Exercises for the Lecture: "Architecture and Programming Models for GPUs and Coprocessors" Exercise Sheet № 8

Priv.-Doz. Dr. rer. nat. Stefan Zellmann

SS 2021

8 GPU Programming

Ex. 8.1

To solve this exercise you need a computer with a graphics card. An integrated GPU should suffice to solve this exercise though. You will also need a web browser that supports the WebGL standard (`https://www.khronos.org/webgl/`). A reasonably new version of Mozilla Firefox (`https://www.mozilla.org/en-US/firefox/new/`) for example should be sufficient.

You're going to solve this exercise using Shadertoy (`https://www.shadertoy.com/`), which allows you to develop fragment shaders without any tedious setup of a graphics application on the host. You just have to create a new shader via the graphical user interface, and copy the source code from the skeleton program to the text editor control. Shadertoy itself uses a language that is inspired by GLSL—just have a look at the skeleton program to familiarize yourself, understanding the language constructs used by the latter should provide you with enough context to solve this exercise.

It's your task to replace the diffuse Lambertian shading model that we also know from the lecture with the diffuse part of Disney's *principled BRDF* [**?**]. The principled BRDF adopts ideas from physically based rendering, but focuses particularly on artist-friendliness and parameters that can be intuitively tuned.

The diffuse part of the principled BRDF uses a microfacet distribution. Microfacets make the light preferentially reflect in the direction $\vec{H} = \frac{\vec{h}}{||\vec{h}||}$, where $\vec{h} = \vec{\omega_i} + \vec{\omega_o}$ and where $\vec{\omega_i}$ and $\vec{\omega_o}$ are unit vectors pointing towards the light source and towards the viewing position, respectively. The *roughness* of the surface due to the microfacet distribution can be controlled via the parameter $\sigma \in [0..1]$. The model also incorporates a Fresnel component so that more light is reflected at grazing angles. The model uses the "Schlick approximation" for Fresnel reflection to compute the reflected color (approximated as "base color × radiance"):

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

where $F_{D90} = 0.5 + 2cos\theta_h^2\ \sigma$. θ_i and θ_o are defined in accordance to the lecture; θ_h denotes the angle between the surface normal and the *half vector* $\vec{H}$. You are advised to familiarize yourself with Burley's paper to learn more about the principled BRDF, it's a reflection model that is becoming predominant these days, both for offline as well as for interactive rendering.

It is your task to implement the diffuse component of the principled BRDF using the designated function in the skeleton program. As base color, you can just use RGB = $\{0.8, 0.8, 0.8\}$.

Ex. 8.2

Discuss the differences between the threading model of NVIDIA GPUs vs. that of contemporary multi-core CPUs. Structure your discussion based on the following questions:

1. Which functional unit (hardware or software) is responsible for scheduling threads on the respective architecture?
2. Are threads potentially executed in lockstep? If so, on which architecture? What's the difference to SIMD?
3. GPUs manage many threads at once, and often there are more threads "in flight" than the GPU has cores and SMs. In that case the GPU needs to assign time slices to threads that perform some calculation. How are those time slices assigned?
4. Context switches on GPUs are particularly fast, why is this so?

Ex. 8.3

In the following you will find a `CUDA` program that uses `thrust` for the data movement between host and device. You are supposed to restructure the program so that it retains its functionality but uses runtime API functions only. To replace `thrust::host_vector<T>` you are allowed to use `std::vector<T>` from the standard template library. Note that CUDA/C++ pseudo code is fine here: the program is not required to successfully compile and run (but it *should* be easy to implement the `__global__` function `kernel`, configure the kernel call appropriately, and check if your program does what it's supposed to do).

```
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>

int main() {
    thrust::host_vector<float> host_data(100);
    for (int i = 0; i < 100; ++i) {
        host_data[i] = i;
    }

    thrust::device_vector<float> device_data(host_data);

    /* Just assume that blockSize and gridSize are set here.
     We also don't care what the function kernel() does - only
     that it takes a device pointer as its only argument! */
    kernel<<<blockSize, gridSize>>>(
        thrust::raw_pointer_cast(device_data.data()));

    thrust::host_vector<float> result(device_data);
}
```

The exercise sheet will be discussed on July 15, 2021.