Paralleles Generieren von Binary Space Partitioning Bäumen

Oliver Skawronek
Berufsakademie Gera
Staatliche Studienakademie Thüringen

27. Dezember 2010

1 Einleitung

In der Computergrafik müssen die Polygone einer 3D-Szene vom Betrachter aus von hinten nach vorne gezeichnet werden, sodass vordere Polygone hintere verdecken können. Das ist allgemein als *Hidden-Surface-Problem* bekannt. 1980 stellten Fuchs, Kedem und Navior auf der Konferenz für Computergrafik SIGGRAPH ein Verfahren vor, das dieses Problem durch vorberechnen einer Baumstruktur – dem Binary Space Portitioning Tree (BSP-Baum) – in Echtzeit löst [1]. Während der Darstellung der Szene bestimmt die Position des Betrachters, wie der BSP-Baum durchlaufen wird und damit verbunden die Reihenfolge, in der die Polygone gezeichnet werden.
Heutzutage wird das Hidden-Surface-Problem pixelgenau auf Hardwareseite durch sogenanntes Z-Buffering gelöst, der BSP-Baum kann aber bei einer Reihe weiterer Probleme eingesetzt werden, darunter Kollisionserkennung und Strahlenberechnung beim Raytracing.
Im Rahmen der Vorlesung Parallele Rechentechnik an der Berufsakademie Gera versucht diese Projektarbeit, den Algorithmus zum Generieren eines solchen BSP-Baums so umzugestalten, dass die Berechnung auf mehrere Prozessoren verteilt werden kann.

2 Hintergrund

Es lässt sich beobachten, dass sich in vielen 3D-Anwendungen nur die Position des Betrachters, nicht aber ein Großteil der zu darstellenden Szene ändert.

In einem Flugsimulator bspw. bleibt die Landschaft auch beim Landeanflug aus unterschiedlichen Positionen stets unverändert. Mann kann daher versuchen, bereits vor der Darstellung, Informationen aus der statischen Szene zu gewinnen, um sie während der Darstellung zu nutzen. Auch beim BSP-Verfahren wird zuvor ein Binärbaum erstellt, auf den während der Darstellung zur schnellen Sortierung der Polygone zurückgegriffen wird.
Für das Verständnis des BSP-Verfahrens sind folgende zwei Erkenntnisse hilfreich:

1. Sei ein dreidimensionaler Raum an einer Ebene in zwei Halbräume S_k und $S_{\overline{k}}$ geteilt, wobei S_k den Halbraum auf der positiven Seite und $S_{\overline{k}}$ den Halbraum auf der negativen Seite der Teilungsebene bezeichnet. Befindet sich der Betrachter auf der positiven Seite, so können vom Betrachter aus die Polygone in $S_{\overline{k}}$ nicht die Polygone in S_k verdecken. Das gilt auch für den umgekehrten Fall, bei dem der Betrachter sich auf der negativen Seite befindet.

2. In einer konvexen Menge lässt sich per Definition keine Verbindungsstrecke zwischen zwei Punkten in der Menge finden, die nicht *vollständig* in der Menge liegt. In der Computergrafik grenzen üblich Polygone einen Raum ab. Sei eine konvexe Menge durch eine Menge von Polygonen abgegrenzt; die Polygone bilden also die konvexe Hülle. Befindet sich ein Betrachter innerhalb der konvexen Menge, dann lässt sich kein Strahl von ihm aus finden, der mehr als eines der Polygone schneidet. Ein Polygon kann vom Betrachter aus also nicht durch ein anderes Polygon der Hülle verdeckt werden. Stellt man sich gedanklich bspw. in einen Würfel mit seinen sechs Würfelseiten, die die konvexe Hülle bilden, kann keine Seite eine andere verdecken.

Das BSP-Verfahren nach Fuchs et al. kombiniert beide Erkenntnisse. Die beiden Halbräume werden wiederum rekursiv an einer Teilungsebene geteilt, bis schließlich konvexe Mengen übrig bleiben. Ein Binärbaum – der BSP-Baum – wird dabei so aufgebaut, dass die Teilungsebenen in den inneren Knoten und die konvexen Mengen in den Blättern abgespeichert werden. Das Verfahren baut auf den Ansatz von Sutherland et al. (1974) unter Überarbeitung der Arbeit von Schumaker et al. (1969) auf. Hier war es noch notwendig, dass die Teilungsebenen vom Gestalter der Szene gesetzt werden mussten. Bei dem Verfahren nach Fuchs et al. (1980) werden die Teilungsebenen aus den Polygonen der Szene automatisch gewählt.

3 Verfahren

Das BSP-Verfahren wird in zwei Phasen, nämlich der *Vorberechnungsphase* und der *Renderingphase*, unterteilt. Die erste Phase baut aus der statischen Szene den BSP-Baum auf und die zweite Phase bestimmt, in welcher Reihenfolge der BSP-Baum durchlaufen (traversiert) wird und damit in welcher Reihenfolge die Polygone gezeichnet werden.

3.1 Algorithmus zum Generieren von BSP-Bäumen

Sei die Szene gegeben als eine Menge von Polygonen $P = \{p_1, p_2, \ldots, p_n\}$. Aus der Menge wird ein Teilungspolygon p_k ausgewählt. Die Ebene, auf der p_k liegt, unterteilt den Rest der Polygone von P in zwei Halbräume S_k und $S_{\overline{k}}$, wobei S_k der Halbraum auf der positiven und $S_{\overline{k}}$ auf der negativen Seite von p_k liegt. Dabei kann es vorkommen, dass ein Polygon von der Teilungsebene geschnitten wird. Dann ist es erforderlich, dieses Polygon entlang der Ebene zu teilen. In Abbildung 1 ist die Aufteilung dargestellt. Polygone, die plan zur Teilungsebene sind, können entweder S_k oder $S_{\overline{k}}$ zugeordnet werden.

