Zusammenfassung

In der folgenden Arbeit wird eine nebenläufige Architektur für die in der Programmiersprache Java geschriebene 3D-Spielebibliothek Blocklib, die von Studierenden der Ostbayerischen Technischen Hochschule Regensburg entwickelt wird, entworfen und implementiert.

Die Anforderungen der Blocklib an eine nebenläufige Architektur werden erörtert und basierend auf den in der Spielindustrie gängigen nebenläufigen Architekturen „System on a Thread“ und „Jobsystem“ wird ein hybrides Architekturmodell entworfen, das Elemente beider Architekturen nutzt. Dieses Modell wird schließlich implementiert und verwendet bekannte Techniken wie Double-Buffering, um Wettkampfbedingungen zu verhindern. Ein Großteil der bestehenden nebenläufigen Strukturen wird durch die neue nebenläufige Architektur ersetzt, die nun zur Nutzung durch zukünftige Studierende zur Weiterentwicklung der Blocklib bereitsteht.

Die Architektur teilt die Blocklib grundlegend in zwei nebenläufige Bereiche auf, ein System zur Simulation der Inhalte der Spielebibliothek und ein System, das für das Rendering der Grafik verantwortlich ist. Des Weiteren wird eine Schnittstelle bereitgestellt, die von der im Java-Umfeld bekannten Schnittstelle `ScheduledExecutorService` abgeleitet und somit grundsätzlich vertraut ist, die dort gebotene Funktionalität aber um die Möglichkeit der Komposition von Jobs mittels der Schnittstelle `CompletionStage` erweitert. Auf diese Weise lassen sich komplexe Aufgaben in aufeinander aufbauende Jobs zerlegen, die automatisch nebenläufig ausgeführt werden.

In einer Performanceanalyse wird die Leistung der neuen Architektur mit der alten Architektur in fünf verschiedenen Szenarien verglichen, indem Messdaten zu den Metriken Bildwiederholrate, Auslastung der CPU und GPU sowie Speicherverbrauch erhoben werden. Aus den Messungen lässt sich eine verbesserte Leistung mit einer um 172 % gesteigerten Bildwiederholrate ermitteln. Die Daten zeigen zudem, dass durch das System eine stärkere Auslastung von CPU und GPU erreicht wird. Aber auch der Speicherverbrauch der Blocklib steigt in der neuen Architektur.

Zuletzt lassen die Messwerte darauf schließen, dass sich die Auslastung und die Bildwiederholrate weiter erhöhen lassen. Das kann erreicht werden, indem die neue Architektur genutzt wird, um mehr Aufgaben in der Simulation der Blocklib nebenläufig durchzuführen. Es wird erwartet, dass dies die Leistung der Spielebibliothek steigern kann, da die gemessenen Werte der Auslastung der CPU darauf schließen lassen, dass diese weiterhin aufgrund mangelnder Nebenläufigkeit einen Flaschenhals darstellt, der das Leistungspotenzial der Blocklib einschränkt.

4 Implementierung der nebenläufigen Architektur

Ebenso wie der Entwurf, ist auch die Implementierung der nebenläufigen Architektur prinzipiell in zwei Bereiche unterteilt, die Implementierung des Render-Threads und die Implementierung des Jobsystems.

Um einen Render-Thread in die Architektur der Blocklib zu integrieren, muss besonders darauf geachtet werden, dass Wettkampfbedingungen vermieden werden. Wie in Abschnitt 3.3.1 beschrieben, kann dies unter anderem durch die Nutzung von Double-Buffern gelingen. Folgend wird die Implementierung des Render-Threads beschrieben. Im darauffolgenden Abschnitt wird die Implementierung des Jobsystems erläutert.

