Tree Data Structures: Quad-/Oc-KD-Trees

Siobhan-Lillian Hönig

13. Oktober 2021

Inhaltsverzeichnis

1 Einführung

Mit der Weiterentwicklung von rechenfähigen Computern wurde zum einen der mögliche Speicherplatz für Daten immer größer zum andern umfangreicher. Um diese Daten besser zu sortieren und um schneller darauf zu zugreifen, wurden und werden sogenannte Datenstrukturen genutzt. Die Daten werden in bestimmten Formen gespeichert, diese Strukturen können zum Beispiel wie eine Schlange aufgebaut werden, sowie ein Graph aber auch einem Baum ähneln. In dieser Seminarabeit werden Baumstrukturen erklärt. Aufgrund dessen, dass die Informatik sehr aufgefächert ist, sind sehr viele Spezifikationen der Bäume auf bestimmte Datenmengen entstanden. Wie bereits erwähnt ermöglichen Datenstrukturen eine effiziente Suche der Daten, ich werde drei dieser Baumstrukturen vorstellen und erklären sowie betrachten, wie effizient sie in bestimmten Situationen sind.
Allgemein sind Baumstrukturen wie der name schon besagt,hierarchische Daten Strukturen wo der Aufbau dem eines Baumes ähnelt. Sie besitzen eine wurzel oder auch root genannt und jeweils nodes (knoten) welche die children/ kinder von der root oder der jeweiligen parent Node sind. In diesen Nodes können Daten gespeichert werden, dies ist aber nicht zwingend. In manchen fällen ist gewünscht nichts zu speichern. In dieser Arbeit wird es um diese drei Baumstrukturen gehen:**Quadtree** ,**Octree** und **KD-Tree** . Diese werden für unterschiedliche Zwecke und Situationen verwenden. Zum einen um Punkte in einem 2-D raum, 3.D raum oder sogar mehr dimensionale räume darzustellen und zu speichern, als auch andere Objekte, beispielsweise Rechtecke oder Polygone. Diese Datenstrukturen können in vielen Berreichen eingesetzt werden und an das jeweilige Kriterium angepasst werden. Nehmen wir das Beispiel das in Deutschland um Freiberg in einem bestimmten Umkreis alle Dörfer mit weniger als 20.000 Einwohner aufgzeigt werden sollen.
Mittels einer Baumstruktur könnte dies ermöglicht werden und anhand einer query Suche einen vordefinerierten Raum untersuchen. Auf diese Funktion werde ich in meiner Arbeit noch etwas genauer im Kaptiel 2.1.5 eingehen.
Ich werde mich mit den Fragen beschäftigen, wodurch zeichnen sich diese Baumstrukturen von anderen aus, wie sind sie aufgebaut, welche Vorteile und Nachteile besitzen diese Bäume? Desweiteren werde ich sowohl mich mit den jeweiligen Komlpexitäten der Strukturen auseinandersetzen als auch Benchmarks verwenden um die Effizienz der Baumstrukturen zu vergleichen.

2 Baumstrukturen

2.1 Quadtree

Der Quadtree ist ähnlich aufgebaut wie fast jede andere Baumstruktur. Er besitzt zum einen einen parent node mit jeweiligen children Nodes. Dieser Knoten wird auch als Wurzel oder Root bezeichet. Was ihn unterscheidet zu z.b. einem Binären baum, ist die Anzahl der nachfolgenden Knoten. Wie im Namen bereits vermerkt, geht nach dem Verfahren, 4 children nodes oder gar keine. Infolgedessen besitzt der Quadtree immer 4 childrenNodes oder gar keine. Dies hat seine Vorteile aber auch seine Nachteile, worauf ich jedoch erst später darauf eingehen werde.
Es gibt verschiedene Arten von Quadtrees, z.b. im Point Quadtree können Punkte gespeichert werden, der RegionQuadtree , wo Flächen unterteilt werden. dies ist sehr hilfreich in dem Themenbereich Geologie, womit für bestimmte Orte bestimmte Erze oder Schichten unterschieden werden können. Aber auch bei Image Processing findet dieser einen nutzen. Dementsprechend hängt die Struktur des Quadtree s von den jeweiligen Daten ab, welche gespeichert werden sollen, womit er sehr anpassbar ist. Noch ein weiteres Beispiel wäre der Edge tree: dieser speichert Linien, Folglich können Bereiche bei einer Kurve so eingegrenzt werden, dass die Funktionswerte minimal sind.
Die Unterteilung erfolgt rekursiv und im besten Fall ist der Quadtree ausbalanciert.
Der Quadtree findet Verwendung bei der Räumlichen Indizierung (GIS programme), zum anderen können im 2-dimensionalen Raum Kollisionen erkannt werden. Dies könnten Punkte sein aber auch bestimmte Regionen. Zur Flächenindizierung aber auch bei der Fraktalen Bildanalyse.
Der Quadtree kann aber auch für Image processing benutzt werden, siehe 1.

