This handout was compiled on 2025-04-10 15:59:09+03:00.

CS-E5520 Advanced Computer Graphics, Spring 2025

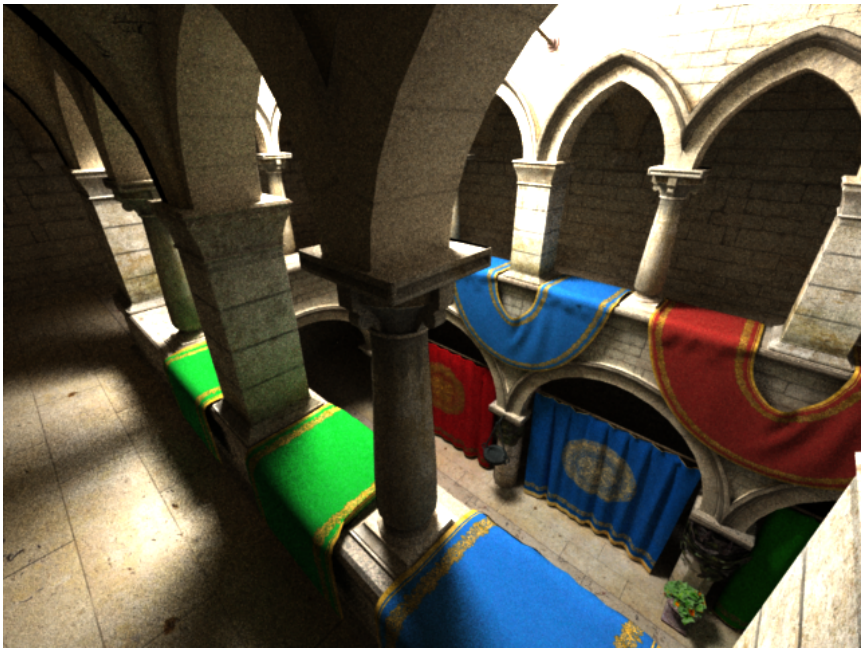
Lehtinen / Härkönen, Timonen

Assignment 3: Path Tracing / Rendering Competition

Due May 11, 2025 at 23:59

The requirements granting points 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 course concludes in a rendering competition where you can unleash all your technical and creative talents to create compelling images. At least submitting an entry in the competition is mandatory, but we hope you put in the effort to create something you can be proud of.



Bidirectional path tracer with MIS and specular BRDF.
Peter Hedman, Spring 2013.

Requirements (maximum 10p) ***on top of which you can do extra credit***

1. **Integrate your raytracer (1p)**
2. **Direct light & shadows (2p)**
3. **Bounced indirect light (5p)**
4. **Russian roulette (2p)**

5. **Rendering competition (mandatory for getting >5p in Assignment 3)**

1 Getting started

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 (ENTER)` 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 (ENTER)` 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 a number of scenes already familiar to you, as well as "Crytek Sponza". Use the keys 1-6 to switch between them; 5 and 6 are different positions in Crytek Sponza. As usual, you can save your own settings by pressing Alt+number key and retrieve the saved parameters by the number key alone, but keep the 1-6 settings at their default values since you are required to produce those particular images.

The starter code provides the usual OpenGL preview mode and the general infrastructure for running the progressive path tracing process. The rendering progresses in blocks, starting from the left upper corner. The rendering process is multithreaded, and each block is given to a single thread. Multithreading is enabled by default and you may need to disable it for easier debugging. The relevant source file is `PathTraceRenderer.cpp`; read through it to get an understanding of how the rendering is initiated and run.

The sample solution contains three rendering modes. The first one, stratified sampling with box filter, takes 3×3 samples per pixel and updates the result to the same pixel. The second one, Sobol with box filter, takes sampling positions and area light samples from a Sobol sequence inside the block, and updates the result to the nearest pixel. The third one, Sobol with Gaussian filter, uses a Sobol sequence for the sample positions and area light samples and adds the contribution to the nearest 3×3 pixels according to a 2D Gaussian kernel. In one pass of rendering a given block, the number of paths traced is 9 times the number of pixels in the block.

2 Detailed instructions

Integrating your raytracer (1p)

Like before, the starter code initially uses our raytracing library. Replace it with your own raytracer code from previous assignments to gain one point. You will also need your area light code. Note that the started code now implements an important writeTriangles method which is necessary to add the light as an intersectable to the ray tracing routine. **For the light, port only the AreaLight::sample method that you wrote in the last assignment.**

Rendering the images on this assignment will take lots of time and computing power, so if your raytracer is very slow, you should use the reference implementation. A raytracer that is 10× slower than the reference will not give you a point for this requirement since it's essentially unusable.