4.1 Implementierung des Render-Threads

Da ein Jobsystem implementiert wird, können alle Simulationsanweisungen auf Job-Threads ausgeführt werden. Damit ist der Thread, der in Java mit dem Start des Programms erzeugt wird, der `main`-Thread, unbenutzt und kann als Render-Thread das Rendering ausführen. Alle anderen Rechenprozesse werden über das Jobsystem auf anderen Threads durchgeführt. Bei der Implementierung der für einen Render-Thread nötigen Funktionalität müssen in der Blocklib viele verschiedene Klassen angepasst werden. Der Übersichtlichkeit halber werden die Anpassungen in Bereiche eingeteilt.

In Abschnitt 4.1.1 wird beschrieben, wie das Laden von Daten auf die *Grafikkarte (engl. graphics processing unit) (GPU)* geändert werden muss, weil dieser Vorgang aus Sicht der Simulation nun wegen der Nebenläufigkeit des Render-Threads asynchron ist. Danach erläutert Abschnitt 4.1.2 den Wechsel von einer Render-Architektur, die sich zu zeichnende Elemente merkt, zu einer zustandslosen Architektur. Abschnitt 4.1.3 beschreibt, wie darauf aufbauend der Zustand der zu zeichnenden Elemente unter Verwendung von Double-Buffers zwischengespeichert wird, damit er während des Renderings unverändert bleibt und so Pipelining ermöglicht wird. In Abschnitt 4.1.4 wird beschrieben, wie der Aufbau der Game-Loop an die neue Architektur angepasst wird. Schließlich wird in Abschnitt 4.1.5 anhand eines Beispiels gezeigt, warum eine zentrale Zwischenspeicherung der Render-Daten zur Vermeidung von Wettkampfbedingungen sinnvoll ist.

4.1.1 Laden von Daten auf die Grafikkarte

In der Blocklib werden häufig Daten an die GPU übergeben. Beispielsweise muss bei der Generierung eines Chunks ein Gitternetz konstruiert werden, das die Form des Terrains im Chunk beschreibt, und dann an die GPU gesendet werden. Dafür wird die OpenGL API genutzt. Alle Aufrufe an OpenGL müssen auf einem Thread ausgeführt werden, die Blocklib wird aber nebenläufig ausgeführt. Daher muss dafür gesorgt werden, dass Anweisungen zwischengespeichert

werden, die mit der GPU interagieren, aber zum Beispiel durch die Chunk-Generierung angestoßen werden. Die zwischengespeicherten Anweisungen können dann von dem Render-Thread gemeinsam ausgelesen und ausgeführt werden.

Um Anweisungen zum Laden von Daten auf die GPU zwischenzuspeichern, muss die dafür zuständige Klasse `Loader` angepasst werden. Die Klasse wird um eine Warteschlange erweitert, die Objekte des Typs `Runnable` enthält. Anstatt die Ladeanweisungen direkt auszuführen, werden sie als Tasks der Warteschlange übergeben und dort als `Runnable`-Objekte gespeichert. Die Warteschlange nutzt intern die von Java bereitgestellte Klasse `ConcurrentLinkedQueue`. Die Warteschlange bietet damit nicht-blockierende Synchronisierung, wodurch mehrere Threads gleichzeitig Elemente der Warteschlange hinzufügen können, ohne dass dies zu Wettkampfbedingungen führen kann. In der Game-Loop der Blocklib wird diese Warteschlange zu Beginn jedes Frames von dem Render-Thread abgearbeitet.

Ein Diagramm der wichtigsten beteiligten Klassen ist in Abbildung 18 gezeigt. Im Klassendiagramm sind die drei Klassen `VAOTransmitter`, `TextureTransmitter` und `DataTransmitter` abgebildet. Die Klasse `Loader` erzeugt Tasks, die dann Methoden dieser Klassen aufrufen, je nachdem, welche Art von Daten an die GPU übertragen oder von dort gelöscht werden soll. Die Klasse `VAOTransmitter` ermöglicht den Umgang mit sogenannten *Vertex-Array-Objects (VAOs)*. Das ist eine Datenstruktur, die alle relevanten Informationen zur Darstellung eines Modells enthält, beispielsweise eines Chunks oder des Spielercharakters. Mit der Klasse `TextureTransmitter` lassen sich Texturen übertragen, die beispielsweise das Aussehen von Blöcken definieren. Der `DataTransmitter` wird genutzt, um die Daten an die GPU zu senden, die in der Blocklib für die Berechnung von Niederschlag, wie beispielsweise Regen oder Schnee, benötigt werden.