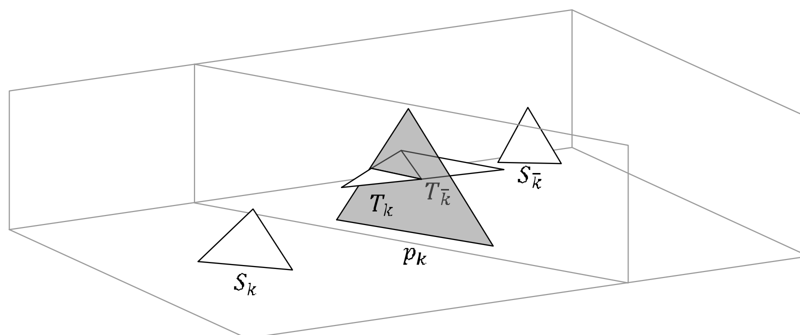


Abbildung 1: Aufteilung entlang p_k in die beiden Halbräume S_k und $S_{\overline{k}}$

Als nächstes werden die beiden Halbräume S_k und $S_{\overline{k}}$ rekursiv auf die gleiche Weise geteilt. Lässt sich aus P kein Polygon mehr auswählen, für das $S_k \neq \emptyset$ und $S_{\overline{k}} \neq \emptyset$ gilt, dann bildet P eine konvexe Hülle und die Rekursion kann abgebrochen werden.
Das rekursive Aufteilen in jeweils einen positiven und negativen Halbraum lässt sich als ein Binärbaum, der „Binary Space Partitioning Tree“ oder kurz BSP-Baum genannt wird, darstellen. Es gibt unterschiedliche Varianten der Darstellung. Diese Projektarbeit wählt die Darstellung als sogenannten „Leafy BSP Tree“. Dabei speichert die Wurzel die erste Teilungsebene. Der

linke Teilbaum stellt den positiven Halbraum und der rechte Teilbaum den negativen Halbraum dar. Die Wurzel der beiden Teilbäume speichern wiederum die Teilungsebenen nach einem weiteren Rekursionsschritt ab. In den Blättern sind die Polygone gespeichert, daher das Adjektiv „Leafy“[1]. Es wird nun der Algorithmus zum rekursiven Generieren eines Leafy-BSP-Baums vorgestellt.

Algorithmus MAKETREE $(P : Set) : Tree$

Eingabe: Eine statische 3D-Szene, beschrieben als eine Menge von Polygonen $P = \{p_1, p_2, \ldots, p_n\}$.

Ausgabe: Ein Leafy-BSP-Baum.

Methode:

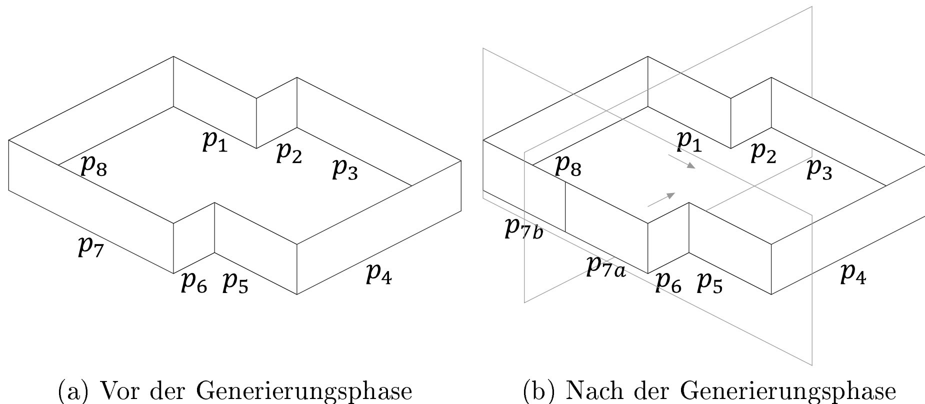
$p_k \leftarrow$ SELECTPOLYGON(P)
$T_k \leftarrow \emptyset$; $T_{\overline{k}} \leftarrow \emptyset$
$S_k \leftarrow \emptyset$; $S_{\overline{k}} \leftarrow \{p_k\}$
for all $p_i \in P \setminus \{p_k\}$ **do**
  {Teile p_i entlang der Ebene von p_k, sodass T_k die Teilpolygone auf der positiven und $T_{\overline{k}}$ die Teilpolygone auf der negativen Seite zugeordnet werden.}
  SPLITPOLYGON$(p_i, p_k, T_k, T_{\overline{k}})$
  $S_k \leftarrow S_k \cup T_k$
  $S_{\overline{k}} \leftarrow S_k \cup T_{\overline{k}}$
end for
if $S_{\overline{k}} = \emptyset$ **then**
  return MAKELEAF(S_k)
else if $S_k = \emptyset$ **then**
  return MAKELEAF$(S_{\overline{k}})$
else
  return COMBINETREE(MAKETREE$(S_k), p_k,$MAKETREE$(S_{\overline{k}})$)
end if

