Strukturen und Algorithmen

Materialien zur Dozentur an der school4games Berlin

Inhalt

Einführung

In diesem Kurs geht es um Themen, die dabei helfen, zu verstehen, was mit heutigen Computern leicht möglich ist und was der Technik größere Schwierigkeiten bereitet.

Dabei wird es vor allem darum gehen, wie große Datenmengen verwaltet werden können, denn es sind meist nicht irgendwelche Berechnungen, die bei Spielen zum **Problem** werden, sondern **die Zeit, die dafür benötigt wird, freien Speicher bereitzustellen und Daten zu durchsuchen**.

Es ist also **für jeden Spieleentwickler hilfreich**, zu wissen, welche **Performanceeigenschaften die heute verwendeten Datenstrukturen** haben, denn wenn das Game Design sich diese Eigenschaften zunutze macht, lässt sich wesentlich mehr erreichen als wenn dem Computer etwas abverlangt wird, wofür er nicht geeignet ist.

Grundsätzlich sollte man bei Überlegungen zur Performanceoptimierung jedoch zwei Grundregeln einhalten:

1. **Nicht vorzeitig optimieren!** Solange man keine Performance-Probleme hat, sollte man vor allem auf Lesbarkeit und Bugsicherheit achten.
2. Nicht theoretisch diskutieren wie sich Änderungen auf die Performance auswirken sollten sondern **ausprobieren und messen!** Bei der Performanceoptimierung wird man immer wieder auf Überraschungen treffen, da sich verschiedene Bauteile im Computer unterschiedlich verhalten und eine Zeiteinsparung an einer Stelle wirkungslos sein kann, da gleichzeitig an ganz anderer Stelle Zeit verschwendet wird.

Grundsätzliches zur Hardware

Wenn man von einem schnellen Rechner spricht, denkt man meist zuerst an die Rechenleistung der CPU. Diese hat sich in den letzten Jahrzehnten recht konstant weiterentwickelt, seit der Einführung von 3D-Grafikkarten ist die Entwicklung sogar schneller geworden.

Die Geschwindigkeitsentwicklung bei Speicher hinkt dem hinterher. So wird heutzutage das Lesen und Schreiben von Daten in den Arbeitsspeicher immer mehr zum Bottleneck. Daher ist es von großer Bedeutung, Daten in einer für den Speicherzugriff möglichst effizienten Ordnung abzulegen.

Dazu muss man wissen, dass der Speicherzugriff über sogenannte Caches passiert. Das bedeutet, dass die CPU nicht direkt auf den Arbeitsspeicher zugreifen kann, sondern stattdessen anfordern kann, dass Daten aus dem Arbeitsspeicher in den deutlich kleineren aber schnelleren Cache-Speicher kopiert werden. Dadurch wird verhindert, dass wenn ein Wert mehrfach in Folge gelesen werden soll, dieser immer wieder neu aus dem langsameren Arbeitsspeicher gelesen werden müsste.

Nachdem die Daten aus dem Arbeitsspeicher in den Cache kopiert wurden, gibt es für die Bauteile, die für diesen Kopiervorgang zuständig sind, nichts zu tun – daher fahren sie einfach damit fort, die folgenden Daten ebenfalls in den Cache zu kopieren („**Precaching**“), da man davon ausgehen kann, dass wenn ein Datensatz angefordert wurde, in der Regel auch der folgende benötigt wird. Wenn ich beispielsweise auf die HP einer Spielfigur zugreife, dann werde ich in der Regel auch noch andere Werte von ihr benötigen, daher ist es von Vorteil, wenn nicht jeder Wert einzeln sondern gleich der ganze Spieler kopiert wird.

Somit ist es von großem Vorteil, wenn zusammenhängende Daten hintereinander im Speicher liegen, damit das Precaching möglichst viele Daten erwischt, die auch tatsächlich benötigt werden.

