spp-support@lists.sc.informatik.tu-darmstadt.de

Systemnahe und Parallele Programmierung (WS 22/23)

Praktikum: CUDA

Die Lösungen müssen bis zum 14. Februar 2023, 16:00 Uhr, in Moodle submittiert werden. Anschließend müssen Sie Ihre Lösungen einem Tutor vorführen. Alle Programmieraufgaben müssen mit CUDA und C/C++ gelöst werden, und dann auf dem Lichtenberg Cluster kompilier- und ausführbar sein. Alle Lösungen müssen zusammen in einer Archivdatei eingereicht werden.

In diesem Praktikum geht es um die CUDA Programmierung. Im Folgenden sollen Sie die Implementierung eines Raytracing Algorithmus vervollständigen. Beim Raytracing schickt man für jedes Pixel der Bildebene einen Sichtstrahl vom Augpunkt (der Kamera) durch das entsprechende Pixel in die Szene und prüft welche Objekte der Strahl trifft. Wird ein Schnittpunkt gefunden, berechnet man ausgehend von diesem die Beleuchtung an dieser Stelle, in dem geprüft wird, welche Lichtquellen nicht verdeckt sind und in welchem Winkel ihre Lichtstrahlen die Oberfläche treffen. Zudem werden am Schnittpunkt sekundäre Strahlen (Reflexionsstrahlen) generiert, die von der Oberfläche gespiegelt und wieder in die Szene geschickt werden. Treffen diese das gleiche (Selbstreflexion) oder ein anderes Objekt in der Szene, wird die dadurch erzeugte Farbinformation zum Ursprungsschnittpunkt hinzugefügt. Dieses Vorgehen lässt sich prinzipiell beliebig oft wiederholen um Spiegelungen von Spiegelungen zu berechnen.

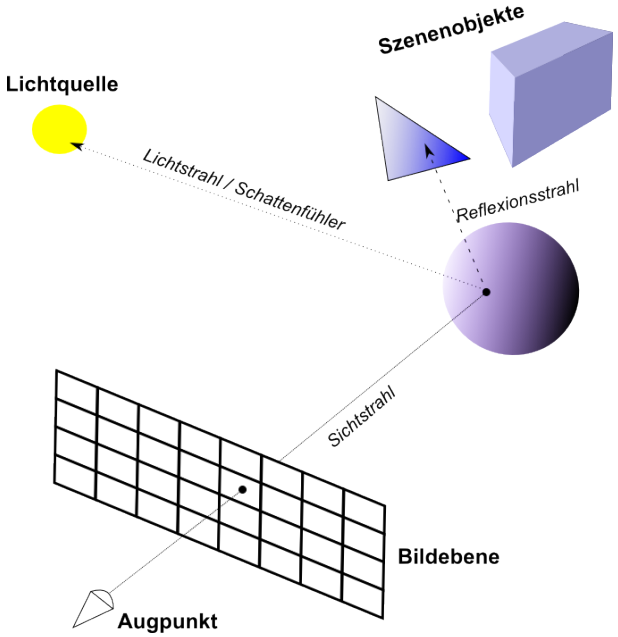


Figure 1: Schema des Raytracings

Der im Praktikum eingesetzte Raytracer besitzt folgende Eigenschaften / Einschränkungen

- Es werden nur Reflexionsstrahlen von Sichtstrahlen berechnet, keine weiteren/tieferen.
- Die Szenenobjekte sind aus Dreiecksnetzen aufgebaut.
- Es gibt nur eine Lichtquelle in der Szene.

Nachdem die Host Funktionen vervollständigt sind, speichert das Programm das berechnete Bild in der Datei `img.bmp`.