Dabei sollte SELECTPOLYGON ein geeignetes Teilungspolygon aussuchen. Ein Polygon ist dann als Teilungspolygon geeignet, wenn es P in zwei möglichst gleichgroße Mengen S_k und $S_{\overline{k}}$ aufteilt. Damit wird verhindert, dass der generierte BSP-Baum zur Liste entartet, die Höhe steigt und damit das

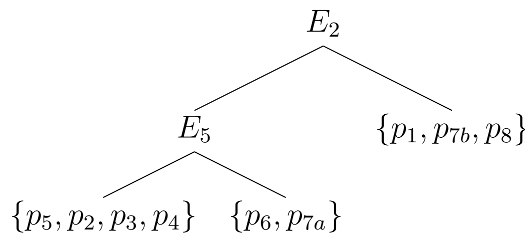
[1]Eine Alternative ist die Darstellung als sogenannter „Node BSP Tree“. Hier sind die Teilungspolygone in den Knoten des Baums gespeichert. Jedoch müssen hier beim Zeichnen viele einzelne Polygone an die Grafikkarte übertragen werden. Das erhöht den Kommunikationsaufwand mit der Grafikkarte. Beim Leafy BSP Tree sind die Polygone als zusammenhängende Liste gespeichert.

Durchlaufen des Baums zur Laufzeit länger dauert. Ein weiteres Kriterium ist die Anzahl an Polygonen gering zu halten, die sowohl auf der positiven als auch negativen Seite des Teilungspolygons liegen. Diese Polygone müssen entlang des Teilungspolygons in mehrere Polygone aufgeteilt werden, weshalb die Anzahl an zu zeichnenden Polygonen steigt.

Beispiel: In Abbildung 2a ist eine Beispielszene mit $P = \{p_1, p_2, \ldots, p_8\}$ dargestellt. Im ersten Schritt wurde p_2 als Teilungspolygon ausgewählt, wobei Polygon p_7 in p_{7a} und p_{7b} geteilt wird. Der positive Halbraum ist somit $S_k = \{p_2, p_3, p_4, p_5, p_6, p_{7a}\}$ und der negative Halbraum $S_{\overline{k}} = \{p_1, p_{7b}, p_8\}$.
Aus $S_{\overline{k}}$ lässt sich nun kein Polygon mehr wählen, an dessen Ebene $S_{\overline{k}}$ in zwei Halbräume geteilt werden kann, daher ist $S_{\overline{k}}$ konvex.
In S_k wird p_5 als Teilungspolygon ausgewählt. Es folgt die erneute Aufteilung, diesmal in $S_k = \{p_5, p_2, p_3, p_4\}$ und $S_{\overline{k}} = \{p_2, p_{7a}\}$.
In Abbildung 2b ist die Szene nach dem Generieren des BSP-Baums dargestellt. Abbildung 2c zeigt den generierten BSP-Baum. Man kann erkennen, dass bspw. die erste Teilungsebene E_2 in der Wurzel des Baums gespeichert wird.



(a) Vor der Generierungsphase (b) Nach der Generierungsphase



(c) Generierter BSP-Baum

Abbildung 2: Beispielszene

3.2 Algorithmus zum Rendern mittels BSP-Baums

Nach dem der BSP-Baum durch den vorigen Algorithmus generiert wurde, kann er genutzt werden, um die Szene aus einer beliebigen Position zu zeichnen (rendern). Das Ordnen der Polygone wird durch die Reihenfolge bestimmt, in der der Baum durchlaufen wird. Von der Wurzel ausgehend muss an jedem Knoten entschieden werden, ob zu erst der linke oder rechte Teilbaum durchlaufen wird. Das ist davon abhängig, ob der Betrachter sich auf der positiven (linker Teilbaum) oder negativen (rechter Teilbaum) Seite der Ebene des Knotens befindet. Mit Hilfe der Hesse-Normalenform kann die Seite bestimmt werden:

$$d' = \vec{r} \cdot \vec{n}_0 - d \quad \text{wobei}$$

- $\vec{r}$ der Ortsvektor der Betrachterposition,
- $\vec{n}_0$ der normierte Normalenvektor der Ebene,
- d der Abstand der Ebene zum Koordinatenursprung und
- d' der Abstand des Betrachters zur Ebene ist.

Ist d' positiv, befindet sich der Betrachter auf der positiven Seite und die Polygone auf der negativen Seite (rechter Teilbaum) müssen zuerst gezeichnet werden. Für den Fall, dass d' negativ ist, müssen die Polygone in umgekehrter Reihenfolge gezeichnet werden.
Es folgt nun die Beschreibung des Algorithmus zum Rendern mittels BSP-Baums.

Algorithmus RENDERNODE($n : Node, \vec{r}$)

Eingabe: Die Wurzel n eines Leafy-BSP-Baums und die Position des Betrachters als Ortsvektor $\vec{r}$.

Ausgabe: Von hinten nach vorne gezeichnete Polygone der Szene.