Single-Value-Container

Containerklassen sind Klassen, deren Aufgabe es ist, andere Objekte zu speichern. Wann immer man mit einer unbekannten Anzahl von Objekten zu tun hat, wird man mit Containern arbeiten und nicht mit einzelnen Objekten arbeiten sondern mit allen Objekten, die im Container gespeichert sind.

Man unterscheidet zwischen Single-Value-Containern und assoziativen Containern. Single-Value-Container speichern eine Sammlung von Werten oder Objekten, während assoziative Container Paare aus einem Schlüssel und einem Objekt speichern, so wie ein Telefonbuch, in dem Paare aus einem Namen und den zugehörigen Daten gespeichert sind.

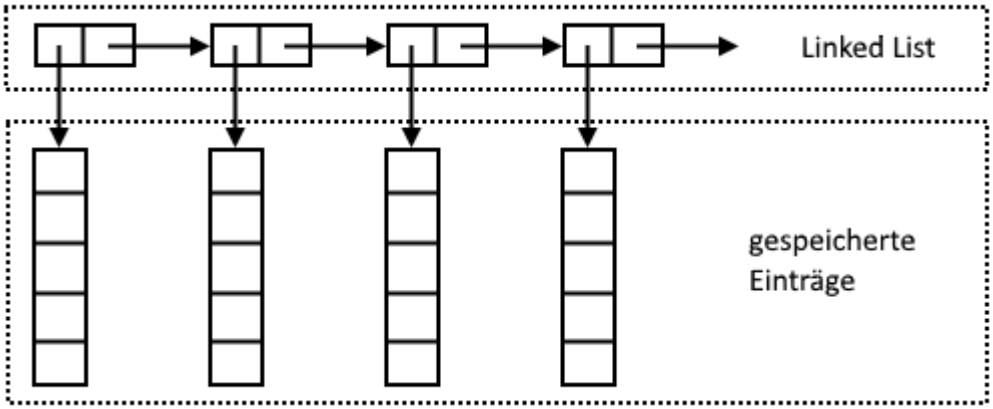
Außerdem lassen sich die verschiedenen Containerklassen auch noch auf zwei andere Arten gruppieren. Dafür sehen wir uns erst an, wie eine Containerklasse überhaupt eine unbekannte Anzahl an Objekten speichert.

Sequentieller und verknüpfter Speicher

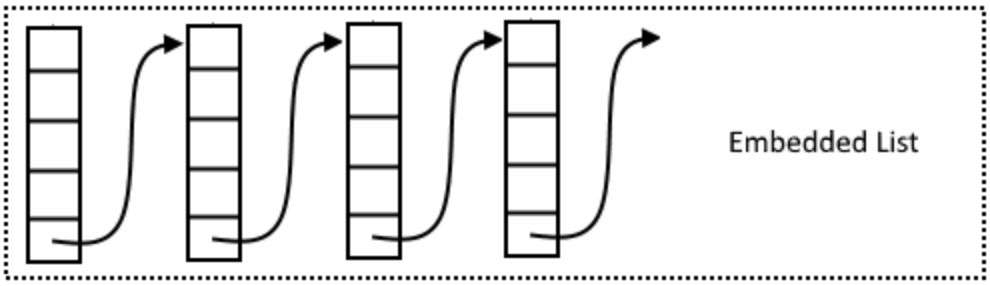
Um viele Objekte speichern zu können, braucht man logischerweise ein Vielfaches des Speicherplatzes, der für eines der Objekte nötig wäre. Diesen Speicher kann man entweder in einem Schritt vom Betriebssystem anfordern oder man fordert ihn in vielen kleinen Schritten an.
Der Vorteil, wenn man nur einmal Speicher anfordert, ist dass man einen zusammenhängenden Block Speicher bekommt. Diesen kann der Computer wesentlich schneller durchsuchen, denn um von einem Eintrag zum nächsten zu kommen, muss nur der direkt folgende Speicher gelesen werden. Eine aufwändige Suche nach der richtigen Adresse entfällt.
Da die einzelnen Objekte als Sequenz im Speicher liegen spricht man hier von **sequentiellem Speicher**. Bei dieser Methode muss nur einmal Speicher angefordert werden, dafür aber gleich eine große Menge, was im Moment deutlich unperformanter ist. Diese Art von Container wird als „**Array**“ bezeichnet.