Abbildung 1: Quadtree Bild Compression

2.1.1 Aufbau

Der Quadtree hat einen ähnlichen Aufbau wie die binären Bäume. Was in unterscheidet ist die Anzahl der nachfolgenden Knoten, was bereits in 2.1 erwähnt wurde. Was ihn weiter abgrenzt ist der Key value im Knoten, der im Quadtree gesplitet wird. Demzufolge besitzt der z.B. Point-Quadtree einen xValue und yValue.
Für diese Arbeit wurde ein Point Quadtree implementiert. Die anderen Ansätze haben einen ähnlichen Aufbau unterscheiden sich jedoch im Unterteilungskriterium als auch was gespeichert wird, wo neben dem punkt ein Polygon weitergegeben wird.
Zuerst muss eine Fläche definiert werden, worin wir die Daten, in meinem Code Beispiel Punkte, einfügen und speichern. Dies erfolgt durch die struct class `struct rectangle`.

```
1  struct Rectangle{
2      // point_t point;
3      randPoint_t point;
4      double width;
5      double height;
6  };
7  typedef struct Rectangle rectangle_t;
```

Wir benötigen einen Punkt, welcher sich in der Mitte des Quadranten befindet, dieser wird desweiteren zur Teilung genutzt. Zum anderen wird die Weite und Höhe definiert. Was im späteren Verlauf nicht vegessen werden darf, ist die beiden Grenzen immer zu halbieren, da der Zentrumspunkt der Ursprung ist und wie im Koordinatensysten positive als auch negative werte besitzt. Darauf werde ich noch etwas genauer in einem späteren Abschitt eingehen.
Da ich mich für den Point Quadtree entschieden habe, darf die Punkte Klasse nicht fehlen. Diese besteht aus einem XValue und einem yValue. Ich habe mich für den Datentyp double entschieden, aus dem Grund das größere Zahlen errreicht werden können und als auch bei der Teilung der Quadranten Kommazahlen entstehen, wrlches durch den Datentyp `int` einfach aufgerundet wird und somit Werte verfälscht werden. Um eine gewisse Anzahl an Punkte zu erreichen wurde die Random Function `rand()` genutzt.

```
1      struct RandomPointGenerated{
2      double xValue;
3      double yValue;
4  };
5  typedef struct RandomPointGenerated randPoint_t;
```

In der Quadtree Struktur wird eine **boundary** definiert, welche vom Datentyp `rectangle_t` ist. Somit besteht die Boundary aus einem Mittelpunkt, Breite und Höhe. Zusätzlich wird ein Array initialisiert des Datentyps `randPoint_t` mit der maximalen Kapazität, welche bei erreichen der Kapazität angibt wann der Quadtree geteilt wird. Die maximale Kapazität habe ich bereits am Anfang mit `#define capacity_of_Points 2` vordefineriert. Somit wird geteilt, wenn mehr als 2 Punkte im Quadranten sind. Um bestimmen zu können wie viele Punkte im Quadranten gerade vorhanden sind, wird eine variable

`n_points` initialisiert vom Datentyp `int`. Nun kommen wir zu einem der wichtigsten Punkte bei der Implementierung des Quadtree s, sprich den children Nodes oder auch inneren Knoten.
Anstatt einen Speicherplatz komplett zu reservieren, greifen wir auf Pointer zurück. Dies ermöglicht es, leichter Referencen zu generieren und ist zum anderen speicherfreundlich. Entsprechend des Aufbaues des Quadtree s, haben wir vier nodes. Um die position genauer bestimmen zu können werden diese nach den Himmelsrichtungen benannt, es ist aber auch möglich sie zu nummerieren. Das kann aber schnell zu Verwirrung führen. Daher habe ich mich für die andere Variante entschieden.
Da bei meiner Implementierung wird der Quadtree in gleich große Quadranten geteilt wird, und es entstehen die neuen Quadranten northWest, northEast, southWest, southEast.

```
    struct QuadTree{
    //the boundary of the node
    rectangle_t boundary;                                    // centerX, centerY
     and width and height
    randPoint_t points[capacity_of_Points];                  //max capacity of
    points in the Quadtree
    int n_points;                                            // number of points
     in quadtree
    //childrenNodes
    struct QuadTree *northWest;                             //creates the
    children of the parentQuadTree
    struct QuadTree *northEast;
    struct QuadTree *southWest;
    struct QuadTree *southEast;
};
typedef struct QuadTree QTree_t;
```

