Week 2

The ray tracers mentioned the first week both use multicore CPU processing for the ray tracing. This week we will investigate how a graphical processing unit (GPU) is used to accelerate rendering. Unless you did not work with shaders nor use OptiX in a previous course, we expect you to do the exercises with two different approaches. First using OptiX, a GPU ray tracer distributed by NVIDIA, then using shader-based rasterization (frameworks are available for both approaches). Using the GPU for ray tracing, you can do all the same things as with traditional ray tracing, only the GPU gives a significant rendering speed-up. Rasterization is even faster, but it complicates the implementation of more advanced rendering effects (such as shadows, reflections, refractions, and likewise).

Learning Objectives

- Compare real-time and off-line rendering.
- Use the GPU to accelerate physically based rendering.
- Write programs (shaders/kernels) that run on the GPU.
- Explain possible ways of exploiting interoperability between rasterization and ray tracing.

GPU Accelerated Rendering

The purpose is now to repeat the work from Week 1, only this time using the GPU. Feel free to use your own rasterization program and/or your own GPU ray tracer. Otherwise, use the `realtime` project in the course framework for the rasterization approach. Use the separate `render_OptiX` framework for the OptiX approach, and notice that you must use CMake to generate a Visual Studio solution for an OptiX program. Work through the following exercises using each approach.[1]

- Load the Stanford bunny (`bunny.obj`) into your real-time shading program. (Take some time to understand what you can do with the keyboard and command line. Rasterization: investigating the `keyboard` function in `realtime.cpp`. OptiX: compile and run the generated executable with the command line option `-h`.)
- Implement a shader that, using the same directional light, produces the same output as you got in Week 1 without the shadow rays turned on. (Rasterization: implement the `set_light` function in `Directional.cpp` and the fragment shader in `Lambertian.cpp`. OptiX: implement the closest hit program `directional_shader` in `directional_shader.cu`.) Save the resulting image both with and without anti-aliasing.
- Load the Cornell box (`CornellBox.obj`) and the blocks inside it (`CornellBlocks.obj`). Implement a shader that handles the area light correctly. It should produce the same output as in Week 1 except for the shadows. (Rasterization: implement the `set_light` function in `AreaLight.cpp` and improve the fragment shader in `Lambertian.cpp` if necessary. OptiX: implement the `sample_center` function in `AreaLight.h` and the closest hit program `arealight_shader` in `arealight_shader.cu`.)
- Finally, implement shadows. To reduce the workload, shadows need only be implemented for the area light in rasterization, as a different approach would be required for the directional light. (Rasterization: use omnidirectional shadow mapping [Gerasimov 2004, see reference below]. Implement the distance fragment shader used for capturing the cube map and the fragment shader that uses the cube map in `Shadow.cpp`. OptiX: implement the closest hit program `shadow_shader` in

[1] If both OptiX and shaders are new to you, state this in your hand-in and we will accept if you only try one of the approaches.

`shadow_shader.cu` to get shadows with the directional light. Update the closest hit program `arealight_shader` to get shadows with an area light source.) Again save the resulting images.

- Make a table that compares render times (also to the renderings of the first week). Discuss advantages and disadvantages of the two different methods. Propose ways of exploiting interoperability.

Week 2 Deliverables

Bunny images and Cornell box images that compare as closely as possible to the results from Week 1. Include relevant code as well as table and text from the last bullet point. Please copy everything into a document and upload at CampusNet under Assignments.

Reading Material

The curriculum for Week 2 is

- Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010) 29*(4), Article 66, July 2010.
- Bærentzen, J. A. *Lecture Notes on Real-Time Graphics*, DTU Informatics, 2009. http://www2.imm.dtu.dk/~jab/rasterization-pipeline.pdf
- Gerasimov, P. S.. Omnidirectional Shadow Mapping. In, R. Fernando (editor), *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, Chapter 12, Addison-Wesley, 2004. Available online at http://http.developer.nvidia.com/GPUGems/gpugems_ch12.html

Additional resources:

- NVIDIA. NVIDIA OptiX Ray Tracing Engine: Quickstart Guide, Version 2.1, December 2010.
- Chen, J. X. OpenGL Shading Language. In *Guide to Graphics Software Tools*, Chapter 9, Springer, 2009. http://www.springerlink.com.globalproxy.cvt.dk/content/p7544x23t1h6l101/
- GLSL quick reference: http://mew.cx/glsl_quickref.pdf
- GLSL tutorial: http://www.lighthouse3d.com/opengl/glsl/
- OpenGL and GLSL manuals: http://www.opengl.org/documentation/