Wenn man stattdessen immer wieder kleine Mengen an Speicher anfordert, muss man das zwar häufiger tun, dafür ist aber jeder Schritt nicht so teuer wie bei einem großen Speicherblock.
Bei dieser Methode liegen die Objekte nicht als Sequenz hintereinander sondern sind verteilt. Um dennoch von einem Objekt zum nächsten gehen zu können, wird in jeden Speicherblock ein Verweis auf den folgenden geschrieben. Man spricht deswegen auch von **verknüpftem Speicher** oder auch einer „**linked list**“.
Es ist also **aufwändiger, von einem Objekt zum nächsten zu gehen**, weil immer eine neue Adresse herausgesucht werden muss. Es ist dafür aber deutlich **leichter, in eine Reihe von Objekten ein neues einzufügen**, weil nur die Verknüpfungen angepasst werden müssen. Wo das neu eingefügte Objekt im Speicher liegt, ist bei dieser Methode irrelevant. Bei sequentiellem Speicher müssten alle Objekte, die in der Reihenfolge hinter dem neuen Objekt kommen, verschoben werden, um an der richtigen Stelle für ein neues Element Platz zu machen.

Linked List und Embedded List



Dieses Schaubild stellt dar, wie eine Linked List üblicherweise aufgebaut ist: Für jeden neuen Datensatz, der gespeichert werden soll, wird ein Listenelement erstellt, das zum einen auf das gespeicherte Objekt verweist, damit die Daten von der Liste aus erreichbar werden, und zum anderen auf das nächste Listenelement verweist, damit die Einträge zu einer Liste verknüpft werden.



Alternativ kann man auch die Listenelemente in die zu speichernden Datensätze integrieren, indem man **den Objekten einen Verweis auf das nächste Objekt hinzufügt**. Wenn man beispielsweise eine Embedded List von Player-Objekten erstellen wollte, müsste man der Player-Klasse einen Wert „next“ vom Typ „Player“ hinzufügen, damit jeder Spieler auf den nächsten Spieler zeigen kann.

Der Nachteil ist, dass jetzt der Listencode nicht mehr in einer eigenen Klasse steht. Die Playerklasse enthält also nicht mehr nur Code, der beschreibt, was ein Player kann, sondern auch etwas Code, der zur Player-Liste gehört.

Der Vorteil gegenüber der vorher skizzierten Implementation einer Linked List ist jedoch, dass jetzt um von einem Datensatz zum nächsten zu kommen nur noch einem Verweis gefolgt werden muss. Bei der ersten Variante muss man dem Verweis zum nächsten Listenelement folgen und dann von diesem erst einem Verweis zum Datensatz. Bei der Embedded List erhält man durch den Verweis auf das nächste Element direkt den gewünschten Datensatz. Das **spart ein wenig Rechenleistung und Speicherplatz**. In der Regel wird dieser Vorteil nicht nennenswert sein, er führt aber dazu, dass man immer wieder mit Embedded Lists zu tun haben wird, weil sie in Libraries oder von Kollegen verwendet werden können.

Stack, Queue und Deque

Eine andere Art, Container zu unterteilen, ist, für welche Verwendung sie optimiert sind. Wenn man beispielsweise Objekte in einem Speicherblock speichert, dann wird es kein Problem sein, am Ende

Objekte zu löschen und wieder welche dranzuhängen. Wenn man hingegen immer wieder am Ende etwas anhängt und die ersten Einträge löscht, dann würde sich ja der verwendete Speicher immer weiter verschieben bis er auf belegte Speicherbereiche stößt.

