Bachelorarbeit (Auszug Hauptteil)

Gestaltung und Entwicklung einer stark parallelisierten dreidimensionalen Simulation von orbitalen Himmelskörpern.

Autor:	Dennis Goßler
Matrikel-Nr.:	11140150
Adresse:	Oswald-Greb-Str. 7
	42859 Remscheid
	dennis.gossler@smail.th-koeln.de
Erstprüfer:	Prof. Dr. Christian Kohls
Zweitprüfer:	Alexander Dobrynin

Remscheid, 19.05.2022

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

[…]

2 Hauptteil

Der Hauptteil beschäftigt sich mit der Entwicklung und den daraus folgenden Problemstellungen der einzelnen Implementierungsphasen. […]

[…]

2.1 Parallels iterieren auf einer Liste

In dem Projekt wird oftmals über eine List iteriert und auf das ausgewählte Objekt eine Operation angewendet. Wenn eine Operation eine längere Zeitspanne benötigt, kann es von Vorteil sein, dass die Berechnungen auf den einzelnen Objekten parallel ausgeführt werden.

Dieser Abschnitt befasst sich mit der Entwicklung einer Funktion höherer Ordnung, die es ermöglicht, auf den gegebenen Objekten eine Operation anzuwenden.

2.1.1 Herangehensweise

Eine Aufgabenstellung auf ein Objekt kann sehr unterschiedliche Zeitspannen in Anspruch nehmen. Deswegen nimmt die *foreachParallel* Funktion als ersten Parameter eine Jobanzahl entgegen. Mit diesem Wert wird die Liste in verschiedene Abschnitte unterteilt. Beim Start der Funktion wird die Abschnittsgröße (*c*) und ein Restwert (*r*) bestimmt.

$$c = floor\left(\frac{list.size}{jobCount}\right)$$

$$r = list.size - (c * jobCount)$$

Eine *For-Schleife* zählt nun von 0 bis zur Jobanzahl und erstellt dementsprechend viele Jobs und fügt diese einer Job-Liste hinzu. Jeder Job bearbeitet somit einen gewissen unabhängigen Bereich.

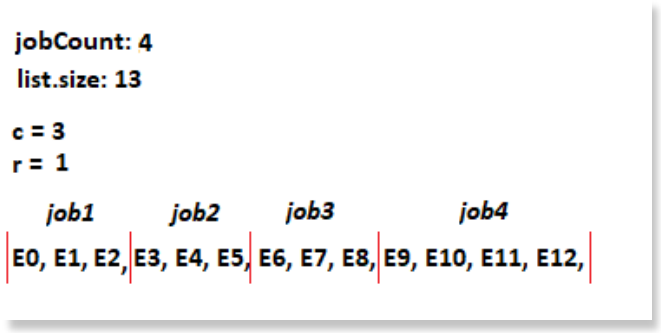


Abbildung 1: [Listenzerteilung zur Parallelisierung]

2.1.2 Funktionsaufbau

Die *foreachParallel* Funktion erweitert die Klasse *List* und nimmt ein *predicate* entgegen, welches die Operation auf das zugehörige Element vornimmt. Zusätzlich kann das *predicate* auch einen Integer aufnehmen, welches den aktuellen Index des Zugriffselements enthält. Um einen redundanten Code zu verhindern, wird die Funktion *foreachParallelIndexed* überladen und somit kann die Funktion auch ohne Zugriffsindex ausgeführt werden.

```
@OptIn(DelicateCoroutinesApi::class)
suspend fun <T> List<T>.foreachParallel(jobCount : Int, predicate : ((T)->Unit)) {
    this.foreachParallelIndexed(jobCount) { t: T, _: Int -> predicate(t) }
}

@OptIn(DelicateCoroutinesApi::class)
suspend fun <T> List<T>.foreachParallelIndexed(jobCount : Int, predicate : ((T, index: Int)->Unit)) {
    val items = this
    val jobs = mutableListOf<Job>()

    val chunkSize = items.size / jobCount
    val remains = items.size - (chunkSize * jobCount)

    for(jobIndex in 0 until jobCount){
        jobs.add(GlobalScope.launch(){
            for(index in jobIndex * chunkSize until (jobIndex + 1) * chunkSize + if(jobIndex != jobCount-1) 0 else remains){
                predicate(items[index], index)
            }
        })
    }
    jobs.joinAll()
}
```

