3.3.1 Design des Renderthreads

Die Blocklib nutzt OpenGL als Grafik Schnittstelle. Da OpenGL, wie in Abschnitt 2.4.2 beschrieben, Multithreading nicht unterstützt, kann das Rendering selbst nicht nebenläufig durchgeführt werden, sondern muss auf einem Renderthread ausgeführt werden.

Der Renderthread kann einerseits als Teil des Jobsystems designt werden, andererseits gibt es die Möglichkeit den Renderthread von diesem zu trennen. Ist der Renderthread Teil des Jobsystems, kann dieser voll zur Bearbeitung von Jobs mitgenutzt werden. Da die Anzahl der Threads des Jobsystems üblicherweise der Anzahl der Hardwarethreads entsprechen soll, lässt sich so automatisch immer die Leistung aller Prozessorkerne nutzen. Trennt man den Thread dagegen ab, entsteht einerseits die Problematik zu entscheiden, wann welche Anzahl von Threads genutzt wird, um möglichst alle Kerne zu nutzen, aber gleichzeitig zu verhindern, dass sich die Threads in der Ausführung gegenseitig behindern. Andererseits gestaltet sich die Implementierung eines getrennten Renderthreads im Vergleich zur Jobsystemintegration als deutlich einfacher und intuitiver [Tat14], da beispielsweise nicht ermittelt werden muss, wann der Renderthread zur Ausführung von Jobs genutzt werden kann.

Über eine Analyse der Methodenlaufzeiten in der Blocklib ergibt sich, dass das Rendering knapp die Hälfte der Zeit eines Frames benötigt. Der prinzipiell mögliche Performancegewinn durch die Integration in das Jobsystem ist damit vernachlässigbar, da das Rendering selbst meist die gesamte Rechenleistung des Kerns beansprucht. Da das Rendering im sequentialisierten Fall circa 50 % der Zeit benötigt, entspricht das, sobald Simulation und Rendering in zwei Threads nebenläufig ausgeführt werden, annähernd 100 %. Wird die Simulation selbst vermehrt nebenläufig durchgeführt, verstärkt sich dieser Effekt, da sich damit die von der Simulation benötigte Zeit weiter verringert. Daher bietet es sich an, den Renderthread zu separieren. Vor diesem Hintergrund lässt sich auch die Anzahl der Threads des Jobsystems bestimmen, obwohl für das Rendering ein SoT Design genutzt wird. Die Anzahl der Jobthreads wird im Vergleich zu der Anzahl der Hardwarethreads um eins verringert. Aufgrund der vollen Auslastung des Renderthreads führt das nicht zu Hardwarethreads, die keine Arbeit entgegen nehmen können.

Da der Renderthread nebenläufig auf Daten des Spiels zugreift, um die sichtbaren Elemente zu zeichnen, muss sichergestellt werden, dass diese Daten keinen Wettkampfbedingungen unterliegen. In der Spielentwicklung ist es üblich einen *Spielzustand* zu definieren, der während der Simulation in jedem Frame angepasst wird. Der Teil des Zustands auf den der Renderthread nutzt muss also während dieses Zugriffs konstant sein. Durch die Simulation wird dieser allerdings verändert. Eine Möglichkeit, dieses Problem zu beheben, besteht darin einen *Double Buffer* [Nys15, S. 143] zu nutzen, um den gesamten Spielzustand zwischenzuspeichern [Tat14]. Die Blocklib ist aber nicht mit diesem Hintergedanken entwickelt worden, weswegen es keine einfache Möglichkeit gibt, den gesamten Spielzustand zwischenzuspeichern. Die Objekte, die den Spielzustand darstellen, sind über die Blocklib hinweg verteilt. Um dennoch einen Renderthread nutzen zu können, müssen die Objekte identifiziert werden, die für das Rendering benötigt werden. Für diese Objekte muss