Daher unterscheidet man zwischen folgenden grundlegenden Containertypen:

- **Stack**
  - Ein Container, der sich wie ein **Stapel** dafür eignet, Objekte auf derselben Seite anzufügen und wieder wegzunehmen. Angelehnt an die Signalverarbeitung spricht man auch von „**LIFO**“-Buffer (last in, first out – was zuletzt reingetan wurde, wird zuerst wieder rausgeholt).
- **Queue**
  - Ein Container, der sich wie eine **Warteschlange** dazu eignet, Objekte auf einer Seite anzuhängen und auf der anderen Seite herauszuholen. Angelehnt an die Signalverarbeitung spricht man auch von „**FIFO**“-Buffer (first in, first out – was zuerst reingetan wurde, kommt auch als erstes wieder heraus).
- **Deque**
  - Deque, ausgesprochen „deck“, steht für „double ended queue“, also „Warteschlange mit zwei Enden“. Eine Deque ist ein Container, der auf beiden Seiten neue Elemente anfügen kann und aus dem man an beiden Enden Elemente entnehmen kann.

Andere Bezeichnungen

Es haben sich auch andere Bezeichnungen in den verschiedenen Sprachen eingebürgert wie z.B. **Array oder Vector**. Diese Bezeichnungen sind jedoch zwischen den **verschiedenen Sprachen so unterschiedlich verwendet**, dass man sich jedes Mal vergewissern sollte, ob man dabei an das gleiche denkt wie der andere.

Algorithmen

Wenn Objekte in vorgegebener Reihenfolge gespeichert werden sollen, ist vor allem das Speicherlayout entscheidend, wie im vorhergehenden Kapitel besprochen.
Sobald aber aus einer Menge von Objekten einzelne herausgesucht werden müssen, kommen Such- und Sortieralgorithmen ins Spiel.

Ansätze

Wenn man sich neue Algorithmen überlegen muss, dann gibt es zwei Ansätze, die einem helfen können, schneller eine Lösung zu finden.

Der eine ist **Rekursion**. Von Rekursion spricht man, wenn **eine Funktion sich selbst aufruft**. Das kann man sich so vorstellen wie die Regeln von „Ich packe meinen Koffer“: Einer fängt an und denkt sich einen Gegenstand aus. Danach muss man alles aufzählen, was vorher aufgezählt worden ist und sich eine neue Sache ausdenken und dranhängen.
Wie auch in diesem Beispiel besteht eine rekursive Funktion in der Regel aus zwei Bestandteilen: einem trivialen Fall (hier der erste Spieler) und einem Vorgang, der einen anderen Fall aufruft (hier die Beschreibung, was jeder folgende Spieler zu tun hat).

Mit Rekursion lassen sich oft Probleme leichter lösen und die Funktionen, die man dabei schreibt, sind oft leichter zu verstehen.

Ein effizienterer Ansatz, der sich jedoch nicht immer anwenden lässt, nennt sich **Divide-and-Conquer**. Bei diesem Ansatz versucht man, ein Problem ähnlich wie bei der Rekursion auf ein einfacheres Problem zurückzuführen, man versucht jedoch, **den Aufwand möglichst zu halbieren**. Ein gutes Beispiel für Divide-and-Conquer ist der Suchalgorithmus binary search (oder Binärsuche), der auf Seite 7 beschrieben wird.

Eine weitere Technik sollte an dieser Stelle erwähnt werden, auch wenn es sich dabei nicht wirklich um einen Algorithmus handelt. Es ist nämlich auch oft eine **gute Alternative, Werte** nicht zu berechnen sondern sie einfach **zu speichern**. Beispielsweise hat man vor der Zeit von Taschenrechnern Logarithmentafeln benutzt, um den Logarithmus einer Zahl zu erhalten. In diesen Tafeln stehen die Ergebnisse des Logarithmus für bestimmte Zahlen, so dass man nur das gewünschte Ergebnis heraussuchen muss. Ganz analog dazu kann man auch in einem Computerspiel Werte errechnen und speichern, was dafür sorgt, dass **keine Rechenzeit, dafür allerdings mehr Speicherplatz verbraucht wird**.

Such- und Sortieralgorithmen

Suchalgorithmen spielen im Zusammenhang mit Containern eine besondere Rolle, denn oftmals will man nicht alle Objekte eines Containers abhandeln sondern einzelne heraussuchen (den Spieler mit einem Namen, den NPC an einer Stelle, das Item im Inventar…).