Methode:

```
if not IsLeaf(n) then
    E ← n.plane
    d' ← r · E.n0 − E.d
    if d' ≥ 0 then {positive Seite}
        {zuerst negativen dann positiven Halbraum rendern}
        RenderNode(n.rightBranch, r)
        RenderNode(n.leftBranch, r)
    else {negative Seite}
```

```
      {zuerst positiven dann negativen Halbraum rendern}
      RenderNode(n.leftBranch, r⃗)
      RenderNode(n.rightBranch, r⃗)
    end if
  else
    RenderPolygons(n, r⃗)
  end if
```

4 Umsetzung

Der oben vorgestellte Algorithmus zum Generieren von BSP-Bäumen ist ein *Divide and Conquer*-Algorithmus (engl. für „teile und herrsche“): Die Polygone sollen vom Betrachter aus von hinten nach vorne sortiert werden. Das Problem wird durch das rekursive Aufteilen in zwei Halbräume (divide) solange in Teilprobleme zerlegt, bis das Problem einfach genug ist, um es zu beherrschen (conquer), in dem Fall bis die Polygone konvexe Hüllen bilden. Sie können sich nicht mehr gegenseitig verdecken und müssen daher nicht sortiert werden.
Der Ansatz dieser Projektarbeit baut auf dem Artikel von Goetz [2] auf. Dabei ist die Idee, die Teilprobleme parallel zu lösen und die Teilergebnisse (Teilbäume) anschließend zusammenzufassen.
Zunächst muss überblickt werden, ob sich die Teilprobleme parallel lösen lassen. Das ist dann der Fall, wenn Teile gar nicht oder nur mit einem geringem Aufwand synchronisiert werden müssen. Auf den ersten Blick ist die Datenunabhängigkeit der Teilprobleme bei zwei Datenstrukturen nicht gegeben:

1. *Änderungen an der Ausgangszene*: Die Menge von Polygonen P wird in zwei disjunkte Mengen S_k und $S_{\overline{k}}$ aufgeteilt. Dabei kann es vorkommen, dass Polygone geteilt werden müssen und damit die Ausgangsszene geändert wird. Es darf daher nicht $P = S_k \cup S_{\overline{k}}$ angenommen werden. Da in den folgenden Rekursionsschritten auf P keinen Bezug mehr genommen wird (die geänderten Polygone werden nach unten weitergereicht), ist es nicht notwendig, P zu aktualisieren und damit auch nicht, den Zugriff auf P zu synchronisieren.

2. *Aufbau des gemeinsamen BSP-Baums*: Jeder Aufruf von MakeTree erzeugt einen Knoten. Muss das Problem nochmal in zwei Teilprobleme geteilt werden, ruft sich die Funktion selber auf und es werden die entstanden Teilbäume als Kinder an den Knoten angefügt. Der Knoten

wird anschließend zurückgegeben. Weil die Teilbäume unabhängig voneinander aufgebaut werden, muss der Aufbau des Gesamtbaums nicht synchronisiert werden.

Damit ist gezeigt, dass die Teilprobleme unabhängig voneinander lösbar sind. Im Folgenden werden Ansätze vorgestellt, mit denen die Teilbäume parallel generiert werden können.

4.1 Naiver Ansatz

Ein naiver Ansatz würde für jedes der Teilprobleme jeweils einen neuen Thread erzeugen. Jeder Thread müsste solange warten, bis seine Teilprobleme gelöst wurden. Anfangs würde ein Thread zwei neue Threads erzeugen die beim nächsten Schritt ihrerseits vier weitere Threads erzeugen usw. Zum einen ist das Erzeugen eines Threads mit viel Aufwand verbunden (Anlegen des Stacks und des Caches etc.), zum anderen entstünde ein großer Aufwand, um die vielen Threads zu verwalten.

4.2 Ansatz mittels Threadpool

In einem *Threadpool* wird eine begrenzte Anzahl an Threads (oft als „Poolthreads" bezeichnet) zur Ausführung von Aufgaben vorrätig gehalten. Die Aufgabe wird an den Threadpool gestellt, und falls ein Thread frei ist, wird ihm die Aufgabe zugewiesen. Steht kein Thread zur Verfügung, wird die Aufgabe in eine Warteschlange eingetragen (siehe Abbildung 3). Sobald ein Thread seine Aufgabe erledigt hat, wird ihm eine Aufgabe aus dieser Schlange zugwiesen oder er wartet im Pool auf die nächste Aufgabe.

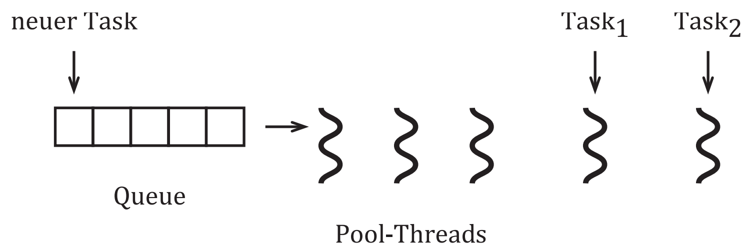


Abbildung 3: Threadpool

Der Einsatz eines Threadpools verhindert, dass für jede Aufgabe ein neuer Thread gestartet wird. Für das parallele Ausführen von Divide and Conquer-Algorithmen ist der Ansatz aber noch ungeeignet: Eine Aufgabe, die in

neue Teilaufgaben unterteilt werden muss, fordert für jede Teilaufgabe einen Thread beim Threadpool an. Danach muss sie warten, bis die Teilaufgaben erledigt sind. Erst, wenn eine Aufgabe erledigt wurde, kann der an die Aufgabe gebundene Thread dem Threadpool übergeben werden. Während die Aufgabe wartet, bis die Teilaufgaben erledigt sind, wird der Thread in Anspruch genommen. Werden alle Threads aus dem Threadpool genutzt und die Aufgaben konnten noch nicht so in Teilaufgaben geteilt werden, dass eine Aufgabe lösbar ist, kommt es zum sogenannten „Thread Starvation Deadlock“. Obwohl andere Aufgaben zu erledigen sind, befinden sich die Threads im Wartezustand und können nicht dem Threadpool übergeben werden.