2.1.2 CreatingQuadtree Funktion

Um die Unterteilung in die jeweiligen Quadranten des Quadtree zu ermöglichen, müssen wir erst einmal Speicher für den parentQuadTree reservieren,was wir durch `malloc()` erreichen,welche den Speicher zuweist. Die Werte werden auf `NULL` initialisiert, damit es im späteren Verlauf zu keinem Fehler kommt. Wenn es zu einem Speicherfehler kommen sollte, wird ein Error geprinted und das Programm gestoppt. Wir haben am Anfang noch keine Punkte im Quadtree , deswegen wird auch das array `n_points = 0;` gesetzt. Somit werden alle werte mit dem startwert Null versehen und sobald ein Quadtree initialisiert wird, werden die jeweiligen Werte übernommen.

```

    QTree_t *createQuadtree(double centerX, double centerY, double width, double
    height){
    QTree_t* result = malloc(sizeof(QTree_t));
    if(result == NULL){
        printf("Malloc returns NULL. \n EXIT! \n");
        exit (1);
    }
```

```
    result->northWest = NULL;                //no childrennodes yet
    result->northEast = NULL;
    result->southWest = NULL;
    result->southEast = NULL;
    result->n_points = 0;
    result->boundary.height = height;
    result->boundary.width = width;
    result->boundary.point.xValue = centerX;
    result->boundary.point.yValue = centerY;

    return result;
}
```

2.1.3 insertPoint funktion

Es folgt die `insertThePoint` Funktion für das Einfügen der Punkte im Quadtree . Dazu brauchen wir zum einem einen parent QuadTree `QTree_t *quadTree` sowie Punkte `randPoint_t randPoint`.
Zunächst wird die Grenze oder boundary definiert und initialisiert mit den neuen x und y werten sowie der Höhe und Breite.

```
    double cx = quadTree->boundary.point.xValue;
    double cy = quadTree->boundary.point.yValue;
    double w = quadTree->boundary.width;
    double h = quadTree->boundary.height;
```

Da ich die Random Function `rand()` nutze und nur in einem bestimmten bereich die x und y Werte liegen sollen, müssen noch die Grenzen bestimmt werden. Meine Überlegung war, dass der minimale Wert die negative hälfte der breite ist und der maximale Wert die positive Hälfte der Breite. durch dies werden die Punkte immer in der Boundary liegen.

```
    double min_range = -w/2;
    double max_range = w/2;
```

Falls es doch zu einer Überschreitung der Boundary kommt, wird diese mit einer `if` Abfrage abgefangen. Das Programm würde sich in diesem Fall beenden.

```
    if(!(randPoint.xValue >= cx - w/2 && randPoint.xValue <= cx + w/2 && randPoint.
    yValue >= cy - h/2 && randPoint.yValue <= cy + h/2)){
        printf("Error! The Point is not in the Boundary of the Quadtree!\n");
        exit(1);
    }
```

Damit der Quadtree nicht mehr weiter unterteilt wird wenn die Breite oder die Höhe null sind und Punkte mit null Werten entstehen, wird dieser Fall ebenfalls mit einer `if` schleife abgefangen.

```
    if(quadTree->boundary.width == 0.0 || quadTree->boundary.height == 0.0){
        printf("The quadtree cant be divided anymore! \n");
        exit(1);
    }
```

Nun kommen wir zu der `for` schleife in welcher die Unterteilung der Node erfolgt als auch der Generierung der Punkte.
Da ich den `double` Datentyp verwende und keine `int` werte als Ausgabe bekommen möchte wenn ich den modulo Operator verwende, habe ich die `fmod()` Funktion verwendet. Hiermit wird der Rest von den zwei `double` Werten zurückgegeben utner berücksichtigung der Nachkommazahlen.

```
    randPoint.xValue = fmod(rand() , quadTree->boundary.width/2);
    randPoint.yValue = fmod(rand() , quadTree->boundary.height/2);
```

