fall bezüglich dem Entfernen von Knoten – ein Knoten ohne Kinder kann jederzeit ohne zusätzlichen Aufwand gelöscht werden. Da das Entfernen eines Knotens immer auch das Suchen desselben erfordert, entsteht somit ein logarithmischer Aufwand.

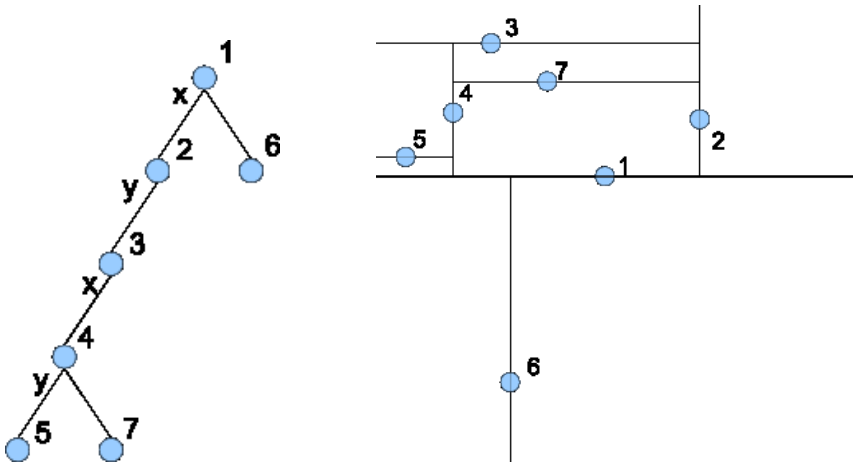
2.3 *k*-d Tree

Diese Variante wurde von Bentley kurz nach dem Point Quadtree 1974 in [Ben75b] entwickelt. Der Name k-d deutet an, dass dieser Baum k-dimensionale Schlüssel verwendet.

Der k-d Tree kann wie der Point Quadtree als Verallgemeinerung des Binärbaums gesehen werden, hier wird hingegen ein etwas anderer Ansatz verfolgt: Anstatt den Raum auf jeder Teilungsebene in allen Dimensionen zu partitionieren, was zu 2^k Kinder pro Knoten führt, wird in jedem Schritt nur eine Dimension berücksichtigt und so eine binäre Unterteilung von eigentlich mehrdimensionalen Schlüsseln ermöglicht. Diese Mehrdimensionalität wird bei k-d Trees weiters dadurch ausgedrückt, dass die Dimensionen, bezüglich derer an einer gewissen

[23] Vgl. [OL82], Seite 8

Tiefe geteilt wird, nach verschiedenen Strategien alternieren. Im zweidimensionalen Fall könnte hier etwa bei geraden Tiefenebenen an der x-Achse bzw. bei ungeraden an der y-Achse geteilt werden, wie etwa Abbildung **??** zeigt. Ursprünglich wurde genau dieses Abwechslungsmuster vorgeschlagen,[24] wobei es, an die Daten angepasst, dynamisch gewählt werden kann[25]. Verwendet man hier einen dynamischen Ansatz, muss jedoch bei jedem Knoten der Diskriminator mitgespeichert werden.



(a) (b)

Abbildung 5: Ein k-d Tree in Baum- bzw. Koordinatendarstellung.

(a) Die Beschriftung der Unterteilungen gibt die Achse an, bezüglich welcher die Knoten aufgeteilt werden.

(b) Hinsichtlich den vorhergegangenen Abbildung wurden hier die Koordinaten der Knoten leicht verändert, da in diesem Fall ein äußerst schlecht balancierter k-d Tree entstehen würde, was der Autor aus Demonstrationszwecken vermeiden möchte.

Dieser Typ bietet gegenüber einem Point Quadtree Vorteile bezüglich des Platzbedarfs: anstatt 2^k sind hier nur 2 Kinder pro Knoten erforderlich. Dies löst das Problem von vielen unnötigen Leereinträgen, welche bei einem Point Quadtree bei höherdimensionalen Datensätzen auftreten können.

Weiters müssen bei einem Point Quadtree bei einer Suche in jedem Schritt alle Dimensionen verglichen werden. Bei einer großen Anzahl von Dimensionen kann dies überflüssig sein,

[24] Vgl. [Ben75b], Seite 2
[25] Vgl. [Sam90], Seite 66

wenn sich etwa Datensätze bereits durch eine echte Teilmenge ihrer Schlüssel einteilen lassen. In diesem Fall reicht das Überprüfen dieser Teilmenge, was in einem Point Quadtree nicht realisierbar ist, sich mit dem k-d Tree jedoch effizient verwirklichen lässt.[26]

Die Operationen Suche und Einfügen ergeben sich direkt aus dem Funktionsprinzip des k-d Trees, beide benötigen offensichtlich eine logarithmische Laufzeit im Average Case[27] (sofern der Baum nicht entartet). Schwieriger ist hingegen das Löschen: Nach dem Entfernen eines Knotens darf die Struktur des k-d Trees nicht verletzt werden, weshalb ein geeigneter Ersatz gefunden werden muss. Für dieses Problem wurde eine Strategie entwickelt, welche auf der Lösung des analogen Problems bei Binärbäumen basiert und zugleich die Mehrdimensionalität im k-d Tree berücksichtigt. Der wesentliche Aspekt dieses Verfahrens ist, dass der Ersatzknoten nicht wie beim Binärbaum immer ein Blatt ist, sondern im Allgemeinen auch Kinder hat, wodurch der Ersatzknoten auch substituiert werden muss.[28] Anstatt im Worst Case (bei Löschung der Wurzel) n Neueinfügungen mit dem naiven Ansatz benötigt diese Methode nur $O(\log n)$ (bei einer Knotenanzahl von n).

