T-111.5310 Vuorovaikutteisen tietokonegrafiikan jatkokurssi

Assignment 4: Path Tracing / Rendering Competition

Due Wednesday May 22 at 23:59.

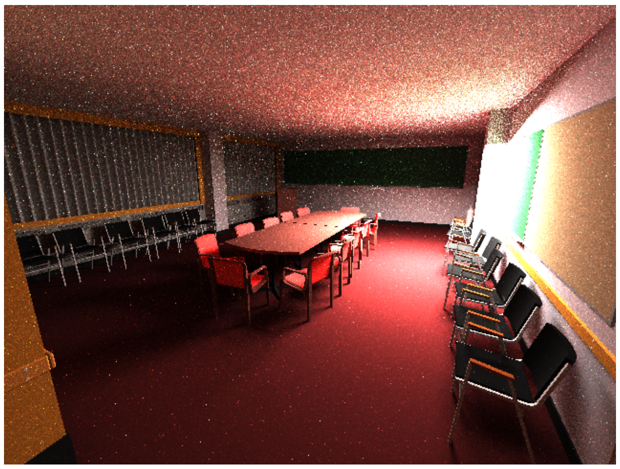
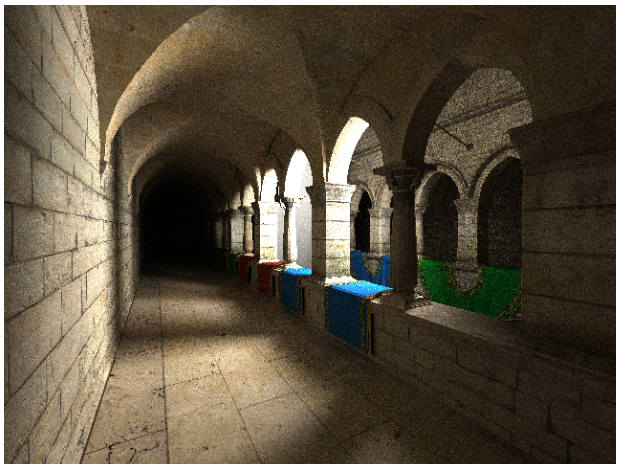
1 Introduction

The mandatory requirements for full credit in this fourth, final assignment are light: you will implement a simple forward Monte Carlo path tracer that supports diffuse materials and computes an unbiased image using Russian roulette.

The basic requirements are:

1. Cast rays into the scene and evaluate the direct light and shadows. (3p)
2. Compute indirect light using an arbitrary (fixed number) of bounces. (5p)
3. Turn your solution into an unbiased one by implementing Russian roulette path termination. (2p)

Here are the images generated by the example solution, after running for a couple of minutes:

The example solution is pretty braindead: it does not use low discrepancy sampling or anything.

1.1 Rendering Competition

This round ends in a *voluntary rendering competition*. The point of the competition is to draw on everything you've learned this spring — not just this final assignment — and produce a few pictures that are as pretty and technically advanced as you can possibly make them. The winner will be chosen by Luca Fascione of Weta Digital, and there will be a prize that we've already named in class[1].

The competition is supposed to be a positive, fun thing. In particular, it's not necessary to participate to earn full credit. However, we strongly encourage you to! Think of it as a basis for your own portfolio you can use later when applying for (summer) jobs and the like.

We'll get together to announce the winners and celebrate your achievements in the end of May or beginning of June (to be announced later). Everyone will be welcome. Maybe we'll get the lab to sponsor something to eat, too!

2 Getting Started

2.1 Sample solution

Again, the package contains a sample solution executable. The user interface is like in the previous assignments. The default mode is an OpenGL-rendered preview, where you can align the camera and the light source. The rendering is initiated by pressing the `Path trace mode (INSERT)` button. The renderer is progressive: you will instantly see a noisy initial result, and the quality will gradually improve over time as more paths are traced. To quit rendering, press the `Path trace mode (INSERT)` button again. Note also the buttons for setting the light source position and toggling the Russian roulette, and the sliders for the number of indirect bounces and the light source size.