Da ich in der Grenze des Quadtree s bleiben möchte, habe ich die Werte, welche x annehmen kann auf TEST Mithilfe einer Weiteren `if` schleife werden die Punkte abgefangen, die gleich null sind:

```
    if(!(randPoint.xValue != 0 && randPoint.yValue != 0)){
        printf("Sorry cant be inserted!\n");
        exit(1);
```

Die Punkte werden anschließend in das Array gespeichert, wenn die maximale Kapazität nicht erreicht wurde.

```
    if(quadTree->n_points < capacity_of_Points){
        printf("direct adding to tree...\n");
        quadTree->points[quadTree->n_points++] = randPoint;
```

Sobald die Kapazität erreicht ist, wird der Quadrant unterteilt. Wenn er noch nicht unterteilt wurde, erfolgt dies zuerst. Danach wird bestimmt wo der Punkt sich genau befindet. Wenn der x wert und der y wert positive sind, wird der punkt in northEast hinzugefügt. Ist der x wert positiv, der y wert jedoch negativ wird der Punkt in southEast hinzugefügt. Sollte der x wert negativ sein, der y wert aber positiv, befinden wir uns in northWest. sind beide werte negativ, wird der Punkt in southwest hinzugefügt.

```
    if(quadTree->northEast == NULL){
        printf(" \n node will be subdivided..\n");
        subdivideNode(quadTree);
        }
            int positiveX = randPoint.xValue - cx >= 0;
            int positiveY = randPoint.yValue - cy >= 0;
        if(positiveX){
            if(positiveY){
                    printf("Point: %.2f/%.2f \n", randPoint.xValue, randPoint.yValue
);
                    printf(" Point in northEast\n");
                    insertThePoint(quadTree->northEast, randPoint);
            }else{
                    printf("Point: %.2f/%.2f \n", randPoint.xValue, randPoint.yValue
);
                    printf("Point in southEast\n");
                    insertThePoint(quadTree->southEast, randPoint);
            }
```

```
        }else{
            if(positiveY){
                    printf("Point: %.2f/%.2f \n", randPoint.xValue, randPoint.yValue
);
                    printf("Point in northWest\n");
                    insertThePoint(quadTree->northWest,randPoint);
            }else{
                    printf("Point: %.2f/%.2f \n", randPoint.xValue, randPoint.yValue
);
                    printf("Point in southWest\n");
                    insertThePoint(quadTree->southWest, randPoint);
```

2.1.4 subdivideNode Funktion

Für die Unterteilung des parent Quadtree s wurde eine eigene Funktion implementiert, welche die node in ihre children Nodes teilt. Wie der parent Node werden die Quadranten mit einem neuen mittelpunkt und neuen breiten und höhen definiert.

```
    // boundary: x - w/4 , y - h/4 , w/2 , h/2    northWest
    // boundary: x + w/4 , y - h/4 , w/2 , h/2    northEast
    // boundary: x - w/4 , y + h/4 , w/2 , h/2    southWest
    // boundary: x + w/4 , y + h/4 , w/2 , h/2    southEast
```

Für die Berechnung der neuen Boundary Werte der neuen Quadranten, müssen die Abstände zu dem Zentrumspunkt neu berechnet werden sowie die neue höhe und breite.

2.1.5 rangeQuery Funktion

Diese Funktion ist dafür da, dass in einem bestimmten bereich alle daten ausgegeben werden und nur in dem bereich gesucht wird. Damit wird zeit gespart. Bei dieser Methode muss beachtet werden, dass die jeweilige abfrage form in den jeweiligen boundarys ist. wenn dies nicht der fall ist, dann beginnt die suche nicht.
außerdem wird ein speicherort für die gefundenen Punkte benötigt, dies wird durch ein Array ermöglicht.