Zur Bearbeitung der Aufgabe müssen Sie nur in der Datei `kernel.cu` arbeiten, in der sich der Code für die Ausführung auf der Grafikkarte, sowie die Verwaltungsfunktionen zur Datenübertragung und zum Aufrufen des Kernels befinden. Vervollständigen Sie die Implementierung in der Datei `kernel.cu` anhand der folgenden Aufgaben.
Allgemeine Hinweise: Zum Kompilieren oder Ausführen eines CUDA Programms auf dem Lichtenberg Cluster müssen sie das CUDA Modul laden: `module load cuda`. Kompilieren können sie bereits auf den Login Knoten. Zum Kompilieren, setzt dieses Praktikum CMake[1] ein. Laden sie zuerst das Modul mit `module load cmake`. Anschließend führen Sie im Hauptverzeichnis `mkdir build`, `cd build`, `cmake ..` aus, gefolgt von `make`.
Zum Ausführen des CUDA Progamms muss ein Batch Job submittiert werden. Nutzen sie dafür die Datei `batch_job.sh` als Beispiel.

Aufgabe 1

(6 Punkte) Allozieren Sie am Beginn der Funktion `cudaError_t setupCudaMemory` unter dem Kommentar *`//Buffer for result picture`* den Ausgabebuffer für das erzeugte Bild auf der Grafikkarte mit der benötigten Größe. Der zu benutzende Buffer ist bereits als globale Variable **`float4`**`* result_Buffer` deklariert. Die jeweils ersten drei Einträge eines **`float4`** werden nachher die RGB-Werte enthalten. Der vierte Eintrag ist ungenutzt.
Stellen Sie sicher, dass der Buffer mit 0 initialisiert wird. Geben Sie im Falle von Fehlern entsprechende Meldungen an den Benutzer zurück welcher Arbeitsschritt fehlgeschlagen ist.

Aufgabe 2

(6 Punkte) Allozieren Sie im weiteren Verlauf von `cudaError_t setupCudaMemory` unter dem Kommentar *`//Buffer for vertices`* einen Eingabebuffer für alle Eckpunkte der Dreiecke, aus denen die Modelle bestehen. Ihre Anzahl ist durch das Attribut `vtxcnt` des `SplitScene*`-Pointers `scene` gegeben. Der zu benutzende Buffer ist bereits als globale Variable **`float4`**`* vertices_Buffer` deklariert.
Kopieren sie anschließend alle Endpunkte aus dem Pointer **`float`**`* vertices` vom Host auf die Grafikkarte. Ein Vertex in **`float`**`* vertices` besteht aus seiner 3D Koordinate und einem vierten Dummy-Wert.
Geben Sie im Falle von Fehlern entsprechende Meldungen an den Benutzer zurück welcher Arbeitsschritt fehlgeschlagen ist.

Aufgabe 3

(6 Punkte) Da die Information für die Position der Kamera und die der Lichtquelle bei der Berechnung jedes Pixels des Ausgabebildes gelesen werden muss, sollen sie im *constant memory* der Grafikkarte abgelegt werden.
Deklarieren Sie hierzu am Beginn der Datei *kernel.cu* unter *`//Constant buffers`* die beiden entsprechenden **`float`**-Arrays `camera_Buffer` und **`float`** `lightdat_Buffer`. Die Kamera benötigt acht Einträge, die Lichtquelleninformation zwölf.
Kopieren Sie die Kameradaten aus `camera` unter dem Kommentar *`//Buffer for camera`* bzw. die Lichtquellendaten `lightdat` unter dem Kommentar *`//Buffer for light`* in den entsprechenden *constant memory* Bereich.
Geben Sie im Falle von Fehlern entsprechende Meldungen an den Benutzer zurück welcher Arbeitsschritt fehlgeschlagen ist.

Aufgabe 4

(5 Punkte) Kopieren Sie nach der Ausführung des Kernels in der Funktion `traceWithCuda` nach dem Kommentar *`//Copy output from GPU to host`* den Ausgabebuffer von der Grafikkarte an den lokalen Pointer **`float`**`* result` und melden Sie auftretende Fehler per Konsolenmeldung.