Beispiel: In Abbildung 4 ist ein Beispiel für einen Thread Starvation Deadlock dargestellt. Es werden anfangs drei Threads bereitgestellt. Der Ablauf geschieht wie folgt:

1. Aufgabe 1 bekommt den ersten Thread zugewiesen. Danach teilt sie sich in die zwei neuen Teilaufgaben 2 und 3.
2. Auch Aufgabe 2 und 3 bekommen einen Thread zugewiesen. Nun befindet sich der Thread von Aufgabe 1 im Wartezustand, da die Teilaufgaben 2 und 3 noch nicht gelöst sind.
3. Aufgabe 2 muss wiederum in die zwei neuen Teilaufgaben 4 und 5 geteilt werden und da sie noch nicht gelöst sind, befindet sich der Thread von Aufgabe 2 im Wartezustand. Gleiches gilt für Aufgabe 3 und deren Teilaufgaben 6 und 7.
4. Aufgabe 4, 5, 6 und 7 werden in die Warteschlange eingetragen.

Alle drei Threads sind vergeben und befinden sich im Wartezustand, daher können die Aufgaben 4, 5, 6, und 7 nicht gelöst werden. Es tritt der Thread Starvation Deadlock ein, da sich alle Threads im Wartezustand befinden und ihn nicht wechseln können – sie „verhungern“ (engl. starvation).

4.3 Ansatz mittels Work Stealing

Ein anderer Ansatz mittels *Work Stealing* baut auf dem Prinzip des Threadpools auf, verhindert aber das Eintreten des Thread Starvation Deadlocks. Work Stealing ist ein Scheduling-Algorithmus. Auch hier steht nur eine begrenzte Anzahl an Threads, den sogenannten „Worker Threads“, zur Verfügung. Aufgaben können von der globalen Aufgabenschlange des Threadpools

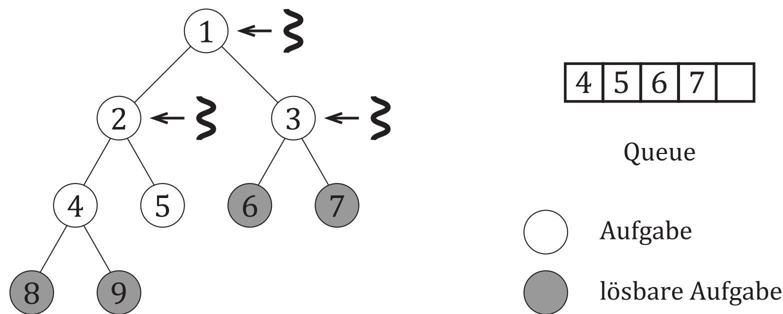


Abbildung 4: Thread Starvation Deadlock mit drei Threads

oder aus der lokalen Aufgabenschlange des Worker Threads abgearbeitet werden. Lässt sich keine Aufgabe finden, wird versucht, einem anderen Worker Thread eine Aufgabe zu entziehen (zu „stehlen“, engl. stealing). So wird versucht, soweit wie möglich alle Threads auszulasten. Umgesetzt wird das Scheduling wie folgt:

- Auch beim Work Stealing werden anfangs Aufgaben in die globale Aufgabenschlange des Threadpools eingetragen.
- Danach können Worker Threads Aufgaben aus der globalen Aufgabenschlange entnehmen und abarbeiten.
- Jeder Worker Thread besitzt eine lokale Schlange mit abzuarbeitenden Aufgaben.
- Die Schlange wird als Double Ended Queue, kurz Deque[2], organisiert. Elemente können bei einer Deque sowohl am Anfang als auch am Ende eingefügt und entfernt werden.
- Wird eine Aufgabe in Teilaufgaben geteilt (fork/spawn), werden die Teilaufgaben am Anfang der Deque eingetragen.
- Während ein Worker Thread auf seine Teilaufgaben wartet (join/sync), sucht er sich in folgender Reihenfolge solange eine andere Aufgabe:
  1. Enthält die lokale Deque eine Aufgabe, entnimmt er eine Aufgabe vom Anfang.
  2. Ist die lokale Deque leer, dann versucht er eine Aufgabe aus der globalen Aufgabenschlange des Threadpools abzuarbeiten.

[2] Ausgesprochen als „Deck“. Anstatt „Dequeue“ wird die Schreibweise „Deque“ bevorzugt, weil „dequeue“ bereits die Bedeutung besitzt, ein Element aus einer Schlange zu entfernen.

3. Kann auch keine Aufgabe aus der globalen Aufgabenschlange gefunden werden, versucht er von einem zufällig gewählten anderen Worker Thread eine Aufgabe vom Ende dessen Deque zu entziehen.

- Findet sich keine weitere Aufgabe, so gibt der Worker Thread seine Rechenzeit an die Laufzeitumgebung ab.

