Übungen zur Vorlesung "Architektur und Programmierung von Grafik- und Koprozessoren"
Übungsblatt 8

Sommersemester 2018

8 Simulation auf der GPU

Aufgabe 8.1

Die Klassische Mechanik fußt auf Isaac Newtons Gesetzen für massebehaftete Körper. Massebehaftete Körper sind *ausdehnungslos* und gegeben durch ihre Position x im $\mathbb{R}^3$, durch ihre Masse $m \in \mathbb{R}$ sowie durch den Vektor für die *Geschwindigkeit* $\mathbf{v} = \frac{dx}{dt}$. Daraus kann man Vektoren für die *Beschleunigung* $\mathbf{a} = \frac{d\mathbf{v}}{dt}$ berechnen. Position, Geschwindigkeit und Beschleunigung massebehafteter Körper können sich im Laufe der Zeit t verändern. Dazu berechnet man Kräfte, die die Körper aufeinander auswirken. Beschleunigung führt zu Veränderungen der Geschwindigkeit, und die Körper bewegen sich im Laufe der Zeit proportional zu Betrag und Richtung der veränderten Geschwindigkeitsvektoren. Für die in der Mechanik häufig benötigten *Zeitableitungen* wie etwa $\frac{dx}{dt}$ oder $\frac{d^2x}{dt^2}$ verwendet man auch die Kurzformen $\dot{x}$ und $\ddot{x}$. Die vektorielle Größe $\mathbf{p} = m\mathbf{v}$ nennt man *Impuls*. Die axiomatischen Newtonschen Gesetze erklären die Bewegung von massebehafteten Körpern, wie sie mit technischen Mitteln *zur Zeit Newtons* beobachtet werden konnte. Sie lauten wie folgt:

1.) Trägheit: ist die Summe aller Kräfte $\mathbf{F}_{ij}$, die auf Körper i durch alle anderen Körper $j \neq i$ wirkt 0, ist dessen Geschwindigkeit $\mathbf{v}_i$ *konstant*:

$$\sum_j \mathbf{F}_{ij} = 0 \Leftrightarrow \frac{d\mathbf{v}_i}{dt} = 0. \tag{1}$$

2.) Die Kraft, die auf einen Körper mit konstanter Masse wirkt, ist proportional zum Produkt aus Masse und Beschleunigung (Veränderungsrate des Impuls):

$$\mathbf{F} = m\mathbf{a}. \tag{2}$$

3.) Sei $\mathbf{F}_{ij}$ die Kraft, die Körper i auf Körper j auswirkt, und sei $\mathbf{F}_{ji}$ die umgekehrte Kraft. Dann gilt

$$\mathbf{F}_{ij} = -\mathbf{F}_{ji}. \tag{3}$$

(Das erste und dritte Gesetz bezieht sich auf das Zusammenspiel mehrerer Körper. Betrachtet man Gesetzmäßigkeiten, die sich nur auf einen Körper beziehen, lässt man, wie beim zweiten Gesetz, das Subskript, das den Körper unter allen anderen identifiziert, der Einfachheit halber häufig weg.)

a.)
Zeigen Sie, dass Newtons erstes Gesetz aus Newtons zweitem Gesetz folgt. (1 Punkt)

b.)
In einem *geschlossenen System* befinden sich n Körper. Außer der Kraft, die die Körper aufeinander bewirken, wirken keine weiteren Kräfte. Zeigen Sie mit Hilfe der Newtonschen Gesetze, dass in einem geschlossenen System das Prinzip der *Impulserhaltung* gilt:

$$\sum_{i}^{n} \dot{\mathbf{p}}_i = 0. \tag{4}$$

(1 Punkt)

c.)
Das ebenfalls axiomatische *Newtonsche Gravitationsgesetz* beschreibt die *Gravitationskraft* zwischen Paaren von Körpern i und j in einem *Inertialsystem* als

$$\mathbf{F}_{ij} = G\frac{m_i m_j}{|x_j - x_i|^2}\frac{x_j - x_i}{|x_j - x_i|}. \tag{5}$$

Dabei bezeichnet G die *Gravitationskonstante* $(6,674 \times 10^{-11} N \cdot kg^{-2} \cdot m^2)$. Es folgt die dem *Mehrkörperproblem* zugrunde liegende Bewegungsgleichung:

$$m_i \frac{d^2 x_i}{dt^2} = \sum_{j,j\neq i} G\frac{m_i m_j (x_j - x_i)}{|x_j - x_i|^3} \Leftrightarrow a_i = \sum_{j,j\neq i} G\frac{m_j (x_j - x_i)}{|x_j - x_i|^3}. \tag{6}$$