Aufgabe 5

(15 Punkte) Um die Parallelität der Grafikkarte ausnutzen zu können, muss die Arbeit der Bilderzeugung auf verschiedene Threads verteilt werden. In unserem Fall soll jeweils ein Thread ein Pixel des

[1] `https://cmake.org/`

Ausgabebildes berechnen. Als Beispiel betrachten wir ein Bild mit zwölf Pixeln Breite und acht Pixeln Höhe. Stellt man sich das Bild als 2-dimensionales Array von Pixeln vor, so würden diese in einem linearen Array folgendermaßen nummeriert sein:

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80	81	82	83
84	85	86	87	88	89	90	91	92	93	94	95

CUDA nummeriert seine Threads abhängig von den Blocks, in denen sie ausgeführt werden. Nehmen wir an, wir wollen das Bild mit einer Blockgröße von (4, 4) auf einem (3, 2) Grid ausführen. Dann wären die Threads global folgendermaßen nummeriert:

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80	81	82	83
84	85	86	87	88	89	90	91	92	93	94	95

Für das Raytracing werden zudem die 2D Koordinaten des entsprechenden Pixels benötigt. Diese sind folgendermaßen nummeriert:

0/0	0/1	0/2	0/3	0/4	0/5	0/6	0/7	0/8	0/9	0/10	0/11
1/0	1/1	1/2	1/3	1/4	1/5	1/6	1/7	1/8	1/9	1/10	1/11
2/0	2/1	2/2	2/3	2/4	2/5	2/6	2/7	2/8	2/9	2/10	2/11
3/0	3/1	3/2	3/3	3/4	3/5	3/6	3/7	3/8	3/9	3/10	3/11
4/0	4/1	4/2	4/3	4/4	4/5	4/6	4/7	4/8	4/9	4/10	4/11
5/0	5/1	5/2	5/3	5/4	5/5	5/6	5/7	5/8	5/9	5/10	5/11
6/0	6/1	6/2	6/3	6/4	6/5	6/6	6/7	6/8	6/9	6/10	6/11
7/0	7/1	7/2	7/3	7/4	7/5	7/6	7/7	7/8	7/9	7/10	7/11

Implementieren Sie die Berechnung der 2D Koordinaten und des seriellen Index des jeweiligen Pixels am Beginn der Funktion `__global__ void traceLA` und speichern Sie das Ergebnis in den vorgegebenen Variablen `globalIdx`, `globalIdy` und `globalIdSerial`. Stellen Sie außerdem sicher, dass durch falsch gesetzte Grid- und Blockgrößen keine Pixelkoordinaten außerhalb des Bildbereichs berechnet werden. Die Variable `int2 resinfo` hat dazu an ihrer ersten Stelle die Breite des Bildes und an zweiter Stelle seine Höhe gespeichert.
Zum Testen können Sie den Codeblock, der mit *`//BLOCK TO TEST INDICES`* markiert und auskommentiert wird, verwenden. Wenn die Berechnung korrekt durchgeführt wird, erzeugt ein Durchlauf des Programms folgenden Farbverlauf:

In x-Richtung wächst der Rotanteil wenn `globalIdx` korrekt berechnet ist, während in y-Richtung (von oben nach unten) der Grünanteil mit `globalIdy` wächst.
Anschließend können Sie den Code, der von *`//BLOCK TO TEST INDICES`* umschlossen ist, löschen.

Aufgabe 6

(6 Punkte) Im Rahmen des Algorithmus ist es an mehreren Stellen nötig Vektoren bzw. deren Richtungsvektoren zu normalisieren, so dass seine Länge 1 ist. Implementieren sie die Normalisierung in der Funktion `normalizeDirection`, die den normalisierten Vektor als **`float4`** zurück gibt.

Aufgabe 7

(6 Punkte) Für die korrekte Beleuchtung, ist die Berechnung von Reflektionsstrahlen notwendig, wie in Abbildung 2 verdeutlicht. Der einfallende Primärstrahl **l** und der ausgehende Reflexionsstrahl **r** haben den selben Winkel bzgl. der roten Flächennormale **n**.