Such- und Sortieralgorithmen arbeiten dabei Hand in Hand, denn **Daten die vorsortiert sind, sind leichter zu durchsuchen** als unsortierte Daten. Wenn allerdings Daten vorsortiert werden müssen, ist es teurer, neue Daten zu speichern, weil sie in die bestehende Ordnung eingegliedert werden müssen.

Verschiedene Sortieralgorithmen **braucht man jedoch als Spieleentwickler nicht unbedingt zu kennen**, denn anders als bei den sequentiellen und verknüpften Containern sind die Nachteile der schnelleren Algorithmen nicht so dramatisch als dass man von Fall zu Fall unterschiedliche Algorithmen benutzen müsste. Deswegen gibt es meist auch nicht die Wahl zwischen verschiedenen Sort-Funktionen, die sich in der Performance unterscheiden würden.

Zwei Eigenschaften gibt es jedoch, die Sortierverfahren haben können, die unter Umständen wichtig sind:

- **In place**
    - Ein Sortieralgorithmus ist dann „in place“, wenn er **keinen zusätzlichen Speicher benötigt** sondern die Daten in dem Container sortiert, in dem sie sich gerade befinden. Wenn große Datenmengen sortiert werden müssen, könnte der Speicher knapp werden, wenn andere Algorithmen verwendet werden.
- **Stable**
    - Ein Sortieralgorithmus ist dann „stable“, wenn er die **Elemente, die er nicht sortieren kann, in der Reihenfolge lässt**, in der er sie vorgefunden hat. Das ist dann wichtig, wenn man beispielsweise eine Spielerliste erst nach Spielernamen und dann nach Herkunftsland sortieren möchte. Wenn die Sortierung nach Land stable ist, dann bleibt die Namenssortierung innerhalb der Länder erhalten.

Verschiedene Algorithmen verhalten sich unterschiedlich, wenn die zu bearbeitende Datenmenge steigt. Um dazu Angaben machen zu können wurde die **O-Schreibweise** erfunden.

O(n) würde beispielsweise bedeuten, dass sich die Performance **proportional zu n**, der Anzahl der Elemente, ändert.
O(log(n)) hieße, dass sie sich **proportional zum Logarithmus der Anzahl** ändert.
In Ausnahmefällen kann auch **O(1)** erreicht werden, was bedeutet, dass ein Vorgang **immer gleich lang** dauert, egal wie viele Elemente bearbeitet werden müssen.

Bubble Sort

Die wahrscheinlich einfachste Art, eine Collection zu sortieren, nennt sich Bubble Sort. Dabei wird der erste mit dem zweiten Eintrag verglichen. Ist der erste Eintrag größer (d.h. wenn er weiter hinten einsortiert werden muss als der zweite), dann tauscht man anschließend die beiden Einträge aus. Danach werden der zweite und der dritte Eintrag auf diese Weise behandelt, dann der dritte und vierte und so weiter. Wenn man auf diese Weise einmal alle Einträge betrachtet hat, dann kann man sich sicher sein, dass der größte Eintrag an der letzten Stelle gelandet ist. Anschließend fängt man

wieder von vorne an, muss dabei allerdings nur alle Einträge bis zum vorletzten betrachten, da ja der letzte Eintrag bereits an der richtigen Stelle sein muss. Nach dem zweiten Durchlauf sind mindestens die letzten zwei Einträge an der richtigen Stelle, nach dem dritten die letzten drei usw.
Auf diese Weise fährt man fort, bis bei einem Durchlauf keine zwei Einträge ausgetauscht werden müssen.

Es ist deswegen hilfreich, Bubble Sort zu kennen, da es der Sortieralgorithmus ist, der sich am leichtesten selbst schreiben lässt. Außerdem gibt es einen Spezialfall, in dem Bubble Sort zu den schnellsten Algorithmen gehört, nämlich wenn die Collection bereits sortiert war. In dem Fall wird jedes Wertepaar nur genau einmal betrachtet . Man sieht also, dass es sich in vielen Fällen nicht eindeutig beantworten lässt, welcher Algorithmus schneller oder langsamer ist. Die mit der O-Schreibweise angegebene Komplexität beschreibt also oft nur die maximalen Kosten.