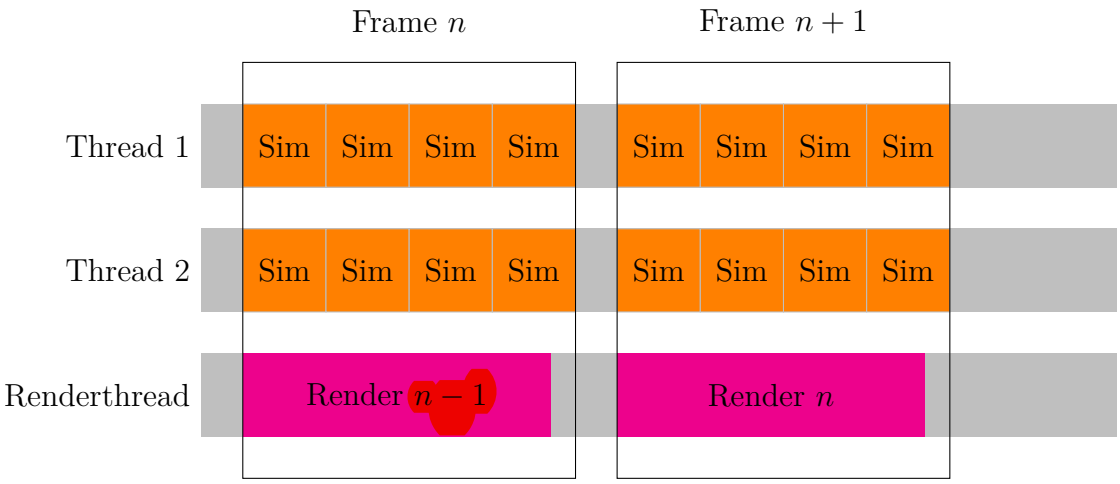

Abbildung 12: Darstellung des gewünschten Designs der Multithreading Architektur der Blocklib. Es existiert ein gesonderter Renderthread, der einen großen Teil der Rechenzeit während eines Frames nutzt (magenta). Die Simulation ist in Jobs (orange) aufgeteilt, die auf beliebig viele Threads aufgeteilt werden können. „Sim“ kennzeichnet einen Simulationsjob.

dann ein geeigneter Double Buffer erzeugt werden, sodass die Daten aus Renderthread-Sicht konstant sind.

Somit ist eine nebenläufige Architektur, wie sie in Abbildung 12 dargestellt wird, erstrebenswert. Es gibt einen Renderthread, der die von der Simulation im vorherigen Frame berechneten Objekte zeichnet. Alle anderen verfügbaren Hardwarethreads können in dem Jobsystem für die Simulation genutzt werden.

3.3.2 Design des Jobsystems

Aufgrund des Umfangs der Blocklib und der Unübersichtlichkeit an einigen Stellen kann das Jobsystem nicht, wie in Abbildung 12 gezeigt, umfassend integriert werden. Um die Anforderungen von Kapitel 3.2 zu erfüllen, wird daher ein Jobsystem implementiert, das nur an ausgewählten Stellen mit der Blocklib konsolidiert wird und ansonsten für die zukünftige Nutzung bereitsteht.

Da das Jobsystem keine vollständige Integration in die Blocklib erfährt, verändert sich die Architektur konzeptuell leicht. Diese Änderung ist in Abbildung 13 zu sehen. Anstatt die gesamte Simulation in viele kleine Jobs zu zerlegen, bleibt eine sequentialisierte Simulation bestehen, die dann aber die Möglichkeit besitzt, weitere nebenläufige Jobs zu starten. Mit dieser Architektur ist es möglich, die Blocklib inkrementell zu der in Abbildung 12 gezeigten Architektur umzuwandeln, indem in zukünftigen Arbeiten immer mehr pseudo-sequentialisierte Anweisungen der Simulation als nebenläufige Jobs definiert werden, bis sie vollständig aus Jobs besteht.