2.2 Octree

2 Wie der Quadtree , ist diese Baumstruktur ähnlich aufgebaut wie der binäre Baum. Beim octree gibt es jedoch nicht zwei oder vier children nodes, sondern acht oder gar keine. Dadurch findet diese Art von Struktur Anwendung in dem 3-dimensionalen Raum. Hier wird der Raum in jeweils 8 octanten unterteilt und dies solange bis keine Teilung mehr möglich ist oder auch allgemein nicht mehr nötig ist. Dies erfolgt nach dem conquer and divide schema. Bei dem empty non empty Octree wird

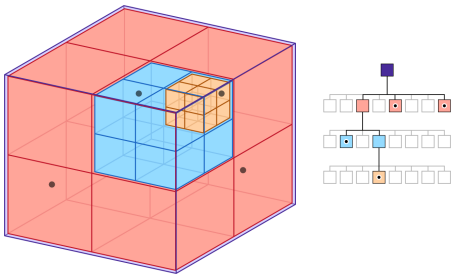


Abbildung 2: Octree

in den jeweiligen knoten leer oder nicht leer gespeichert. Wenn der Knoten mit leer gekezeichnet ist, wird verdeutlicht das die datenmenge, welche in diesem octant ist, nicht verarbeitenswert ist. sobald der knoten nicht leer ist, wird die datenmenge verarbeitet. neben dem empty non empty Octree gibt es auch die variante Min max. in den jeweiligen unterknoten werden jeweils die min und max werte hinterlegt somit ist eine effizientere suche gegeben. den durch die min max werte können die datenmengen schneller unterteilt werden und es nicht jeder knoten untersucht werden. der Octree findet verwendung in zum einen 3D grapfiken, wird zur Bildrepräsentation aber auch raumindizierung wie der Quadtree verwedet. Zum anderen können somit partikel gruppiert werden und zustandsschätzungen vorgenommen werden. Zusätzlich wird der Octree im marching cubes algorithm ...ref hier... somie in dem 3D game developüement zur kollisionserkennug genutzt.
Vorteile des Octree ist die effiziente suche wie bei vielen anderen Strukturen, was eines der wichtigsten punkte ist. Jedoch ist es schwer genaue bestimmungen zu ermöglichen da oftmals abgeschätzt wird wo genau sich z.b. die Partikel befinden. Zum anderen wie bei vielen Baumstrukturen tendiert der Octree zur nicht ausbalancierung, womit viele unnötige berechnungen erfolgen.

Was sie von den Quadtree strukturen unterscheidet sit die möglichkeit, in einem 3D raum daten darzustellen wie punkte aber auch dreiecke. Mit dem Octree kann ähnlcih wie der marching cubes algorithm eine iso oberfläche generiert werden. jedoch ist dies Implementierung schwieriger mit der Struktur zu verwirklichen als hingegen der algorithmus. Der marching cubes algorithm teilt den raum in gleich große dreiecke, wohin gegen der Octree ungleichmäßig unterteilt. Dies hat zwar den Nachteil, dass mehr speicher benötigt wird und die rechenzeit sich erhöht. Allerdings lassen sich damit formen detalierter darstellen. [2]

[2]https://stackoverflow.com/questions/15827385/what-is-the-difference-between-marching-cube-and-octree , aufgeru-

Ray tracing ist ein weiteres Beispiel, in welchem der Octree angewendet wird. Ray tracing wird verwendet um die sichtbarkeit von Objekten zu berechnen. Durch dies können 3D-Szenen dargestellt werden, wie in Video spielen. Es ist möglich mit dem Octree dies zu Implementieren jedoch gibt es eine adaption der Datenstruktur, dem Octree -R für effizientes ray tracing. Der Raum wird nichtmehr mit einem spatialen mittelwert in voxeln geteilt,sondern durch einen Punkt. Anhands des Octree können die benötigten tests für den ray tracing algorithm minimiert werden, indem berechnet wird wo die tests verlaufen und durch teilen des raumes wird minimiert. [3] Ich werde in meiner Arbeit nicht genauer auf diesen Algorithmus eingehen, allerdings kann ich diesen artikel empfehlen welcher sich damit genauer beschäftigt. [WSC+95]

