02562 Rendering - Introduction DTU Compute

Worksheet 7

Graphics is the visual interface between the user and the computer. The display driver thus prefers that the graphics processing unit (GPU) is responsive. Stalling the GPU to compute one expensive frame might lead to a timeout and recovery action in the display driver, which oftentimes means that your rendering program is not allowed to do what you asked it to do. When rendering using the GPU, we therefore prefer many frames of low computational cost over one frame very costly to compute. Progressive unidirectional path tracing is an algorithm for realistic image synthesis that can adhere to these constraints. We will use it in this worksheet to add indirect illumination to our renderings.

Learning Objectives

- Generate pseudo-random numbers on the GPU.
- Do progressive updating using a ping-pong rendering scheme.
- Use Monte Carlo integration for solving the rendering equation.
- Implement basic progressive unidirectional path tracing.

Progressive Path Tracing

Progressive rendering is useful in the sense that we do not need to start over if the initial number of samples was insufficient to get the desired image quality. With a progressive technique, we can keep improving the image until the desired quality is obtained. We can also quickly spot a problem by inspecting the rendering as it progressively improves.

1. In this worksheet, we will render the Cornell box with blocks (`CornellBoxWithBlocks.obj`, comment out the spheres). Our objective is to enable progressive random sampling of the pixels. Create two textures for a ping-pong rendering scheme. Ping-pong rendering means that the previous rendering result is used to render the next. To accomplish this, set up one texture to be a render target and a copy source and one to be a texture binding and a copy destination. After rendering to the render target, copy the result to the texture bound for shader look-ups. Upload the frame number as well as the rendering resolution (canvas width and height) to WGSL as uniform variables. Use these uniform variables to seed a pseudo-random number generator and to get a random offset within each pixel. Implement progressive updating using the result from the previous frame such that you always store the average of all the frames as the next result. Create a button or a checkbox for switching progressive updating on/off. Your solution works when the user can use progressive updating until the image has nice anti-aliased edges.
2. Pass a pointer to the seed for the pseudo-random number generator (`t`) from the main function over your shader for Lambertian materials to the function for sampling an area light. Then change the simplified area light sampling from earlier to Monte Carlo sampling of a random position on the surface of the light source (first sample a random triangle index, then sample a random position on that triangle). As a result you should obtain an image of the Cornell box where the blocks cast soft shadows.
3. Update your `HitInfo` struct to include an `emit` flag and an RGB `throughput` factor. The emit flag is for combining different methods for evaluating direct and indirect illumination. The throughput factor is for weighting the result returned from later in the path. Implement sampling of a cosine-weighted hemisphere and add sampling of indirect illumination to your shader for Lambertian materials. Use these techniques to render the Cornell box with blocks including path traced indirect illumination. Test

the effect of rendering with a blue versus a black background. Create a button or a checkbox that switches the blue background on/off. Remember to reset the frame counter back to zero after making a change that requires re-rendering of the image.

Reading Material

The curriculum for Worksheet 7 is (34 pages, 9 of these pages are repetition)

- **B** Sections 2.10-2.12. *Discrete Probability*, *Continuous Probability*, and *Monte Carlo Integration*.
- **B** Chapter 13. *Sampling*.
- **B** Sections 14.8 and 14.10. *Transport Equation* and *Monte Carlo Ray Tracing*.