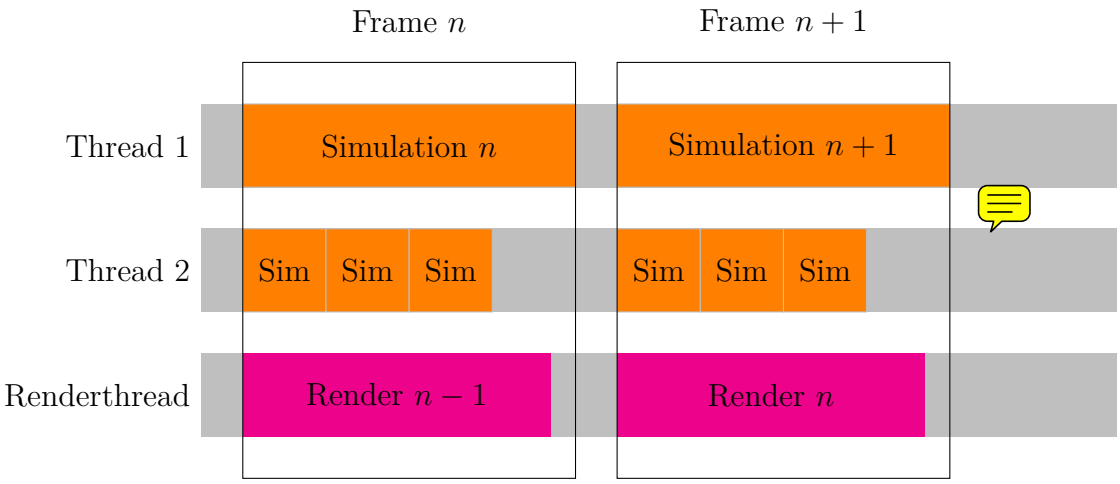

Abbildung 13: Design der tatsächlichen Threadingarchitektur der Blocklib. Statt einer Simulation, die vollständig aus Jobs besteht, gibt es einen langen Simulations-Job, der aber weitere Jobs starten kann. „Sim“ kennzeichnet einen Simulationsjob.

Java bietet mit dem Interface `ExecutorService` bereits eine gute und bekannte Schnittstellendefinition, die für ein Jobsystem genutzt werden kann. Daher baut das Design des Jobsystems der Blocklib auf diesem Interface auf.

In Abbildung 14 ist die Struktur des Designs für das Jobsystem der Blocklib dargestellt. Es wird ein Interface `BlocklibExecutorService` definiert, das von dem Interface `ScheduledExecutorService` der Java Bibliothek abgeleitet ist. Das Interface wird so erweitert, dass die verschiedenen `submit(...)` Methoden jeweils Objekte des Typs `CompletableFuture` zurückgeben. Die `schedule(...)` Methoden geben jeweils ein `ScheduledCompletableFuture` Objekt zurück, das später noch näher erläutert wird.

Das Interface `BlocklibExecutorService` wird durch die Klasse `BlocklibExecutor` implementiert. Sie nutzt zur Durchführung von Jobs den von der Java Bibliothek definierten `ScheduledThreadPoolExecutor`. Um die für die Rückgabewerte nötigen `CompletableFuture` Objekte zu erzeugen, wird eine Klasse `CompletableFutureWrapper` erstellt. Eine vollständige Auflistung der von den drei Interfaces bereitgestellten Methoden ist in Anhang D zu finden.

Die API des Jobsystems bietet verschiedene Methoden, um nebenläufige Anweisungen zu starten. Mittels der `submit(...)` Methoden können Anweisungen definiert werden, die so bald wie möglich ausgeführt werden. Da diese Methoden ein `CompletableFuture` Objekt zurückgeben, lassen sich über die dort definierten Methoden einfach nachgelagerten nebenläufige Anweisungen definieren, die abhängig von der Vollendung der ursprünglichen Anweisung ausgeführt werden. Mit den `schedule(...)` Methoden wird analog dazu eine Möglichkeit geboten Anweisungen zu definieren, die nach Ablauf eines bestimmten Zeitintervalls nebenläufig ausgeführt werden. Mittels `scheduleAtFixedRate(...)` und `scheduleWithFixedDelay(...)` können periodisch durchzuführende Anweisungen zur Ausführung gebracht werden.