Zeigen Sie, wie sich $\mathbf{F}_{ij}$ für $|x_j - x_i| \to 0$ sowie für $|x_j - x_i| \to \infty$ verhält. (2 Punkte)

d.)
Beim Mehrkörperproblem werden für n Körper deren paarweise Kräftewechselwirkungen bestimmt, um zu diskreten Zeitpunkten neue Positionen und Geschwindigkeitsvektoren zu berechnen.Das Mehrkörperproblem kann man z. B. mittels Simulation lösen. Implementieren Sie mit Hilfe des Gerüstprogramms ein Vulkan Programm, das das Mehrkörperproblem für n Körper parallel auf der GPU löst. Zur Lösung der Aufgabe können Sie wieder, wie für die Bearbeitung des vorherigen Aufgabenblatts, das Entwicklungssystem verwenden. Im Gerüstprogramm sind noch zwei Anpassungen vorzunehmen, damit die Mehrkörpersimulation funktioniert.

Die Simulation verteilt sich auf *zwei* Compute Shader. Den ersten Compute Shader implementieren Sie. Der zweite Compute Shader verwendet Ihr Ergebnis und passt auf Basis dessen die Positionen der Körper an und verschiebt sie entsprechend der wirkenden Kräfte. Der von Ihnen zu implementierende Compute Shader

1. läuft parallel und wird genau ein Mal (pro Zeitschritt) pro Körper i aufgerufen,
2. identifiziert den Index des Körpers, für den die Compute Shader Instanz zuständig ist, mittels der Variable `gl_GlobalInvocationID.x`,
3. rechnet die Beschleunigung a_i durch *serielles* Aufsummieren aus und speichert sie im SSBO mit `binding 1`. Der zweite Compute Shader verwendet diese, um die Geschwindigkeitsvektoren und Positionen der Körper zu aktualisieren.

Wegen der Annahme, dass die Körper ausdehnungslos sind, können sich auch Körper mit betraglich sehr großer Masse unendlich nahe kommen (vgl. Aufgabenteil **c.)**). Um die dadurch auftretenden Singularitäten zu vermeiden, führt man bei der Berechnung der Beschleunigungsvektoren eine *Abschwächungskonstante* ϵ^2 ein. Verwenden Sie für Ihre Berechnungen die entsprechend angepasste Formel für die Beschleunigung

$$a_i \approx \sum_{j} G\frac{m_j (x_j - x_i)}{\left(|x_j - x_i|^2 + \epsilon^2\right)^{\frac{3}{2}}}. \tag{7}$$

Für ϵ^2 ist mit Bezug auf die Initialisierung der Beispieldaten der Wert $0,000125$ sinnvoll. Durch die Abschwächungskonstante brauchen Sie den Fall $j \neq i$ nicht mehr gesondert zu behandeln, da a_i dann automatisch den Wert 0 erhält.

Ihre zweite Aufgabe ist es, die Ausführung der beiden Compute Shader im Vulkan Programm zu synchronisieren. Der zweite Compute Shader sollte erst gestartet werden, wenn alle Instanzen des zweiten Compute Shaders durchlaufen wurden. Andernfalls werden beim Integrationsschritt, bei dem die Positionen aktualisiert werden, eventuell noch nicht korrekt aktualisierte Geschwindigkeitsvektoren benutzt und das Simulationsergebnis ist nicht korrekt. Fügen Sie dazu mit Hilfe des *Kommandos* `vkCmdPipelineBarrier()` ein `VkMemoryBarrier` Objekt in den Command Buffer des Vulkan Programms ein, sodass *zwischen* der Ausführung des ersten Compute Shaders und dem Binden der zweiten Compute Pipeline gewartet wird. Die offizielle Dokumentation zum Vulkan API beschreibt, wie man *Compute to Compute Dependencies* synchronisiert:

`https://github.com/KhronosGroup/Vulkan-Docs/wiki/Synchronization-Examples`.

Zum Schluss können Sie die Ausgabe der Simulation noch visualisieren, um zu überprüfen, ob Sie richtig gerechnet haben. Das Gerüstprogramm exportiert die Ausgabe im `.csv` Format, sodass das Ergebnis mit *ParaView* (`https://www.paraview.org/`) visualisiert werden kann. Installieren Sie ParaView und folgen Sie den Anweisungen unter

`https://www.olcf.ornl.gov/using-paraview-to-view-csv-files/`,

um die Ausgabedateien, die Sie vorher vom Entwicklungssystem herunterladen müssen, zu importieren und zu visualisieren. Eine Beispielsequenz mit 256 Zeitschritten liegt dem Gerüstprogramm bei, Ihr Ergebnis sollte vergleichbar aussehen. (6 Punkte)

Aufgabe 8.2

Zelluläre Automaten sind gitterbasierte Modelle. Gitterzellen sind definiert durch ihren *Zustand*, der sich aufgrund von Nachbarzuständen im Verlauf der Zeit verändert. Zustandsveränderungen werden aufgrund eines Regelwerks beschrieben.