Um diesem Problem gänzlich zu entgehen kann ähnlich dem Pseudo Quadtree[29] auch ein Pseudo k-d Tree[30] verwendet werden, in welchem analog zum Pseudo Quadtree die Unterteilung des Raumes nicht an den Koordinaten der Knoten, sondern an beliebigen Punkten zwischen den Knoten vorgenommen wird, und so Daten nur in Blättern gespeichert werden.

3 Bereichsbasierte Quadtrees

Wie im Kapitel 2 ausführlich erläutert wurde, erfolgt die räumliche Dekomposition bei punktbasierten Quadtrees immer anhand der gegebenen Punkte.[31] Dieses Kapitel beschäftigt sich mit Quadtrees, deren Form vordergründig a priori durch die Struktur des Raumes determiniert wird, was in etwa mit dem Prinzip von Tries vergleichbar ist.[32]

[26] Vgl [Sam90], Seite 66
[27] Vgl. [Ben75b], Seite 1
[28] Für eine detaillierte Beschreibung dieses Verfahrens siehe [Ben75b], Seite 7
[29] Siehe Kapitel 2.2
[30] Vgl. [OL82], Seite 13ff
[31] Bei Point bzw. k-d Trees gilt dies direkt; bei Pseudo Quadtrees hingegen besteht die Abhängigkeit vielmehr indirekt: die Aufteilung stellt eine Funktion der Punkte dar.
[32] Vgl. [Sam90], Seite 85

Weiters wird in diesem Abschnitt nur insofern auf die verschiedenen Operationen eingegangen, als sie sich von punktbasierten Quadtrees unterscheiden.

3.1 MX Quadtree

Der MX Quadtree zeichnet sich durch seine Art der Raumunterteilung aus: Der Bereich, in dem die Datenpunkte liegen, wird unabhängig von den konkreten Daten unterteilt, und zwar rekursiv in vier gleichgroße Quadranten (im zweidimensionalen Fall), bis auf der untersten Ebene 1×1 große Felder erreicht werden, welche die Daten beinhalten.[33]

Diese Art der Dekomposition setzt folgendes voraus:

(1) **Konstanter Bereich.** Die Region, in welcher sich Daten befinden, muss a priori bekannt sein und während des gesamten Programmablaufs konstant bleiben.

(2) **Diskrete Koordinaten.** Die Schlüssel dürfen nur diskrete Werte annehmen, ähnlich den Einträgen einer Matrix, woher sich auch der Name „MX“ ableitet.[34]

(3) **Die Fläche des Raumes ist quadratisch mit einer Zweierpotenz als Seitenlänge.** Da der Raum aus praktischen Gründen in 4 gleichgroße Teile partitioniert wird, entsteht eine Gesamtgröße von $2^n \times 2^n$. Bei Datenregionen, welche nicht diese Ausdehnung besitzen, können Leereinträge verwendet werden, um diese Größe zu erreichen.

Abbildung 6 zeigt einen MX Quadtree auf Basis der bei den vorherigen Grafiken verwendeten Datensatzes. Dabei fällt auf, dass die Aufteilung des Raumes hier unabhängig von der Reihenfolge des Einfügens der Knoten ist, was eine Konsequenz aus der Partitionierung auf Basis der Raumstruktur ist. Dies führt auch zu einer konstanten Höhe h des Baumes ohne Bezug zu den konkret vorhandenen Datenblättern, wodurch wiederum nur eine begrenzte Anzahl von Knoten $(2^h * 2^h)$ in einem MX Quadtree gespeichert werden können.[35] Diese Einschränkung ergibt sich durch die obigen Bedingungen.

Ein klassischer Anwendungsfall dieses Prinzips sind Rastergrafiken, welche ein geeignetes mentales Modell für MX Quadtrees darstellen, da hier jeder Datenpunkt (Pixel) offensichtlich

[33] Vgl. [SW85], Seite 1f
[34] Vgl. [Sam90], Seite 86
[35] Vgl [Sam90], Seite 88