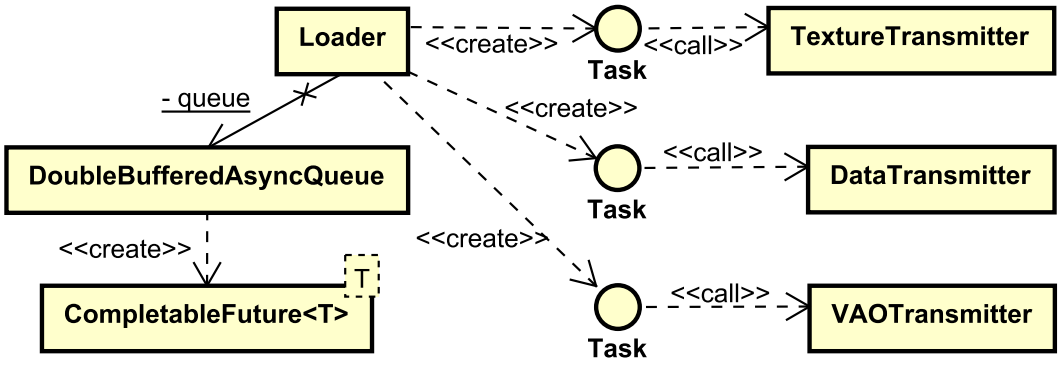

Abbildung 18: Klassendiagramm der wichtigsten Klassen, die am Laden und Entfernen von Daten auf die und von der GPU beteiligt sind. Zum Laden wird der `Loader` aufgerufen, dieser erstellt einen Task, der eine der drei Transmitter-Klassen aufruft. Die so erstellten Tasks werden an die `DoubleBufferedAsyncQueue` übergeben, die ein `CompletableFuture`-Objekt zurückgibt.

Die Klassen `RawModel` und `ModelTexture`, welche die Informationen der VAOs und Texturen

Abbildung 19: Bildschirmfoto der Spielerfigur, die in diesem Kapitel als Beispiel für die Implementierung des Render-Threads verwendet wird.

speichern, implementieren das neu hinzugefügte Interface `AsyncGraphicsObject`, das die Methoden `setLoaded(...)`, `setUnloaded()` und `isLoaded()` bereitstellt. Damit kann überprüft werden, ob die Daten bereits auf die GPU geladen sind. Die Renderer-Klassen, auf die in Abschnitt 4.1.3 genauer eingegangen wird, prüfen damit, ob sie ausschließlich auf der GPU befindliche Objekte rendern sollen, und geben andernfalls einen Fehler aus.

Die Abarbeitung aller Ladevorgänge der Klasse `Loader` muss synchronisiert sein, da die Simulation sonst beispielsweise versuchen könnte, ein Objekt rendern zu lassen, das später nebenläufig von der GPU entfernt werden würde. Nachdem der Ladevorgang von Objekten auf die GPU abgeschlossen ist, kann auf allen Threads unabhängig nebenläufig überprüft werden, dass die Objekte gerendert werden können. Dieser Vorgang wird nun mit einem Beispiel verdeutlicht.