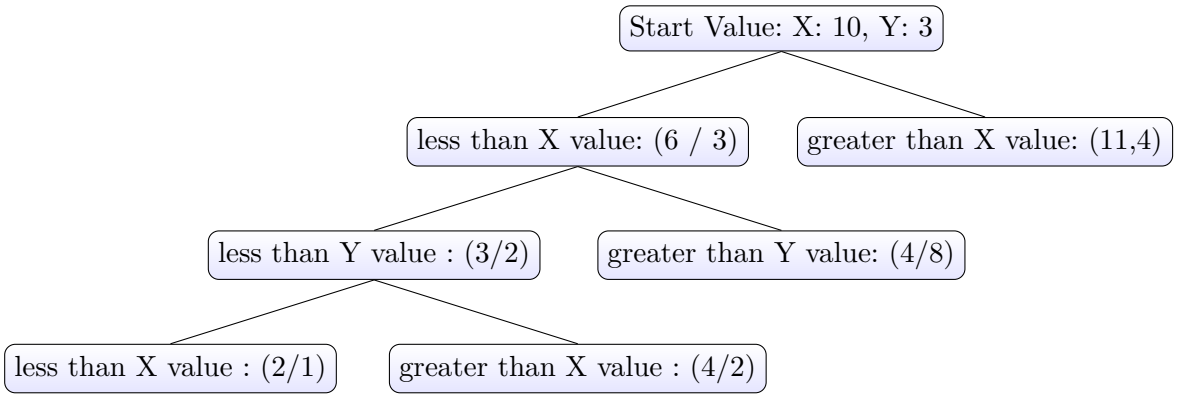
2.3 KD-Tree

Bei dieser Datenstruktur handelt es sich um einen ausbalancierten Suchbaum. In diesem werden Punkte aus dem $\mathbb{R}^k$ raum gespeichert. jeder Knoten stellt hier den k-dimensionalen Punkt dar. der KD-Tree tree wird für ray tracing und nearest neighbour search benutzt.Zudem können sogenannte Point Clouds generiert werden. Die teilung in subsektionen erfolgt nach bestimmung eines Kriteriums. Bei Punkte könnte einer gewählt werden, welche sich in der Mitte des quadtranten oder rechtsecks befindet. auf der linken seite würden dann alle werte dargestellt werden, welche kleiner als der Kriteriumspunkt sind, auf der rechten alle die größer sind. außerdem wird abwechselnd das kriterium geändert. somit wenn als erstes nach dem mittelwert der x achse gesucht wurde, wird als nächstes nach dem mittelwert der y achse gesucht. somit können daten effiziernter unterteilt werden.
es gibt verschiedene Arten dieser Struktur, zum einenen Homogene als auch inhomogene trees. Nebenen Punkten ist auch die speicherung von rechtecken möglich. Die speicherung der Punkte in nur den leafs/ blättern ist auch ausführbar. wie der Octree tree können uach nur min und max werte in den nodes gespeichert werden. Durch die bessere ANpassung und das abwechgseln der jeweiligen divide Kriterien wird die Struktur besser unterteilt und es folgt das die knoten dieser baum truktur besser ausbalancierbar sind als die des Quadtree s. Im Quadtree wird nur nach einem Kriterium unterteilt. Es gibt verschiedene Möglichkeiten, den KD-Tree aufzubauen. Da wir entlang der Achse gehen, haben wir dadurch viele wege zu starten. Bei einem 2-d tree würde die parent node erst die x achse als teilebene nutzen, anschließend die y achse und in diesem ablauf bis zur endbedingung durchlaufen. Die punkte werden unter berücksichtigung des berechnet medians eingefügt. durch diesen Ansatz kann der KD-Tree but ausbalanciert werden, da durch die berechnung des medians der Entsprechenden punkte es zu keinem großen unterschied in den bereichen kommt. Die unterteilung der leaves unterläuft meist nach links werden alle kleineren werte als der Median wert eingefügt, recht alle daten die größer sind.
pointlist: (10,3) (6,3) (11,4) (3,2) (4,8) (2,1), (4/2)

---

fen: 10.10.2021

[3] Kyu-Young Whang et al., Öctree-R: an adaptive octree for efficient ray tracing,ïn IEEE Transactions on Visualization and Computer Graphics, vol. 1, no. 4, pp. 343-349, Dec. 1995, doi: 10.1109/2945.485621.