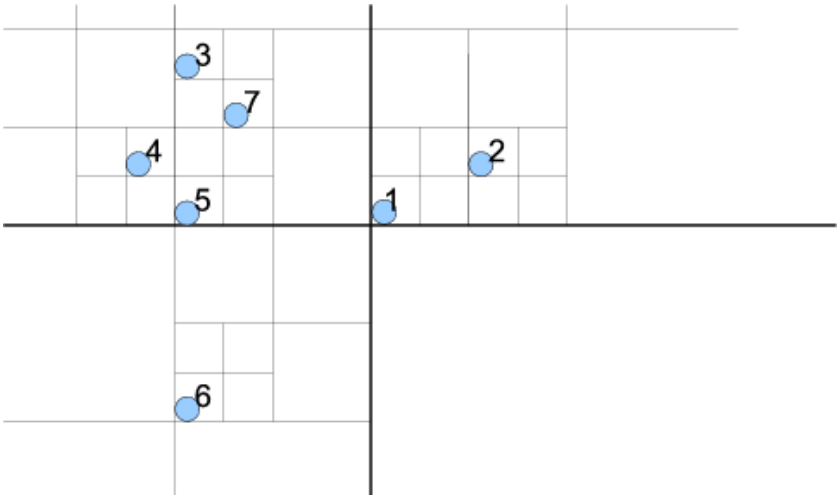


Abbildung 6: MX Quadtree. Diese Instanz ist eher spärlich besetzt.

genau 1×1 groß ist und einer Koordinate zugeordnet ist.[36] Davon abgeleitet spricht man in diesem Kontext von weißen, schwarzen und grauen Knoten[37]:

Schwarze Knoten sind entweder Datenblätter oder Knoten mit vier schwarzen Kindern.

Graue Knoten stellen Knoten dar, welche mindestens ein graues Kind besitzen oder mindestens je ein schwarzes und ein weißes Kind.

Weiße Knoten sind leere Blätter ohne Daten bzw. Knoten, die weder direkt noch indirekt mittels Kindverweise auf Daten verweisen. Diese werden im Allgemeinen nicht explizit repräsentiert, also in der implementierten Datenstruktur nicht gespeichert.

Eine konkrete Instanz eines MX Quadtree enthält somit nur schwarze und graue Knoten. Das Nicht-Repräsentieren weißer Knoten führt dazu, dass die Struktur des Baumes erst beim Einfügen aufgebaut wird. Dies ist jedoch nur eine Optimierung; es wäre genauso möglich, den vollständigen Baum mit weißen Knoten bzw. Blättern aufzufüllen, welche später grau oder schwarz eingefärbt bzw. von schwarzen Blättern ersetzt werden.

Das Löschen von Knoten im MX Quadtree ist weiters trivial, da im Gegensatz zu Point Quadtrees hier die Daten keinen Einfluss auf die Struktur des Baumes haben und nur in Blättern gespeichert werden. Nach dem Entfernen des eigentlichen Datenblattes muss nur die Färbung

[36] Vgl. [HS79]
[37] Vgl [Gar82], Seite 2

der Knoten am Pfad von der Wurzel zu diesem Blatt überprüft und gegebenenfalls angepasst werden.

3.2 PR Quadtree

Die Einschränkung durch diskrete Koordinaten bei MX Quadtrees verhindert oft deren praktische Anwendung. Nichtsdestoweniger ist die Eigenschaft, dass dieser Baumtyp nicht entarten kann (da die Einfügereihenfolge keine Rolle spielt), oft sehr sinnvoll.

Der PR Quadtree stellt eine Möglichkeit dar, diesen Zwiespalt zu lösen. Er ist eine Kombination von Point und Region Quadtree, wovon sich auch der Name ableitet (**P**oint **R**egion Quadtree).[38] Hier orientiert sich die Aufteilung des Raumes ähnlich den MX Quadtrees wiederum an der rekursiven Dekomposition eines fixen Bereiches, diese kann jedoch beliebig tief werden.

Die Einfügeoperation läuft wie folgt ab: Der tiefste Quadrant, welcher die Koordinaten eines einzufügenden Knotens beinhaltet, wird gesucht. Ist dieser leer, kann der Knoten hier platziert werden. Andernfalls muss eine weitere Unterteilung vorgenommen werden, was durch Partitionierung des Quadranten in gleichgroße Teile geschieht. Der Knoten, welcher sich bereits in diesem Quadranten befand, wird in die neue Unterteilung überführt, und es wird neuerlich versucht, den einzufügenden Knoten zu platzieren. Wenn hier erneut beide Knoten in den selben Quadranten gespeichert werden würden, muss der Prozess der Partitionierung rekursiv wiederholt werden. Sind die Koordinaten der Knoten voneinander verschieden, ist trivialerweise garantiert, dass sie nach endlich vielen Schritten in verschiedene Quadranten eingeteilt werden. Sollte es in der Anwendung möglich sein, dass zwei Knoten mit äquivalenten Koordinaten auftreten, müssen diese speziell behandelt werden, etwa durch eine Überlaufliste.

