Übungen zur Vorlesung "Architektur und Programmierung von Grafik- und Koprozessoren"
Übungsblatt 7

Sommersemester 2018

7 Shader Programmierung

Aufgabe 7.1

Zur Lösung dieser Aufgabe werden Sie zum ersten Mal Code für die GPU entwickeln. Sie benötigen dazu einen PC mit Grafikkarte (die meisten Grafikkarten, z. B. auch integrierte Grafik-Chips, sollten für diese Aufgabe ausreichen) sowie einen Webbrowser, der den Standard WebGL (`https://www.khronos.org/webgl/`) unterstützt.

Für die Aufgabe verwenden Sie das Tool Shadertoy (`https://www.shadertoy.com/`), mit dem Sie ohne aufwendiges Setup Fragment Shader entwickeln können. Erzeugen Sie dazu über das Benutzer-Interface einen neuen Shader und kopieren Sie den Quelltext des zur Aufgabe gehörenden Gerüstprogramms dort hinein. Shadertoy Programme entwickelt man in einer GLSL sehr ähnlichen Sprache.

In der Vorlesung und auch den Übungen haben wir für diffuse Reflektion bisher immer das sehr einfache Lambertsche Modell angenommen. Sie sollen nun das diffuse Reflektionsmodell implementieren, das Burley als Teil von *Disneys Principled BRDF* vorschlägt [1]. Disneys Principled BRDF ist ein empirisches Beleuchtungsmodell, das in Teilen physikalisch plausibel ist, und das in Teilen dadurch motiviert ist, dass Modellierer Parameter intuitiv manipulieren können.

Das diffuse Modell von Burley nimmt eine Mikrofacettenverteilung an. Mikrofacetten reflektieren das meiste Licht entlang der Richtung $\vec{H} = \frac{\vec{h}}{||\vec{h}||}$, wobei $\vec{h} = \vec{\omega_i} + \vec{\omega_o}$ und $\vec{\omega_i}$ sowie $\vec{\omega_o}$ Einheitsvektoren zur Lichtquelle sowie in Reflektionsrichtung sind. Mikrofacetten werden über die Oberflächenrauheit $\sigma \in [0..1]$ modelliert. Burleys Modell sieht eine Fresnel Komponente vor: schaut man in flachem Winkel auf die Oberfläche, wird mehr Licht in die Betrachtungsrichtung reflektiert, als wenn man von oben auf die Oberfläche schaut. Im Modell wird die Schlick Approximation für Fresnel Reflektion verwendet, sodass sich für die reflektierte *Farbe* (Basisfarbe × Strahlstärke) ergibt:

$$f_d = \frac{\text{BaseColor}}{\pi}(1 + (F_{D90} - 1)(1 - cos\theta_i)^5)(1 + (F_{D90} - 1)(1 - cos\theta_o)^5), \quad (1)$$

wobei $F_{D90} = 0.5 + 2cos\theta_h^2\ \sigma$. θ_i und θ_o sind definiert wie in der Vorlesung und θ_h bezeichnet den Winkel zwischen der Oberflächennormale und dem *Halbvektor* $\vec{H}$.

Erweitern Sie das Gerüstprogramm, indem Sie Burleys Beleuchtungsmodell für diffuse Reflektion in der dafür vorgesehenen Funktion implementieren. Als Basisfarbe nehmen Sie RGB = $\{0.8, 0.8, 0.8\}$ an. (5 Punkte)

Aufgabe 7.2

a.)
Erläutern Sie kurz den Unterschied zwischen Fragment Shadern und Vertex Shadern. (2 Punkte)

b.)
Kopieren Sie das Gerüstprogramm, z. B. mit dem Programm `scp` unter Unix oder dem Programm `WinSCP` unter Windows, in Ihr *Home Verzeichnis* (`/home/username/`) auf dem Entwicklungssystem `vistitanv.rrz.uni-koeln.de`. Dazu müssen Sie per VPN mit dem Uni-Netz *UKLAN* verbunden sein (vgl. https://rrzk.uni-koeln.de/vpn.html). Entpacken Sie das Programm dort und übersetzen Sie es anschließend mit der Kommandozeile

```
g++ shaders.cpp -lboost_filesystem -lboost_system -lX11 -lGL -lGLEW
```

Um das Programm ausprobieren zu können, müssen Sie auf dem entfernten System ein lokales Fenster erzeugen. Setzen Sie dazu die Umgebungsvariable `DISPLAY` auf den Wert ":0.0" (mit Bash z. B. `export DISPLAY=:0.0`), um auf das Primär-Display zugreifen zu können.