in der grafik wird einer beispielehafter Aufbau eines KD-Tree dargestellt. Der erste Punkt der eingefügt wird ist (10/3). Sobald der zweite Punkt hinzugefügt wird, werden die X-werte verglichen. (6,3) ist kleiner und wird auf der linken Seite eingeordnet. (11,4) hat einen größeren X wert und wird somit auf der rechten Seite angeordnet. Die nächstes hinzugefügten Punkte (3,2) und (4,8) haben jeweils einen kleineren X-Wert. Somit werden sie auf der linken seite eingeordnet. Dort ist aber schon der Punkt (6/3). Diesmal wird der Y-Wert der beiden Punkte mit dem (6/3) Punkt verglichen. Damit ergibt sich, dass (3,2) wiederrum links eingeordnet wird da der Y-Wert kleiner ist und (4,8) rechts zugeordnet wird. (2,1) und (4/2) haben sowohl einen kleineren X-Wert gegenüber (10/3), als auch Y-Wert zu (6/3). Nun wird wieder der X-Wert der beiden mit (3/2) betrachtet und richtig eingeordnet. Nun haben wir schematisch einen beispielehaften Aufbau eines KD-Tree .
Ein anwengungsgebiet für den KD-Tree ist die nearest neighbour search. Dies ist ein algorithmus in welchem in der nähe eines bestimmten startwertes der nächstgelegene Punkt gsucht wird. Vorteil an diesem ist ,das die suche effizienter und schneller ist indem nicht jede node durlaufen werden muss. Durch die abwechslung der rahmenbedingungen ist der KD-Tree besser für die suche von daten geiegnet als de rQuadtree . außerdem ist er besser ausbalanciert, was eins der probleme bei dem Quadtree ist.

3 Variationsmöglichkeiten

3.1 Zusammenfassung

Wie bereits in den Kapiteln davor behandelt, ist es möglich die Strukturen unterschiedlich zu Implementieren. Der Quadtree kann zum einen in gleich große Formen geteilt werden, was bei dem Image processing vom vorteil ist. Zumal wir das Bild fragmentieren wollen und dies gleichmäßig als auch die jeweiligen children nodes immer die gleiche breite und höhe haben. anderseits können die jeweiligen grenzen angepasst werden, wenn in einem Raum Punkte an einem Ort sehr angereichert sind, würde es wenig sinn machen gleichmäßig zu unterteilen. Dies würde nur zu große leere quadtranten führen, wodurch zum einen Speicher für die leeren Blättern verschwendet wird als auch es zu einem unausbalancierten Baum resultiert.
wenn wir den Ansatz verwenden, wo wir die quadraten immer mit einer bestimmten Bedingung unterteilen, kann der speicher effizienter verwenden werden. Somit sollte immer darauf geachtet werden, ob es sich in dem fall anbieten würde, diese Unterteilung zu benutzen.

Das gleiche gilt bei dem Octree. Wenn dort im 3D raum punktmengen angehauft sind, wird sich das Voxel öfters teilen, allerdings führt das schnell zu ausbalancierten Bäumen.

3.2 Programmbibliotheken

Für den Quadtree gibt es die Libary `quadtree-lib`, mit den Funktionen: hinzufügen von Elementen, entfernen von Elementen und weiteren Funktionen wie den Baum filtern. Hier wurde die Programmiersprache **CoffeeScript** verwendet. [1] eine weitere Implementierung in der Programmiersprache **Python** ist der `Pyqtree`. Genutzt wird diese Art von Quadtree zur raum indezierung bei GIS systemen oder rendering. [2]

Für das umwandeln von Point cloud daten in die Baumstruktur Octree ist die Point Cloud Libary `pcl_octree` in **C++** nützlich. Mit dieser können auch effiziente nearest neighbor suchen genutzt werden. Für ray tracing und shadow casting kann die Bibliothek `pyoctree 0.2.10` genutzt werden. [3]

Auch für den KD-Tree ist eine PCL libary vorhanden. `pcl_kdtree` kann für die schnelle nearest neighbor suche verwendet werden. Diese Libary ist in der Programmiersprache **C++**. Des weiteren gibt es noch in **Python** zur erstellung eines KD-Tree die Bibliothek `scipy.spatial.KDTree`.
Dies sind nur ein paar Beispiel libaries von vielen

4 Benchmarks

Ich habe mich weitestgehen mit dem Quadtree beschäftigt, doch um die unterschiedlichen Leistungen darstellen zu können, habe ich beispiel Implementierungen für den Octree und KD-Tree tree genutzt.