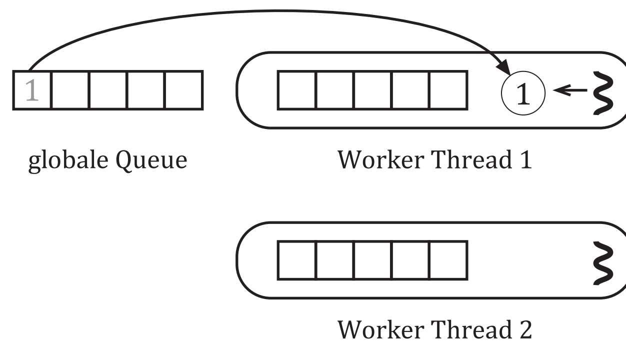
Das eigene Aufgaben am Anfang eingefügt und fremde Aufgaben vom Ende der Deque entzogen werden, verringert die Wahrscheinlichkeit des gleichzeitigen Zugriffs auf gleiche Elemente der Deque. Desweiteren bringt es den Vorteil bei Divide and Conquer-Algorithmen, dass schnelle, einfach zu lösende Aufgaben zu erst abgearbeitet werden. Später folgende, größere Aufgaben können auf die Ergebnisse der kleineren Aufgaben zurückgreifen.

Beispiel: Abbildung 5 soll das Work Stealing mit zwei Worker Threads veranschaulichen. Der Ablauf geschieht wie folgt:

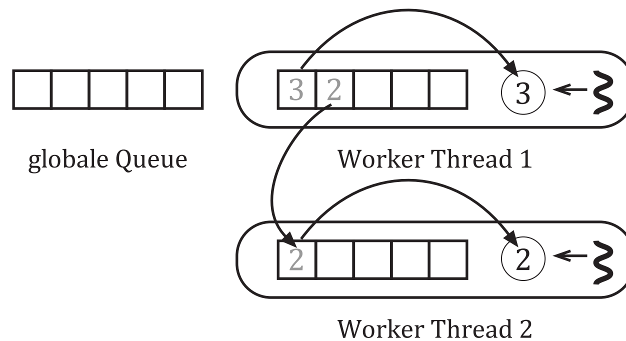
1. Aufgabe 1 wird in die globale Aufgabenschlange des Threadpools eingetragen. Worker Thread 1 und 2 sind unbeschäftigt.
2. Worker Thread 1 arbeitet Aufgabe 1 aus der globalen Aufgabenschlange ab (Abbildung 5a).
3. Aufgabe 1 teilt sich in die zwei Teilaufgaben 2 und 3. Zuerst wird Teilaufgabe 2 und dann Teilaufgabe 3 an den Anfang der lokalen Deque eingetragen.
4. Worker Thread 1 entnimmt Teilaufgabe 3 vom Anfang seiner lokalen Deque und Worker Thread 2 entzieht Teilaufgabe 2 vom Ende der Deque (Abbildung 5b).

Work Stealing eignet sich insbesondere dann, wenn die Aufgaben fein unterteilt werden können. Sind die Aufgaben zu grobgranular, können sie nicht optimal auf freie Worker Threads verteilt werden. Auf der anderen Seite sind zu fein unterteilte Aufgaben mit hohen Verwaltungskosten verbunden. Es sollte daher eine Grenze festgelegt werden, ab wann die Aufgabe sequenziell abzuarbeiten ist.

Insbesondere bei rechenintensiven Aufgaben sollte die Anzahl der bereitgestellten Worker Threads mit der Anzahl der verfügbaren Prozessorkerne übereinstimmen. Eine kleinere Anzahl würde nicht alle Prozessorkerne ausnutzen. Mit Erhöhung der Anzahl würde nicht die Auslastung der Prozessorkerne



(a) Worker Threads 1 arbeitet Aufgabe 1 ab



(b) Worker Thread 2 entzieht Teilaufgabe 2

Abbildung 5: Work Stealing mit zwei Worker Threads

steigen, dafür aber der Verwaltungsaufwand der Worker Threads.
Da die Anzahl an Worker Threads dynamisch an die Anzahl an Prozessorkernen angepasst werden kann, skaliert Work Stealing mit steigender Anzahl an Prozessorkernen.

4.4 Implementierung mittels Fork-Join-Framework

Der Ansatz mittels Work Stealing verhindert die Probleme der vorhergehenden Ansätze. Zum einen muss nicht für jede Teilaufgabe ein weiterer Thread verwaltet werden, zum anderen suchen sich die Worker Threads selbstständig neue Aufgaben, oder stehen dem Threadpool wieder zur Verfügung. Bei feingranularen Aufgaben lässt sich die Last gut auf die Worker Threads verteilen. Diese Arbeit verfolgt daher zum parallelen Generieren von BSP-Bäumen den letzten Ansatz.

Mit dem Fork-Join-Framework steht Java-Anwendungen das Work Stealing zur Verfügung. In [3] erläutert Lea den Entwurf, die Implementierung und

die Performance des Frameworks. Das Framework liegt nun als Java Specification Request 166 (JSR 166) vor, und soll mit kommenden Java 7 als Standard Einzug halten. Bis dahin kann das Framework von Entwicklern getestet werden.

Prinzipiell werden zwei Klassen zum Implementieren benötigt: *RecursiveAction* und *ForkJoinPool*. Die erste Klasse muss abgeleitet und deren Methode *compute* überschrieben werden, die zweite Klasse setzt das Work Stealing um. Die Berechnung wird gestartet, indem die Methode *invoke* von ForkJoinPool mit einer Instanz von RecursiveAction aufgerufen wird. Soll ein Problem in Teilprobleme aufgeteilt werden, muss aus *compute* heraus *invokeAll* mit den neuen Teilaufgaben aufgerufen werden.