John Conways *Game of Life* basiert auf einem Zellulären Automaten, der auf einem zweidimensionalen, uniformen, rechteckigen Gitter definiert ist: jede Zelle kann durch ihre Position $(x, y) \in (0 \dots N-1, 0 \dots M-1)$ eindeutig identifiziert werden und speichert Zustände $s(x, y, t) \in \{0, 1\}$. Die Zustände sind zeitabhängig und die Dimensionen des Gitters gegeben durch Breite N und Höhe M. Bezeichne $\mathfrak{S}(x, y, t)$ die Anzahl der Nachbarzellen der Zelle an Position (x, y), für die $s = 1$. Um die $s(x, y, t)$ für alle Gitterzellen zu ermitteln, wendet man die nachfolgenden Regeln auf die Gitterzellen an:

$$s(x,y,t) = \begin{cases} 1 \text{ falls } s(x,y,t-1) = 1 \wedge (\mathfrak{S}(x,y,t-1) = 2 \vee \mathfrak{S}(x,y,t-1) = 3) \\ 1 \text{ falls } s(x,y,t-1) = 0 \wedge \mathfrak{S}(x,y,t-1) = 3 \\ 0 \text{ sonst.} \end{cases}$$

Programmieren Sie mit Hilfe des Gerüstprogramms Conways Game of Life mit einem Compute Shader. Der Compute Shader wird diesmal für $N \times M$ Threads auf einem zweidimensionalen Gitter aufgerufen, sodass Sie die Threads mittels ihrer 2D Position im Gitter (`gl_GlobalInvocationID.x` und `gl_GlobalInvocationID.y`) identifizieren können.

Die Updates des Zellulären Automaten sollten Sie so programmieren, dass für zwei aufeinanderfolgende Zeitschritte zwei Buffer verwendet werden, die nach jedem Zeitschritt ihre Rolle tauschen. Im Zeitschritt t wird der alte Zustand aller Zellen aus Zeitschritt $t-1$ aus Buffer 0 *gelesen*. Der

aktualisierte Zustand wird in Buffer 1 *geschrieben*. Um nun den nächsten Zustand für Zeitschritt $t+1$ zu ermitteln, werden die Rollen der beiden Buffer vertauscht, sodass aus Buffer 1 gelesen und in Buffer 0 geschrieben wird. Danach vertauschen Sie immer wieder die Buffer, bis alle Zeitschritte berechnet wurden. Das Tauschen der Buffer kann man z. B. erreichen, indem man beim Aufruf von `vkUpdateDescriptorSets()` mit der zugehörigen `struct` den jeweiligen Buffer mittels des `BufferInfo` Objekts an der richtigen Position im Descriptor Set bindet.

Rufen Sie das Gerüstprogramm ohne Parameter auf, wird eine einfache Startkonfiguration initialisiert. Außerdem können Sie dem Programm Initialisierungsdateien in der Form ähnlich der beigefügten `.txt` Datei übergeben. Sie müssen dann jedoch die Konstanten in `main.cpp` sowie im Shader anpassen, sodass Breite und Höhe des Gitters zur Eingabe passen. Ebenso können Sie die Anzahl berechneter Zeitschritte mittels einer Konstante in der Funktion `main()` zur Compile Zeit des Programms manipulieren.

Das Gerüstprogramm schreibt die Zeitschritte in eine Datei im *NetCDF* Format (vgl. `https://www.unidata.ucar.edu/software/netcdf/docs/netcdf_introduction.html`) in das aktuelle Arbeitsverzeichnis. NetCDF Dateien können Sie z. B. mit dem Programm `ncview` anschauen, das auf dem Entwicklungssystem installiert ist. Dies können Sie bspw. per X11 Forwarding benutzen. Dazu verbinden Sie sich (am besten mit einer zweiten Session, da in der X11 Forwarding Session die lokale Grafikkarte nicht für Berechnungen zur Verfügung steht) z. B. mit OpenSSH und Parameter `-X`, um X11 Forwarding zu aktivieren. Unter Windows müssen Sie einen X11 Client (z. B. `Xming`) installieren, um X11 Forwarding mit `PuTTY` verwenden zu können. Achten Sie dann darauf, vor dem Verbinden in der `PuTTY` Konfiguration unter `Connection->SSH->X11` X11 Forwarding zu aktivieren. (10 Punkte)

Abgabe bitte bis zum 04.07.2018, 22:00h in Ilias.