Abbildung 2: [Erweiterungsfunktion foreachParallel]

2.2 Sweep-and-prune Algorithmus

Der Sweep-and-prune Algorithmus (*SAP*) [Q3 Sweep and prune S.I] ist ein Algorithmus zur effizienten Kollisionserkennung von Objekten im dreidimensionalen Raum. Ein Objekt (*Hitbox*) im *SAP* definiert sich durch seinen achsenorientierten Begrenzungskasten. Auf jeder Achse des dreidimensionalen Koordinatensystems besitzt jedes Objekt bei dieser Darstellungsform einen minimalen (*min*) und einem maximalen *(max*) Wert. Daher hat jedes Objekt in einer dreidimensionalen Umgebung sechs Werte. Die Gesamtheit aller Werte einer Achse werden in einer sortierten Liste gespeichert.

2.2.1 Datenstruktur

Um den Algorithmus darzustellen, werden gewisse Datenstrukturen benötigt. Hierbei orientiert man sich stark an den Strukturen aus dem Paper [Q3 Sweep and prune S.I].

```
class SAP{

    var idCounter = 0
        get() = field++

    val hitBoxes : MutableList<IHitBox>

    private val endPointsX = mutableListOf<EndPoint>()
    private val endPointsY = mutableListOf<EndPoint>()
    private val endPointsZ = mutableListOf<EndPoint>()

    //[...]

}
```

Abbildung 3: [Klassendefinition SAP ohne Funktionen]

Das *SAP* enthält alle Objekte und drei sortiere Listen aus deren Endpunkten. Die Endpunktlisten sind immer doppelt so lange, wie die Liste der *Hitboxen*.

```
data class EndPoint(
    val owner : IHitBox,
    var value : Float,
    val isMin : Boolean
)
```

Abbildung 4: [Klassendefinition EndPoint]

Ein Endpunkt enthält immer die Referenz auf seinen Besitzer, den jeweiligen Koordinatenwert und enthält die Festlegung der Frage, ob es sich um den *min* Wert seines Besitzers handelt.

```
interface IHitBox {

    val id : Int

    val minEndPoints : Array<EndPoint>
    val maxEndPoints : Array<EndPoint>

    var collided : AtomicBoolean
    var collisionChecked : AtomicBoolean
    var collidedWith : MutableList<IHitBox>

    fun addCollidedWith(hitBox : IHitBox)
    fun removeCollidedWith(hitBox : IHitBox)

    fun updateEndPoints()
}
```

Abbildung 5: [Schnittstelle IHitbox]

Jede *Hitbox* besitzt einen unikalen Identifizierer, der bei der Erstellung vom *Sap* zugewiesen wird. Außerdem enthält jede *Hitboxstruktur* die Kollisionspartner, die durch das *SAP* ermittelt werden.

2.2.2 Sequenzielle Kollisionserkennung

Um zu erkennen, ob ein Objekt mit den anderen *n* Boxen im *Sap* kollidiert, wird über die *endPointXListe* iteriert. Wenn es sich um einen minimalen Endpunkt handelt, wird eine zweite Schleife gestartet, die ab dem Endpunktindex beginnt und so lange läuft, bis der zugehörige maximale Endpunktwert gefunden werden konnte. Alle minimalen Endpunkte zwischen den Objektendpunkten können als Kollisionen auf der X-Achse betrachtet werden.

```
endPointsX.forEachIndexed { index, endpoint ->
    if (endpoint.isMin) {

        var i = index + 1
        while ( i < endPointsX.size){
            val endpointNext = endPointsX[i]

            if (endpointNext.isMin){
                val collideWith = endpointNext.owner
                collideWith.collided.set(true)
                collideWith.collidedWith.add(endpoint.owner)
                endpoint.owner.collided.set(true)
                endpoint.owner.collidedWith.add(collideWith)

            }else
                if(endpoint.owner.id == endpointNext.owner.id)
                    break

            i++
        }
    }
}
```

Abbildung 6: [SAP-Kollisionsüberprüfung der X-Achse]