Bei diesem Vorgang ist die Dekompositionsstruktur offensichtlich unabhängig von den konkreten Schlüsseln, wodurch die Einfügereihenfolge irrelevant wird und der Baum nicht im Sinne von Point Quadtrees entarten kann.[39]

Diese Strategie kann jedoch problematisch werden, wenn Datenpunkte nahe aneinander liegen. Hier müssen unter Umständen viele Unterteilungen getroffen werden, bis sie in verschiedene Quadranten eingeteilt werden können. Diese Struktur ist daher nur für entsprechend

[38] Vgl. [Sam90], Seite 92f
[39] Vgl [Sam90], Seite 96

verschiedene Schlüsseltypen geeignet.[40]

Das Löschen aus einem PR Quadtree ist wiederum unkompliziert, da sich die Daten nur in Blättern befinden. Durch das Entfernen eines Blattes können Unterteilungen überflüssig werden, wenn etwa ein Knoten nur ein Kind besitzt. In diesem Fall kann der entsprechende Knoten einfach durch das Kind ersetzt werden.

[40]Vgl [Sam90], Seite 95f

Teil II - Praktikum

4 Problemstellung

In dem an die Bachelorarbeit angeschlossenen Praktikum habe ich mich mit einer Problemstellung aus dem Spiel „Unknown Horizons“ beschäftigt. Um das Umfeld der Aufgabenstellung deutlich zu machen, werde ich zuerst auf dieses Programm näher eingehen.

4.1 Unknown Horizons

„Unknown Horizons“ ist ein klassisches Aufbaustrategiespiel, in welchem eine Inselgruppe besiedelt wird. Das Szenario ist historisch gesehen vergleichbar zur Besiedelung Amerikas durch europäische Siedler.

Ziel des Spieles ist, eine florierende Wirtschaft aufzubauen, welche die Bevölkerung mit Waren versorgt, die wiederum mit Steuern die laufenden Kosten der Produktionsketten und den Bau neuer Gebäude finanzieren. Aus Spieler_innensicht geschieht der Aufbau der Siedlung durch das Errichten von Gebäuden, welche anschließend weitestgehend autonom arbeiten.

Das Programm ist unter der GPL-2[41] lizenziert und somit freie, offene Software. Es befindet sich noch in einer relative frühen Phase der Entwicklung, obschon der bereits implementierte Teil „spielbar“ ist. Nähere Informationen sind unter `http://www.unknown-horizons.org` verfügbar.

4.2 Technische Details

Die Spiellogik ist in Version 2.6 der Sprache Python[42] implementiert und baut auf der Engine FIFE[43] auf. Weiters läuft das Spiel in nur einem Thread ab, weswegen jede Berechnung den Programmablauf blockiert, und somit sehr effizient sein muss.

Die Spielwelt stellt eine zweidimensionale Fläche dar. Sie ist in 1x1 große Tiles unterteilt, welche daher die kleinste Größeneinheit sind. Die Zuordnung von Tiles zu Koordinaten ist

[41] `http://www.gnu.org/licenses/gpl-2.0.html`, abgerufen am 01.03.2011
[42] `http://www.python.org`, abgerufen am 01.03.2011
[43] `http://fifengine.de`, abgerufen am 01.03.2011

bijektiv und über den ganzen Spielverlauf hindurch konstant. Diese Tiles (dt.: „Fliesen“) sind in verschiedene geographische Typen unterteilt (unter anderem Wasser, Land, Küste), denen zur graphischen Darstellung ein entsprechendes Bild zugeordnet ist. Desweiteren enthalten sie einen Verweis auf etwaige Gebäude, welche den Platz an dieser Koordinate einnehmen und stellen so die Basis der räumlichen Verwaltung der Inseln dar.

Aus diesem System folgt, dass alle Objekte im Spiel nur ganzzahlige, diskrete Koordinaten aufweisen.

4.3 Problembeschreibung

Die Problemstellung, welche in diesem Praktikum behandelt wird, ist eine Bereichsabfrage:

Im Spiel hat jedes Gebäude einen Einflussbereich, innerhalb welchem gewisse Aktionen ausgeführt werden können. Ein Holzfäller etwa kann nur Bäume fällen, welche sich innerhalb seines Radius befinden, eine Schule nur Einwohner des entsprechenden Bereiches betreuen. Abbildung 7 zeigt ein Gebäude, dessen Einflussbereich eingefärbt wurde.

Abbildung 7: Screenshot aus „Unknown Horizons“. Abbgebildet ist hier ein Bootsbauer inklusive seinem Aktionsradius, welcher den Bereich beschreibt, aus welchem dieser Rohstoffe für den Bootsbau einsammeln kann.

Dieses Gebiet umfasst alle Tiles, welche folgende Ungleichung erfüllen

$$Distanz(Gebaeude, Tile) \leq Radius\ des\ Einflussbereichs \tag{1}$$

sowie sich

(1) im Areal der Siedlung

(2) an markierbaren Stellen

befinden. Da die Fläche eines Gebäudes eine beliebige rechteckige Form darstellen kann, entsteht durch diese Ungleichung ein Rechteck mit abgerundeten Ecken, welches durch Bedingungen 1 und 2 beliebig beschränkt sein kann.