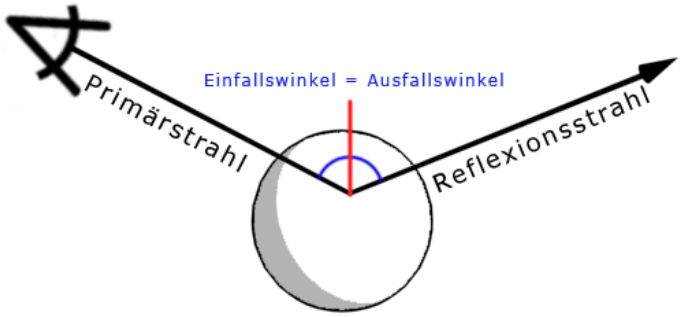


Figure 2: Reflexionen an der Oberfläche

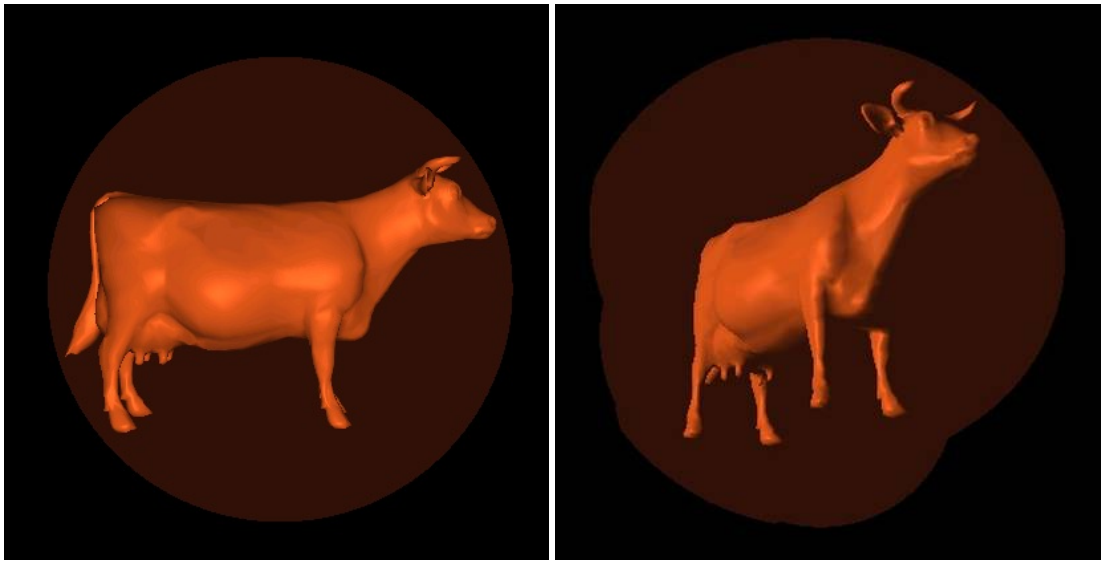
Somit lässt sich **r** aus **l** und **n** mittels folgender Formel berechnen ($\cdot$ =Skalarprodukt, $*$ = komponentenweise Multiplikation):

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{l}) * \mathbf{n} - \mathbf{l}$$

Implementieren Sie die Reflexion in der Funktion `reflect3DLA`, die den Reflektionsstrahl zurück gibt. Alle Vektoren sind als **`float4`** realisiert.

Aufgabe 8

(15 Punkte) Um die Anzahl an Schnitttests zwischen Dreiecken und Sichtstrahlen zu reduzieren, generiert der C-Code sogenannte Hüllkugeln (Bounding Spheres), die jeweils eine gewisse Menge an Dreiecken umschließen. Somit kann entweder ein vollständiges Modell wie die komplette Kuh im folgenden Bild von einer Kugel eingeschlossen sein, oder aber Teil-Meshes eines Modells von Kugeln umschlossen werden, wie im rechten Bild gezeigt.