Linear Search

Unter dem Namen linear search oder Linearsuche verbirgt sich die einfachste Art, eine Collection zu durchsuchen: man betrachtet jeden Eintrag vom Anfang bis man am Ende angekommen ist oder den gesuchten Wert gefunden hat. In der Regel bedeutet eine Angabe wie „verwendet eine Linearsuche“ also, dass die Suche nicht optimiert ist.

Auch hier gibt es allerdings einen Spezialfall, in dem die Linearsuche zu den schnellsten Lösungen gehört, wenn nämlich nach dem ersten Eintrag gesucht wird. Wenn man also davon ausgeht, dass man in den meisten Fällen den ersten Wert in der Liste haben möchte, dann kann eine Linearsuche sogar performanter sein als kompliziertere Alternativen.

Außerdem benötigt eine Linearsuche im Gegensatz zur Binärsuche keine vorsortierte Liste sondern kann auf jeden Container angewendet werden.

Binary Search

Ein sehr einfach umzusetzender Suchalgorithmus ist die Binärsuche oder binary search, die nach dem **Divide-and-Conquer-Prinzip** arbeitet. **Voraussetzung ist, dass die Liste**, in der etwas gesucht werden soll, **sortiert ist**. Wenn man in dieser sortierten Liste einen Eintrag sucht, braucht man sich nur den Eintrag in der Mitte anzusehen um zu wissen, ob der gesuchte Eintrag links oder rechts davon zu finden ist. Anschließend sieht man sich die Mitte der linken oder rechten Hälfte an und halbiert somit wieder die zu durchsuchende Datenmenge, bis der gesuchte Eintrag gefunden wurde.

Assoziative Container

Assoziative Container sind Container, die neben Objekten auch noch Schlüsselwerte speichern, nach denen die Objekte sortiert werden. Man kann sie sich wie ein Telefonbuch oder ein Wörterbuch vorstellen, weswegen sie in manchen Sprachen auch als Dictionary bezeichnet werden.

Bei den assoziativen Containern gibt es zwei unterschiedliche Verfahren. Im einen Fall werden die Daten in einer **Baumstruktur** gespeichert, die so aufgebaut ist, dass sie den binary search-Algorithmus optimal unterstützt. Jeder Eintrag des Baums enthält einen mittleren Wert, der mit einem größeren und einem kleineren Wert verknüpft ist. Wenn man einen Wert sucht, muss man nur den Verknüpfungen folgen und dabei den größeren oder den kleineren Ast benutzen, je nachdem ob der gesuchte Wert größer oder kleiner ist als der Wert, an dem man sich befindet.

Das andere Verfahren sortiert **alle Elemente in Gruppen**. Beispielsweise könnte eine Liste von Ganzzahlen anhand der letzten Stelle in zehn Gruppen eingeteilt werden. Eine 13 käme in die dritte Gruppe, eine 217 in die siebte. Auf diese Weise kann man in einem Schritt ermitteln, wo ein bestimmtes Element zu finden ist. Dieses Verfahren ist also unter Umständen sehr viel schneller, es benötigt jedoch mehr Speicher, und vor allem braucht es eine geeignete Hashfunktion, die das Verteilen der Elemente in Gruppen übernimmt.

Eine solche **Hashfunktion** müsste idealerweise zwei Eigenschaften haben: sie müsste **für jedes Element einen anderen Hashwert** ermitteln, denn dann würde in jede Gruppe nur ein Element sortiert. Man würde also durch die Hashfunktion direkt auf das gesuchte Element geführt. Eine solche Hashfunktion nennt man eine „**perfekte Hashfunktion**“.
Außerdem sollte die Menge der möglichen Ergebnisse möglichst klein sein, denn für jedes mögliche Ergebnis muss ja ein Speicherplatz für Elemente bereitgestellt werden.