Nachdem dieser Bereich beim Gebäudebau ein zentrales Kriterium darstellt, muss dieser dargestellt werden, wenn die Maus über die Spielwelt bewegt wird, um eine geeignete Position für ein zu bauendes Gebäude zu suchen, was offensichtlicherweise auch sehr schnell geschehen muss, um hier Verzögerungen zu vermeiden. Die Einfärbungsroutine wird nur dann aufgerufen, wenn der Mauszeiger einen Moment über einer gewissen Koordinate verweilt, um „Ruckler“ zu vermeiden. Im allgemeinen Anwendungsfall wird anschließend auf die Darstellung des Bereiches gewartet, um sicher zu stellen, dass das Gebäude mit diesem Aktionsradius die gewünschte Funktion übernehmen kann.

Die ursprüngliche Implementierung dieser Aufgabe ist für einen flüssigen Spielablauf zu ineffizient, besonders erfahrene SpielerInnen stoßen hier durch ihr zügiges Spieltempo auf Verzögerungen. Wie später noch deutlich wird, entstehen die Verzögerungen im Wesentlichen durch die Berechnung der zu markierenden Tiles, das Übergeben dieser an die Engine sowie das eigentliche Einfärben. In dieser Arbeit werde ich mich auf die Optimierung des ersten Teils beschränken, was für zufriedenstellende Laufzeitergebnisse alleine in der Praxis nicht ausreicht. Die Performance der Engine ist hier weiters schwer messbar, was verschiedene Gründe hat: Diese ist in einer anderen Programmiersprache (C++) als die Spiellogik implementiert, wodurch bei jedem Funktionsaufruf (etwa die Übergabe der einzelnen Tiles) Zeit im Wrapper verbraucht wird, welcher das Interface bereitstellt. Die Markierungsfunktion erledigt weiters nur einen Teil

der Arbeit – das eigentliche Rendern findet in periodischen Durchläufen statt, in welchen das gesamte Bild berechnet wird. In diesen Prozess fließen somit viele äußere Faktoren ein.

4.4 Ursprünglicher Ansatz

Die bisherige Methode zur Speicherung der Tiles stellt eine Hashmap dar, in welcher die Koordinaten der Insel als Schlüssel auf die Tileobjekte verweisen. Es sei darauf hingewiesen, dass hier Nähebeziehungen nicht repräsentiert werden.

Der Lösungsansatz besteht im Wesentlichen aus drei Schritten:

(1) Berechnung der Koordinaten innerhalb des Einflussbereichs.

(2) Lookup dieser Koordinaten in der Hashmap.

(3) Überprüfung der Tiles bezüglich Siedlungszugehörigkeit und Markierbarkeit.

Schritt 1 kann äußerst schnell durchgeführt werden. Zusammengefasst berechnet der verwendete Algorithmus die Grenze eines Viertels des Bereiches, die restlichen Viertel werden gespiegelt. Diese Begrenzungen können schließlich einfach zeilenweise aufgefüllt werden. In dieser Form ist der Ansatz nicht optimal (die Berechnung eines Achtels würde auch ausreichen), in der Praxis ist hingegen der Anteil an der gesamten Laufzeit so gering, dass weitere Optimierungen nicht gerechtfertigt werden könnten.

Der nächste Schritt kann für einzelne Elemente auch sehr schnell umgesetzt werden, bei großen Flächen entsteht hier jedoch durch die Anzahl der Tiles eine messbare Verzögerung. Es ist nicht möglich, mehrere angrenzende Tiles gleichzeitig anzusprechen, jedes muss einzeln aus der Hashmap abgefragt werden. Besonders wenn viele Tiles am Meer liegen, wie es etwa in Abbildung 7 der Fall ist, scheitern alle diese Lookups, denn die Koordinatenberechnung hat keine Informationen über die Struktur der Insel.

Der dritte Schritt ist notwendig, da die Hashmap der Insel für verschiedene andere Anwendungsfälle sämtliche Tiles der Insel beinhalten muss; Beispiele hierfür sind etwa das Überprüfen, ob sich eine Koordinate auf einer Insel befindet, oder Pathfinding für unabhängige Einheiten. Eine spezielle Hashmap zur Markierung könnte dies vereinfachen.

5 Neuer Ansatz

Nachdem bei diesem Markierungsproblem die Lage der Tiles zueinander eine entscheidende Rolle spielt, ist es naheliegend, dieses Problem mittels Quadtrees zu lösen. Diese Datenstruktur kann die Struktur der Inseln auf eine Weise abgebilden, welche der Suche ermöglicht, Gebiete am Meer bzw. außerhalb der Siedlung schnell zu ignorieren sowie größere, zusammenhängende Blöcke effizient als innerhalb des Bereiches liegend zu identifizieren.

Der Aufbau und die Wartung des Quadtrees können zwar je nach Quadtreetyp einen Mehraufwand darstellen, dieser ist jedoch nur beim Spielstart (währenddessen ohnehin mit einer Verzögerung gerechnet wird) bzw. seltener beim Bau verschiedener Gebäude, welche den Einflussbereich der Siedlung erweitern, erforderlich. Insofern ist es hier sinnvoll, diesen Overhead in Kauf zu nehmen, um Verzögerungen während des Baumodus zu vermeiden.