Um zu überprüfen, ob die Objekte auch auf den anderen Achsen kollidieren, wird nun über die Liste der Objekte iteriert. Wenn ein Objekt ein oder mehrere Kollisionen aufweist, werden diese sukzessiv überprüft. Die Überprüfung vergleicht die minimalen und maximalen Werte der potenziell kollidierenden Objekte und entscheidet, ob eine Kollision auf der Y- und Z- Achse stattfindet.

```
hitBoxes.forEach { hitBox ->
    if(hitBox.collided.get()){
        hitBox.collidedWith.toList().forEach { collideHitBox ->

            if(!collideHitBox.collisionChecked.get() &&(
                (hitBox.maxEndPoints[1].value < collideHitBox.minEndPoints[1].value
                || hitBox.minEndPoints[1].value > collideHitBox.maxEndPoints[1].value) ||
                (hitBox.maxEndPoints[2].value < collideHitBox.minEndPoints[2].value
                || hitBox.minEndPoints[2].value > collideHitBox.maxEndPoints[2].value)
            ))
            {
                if(hitBox.collidedWith.size < 2)
                    hitBox.collided.set(false)

                if(collideHitBox.collidedWith.size < 2)
                    collideHitBox.collided.set(false)

                collideHitBox.collidedWith.remove(hitBox)
                hitBox.collidedWith.remove(collideHitBox)
            }
        }
        hitBox.collisionChecked.set(true)
    }
}
```

Abbildung 7: [SAP-Kollisionsüberprüfung der Y /Z-Achse]

2.2.3 Parallele Kollisionserkennung

Beim Ausführen des Algorithmus wird statt der normalen *foreachIndexedSchleife* die neu entwickelte *foreachParallelIndex* Erweiterungsfunktion der Listklasse verwendet [2.1 Parallels iterieren auf einer Liste S.1].

Da beim Durchlaufen der Liste auf andere Listenobjekte zugegriffen wird, muss gewährleistet sein, dass ein Job nicht mit einem anderen Job gleichzeitig versucht, auf eine Variable zuzugreifen. Dieses Problem wird gelöst, indem atomare Typen, wie zum Beispiel *AtomicBoolean,* verwendet werden. Außerdem werden Operationen, wie ein Kollisionspartner hinzuzufügen/ entfernen, über eine synchronisierte Funktion in der *Hitbox* Klasse ausgeführt.

Dieses System wird auch beim zweiten Teil der Kollisionserkennung auf der Y-, Z-Achse verwendet.

2.2.4 Veränderung der Objektpositionen

Wie im Abschnitt [2.2 Sweep-and-prune Algorithmus S.3] beschrieben, müssen alle Endpunktlisten vor der Ausführung der Kollisionserkennung stets sortiert sein. Wenn ein oder mehrere Objekte verschoben oder skaliert werden, müssen seine Endpunktpositionen in den Endpunktlisten neu einsortiert werden. Da in dieser Applikation davon ausgegangen werden kann, dass fast alle Objekte in einem Updatezyklus die Position verändern, ist es effizienter, die drei Listen komplett durchzusortieren.

In der *sortParallel* Funktion des SAP-Objektes, werden die drei Listen in drei verschiedenen Jobs parallel sortiert.

2.2.5 Sequenzielle Teststruktur

Um sicherzustellen, dass der SAP-Algorithmus sequenziell als auch parallel fehlerfrei funktioniert, sind verschiedene Tests entwickelt [A1 Sweep-and-prune Testfälle S.II]. Die einzelnen Tests benutzen statt den Hitboxobjekten eigene Testobjekte, welche auch die Schnittstelle *IHitbox* implementieren.

Die sequenzielle Kollisionserkennung wird getestet, indem drei zuvor bestimmte Anordnungen von Objekten auf Kollisionen hin überprüft werden. Diese Anordnungen wurden mit der Software *Blender*[1] erstellt. Die Anzahl der stattgefundenen Kollisionen entscheidet über den Erfolg oder Misserfolg eines Tests.

2.2.6 Parallele Teststruktur

Die sequenzielle als auch die parallele Teststruktur greifen auf ähnliche Testmethoden zurück. Bei der parallelen Teststruktur muss allerding streng darauf geachtet werden, dass die Variablen bei laufendem Algorithmus threadsicher gesetzt werden. Wenn eine Variable nicht threadsicher gesetzt wird, kann es passieren, dass gewisse Kollisionen nicht erfasst werden.