Das hat in der Regel zur Folge, dass Hashfunktionen keine für Außenstehende sinnvollen Werte ausgeben. Die Daten in einer Hashmap sind nicht alphabetisch oder nach Größe sortiert. Somit ist es bei diesem Verfahren nicht möglich, sich alle Werte in einem bestimmten Bereich geben zu lassen, sondern man kann nur gezielt nach einzelnen Einträgen suchen.

Das erste Verfahren wird Binary Tree genannt, während das zweite als Hashmap bezeichnet wird. Auf den Begriff Binary Tree wird man in Containernamen seltener stoßen, weil das eher als das normale angesehen wird, während eine Hashmap oft ausdrücklich als eine solche bezeichnet wird.
So gibt es in C++ beispielsweise Klassen für map und hashmap.

Anwendungsbeispiel: Verwaltung von Spielobjekten

Der wichtigste Anwendungsfall für Container ist die Verwaltung einer Spielwelt. In vielen Situationen muss in einem Computerspiel möglichst schnell ermittelt werden, welche Objekte sich in einem bestimmten Bereich befinden. Dazu gehört beispielsweise die wichtigste Funktion einer Grafikengine, das Culling.

Analog zum Binary Tree und zur Hashmap sollen hier zwei Container vorgestellt werden, die zur Verwaltung von Spielobjekten geeignet sind.

BSP-Trees

Beim **Binary-Space-Partitioning**-Verfahren wird die Spielwelt halbiert. Da wir es jedoch mit zwei oder drei Dimensionen zu tun haben, wird sie dadurch **in Quadranten oder Oktanten unterteit**. Jeder dieser Bereiche wird dann erneut unterteilt bis eine vorgegebene Menge von Unterteilungen erreicht worden ist.

Ähnlich wie in einem Binary Tree braucht man jetzt nur noch zu prüfen, ob ein gesuchtes Objekt links oder rechts und oben oder unten von der Mitte der Spielwelt liegt, um zu wissen, in welchem Quadranten es zu finden ist. Man benötigt aber auch hier mehrere Schritte um ein gewünschtes Objekt zu finden oder um ein neues einzusortieren, weswegen **BSP-Trees sich nicht so gut eignen, wenn Objekte oft verschoben werden sollen** und deswegen aus einer Zelle herausgeholt und neu einsortiert werden müssen.

Hashgrid

Während BSP-Trees das Prinzip von Binary Trees auf Positionen übertragen, arbeiten Hashgrids mit Hashmaps. Da man aus einer Hashmap allerdings keine Bereiche abfragen kann, muss man die zu speichernden Werte vorher sinnvoll gruppieren. Dazu rundet man die Positionen beispielsweise auf Vielfache von 10, so dass alle Objekte, deren Positionen sich nur in der letzten Stelle unterscheiden, an der gleichen Stelle abgespeichert werden.

Um für die Positionen (12.5, 23.8) und (19.0, 27.0) zu ermitteln, wo sie gespeichert werden müssen, würden sie beide auf (10,20) abgerundet und unter diesem Wert in einer Hashmap gespeichert. Bildlich kann man sich das wie ein Raster vorstellen, bei dem die Gitterlinien 10 Einheiten auseinander sind. Alles, was im gleichen Rasterquadrat landet, wird in einer Sammlung in der Hashmap gespeichert.

Somit kann man später sehr schnell alle Werte erhalten, die in einem Rasterfeld gespeichert sind, da man nur alles aus der Hashmap auslesen muss, was unter der gerundeten Position des Rasterfelds gespeichert ist.

Vor- und Nachteile von BSP-Trees und Hashgrids

In der Regel entspricht der Bereich, für den man alle Objekte erhalten möchte, nicht genau einem Rasterfeld eines Hashgrids. Wenn man beispielsweise alle Objekte auf dem Bildschirm erhalten möchte und dafür diese vorher in ein Hashgrid mit einer Zellengröße gespeichert hat, die der Bildschirmgröße entspricht, dann berührt der Bildschirm in der Regel vier Rasterfelder. Nur wenn die Kamera genau auf die Mitte eines Rasterfelds zeigt, braucht man nur die Daten eines Felds.