Mit einer schnelleren Laufzeit kann vor allem dann gerechnet werden, wenn viele Bereiche bei der Suche ausgeschlossen werden können. In Abbildung 7 liegen ungefähr die Hälfte der Tiles im Einflussgebiet des Gebäudes im Meer und sind somit im Quadtree nicht vorhanden. Bei einer günstigen Partitionierung des Raumes ist daher zu erwarten, dass auf relativ hoher Ebene im Quadtree gewisse Kinder von Knoten, welche diesen Bereich abdecken, in der Terminologie von Abschnitt 3.1 „weiß“ und somit durch *NULL* repräsentiert sind, wodurch der Suchalgorithmus die Information bekommt, dass dieses gesamte Gebiet nicht weiter berücksichtigt werden muss. In der ursprünglichen Implementierung ist diese Information nicht vorhanden, wodurch jede Koordinate dieses Bereichs in der Hashmap unnötigerweise abfragt wird.

Da die Unterteilung des Raumes in Quadraten vorgenommen wird und der Einflussbereich der Gebäude ein Rechteck mit abgerundeten Ecken ist, muss der Algorithmus an den Rändern immer bis zur Blattebene absteigen, um die genaue Grenze zu bestimmen.

5.1 Wahl des Quadtreetyp

Die Art der im Quadtree zu speichernden Daten sind ausschließlich 1×1 große Tiles. Da man diese ebenso als Punkte interpretieren kann, eignen sich für diese Anwendung punktbasierte sowie bereichsbasierte Quadtreetypen gleichermaßen.

Weiters liegt hier eine fixe Raumaufteilung vor: Siedlungen sind durch die Ausmaße der Insel begrenzt und bedecken während des Spielverlaufs im Allgemeinen das gesamte Areal. Dies bedeutet, dass im Falle von punktbasierten Quadtrees, welche an sich entarten können, optimierte Bäume berechnet werden können, um diesem Problem zu entgegnen. Dadurch würde die Struktur dieses Typs jener der bereichsbasierten Varianten ähneln, da alle Koordinaten innerhalb einer Inseln im Quadtree repräsentiert werden müssen, und die effizienteste Möglichkeit, dies zu tun, ist eine Aufteilung jedes Bereiches genau in der Mitte, was exakt der Strategie des MX Quadtrees entspricht.

Aus diesen Gründen sowie der Einfachheit des Löschens, habe ich mich dazu entschieden, die gestellte Aufgabe mit einem MX Quadtree zu lösen. Es ist jedoch anzumerken, dass im Prinzip jeder der vorgestellten Typen für diese Problemstellung verwendet werden kann.

5.2 Implementierung

Für diese Problemstellung ist nur das Suchen von Tiles innerhalb eines Bereiches wirklich zentral, daher werde ich hier nur darauf eingehen und die Einfüge- und Löschoperationen außen vor lassen. Listing 1 zeigt diese Suchoperation schematisch in der Programmiersprache Python.

Diese Implementierung stellt eine klassische rekursive Suche dar. Anstatt wie oft üblich die entsprechenden Tiles in einer Liste zu sammeln und diese zurückzugeben, wurde hier die Variante gewählt, eine Callback-Funktion auf die betreffenden Objekte anzuwenden. Dies erspart eben den Listenerstellungsoverhead, sowie das Zurückgeben durch möglicherweise stark geschachtelten Aufrufe.

Die Suche beginnt bei einem Knoten, welcher in der Folge die Suche bei allen betroffenen Kindern fortsetzt. In Zeile 19 werden durch die `if`-Abfrage „weiße“ Knoten ausgeschlossen. Direkt darunter findet eine Optimierung statt, in welcher in jedem Schritt überprüft wird, ob sich ein Kind vollständig im zu markierenden Bereich befindet. Ist dies der Fall, werden ohne weitere Checks alle Kinder dieses Quadranten besucht. Auf der Blattebene (Zeile 13) wird schließlich trivialerweise nur mehr der aktuelle Knoten abgearbeitet.

Listing 1: Python-ähnlicher Pseudocode zur Lösung des Markierungsproblems

```
"""
Klasse der Knoten des MX Quadtrees
"""
class Node(object):

  [...]

  """
  Rekursive Methode der Klasse Node, welche einen Callback auf alle
  Blaetter anwendet, die innerhalb der Bereichsspezifikation "area"
  liegen. Sie kann dazu verwendet werden, alle Tiles im Aktionsradius
  eines Gebaeudes einzufaerben.
  """
  def visit_radius_tiles(self, area, callback)
    if self.is_leaf():
      # Auf Blattebene muss nur ueberprueft werden, ob der Callback auf
       den aktuellen Knoten angewendet werden soll
      if area.includes(self):
        callback(self)
    else:
      quadrants = self.get_overlapping_quadrants(area)
      for quadrant in quadrants:
        if quadrant != None:
          # Optimierung: Sollte der ganze Quadrant im Bereich liegen,
          muessen die einzelnen Subknoten nicht mehr einzeln
          ueberprueft werden.
          if self.check_full_quadrant_included(quadrant, area):
            quadrant.visit_all_tiles(callback)
          else:
            quadrant.visit_radius_tiles(area, callback)
```

6 Analyse

Zur Überprüfung der Laufzeit der Verfahren habe ich einige Testfälle bestimmt, welche aus je einem Gebäude einer Siedlung bestehen. Diese werden vom Testcode selektiert, was in der Folge den gleichen Code ausführt, wie das Markieren der Tiles im Einflussgebiet im Baumodus. Um Umgebungseinflüsse zu minimieren wird diese Selektion 100 Mal durchgeführt und anschließend der Mittelwert und die Standardabweichung berechnet.