Beispiel:

Die Klasse `Player` ist für die Simulation des Spielercharakters zuständig. Zur Veranschaulichung ist ein Bildschirmfoto der Spielerfigur in Abbildung 19 dargestellt. Wird ein neuer Spielercharakter erstellt, so ruft die Klasse `Player` die Methode `init()` auf, wodurch über verschiedene Methodenaufrufe zum Schluss auch `Loader.loadToVAO(...)` aufgerufen wird. Der `Loader` erzeugt einen Task mit Daten des `Player`-Modells und fügt diesen Task in die Warteschlange ein. Im darauffolgenden Frame wird die Warteschlange abgearbeitet und das Modell auf die GPU geladen. Damit steht das zu ladende Spielercharakter-Modell ab diesem Frame zur Verfügung und kann genutzt werden, um den Charakter zu zeichnen.

Wie der Vorgang des Zeichnens aufgebaut ist, wird in den folgenden Abschnitten erklärt. Dort wird dieses Beispiel fortgeführt.

4.1.2 Zustandsloses Rendering

In der Blocklib implementieren alle zu zeichnenden Elemente, auch Renderables genannt, das Interface `Renderable`. Dieses definiert die Funktionalität, die zum Zeichnen eines Elements notwendig ist, sowie die Methoden `show()` und `hide()`. Das Rendersystem ist so aufgebaut, dass ein Element nach einem Aufruf von `show()` so lange gezeichnet wird, bis `hide()` aufgerufen wird.

Dies wird erreicht, indem eine Datenstruktur im `MasterRenderer` alle Renderables speichert. Diese zustandsbehaftete Zeichenmethode birgt die folgenden Vor- und Nachteile im Vergleich zu einem Rendering, das diese Informationen nicht speichert (hier *zustandsloses Rendering* genannt):

+ Da `show()` nur einmal aufgerufen werden muss, können auch Systeme ohne Update-Methode das Zeichnen von Elementen veranlassen.
+ Da die Elemente gespeichert sind, müssen sie nicht jedes Mal neu hinzugefügt werden. Das verringert den Rechenbedarf.
– Werden Renderables häufig ausgetauscht, müssen die alten Elemente jedes Mal entfernt werden.
– Da die Einführung eines parallelen Render-Threads ansonsten zu Wettkampfbedingungen führen würde, muss diese Datenstruktur aus Sicht des Render-Threads während des Zeichnens unverändert sein. Wie beschrieben wird dazu ein Double-Buffer eingesetzt. Bei einem zustandsbehafteten Double-Buffer müssen beim Swap die Elemente des einen Buffers in den anderen Buffer kopiert werden. Das erfordert Zeit, die nicht parallelisiert werden kann, da das Wechseln des Buffers synchronisiert sein muss.
– Die Existenz von paarweise aufzurufenden Funktionen birgt die Gefahr, dass der zweite Aufruf vergessen wird. Wie bei `malloc()` und `free()` in der Programmiersprache C entsteht durch einen fehlenden Aufruf von `hide()` ein Speicherleck. Zudem wird dann die Anzahl der zu zeichnenden Elemente immer größer und der Rendervorgang wird verlangsamt. Des Weiteren werden möglicherweise Renderables gezeichnet, die nicht gezeichnet werden sollen.

Zur Verringerung des Zeichenaufwands implementiert die Klasse `ChunkManager` das sogenannte *Frustum Culling*. Es wird berechnet, welche Chunks sich im Sichtfeld der Kamera befinden. Nur diese Chunks sollen gezeichnet werden. Dazu entfernt der `ChunkManager` jeden Frame alle Chunks mittels `hide()` aus der Datenstruktur der zu zeichnenden Elemente und fügt nur die als sichtbar ermittelten Chunks wieder ein. Die Chunks machen mit etwa 75 % bis 82 % einen Großteil aller Renderables aus. Es wird also bereits ein Großteil der zu zeichnenden Elemente in jedem Frame neu hinzugefügt. Des Weiteren existiert aktuell in der Blocklib keine Klasse, die Renderables zu der Datenstruktur hinzufügt, aber keine Update-Methode besitzt. Deswegen werden die Gefahr von vergessenen `hide()` Aufrufen und der Kopieraufwand des Double-Buffers als gewichtiger eingeschätzt als die zuvor beschriebenen Vorteile.