Das führt dazu, dass man sich gut überlegen muss, wie groß das Raster gewählt wird. Ist es zu klein, müssen mehr kleine Felder abgefragt werden, was unnötig Rechenleistung kostet.
Ist es zu groß, werden zu viele Objekte herausgegeben, die gar nicht im gewünschten Bereich liegen. Die Rastergröße muss also daran angepasst werden, wie groß der Bereich ist, der in der Regel abgefragt wird.

Wenn diese Größe variiert (zum Beispiel wenn der Spieler rein- und rauszoomt), dann lässt sich die Rasterweite schlechter optimieren. Dieses Problem gibt es bei BSP-Trees nicht, da jede weitere Ebene im Baum die Zellen viertelt und somit verkleinert.

Auf der anderen Seite benötigt man beim Einsortieren und beim Suchen von Objekten im Hashgrid nur einen Schritt: aus der gerundeten Position wird ein Hashwert ermittelt, und diese gibt unmittelbar an, wo die Daten liegen sollen.
Im BSP-Tree hingegen muss für ein Objekt mehrfach geschaut werden, ob es sich links, rechts, oben oder unten von der Mitte der aktuellen Zelle befindet.

Das hat zur Folge, dass es merklich teurer ist, ein Objekt in einem BSP-Tree zu verschieben, da es erst herausgesucht und entfernt werden muss um dann neu hinzugefügt zu werden. Für bewegte Objekte sind somit Hashgrids deutlich besser geeignet.

Gebräuchliche Containerklassen und ihr Anwendungsfall

Array

Als Array oder Vector bezeichnet man einen Container, der dafür gedacht ist, dass alle Einträge einmal behandelt werden. In hardwarenahen Sprachen sind Arrays daher mit sequentiellem Speicher implementiert, damit es möglichst performant ist, von einem Eintrag zum nächsten zu gelangen, der Dank Precaching bereits im CPU-Cache liegen sollte.

(Linked) List

In einer Linked List lässt sich leichter etwas einfügen oder entfernen. Daher wird sie oft verwendet, wenn die Anzahl von Einträgen in der Liste stark schwanken kann.

Set

Ein Set ist ein Container, der jeden Eintrag nur einmal enthalten kann und der dafür optimiert ist, festzustellen, ob ein Eintrag bereits enthalten ist oder nicht.

Map oder Dictionary

Eine Map (auch Dictionary genannt) ist dafür gedacht, Daten unter einem Schlüsselwert zu speichern und sie anhand des Schlüsselwerts wieder herauszusuchen. Wenn man beispielsweise regelmäßig auf den Spieler mit einem bestimmten Namen zugreifen muss, kann es sich lohnen, die Spieler unter ihrem Namen in ein Dictionary zu speichern.

Kombinationen von Containern

Wenn man für einen Anwendungsfall keinen optimalen Container findet, kann es sinnvoll sein, Daten mehrfach in verschiedenen Containern zu speichern, um **die Vorteile mehrerer Containertypen zu kombinieren**.

Ein typisches Beispiel wäre es, alle Spieler in einem Array zu speichern, damit man möglichst effizient in jedem Frame alle Spieler updaten kann. Gleichzeitig würde man alle Spieler in einer Map unter ihrem Namen speichern, damit man möglichst schnell einen Spieler mit einem gegebenen Namen finden kann.

Dadurch wird es natürlich teurer, neue Objekte hinzuzufügen und zu entfernen, da alle Container die neuen Daten hinzufügen bzw. die alten finden und entfernen müssen. Man muss also immer überlegen, was wie oft mit den Daten passiert. Werden die Container im Beispiel nur einmal erstellt und mit Daten gefüllt, dann kann sich der Geschwindigkeitsvorteil beim Suchen und beim Iterieren schnell lohnen.