Die verschiedenen Gebäudetypen besitzen unterschiedliche große Grundflächen und Radien. Aus diesem Grund kommen in den Testfällen verschiedene Typen vor. Tabelle **??** zeigt hierbei die Ergebnisse der Messung. Es fällt schnell auf, dass die Differenzen zwischen den Ansätzen variieren. Genauer gesagt wird hier eine Korrelation zwischen der Größe des zu berechnenden

Bereichs und den Unterschieden der Laufzeiten deutlich. Daher werde ich die Ergebnisse in drei Kategorien analysieren:

Große Bereiche. Die ersten beiden Testfälle zeigen etwa, dass das neue Verfahren besonders bei Gebäuden mit hohen Radien besonders effizient ist. Die Ursache dafür ist der wirkungsvolle Umgang mit großen Quadranten, welche vollständig innerhalb des zu berechnenden Bereiches liegen. Hier muss unter Umständen am Rand des Bereiches nicht einmal bis zur letzten Ebene abgestiegen werden. Dies trifft zu, sofern die Grenze der Insel innerhalb des Bereiches liegt. In diesem Fall können oftmals Quadranten auf hoher Ebene im Baum als komplett im Bereich liegend identifiziert und markiert werden, auch wenn der Quadrant auch Flächen auf der Karte beinhaltet, welche nicht zur Siedlung gehören. Diese sind nämlich „weiß“, also nicht repräsentiert, und werden folglich implizit ignoriert und müssen nicht explizit berücksichtigt werden.

Da beim ursprüngliche, auf Hashmaps basierende Ansatz jedes Tile einzeln abgefragt werden muss, wächst hier die Laufzeit direkt mit dem Quadrat des Radius. Dieser Effekt kann mit dem Quadtreeansatz abgefedert werden, was sich in der Verbesserung von fast der Hälfte der ursprünglichen Zeit widerspiegelt.

Mittlere Bereiche. Obwohl der Marktplatz und der Siedler den selben Radius aufweisen, ist hier die Laufzeit deutlich verschieden, da die Grundfläche unterschiedlich groß sind. Bei der

	Alter Ansatz		Neuer Ansatz		Differenz	
Use Case	time	std deriv	time	std deriv	absolut	relativ
Kontor Strand, viele Gebäude	0,00541	0,00019	0,00282	0,00005	-0,0025	52%
Lager Strand, viele Gebäude	0,00488	0,00008	0,00276	0,00005	-0,0021	56%
Marktplatz in Strandnähe, viele Gebäude	0,00240	0,00005	0,00209	0,00005	-0,0003	87%
Marktplatz Strand, viele Gebäude	0,00229	0,00005	0,00184	0,00005	-0,0004	80%
Siedler Siedlungsrand, viele Gebäude	0,00128	0,00006	0,00092	0,00006	-0,0003	72%
Siedler Land, viele Gebäude	0,00188	0,00017	0,00177	0,00004	-0,0001	94%
Siedler Strand, viele Gebäude	0,00141	0,00007	0,00104	0,00003	-0,0003	74%
Siedler Strand, wenige Gebäude	0,00138	0,00011	0,00124	0,00007	-0,0001	90%
Siedler in Strandnähe, viele Gebäude	0,00166	0,00004	0,00151	0,00005	-0,0001	91%
Weber Strand, wenige Gebäude	0,00064	0,00003	0,00068	0,00006	0,00004	106%
Weber Land, viele Gebäude	0,00088	0,00003	0,00142	0,00009	0,00053	160%

Tabelle 1: Laufzeitmessungen beider Ansätze im Vergleich

Legende:

Gebäude	Größe	Radius
Kontor	3×3	24
Lager	2×2	24
Marktplatz	6×6	12
Siedler	2×2	12
Weber	2×2	8