[1] https://www.npmjs.com/package/quadtree-lib ,aufgerufen am 01.10.2021
[2] https://pypi.org/project/Pyqtree/, aufgerufen am 10.06.2021
[3] https://pypi.org/project/pyoctree/ , aufgerufen am 12.06.2021

4.1 Quadtree

4.2 Octree

4.3 KD-Tree

5 Komplexität der jeweiligen Baum-Daten Strukturen

In diesem Kapitel geht es um die Komplexität der jeweiligen Strukturen. Somit um den Ressourcenverbrauch bei der Verwendung der jeweiligen Algorithmen. Diese ist abhängig davon ,wie die die Datenstrukturen aufgebaut sind und wie viele Daten sie verarbeiten/ speichern müssen. Zudem kommt hinzu, wie gut die Implentierung umgesetzt wurde. Zwar wird immer vom besten Ergebnis ausgegangen, hingegen wird dieses Ergebnis nie komplett ereicht werden, da viele Faktoren dies verhindern. Zum einen ist es abhängig von der Hardware, ob ich hochlesitungsrechner benutzt wird oder nur ein laptop. Zum anderen kommt hinzu, dass die Implementierung vielleicht fehlerhaft ist. Diese Faktoren dürfen nicht unterschätzt werden.

5.1 Quadtree

Betrachten wir einmal die hierarchische Struktur, den Quadtree . Bei einer anzahl von n= 100 und n= 10000 beträgt die allgemeine Rechenzeit *log* n. In einem worst case Szenario würde die Rechenzeit gleich der Node anzahl stehen. Dies wäre der fall wenn zum einen der Baum nicht ausbalanciert ist, bedeutet wir hätten nur einenode mit einer großen tiefe, was zu eienr linked lsit führen würde. Allgemein gubt es eine formel die den Aufwand berechnet: $\mathrm{C}_n^d = \frac{a}{b}\ log\ \mathrm{n}$. [1]

Nun werfen wir einen Blick auf die Such funktion mit s Punkten aus D Koordinaten. $\mathrm{Q}_n^{(s,d)} = \theta^{(1-\frac{s}{d})}\mathrm{n}$

dies wäre die berechnung der Komplexität für den perfekten Baum. Jedoch wird die Baumstruktur nie perfekt sein. deswegen die neue Formel, in der die Abweichung mit einbezogen wird. $\mathrm{Q}_n^{(s,d)} = \Theta^{(1-\frac{s}{d}+\theta\frac{s}{d})}\mathrm{n}$

die korriktur funktion $\theta(x)$ im exponenten ist die Lösung der gleichung mit $\theta \in [0,1] : (\theta + 3 - x)^x(\theta + 2 - x)^{1-x} - 2 = 0$

Allgemein kann man sagen, dass die Kosten , unabhängig ob der Quadtree erfolgreich war oder nicht $\log n + O(n)$ ist siehe [FGPR93, S.475ff] .

das worst case dieses alogrithmuses wäre der fall, dass die anzahl aller Nodes gleich der ausführungszeit ist somit $O(n)$

5.2 Octree

Der Rechenaufwand funkgiert bei dem Octree genauso wie bei dem Quadtree . betrachten wir die find function haben wir einen Aufwand von $O(\log 2\ \mathrm{N})$.

ähnlich siht es be ider insert und der seartch funktion aus. Diese besitzen den gleichen aufwand $O(\log 2\ \mathrm{N})$. Das worst case szenario wäre hier auch $O(n)$.

WEnn wir die Raumkomplexität betrachten, und k definiert ist als die anzahl der punkte im raum, kommen wir auf die Formel: $O(k \log 2N)$

[1] Flajolet, Philippe, et al. Änalytic variations on quadtrees.Älgorithmica 10.6 (1993): 473-500.

5.3 KD-Tree

Der AUfwand allgemein ist $O(n \log n)$, wie auch bei den beiden anderen Datenstrukturen ist der worst case hier $O(n)$. einen Punkt in einem blaancierten KD-Tree hinzuzufügen braucht um die $O(\log n)$ zeit. einen punkt zu entfernen $O(\log n)$. dem baum zu durcgsuchen pro range point dauert um die $O(\log n)$, somit ungefäht $O(n \log n)$ im ganzen betrachtet.

6 Conclusion

reihenfolgenabhängigkeit

- unterteilungsart
- freie bereichen
- ausbalanciertheit
- test
- speicherorte

Literatur

[FGPR93] Philippe Flajolet, Gaston Gonnet, Claude Puech, and John Michael Robson. Analytic variations on quadtrees. *Algorithmica*, 10(6):473–500, 1993.

[WSC+95] Kyu-Young Whang, Ju-Won Song, Ji-Woong Chang, Ji-Yun Kim, Wan-Sup Cho, Chong-Mok Park, and Il-Yeol Song. Octree-r: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349, 1995.