Die Datenstruktur ist nun wie folgt implementiert: Die Methode `hide()` entfällt ersatzlos. Stattdessen wird der Puffer der Renderables des letzten Bilds nach jedem Double-Buffer-Tausch geleert. So zeichnet der Renderer Renderables automatisch nicht mehr, sobald sie nicht mehr hinzugefügt werden. Die Methode `show()` wird zu `draw()` umbenannt, um zu signalisieren, dass es sich um einen einmaligen Vorgang handelt. Alle bisherigen Aufrufe von `show()` werden

6 Fazit

In dieser Arbeit wurde basierend auf der Analyse der Blocklib eine nebenläufige Architektur entworfen und implementiert. Die Architektur baut auf den Strukturen und Ideen System on a Thread (SoT) und Jobsystem auf, die in der Computerspielindustrie für die Entwicklung nebenläufiger Architekturen verwendet werden. Da die Blocklib OpenGL nutzt, sind neben des Jobsystem-Designs notwendigerweise auch Aspekte des SoT integriert worden. Dadurch ist eine hybride Form der Modelle entstanden.

Der SoT-Teil der Architektur nutzt einen Render-Thread, um das Rendering nebenläufig zu der Simulation durchzuführen. Zentrale Zwischenspeichersysteme, die aus Double-Buffern bestehen, sorgen dafür, dass Wettkampfbedingungen auf den zu rendernden Objekten vermieden werden. Um die Leistung zu optimieren, werden die Objekte, die zur Zwischenspeicherung genutzt werden, nach Gebrauch wiederverwendet.

Das implementiere Jobsystem erfüllt die ermittelten Anforderungen. Es definiert eine Stelle, an der Threads kontrolliert werden, den `BlocklibExecutor`, und eine zugehörige Schnittstelle `BlocklibExecutorService`. Das Jobsystem bietet die Möglichkeit, Aufgaben zu definieren, die nebenläufig von Threads abgearbeitet werden. Durch die Dekorierung dieser Aufgaben können über die Schnittstelle `CompletionStage` komplexe Aufgaben gelöst werden. Das wird ermöglicht, indem `CompletionStage` Methoden bereitstellt, um einzelne Jobs zu einem komplexen Job-Graphen zu komponieren. Des Weiteren ermöglicht das Jobsystem, wiederkehrende Aufgaben zu definieren, die automatisiert zu bestimmten Zeiten nebenläufig ausgeführt werden.

Das Jobsystem ermöglicht eine rudimentäre Definition von Prioritäten für die aufgegebenen Tasks, indem zwei getrennte Thread-Pools genutzt werden. Das System ist so vorbereitet, dass ein verbessertes Prioritätensystem basierend auf einer Prioritäts-Warteschlange einfach zu integrieren ist, da die Schnittstelle `BlocklibExecutorService` bereits entsprechend definiert ist und der Aufzählungstyp `TaskPriority` bereitgestellt wird.

Ein Großteil der bereits bestehenden Nutzung von Multithreading konnte so geändert werden, dass nun die neue nebenläufige Architektur verwendet wird. Einzig die Netzwerkfunktionalitäten der Blocklib bilden hier eine Ausnahme.

Der Start der Blocklib wird durch die neue Architektur um eine Zeit im einstelligen Sekundenbereich verlangsamt. Verglichen mit der alten Architektur erreicht die neue nebenläufige Architektur eine um durchschnittlich 172 % höhere Bildwiederholrate und weist damit eine deutlich gestiegene Leistung auf. Dabei nutzt die Blocklib mit der neuen Architektur auch einen größeren Teil der zur Verfügung stehenden Leistung des Testsystems aus. Durchschnittlich wird die CPU nun 40 % und die GPU 37 % stärker ausgelastet als zuvor. In den Messungen ist die CPU auch mit der neuen Architektur weiterhin die leistungsbeschränkende Komponente. In beinahe keinem Szenario ist die GPU voll ausgelastet.