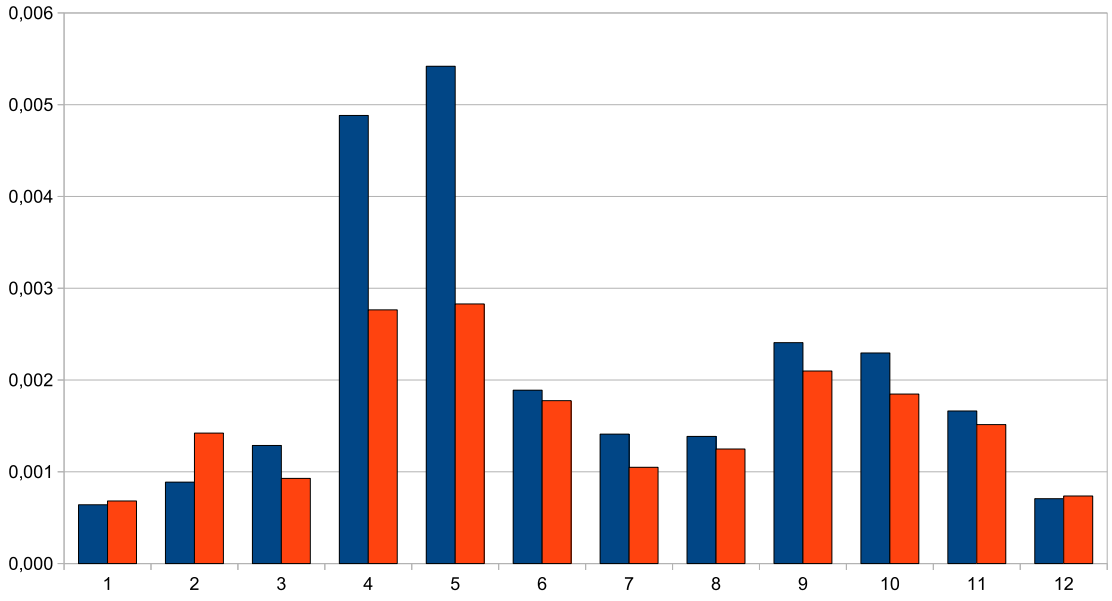


Abbildung 8:

Abbildungsverzeichnis

Alle in dieser Arbeit verwendeten Bilder wurden vom Autor erstellt.

Listings

Tabellenverzeichnis

Literatur

[Ben75a] Bentley, Jon L.: A survey of techniques for fixed radius near neighbor searching. Stanford, CA, USA : Stanford University, 1975 (333). – Forschungsbericht. – 444

[Ben75b] Bentley, Jon L.: Multidimensional binary search trees used for associative searching. In: *Commun. ACM* 18 (1975), September, 509–517. http://dx.doi.org/http://doi.acm.org/10.1145/361002.361007. – DOI http://doi.acm.org/10.1145/361002.361007. – ISSN 0001–0782

[BF79] Bentley, Jon L. ; Friedman, Jerome H.: Data Structures for Range Searching. In: *ACM Comput. Surv.* 11 (1979), December, 397–409. http://dx.doi.org/http://doi.acm.org/10.1145/356789.356797. – DOI http://doi.acm.org/10.1145/356789.356797. – ISSN 0360–0300

[BKOS00] Berg, Mark de ; Kreveld, Marc van ; Overmars, Mark ; Schwarzkopf, Otfried: *Computational Geometry: Algorithms and Applications*. Second Edition. Springer-Verlag, 2000. – 367 S. http://www.cs.uu.nl/geobook/

[FB74] Finkel, Raphael A. ; Bentley, Jon L.: Quad Trees: A Data Structure for Retrieval on Composite Keys. In: *Acta Inf.* 4 (1974), S. 1–9

[FGPR91] Flajolet, Philippe ; Gonnet, Gaston ; Puech, Claude ; Robson, J. M.: The analysis of multidimensional searching in quad-trees. In: *Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 1991 (SODA '91). – ISBN 0–89791–376–0, 100–109

[Fre60] Fredkin, Edward: Trie memory. In: *Commun. ACM* 3 (1960), September, 490–499. http://dx.doi.org/http://doi.acm.org/10.1145/367390.367400. – DOI http://doi.acm.org/10.1145/367390.367400. – ISSN 0001–0782

[Gar82] Gargantini, Irene: An Effective Way to Represent Quadtrees. In: *Commun. ACM* 25 (1982), Nr. 12, S. 905–910

[HS79] Hunter, Gregory M. ; Steiglitz, Kenneth: Operations on Images Using Quad Trees. In: *PAMI* 1 (1979), April, Nr. 2, S. 145–153

[Knu98] Knuth, Donald E.: *The Art Of Computer Programming, Volume 3: Sorting and Searching*. Second Edition. Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 1998. – ISBN 0–201–89685–0

[Lue78] Lueker, George S.: A Data Structure for Orthogonal Range Queries. In: *FOCS*, 1978, S. 28–34

[OL82] Overmars, Mark H. ; Leeuwen, Jan van: Dynamic Multi-Dimensional Data Structures Based on Quad- and *K-D* Trees. In: *Acta Inf.* 17 (1982), S. 267–285

[Sam80] Samet, Hanan: Deletion in Two-Dimensional Quad Trees. In: *Commun. ACM* 23 (1980), Nr. 12, S. 703–710

[Sam84] Samet, Hanan: The Quadtree and Related Hierarchical Data Structures. In: *ACM Comput. Surv.* 16 (1984), Nr. 2, S. 187–260

[Sam90] Samet, Hanan: *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990

[SW85] Samet, Hanan ; Webber, Robert E.: Storing a collection of polygons using quadtrees. In: *ACM Trans. Graph.* 4 (1985), July, 182–222. `http://dx.doi.org/http://doi.acm.org/10.1145/282957.282966`. – DOI http://doi.acm.org/10.1145/282957.282966. – ISSN 0730–0301