The starter package contains three scenes: the Cornell box, the conference room, and the "Crytek Sponza" (Web link). Use the keys 1, 2, and 3 to switch between them. You can save your own settings by pressing Alt+number key and retrieve the saved parameters by the number key alone.

2.2 Starter code

The starter code provides the usual OpenGL preview mode and the general infrastructure for running the progressive path tracing process. However, it does not contain the path tracing logic itself. The relevant source code file is `Renderer.cpp`; read through it to get an understanding of how the rendering is initiated and run.

You will need to integrate your ray tracer from the first assignment again. Alternatively, you can use the ready-made raytracer library, but this will again cost you three points. To use it, choose the `Release_RTLIB` or `Debug_RTLIB` configuration from the Visual Studio configuration menu.

You might want to consider using the ready-made library if your ray tracer is very slow, despite the point penalty. The path tracer casts quite a lot of rays, so instead of spending hours waiting for

[1]NVIDIA employees in the class are not eligible for the prize.

simple images to render, it might be less frustrating to just implement a couple of extra features to make up for the lost points. Also, this time, if you hand over a solution that is 100x slower than the reference, we will consider you to have failed this requirement.

3 Path tracer (3+5+2 p)

The implementation is contained mainly in the function `Renderer::pathTraceScanline()`. It loops through the pixels in a single scanline of the image, and for each of them performs the path tracing operation once. The progressive renderer takes care of repeatedly looping through all the scanlines and averaging the results; the image quality will improve over time, as the different random choices will average towards the true solution.

The path tracing logic closely follows the pseudocode presented in the lecture slides; refer to them for details on the algorithm and the theory behind it. Below we will briefly review the overall structure of the algorithm (but not all of the critical details!), and a few practical implementation matters.

The overall procedure is as follows.

- Generate the initial ray direction through the desired pixel and cast it. You will need to generate a ray direction in the NDC or camera eye space, and then transform it into the world space using the appropriate camera pose matrices. Assignment 1 might contain useful code for this purpose. Note that you will have to randomize the position inside the pixel as well in order to perform antialiasing.
- If the ray misses the scene, just set the pixel black. Otherwise enter the path tracing stage.
- Despite the recursive nature of the path tracing algorithm, it is simpler to implement the procedure as a loop rather than using recursive function calls. Throughout the loop, keep track of the radiance returned by the path, the probability density of generating it, and the current number of bounces. Each iteration of the loop will handle the current intersection, after which it either terminates or generates a single indirect sampling ray for the next iteration. The loop internals are as follows:
  - Determine the hit point coordinates, normal, barycentric coordinates and the surface color. Like in the previous assignments, the color is determined from the surface diffuse color or the texture if it is present. Implement the color fetching in `Renderer::albedo()`.
  - If this was the first ray from the camera and the hit point is the actual light source, add the emission to the radiance returned by the path. Note that this doesn't happen in our code, unless you change your scene handler to have polygons emit light; our light source resides in its own separate class and is not part of the scene geometry proper.
  - Draw a point from the light source surface, trace a shadow ray, and add the appropriate contribution to the radiance returned by the path. Be careful with the probabilities.

  - At this point you should determine if the path should be terminated. There are two alternative modes: a fixed number of bounces, or Russian roulette started after a given number of bounces. The mode and the numbers are set by the user, and passed to the function in the variable `ctx.m_bounces`. The variable has has a negative value if Russian roulette is enabled, and a positive value if not. So for example if the value is -3, this means that the first 3 iterations will always cast continuation rays, but after that the loop will be terminated with probability 50% on each iteration, with the probabilities compensated accordingly. Play around with the example solution to familiarize yourself with the workings of these settings.
  - Indirect ray casting. If the loop did not terminate in the previous step, generate a random cosine-weigted direction (recall that we are assuming a perfectly diffuse material) using the familiar procedure from the previous assignments. Cast the ray and pass it to the next iteration of the loop (but terminate if it misses the scene.) Remember to give the ray a reasonable maximum length.
- After the path has terminated, write the radiance to the corresponding pixel.