Um diese Problemstellung zu testen, werden in einem 500 x 500 x 500 großen Bereich 5000 zufällig platzierte Testobjekte erschaffen. Vorab werden diese mit dem

[1] *Blender* ist eine Software, um dreidimensionale Bühnenbilder/ Modelle zu modellieren und zu gestalten.

sequenziellen Algorithmus auf Kollisionen überprüft. Die sequenziellen Testergebnisse ermöglichen einen Vergleich Ergebnisse mit den parallelen Ausführungen. Der Algorithmus wird mit einer Jobanzahl von 1 bis 100 getestet.

[…]

3 Fazit

[…]

Quellenverzeichnis

Q1 Mechanik und Wärme

Demtröder, W. (2006). Mechanik und Wärme (4. Aufl., Bd. 1). Springer.

Q2 Mathematics of Satellite Motion

Henderson, T. (o. D.). *Mathematics of Satellite Motion*. The Physics Classroom. Abgerufen am 17. Mai 2022, von https://www.physicsclassroom.com/class/circles/Lesson-4/Mathematics-of-Satellite-Motion#:%7E:text=As%20seen%20in%20the%20equation,2.

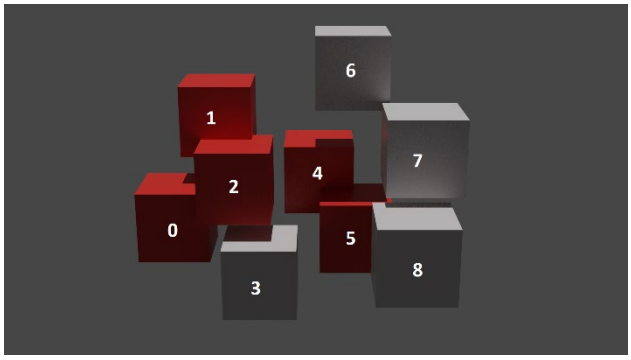
Q3 Sweep and prune

Terdiman, P. (2017, September). Sweep-and-prune (Version 0.2). http://www.codercorner.com/SAP.pdf

[…]

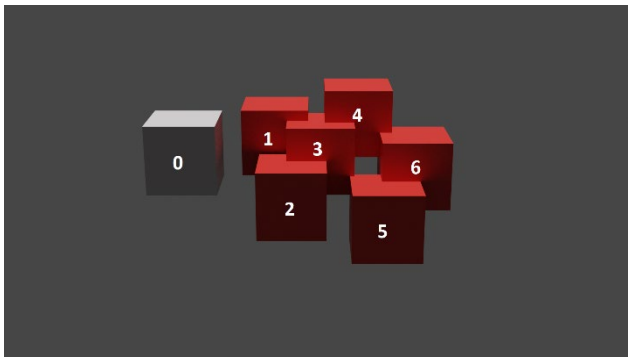
Anhang

A1 Sweep-and-prune Testfälle



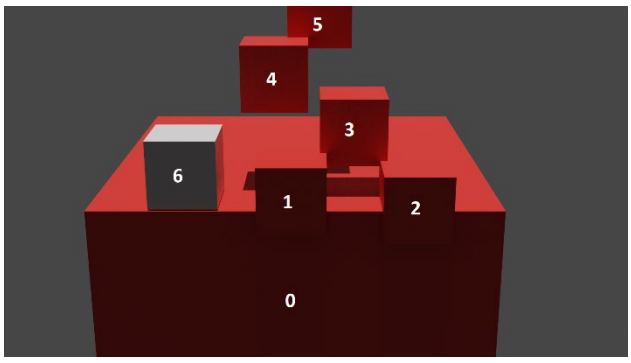
1. Testfall
Würfelanzahl: 9
Würfel kollidieren: 5

Kollisionen:
0: {2}
1: {2}
2: {0,1}
3: {}
4: {5}
5: {4}
6: {}
7: {}
8: {}



2. Testfall
Würfelanzahl: 7
Würfel kollidieren: 6

Kollisionen:
0: {}
1: {3}
2: {3,4}
3: {1,2,4,5}
4: {2,3,6}
5: {3,6}
6: {4,5}



3. Testfall
Würfelanzahl: 7
Würfel kollidieren: 6

Kollisionen:
0: {1,2}
1: {0,3}
2: {0,3}
3: {1,2}
4: {5}
5: {4}
6: {}

[…]