Im Anhang A ist ein vollständiges Beispiel zum Fork-Join-Framework angegeben. Es sucht aus einem Integer-Array den größten Eintrag und gibt ihn auf der Konsole aus. Hier wurde nicht RecursiveAction sondern die generische Klasse *RecursiveTask<V>* abgeleitet. Der Unterschied besteht darin, dass bei RecursiveTask die Methode compute nicht den Rückgabetyp void sondern V (hier $V =$ Integer) besitzt.

5 Auswertung und Aussichten

Wie schon oben besprochen, müssen beim Generieren des BSP-Baums lediglich die Teilaufgaben synchronisiert werden. Der Zugriff auf andere Datenstrukturen benötigt keine Synchronisation. Auch ein Zugriff auf blockierende Ressourcen, wie Dateien oder Datenbanken erfolgt (während der Berechnung) nicht: Die Szene wird zuvor komplett in den Hauptspeicher geladen. Die Annahme ist daher, dass alle verfügbaren Prozessoren nahezu vollständig ausgelastet werden können.

Getestet wurde die Anwendung auf einem Laptop mit einem Intel Core 2 Duo (Zweikern-Prozessor) und einem 2 GByte großen Hauptspeicher. Dabei wurde die benötigte Zeit gemessen, um aus einer gegebenen 3D-Szene einen vollständigen BSP-Baum zu generieren. In der Messreihe wurde die Anzahl der Worker Threads verändert, das Laufzeitsystem und die Szene (q3dm1.obj) blieben gleich. Die Messwerte sind folgender Tabelle zu entnehmen:

	1 Worker Thread	2 Worker Threads	16 Worker Threads
Intel Core 2 Duo	58563 ms	39047 ms	39140 ms

Wie zu sehen ist, benötigt die parallele Variante mit zwei Worker Threads nur rund 66,7% der Zeit im Bezug auf die sequenzielle Variante. Es bestätigt sich ebenfalls, dass eine höhere Anzahl an Worker Threads sich gegenteilig auf die Berechnungszeit der Anwendung auswirkt, wenn auch die 16 Worker Threads noch keinen so großen Kommunikationsaufwand erzeugen, dass er wesentlich an der Berechnungszeit beteiligt ist.

Optimal wäre bei n verfügbaren Prozessoren eine Berechnungszeit von $1/n$ im Bezug auf die sequenzielle Ausführung. Offensichtlich ist das Optimum mit rund 66,7% noch nicht erreicht. Das ist damit zu erklären, dass in den ersten Rekursionsschritten noch nicht alle Worker Threads in Anspruch genommen werden können. Beispielsweise ist im ersten Schritt nur ein Worker Thread dafür verantwortlich, das erste Teilungspolygon zu finden und die Szene in zwei Halbräume aufzuteilen. Insbesondere das Suchen nach einem geeigneten Teilungspolygon bietet Ansatz zur Verbesserung:

for all $p_i \in P$ **do**
  for all $p_j \in P \setminus \{p_i\}$ **do**
    {Berechne, auf welcher Seite p_j bezüglich der Ebene von p_i liegt.}
  end for
end for

Für jedes Polygon p_i (innere Schleife) wird untersucht, ob es als Teilungspolygon am besten geeignet ist. Dabei muss ermittelt werden, wie viele Polygone sich von p_i aus auf der positiven, negativen oder sich auf beiden Seiten befinden (äußere Schleife) um einen Vergleichswert berechnen zu können. Setzt man $n = |P|$, so besitzt die Suche einen Zeitaufwand von $\Theta(n^2)$. Liese sich die Suche beschleunigen, könnte schneller zu den nächsten Rekursionsschritten übergegangen und damit auch schneller mehr Worker Threads ausgelastet werden. Dazu gibt es zwei Möglichkeiten:

- *Ändern des Suchalgorithmus*: Man könnte nur eine begrenzte Menge an zufällig ausgewählten Polygonen betrachten, und daraus das geeignetste Polygon auswählen. Alternativ könnte nur solange gesucht werden, bis p_i ein gewisses Kriterium erfüllt (im Vergleich zu, das geeignetste Polygon aus allen Polygonen zu suchen).
- *Parallelisieren der Suche*: Sind noch $n-1$ Worker Threads ungenutzt, kann die äußere Schleife in n Bereiche auf die n Worker Threads aufgeteilt werden.

Natürlich lassen sich auch beide Möglichkeiten miteinander kombinieren.

6 Zusammenfassung

In den ersten Abschnitten wurde das BSP-Verfahren mit den zwei Phasen zum Generieren des BSP-Baums und Rendern mittels BSP-Baums aufgezeigt. Danach wurden Ansätze besprochen, mit dem sich das einst sequenzielle Generieren parallelisieren lässt. Implementiert wurde der Algorithmus mittels Fork-Join-Framework, um den Ansatz mittels Work Stealing umzusetzen. Das Ziel, die Berechnungszeit mittels Parallelisierung zu verkürzen, bestätigte sich in der Messreihe, wenn auch nicht aussagekräftig auf nur einer Testumgebung. Dennoch gibt es Möglichkeiten, dass Verfahren noch weiter zu parallelisieren, um auch in den ersten Rekursionsschritten eine hohe Auslastung der verfügbaren Prozessoren zu gewährleisten.

Abschließend sind im Anhang B zwei Beispielszenen dargestellt. So zeigt bspw. Abbildung 6a die Ausgangsszene und Abbildung 6b die Szene nach dem ersten Rekursionsschritt. Das gewählte Teilungspolygon teilt die Szene in nahezu gleichgroße Halbräume (hier rot und blau gefärbt) auf. In Abbildung 7 ist die Szene, die zur Messung genutzt wurde, zu sehen.