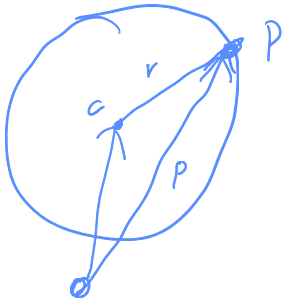


Die Funktion `__device__ bool traceRay` prüft für jeden Sichtstrahl mittels der Funktion `__device__ bool rayIntersectsSphere`, ob die Hüllkugel des Meshs getroffen wird. Nur wenn dieser Test erfolgreich ist, werden die einzelnen Dreiecke des Meshs getestet.
Der Sichtstrahl **l** ist weiterhin gegeben als:

$$\mathbf{l}(\lambda) = \mathbf{o} + \lambda\mathbf{v}$$

Eine Hüllkugel **s** hat einen Mittelpunkt **c** und Radius r. Alle Punkte **p** auf der Oberfläche von **s** erfüllen:

$$\mathbf{p} - \mathbf{c} = r^2$$



Nun setzt man **l** für **p** ein, multipliziert aus und sortiert. Daraus ergibt sich:

$$\lambda = -\beta \pm \sqrt{\beta^2 - \gamma}$$

mit:

- $\beta = 2(\mathbf{v} \cdot (\mathbf{o} - \mathbf{c}))$
- $\gamma = 2((\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c})) - r^2)$

Wenn diese Gleichung mindestens eine Lösung hat, so schneidet der Sichtstrahl die Hüllkugel, ansonsten nicht. Implementieren sie diese Vorgehensweise in der Funktion `__device__ bool rayIntersectsSphere`.

- Der erste Parameter **`float4`** `s` enthält **c** sowie r auf dem vierten Eintrag.
- Der zweite Parameter **`float4`** `r` ist die Richtung des Sichtstrahls **v**.
- Der dritte Parameter **`float4`** `o` ist der Anfangspunkt des Sichtstrahls **o**.

Aufgabe 9

(5 Punkte) Geben Sie zu Beginn der Funktion `freeCudaMemory()` allen Speicher wieder frei, der von Ihnen im Rahmen der Aufgaben auf der Grafikkarte alloziert wurde. Prüfen Sie auch wieder die Korrektheit der Ausführung und informieren Sie den Benutzer im Fehlerfall.

Prüfung der Richtigkeit der Implementierung: Das mitgelieferte Musterlösungsbild, benannt `img_solution.bmp`, befindet sich im Hauptordner (top-level) des Cuda-Projekts. Eine richtige Implementierung, die mit dem default Befehl im `batch_job.sh` ausgeführt wird, muss ein Bild ausgeben, das genau so wie das Musterlösungsbild aussieht.

Aufgabe 10

(5 Punkte) Berechnen Sie die Dimensionen der Blöcke und des Grids für den Kernel `traceLA`. Nutzen

Sie die vorgegebene Funktion `cudaOccupancyMaxPotentialBlockSize`, um eine möglichst hohe Occupancy zu erreichen. Fügen sie die Berechnung des Grids und der Blöcke unter dem Kommentar `// Use cudaOccupancyMaxPotentialBlockSize` in der Funktion `traceWithCuda` ein und überschreiben sie die hard-codierten Standardwerte der Variablen `dim3 grid` und `dim3 threads`.
Welche Werte für die Grid-Größe und die Blockgröße ermittelt Ihr Code?
Aus den Werten soll eine sinnvolle Blockgröße gesetzt werden. Das heißt, dass die gesetzte Blockgröße der vorgeschlagenen Blockgröße nah liegen soll. Gemäß der gewählten Blockgröße wird eine passende Grid-Größe berechnet, um das ganze Bild abzudecken. Außerdem soll die vorgeschlagene minimale Grid-Größe auch erfüllt werden.
Wie verhält sich seine Ausführungszeit verglichen mit den Standardwerten des Codes?

Hinweis zu Occupancy in CUDA: Bevor ein CUDA Kernel gestartet wird, muss seine Blockgröße festgelegt werden. Zum Berechnen einer geeigneten Blockgröße, die die Ausführungszeit des Kernels minimiert, eignet sich das Konzept der *Occupancy*. Die Occupancy ist das Verhältnis von der Anzahl aktiver *Warps* auf einem Multiprozessor der GPU, zu der maximal möglichen Anzahl von Warps, die auf einem Multiprozessor aktiv sein können. Beachten Sie, dass die höchste Occupancy nicht notwendigerweise die beste Performance liefert. Nichtsdestotrotz bietet sie eine gute Heuristik für die geeignete Wahl der Ausführungskonfiguration eines Kernels.