Again, this is only a description of the general structure of the algorithm. It does not contain the sufficient step-by-step details on handling the probabilities, radiometric quantities and other matters. See the lecture slides for the details. It's what we meant when we said in the beginning we won't have the trainer wheels on all the time ☺

4 Extra credit

This list is not exhaustive: the requirements are short this time on purpose to allow you to come up with your own ideas! Do contact us beforehand, though, so we can agree on the scoring.

4.1 Medium

- (6p) Implement Photon Mapping with final gathering. See Henrik and Per's SIGGRAPH 2007 course notes and the more recent 2012 notes for help!
- (4p) Implement a glossy reflection model in addition to pure diffuse. Cook-Torrance, Torrance-Sparrow, or Blinn-Phong are good choices. Note that you will have to implement importance sampling of the BRDF (i.e. the outgoing angle of the continuation ray) as well in order to not get terrible amounts of noise! You can play around with the .MTL file that accompanies the Crytek Sponza model to enable specularity on some select materials. (We know it's a little bit of a pain using this simple file format — you will most likely have to resort to some form of hardcoding, e.g. read the BRDF type off the material name or something.)

- (3p) BRDFs part 2. If you want to do the above really right, see this paper for a state-of-the-art model and parameters that have been fit to real materials. Use some of the measured materials in your renderings.
- (4p) If you implement glossy BRDFs, implement Multiple Importance Sampling (MIS) between the light sample and BRDF sample (continuation ray), as described in class. See Veach's paper for details.
- (5p) Implement Irradiance Caching for the diffuse part of the GI solution. See the SIGGRAPH 2008 course notes for details!
- (4p) Implement single scattering in a participating medium using ray marching. This means cool volumetric shadows like in the video found on this page. See the paper for an introduction and other references!

4.2 Hard

- (5-15p) Implement bidirectional path tracing (BDPT) with Multiple Importance Sampling. Again, see Veach's MIS paper (above) or his thesis. You should convince us that your solution converges to the same solution as the forward tracer, just faster! Your total points depend on the features you build in: if you include glossy materials and other difficult cases like caustics — which is where bidirectional methods really shine — you will receive upwards of 15 points. For a real tour de force, implement both photon mapping and BDPT so that you can compare how they do on caustics.
- (5p) Implement the Kelemen variant of Metropolis Light Transport.

4.3 Extremely Hard

These really are hard. You'll receive a special honorary mention if you try!

- (10+p) Implement Veach-style Metropolis Light Transport, including at least the bidirectional and lens mutators. For 10 points, you need the basic algorithm working in a diffuse environment. Other bells and whistles like glossy materials (and proper handling of perfectly specular surfaces) will earn you more. We will be generous.
- (10p) Implement Jakob's Manifold Exploration on top of Metropolis Light Transport.

5 Submission Instructions

5.1 Regular package

You are to write a README.txt that answers the following questions:

- Your name and student number at the beginning of the document.
- Indicate if you used the ready-made ray tracing library.
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.
- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.
- Did you do any of the extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what how you did it.
- Got any comments about this assignment that you'd like to share?

You should create a single .zip archive containing:

- Your whole Visual Studio project.
- The README.txt file.
- Screenshots of the scenes rendered in the startup position using your path tracer.

Submit it online in Optima by Wednesday May 22 at 23:59.

5.2 Competition entry

You have two extra days, until Friday May 24 23:59, to submit your rendering competition entry.

You should prepare a web page in a directory that contains `index.html` and all the images and other content referenced in the HTML document under that directory using relative paths. Zip this directory up, put it somewhere where it's publicly accessible, and email a link to Miika (cc Jaakko). (Our emails are `first.last@aalto.fi`.)

The above explanation may be unclear, but the point is that when we unzip the package and point our web browser to index.html, the page should display correctly.

Good luck!