Ergänzen Sie nun die Funktion `compile_shaders()`, sodass diese, wie in der Vorlesung, den Vertex und Fragment Shader kompiliert und zu einem Programm linkt. Außerdem sollen Fehlermeldungen und Warnungen, die beim Kompilieren oder Linken generiert werden, auf die Kommandozeile ausgegeben werden. Schauen Sie sich dazu die Funktionen `glGetShaderInfoLog()` und `glGetProgramInfoLog()` an. Beheben Sie mit Hilfe der Fehlermeldungen eventuelle Programmierfehler in den beiden Shadern. (3 Punkte)

Aufgabe 7.3

Grafik APIs wie DirectX, OpenGL oder Vulkan unterstützen auch eine Shader Art, die nicht im Rahmen der herkömmlichen Grafik Pipeline ausgeführt wird: *Compute Shader*. Mit Compute Shadern kann man General Purpose Programme für GPUs schreiben (GPGPU). Mit CUDA lernen wir im Laufe der Vorlesung eine mächtige Programmiersprache für GPUs kennen. Mit Hilfe von Compute Shadern hat man aber grundsätzlich auf die gleichen Ressourcen und auf das gleiche Speichermodell Zugriff, wie mit dedizierten GPGPU Tools.

Um die Aufgabe lösen zu können, kopieren Sie das Gerüstprogramm wie in Aufgabe 7.2 **a.)** auf das Entwicklungssystem. Wechseln Sie in den entpackten Ordner und führen Sie dort zunächst den Befehl

```
direnv allow .
```

aus. Dieser Befehl verursacht, dass Umgebungsvariablen für Ihr Projekt gesetzt sind, wann immer Sie sich im Projektordner oder einem Unterordner davon befinden. Legen Sie dann ein CMake Build-Verzeichnis als Unterordner des Projektverzeichnisses an und übersetzen Sie das Projekt wie gewohnt.

Das Vulkan Programm soll die `Saxpy` Operation `y[i] = a[i] * x[i] + y[i]` aus der Vorlesung auf der GPU durchführen. Teile des GPU Programms sind bereits implementiert, so auch das gesamte Host Programm, das u. a. die Input Buffer befüllt und das Ergebnis der Berechnung zurück in CPU Speicher liest.

a.)
Vulkan Compute Shader schreibt man auch in GLSL[1], übersetzt diese aber, bevor man das Vulkan Programm übersetzt, in SPIR-V Binärcode. Dazu verwenden Sie das Tool `glslangValidator`. Erweitern Sie die beigefügte CMake Datei `CMakeLists.txt`, sodass die Compute Shader im Verzeichnis `shaders/` automatisch mit dem Programm `glslangValidator` und der Option `-V` in SPIR-V Binärcode übersetzt werden, falls sich der Quellcode geändert hat. Mit der Option `-o` können Sie, wie auch bei C Compilern, der Output Datei einen alternativen Namen und Pfad geben. Schauen Sie sich dazu die Dokumentation zu den CMake Befehlen `find_program` sowie `add_custom_command` an. Wenn Sie das SPIR-V Kompilat neben den Shader Quellcode in den Ordner `shaders/` ablegen und `saxpy.comp.spv` nennen, kann es vom Vulkan Host Programm gefunden und integriert werden. (3 Punkte)

b.)
Vulkan Compute Shader werden parallel auf der GPU ausgeführt. Das dazugehörige Threading Modell lernen wir in der Vorlesung noch ausführlich kennen. Zur Lösung der Aufgabe ist aber nur nötig zu verstehen, dass jede Shader Instanz eindeutig über ihre Thread ID identifiziert werden kann, die man über die Variable `gl_GlobalInvocationID.x` abfragt. Mit dieser kann man datenparallel auf die Buffer `y`, `a` und `x` zugreifen, die im Shader als *Shader Storage Buffer Objekte* (SSBO) zur Verfügung stehen. SSBOs verhalten sich ähnlich wie die in der Vorlesung kennengelernten uniformen Variablen, sind aber u. a. für größere Datenmengen vorgesehen. Die Deklaration

```
layout(std430, binding = 1) buffer A
{
    float a[];
};
```

assoziiert ein SSBO mit *binding point* `1`. Im Shader kann auf das SSBO wie in ANSI-C per Array Notation über die Variable `a` zugegriffen werden. Mehr Informationen zu GLSL Compute Shadern und SSBOs finden Sie unter `https://www.khronos.org/opengl/wiki/Compute_Shader` und `https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object`.

Erweitern Sie den beigefügten Compute Shader `saxpy.comp`, sodass er die SAXPY Operation durchführt und das Ergebnis im Array `y` speichert. (4 Punkte)

c.)
Mit der Option `-H` gibt `glslangValidator` den SPIR-V Code in für Menschen lesbarer Form aus. Analysieren Sie den Output und hängen Sie die Instruktionen, die zum SAXPY Statement gehören, Ihrer Lösung an. Markieren Sie die Instruktionen, die unmittelbar für das Laden der Daten aus den drei Arrays, für die Multiplikation und die Addition, sowie das Speichern des Ergebnisses verantwortlich sind. (3 Punkte)

Literatur

[1] Brent Burley. Physically-Based Shading at Disney. Technical report, Walt Disney Animation Studios, 2012.

Abgabe bitte bis zum 27.06.2018, 22:00h in Ilias.

[1] oder auch in anderen Shader Sprachen wie HLSL