Literatur

[1] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, New York, NY, USA, 1980.

[2] B. Goetz. Java theory and practice: Stick a fork in it. `http://www.ibm.com/developerworks/java/library/j-jtp11137.html`, 2007. [Online; accessed 25-November-2010].

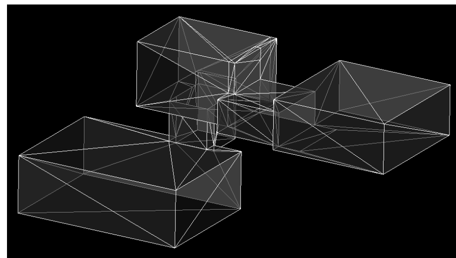
[3] D. Lea. A java fork/join framework. State University of New York, 2000. `http://gee.cs.oswego.edu/dl/papers/fj.pdf` [Online; accessed 25-November-2010].

A Beispiel zum Fork-Join-Framework

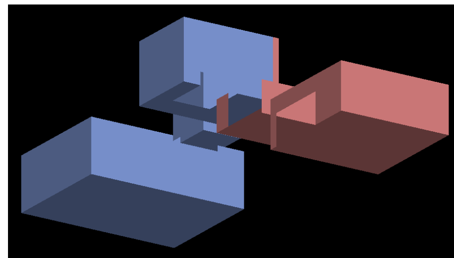
```java
package org.example.selectmax;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class SelectMaxTask extends RecursiveTask<Integer> {
  private static final int SEQUENTIAL_THRESHOLD = 4;

  private final int[] numbers;
  private final int start, end;

  public SelectMaxTask(int[] numbers) {
    this(numbers, 0, numbers.length - 1);
  }

  private SelectMaxTask(int[] numbers, int start, int end) {
    this.numbers = numbers;
    this.start = start;
    this.end = end;
  }

  @Override
  protected Integer compute() {
    int size = end - start + 1;
    if (size <= SEQUENTIAL_THRESHOLD) {
      // Problem kann sequenziell geloest werden
      return solveSequential();
    } else {
      int middle = (end - start) / 2 + start;
      SelectMaxTask left, right;
      // teile
      left = new SelectMaxTask(numbers, start, middle);
      right = new SelectMaxTask(numbers, middle + 1, end);
      // right.fork(); left.invoke(); right.join
      invokeAll(left, right);
      // herrsche
      return Math.max(left.getRawResult(), right.getRawResult());
    }
  }

  private int solveSequential() {
    int max = numbers[start];
    for (int i = start + 1; i <= end; i++) {
      if (numbers[i] > max)
        max = numbers[i];
    }
    return max;
```

```
  }

  public static void main(String[] args) {
    int[] numbers = { 10, 52, 32, 155, 96, 14, 55, 83, 12 };

    // Anzahl logischer Prozessoren abfragen
    int numThreads = Runtime.getRuntime().availableProcessors();
    ForkJoinPool pool = new ForkJoinPool(numThreads);
    SelectMaxTask task = new SelectMaxTask(numbers);
    pool.invoke(task);
    // Ausgabe max element: 155
    System.out.format("max element: %d\n ", task.getRawResult());
  }
}
```

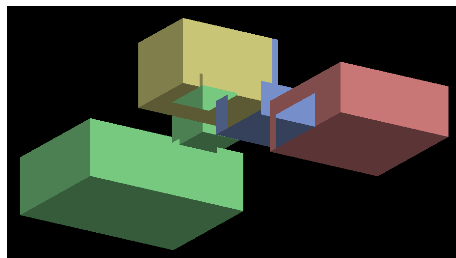
B Beispielszenen



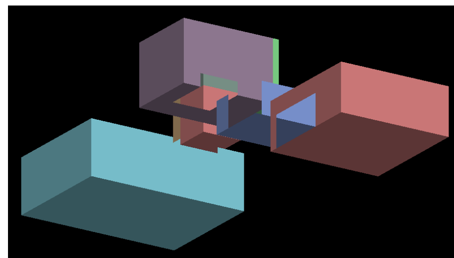
(a) Ausgangsszene



(b) Rekursionsschritt 1

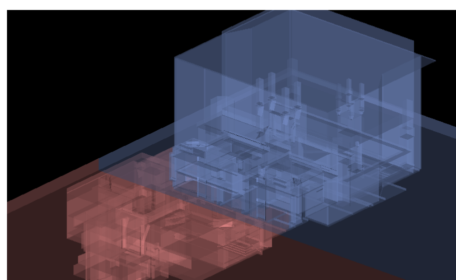


(c) Rekursionsschritt 2

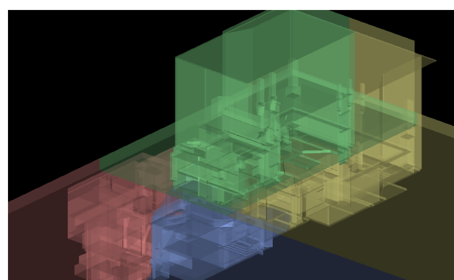


(d) Rekursionsschritt 3

Abbildung 6: Beispielszene level.obj



(a) Rekursionsschritt 1



(b) Rekursionsschritt 2

Abbildung 7: Beispielszene q3dm1.obj