Direct light & shadows (2p)
Bounced indirect light (5p)
Russian roulette (2p)

The implementation lies mainly in the function `PathTraceRenderer::pathTraceBlock()`. It casts paths into the scene and add their contributions to the result image. It may, for example, loop through the pixels of the block in scanline order, or it may use a low discrepancy sequence for selecting the sampling positions. The progressive renderer takes care of repeatedly looping through all of the blocks and averaging the results; the image quality will improve over time, as the different random choices will average towards the correct 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 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 may need to randomize the position inside the pixel as well in order to perform antialiasing.
- If the ray misses the scene, just output zero to the pixel. Otherwise enter the path tracing stage.
- Despite the recursive nature of the path tracing algorithm, it may be 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 `PathTraceRenderer::getTextureParameters()`. Note that you will have to apply gamma correction to diffuse values read from a texture. This means that you will have to raise all components of diffuse to power 2.2. This is necessary, because images are gamma encoded in order

to optimize bit usage. To get smooth normals, interpolate the vertex normals to the hit position and normalize it. The normal is interpolated according to the hit barycentrics similarly to texture coordinates.

- 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. The starter code should already implement this for you.
- 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. This is what we called "next event estimation" at the lectures.
- 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 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 some probability on each iteration, with the ray contribution compensated accordingly. The optimal value depends on the scene, but the reference solution uses a fixed termination probability of 20 %. 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-weighted 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.)
- If you happen to hit the light by chance with the indirect ray, you can treat it as completely absorbing and end the path. We recommend to implement the brute force path tracing variant and Multiple Importance Sampling as extra credit where you will need to account for this case properly.

- After the path has terminated, update 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 training wheels on all the time ☺

Rendering competition (mandatory for getting >5p)

The point of the rendering competition is to draw on everything you've learned this spring — not just this final assignment — and render two or more compelling images of a scene. Apply advanced techniques, be creative and amaze us. Think of it as a basis

for your own portfolio you can use later when applying for jobs and the like. We will be lenient in awarding points for participation, *but simply submitting images of the given reference scenes with absolutely no bells and whistles on top of the requirements will not yield any points*.

For assets, you can use any freely available scenes, models and textures (attributing the sources correctly, of course) or make your own. Any code you use for the competition renderings must be included in the assignment submission.

While participating is sufficient, we strongly encourage you to pull all the stops on this one! For inspiration, you can check out for example these student works from EPFL, Switzerland.

For your rendering competition entry, include a separate folder in your submission zip archive that contains the following:

- A PDF document with one page per image. The page should contain the image itself, plus a short paragraph of text which lists key techniques that were used to create the image, and sources of significant custom assets you used.
- The same images as stand-alone files in PNG format.
- If you submit videos, we suggest uploading them to YouTube or Vimeo and include the links in your document. It's also OK to submit videos in H.264 format inside the MP4 container. FFMPEG can be used for generating such videos. It's recommended you use one of the High profiles, see here.

3 Extra credit

You can also implement any relevant extras from earlier assignments, for example QMC sampling, texture filtering, pixel filtering, alpha and specular textures. (Of course, if you received credit for them earlier, they won't be counted this time.)

The extra credit suggestions below focus mostly on light transport algorithms. We are very much open to your implementations of e.g. noise reduction filters for interactive path tracers; as always, don't be limited by the suggestions here! We'll be generous with points, but if you want to attempt something big that's not listed, it's of course a good idea to contact us first.

3.1 Recommended

- (1p) Brute force path tracing. Instead of using next event estimation like in the requirements, engineer your path tracing loop to wait for light intersections when you only sample using the (diffuse cosine) BRDF. You will see that this is in many cases a bad idea, even though the images should eventually converge to the same result. This also serves a basis for Multiple Importance Sampling (Medium extra), where you combine the samples of the two samplers in a clever way. Use the GUI-elements to toggle this on/off. The reference has an implementation for this.

- (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 glossiness 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.
- (1-3p) Properly uncorrelated QMC samples. Draw high-dimensional samples from a low discrepancy sequence, making sure that each dimension of your integral function has its corresponding dimension in the sample. This means that you sould not use any kind of stratification for anti-aliasing samples.

  A concise checklist for what should happen:

  - AA samples (1st and 2nd dimension) should be drawn over whole image, not just single pixels or tiles
  - 1st bounce draws from 3rd and 4th dimensions
  - 2nd bounce gets dimensions 5 and 6
  - and so on

  Two points for a working sampler (if you haven't received points in an earlier round) and one point if the samples are uncorrelated.
- (3p) Implement tangent space normal mapping. Normal mapping adds high precision detail to the scene with little additional cost and therefore allows you to render much more complex-appearing scenes in equal time. Crytek Sponza has tangent space normal maps which could be utilized. The reference solution has a working implementation, you can use it to make sure your solution is correct. The Cornell Chesterfield scene is also useful for quick testing, and it has large shapes so you can easily see mistakes in the tangent space.
- (1p+) Implement a system that supports light-emitting triangles in the scene. You should gather all triangles whose emission `Material::emission` is non-zero into a list that can later be used for light sampling. Assignment 1 starter code had a method for doing this that you can use. You also need to take a triangle's emission into account when computing the radiance of camera rays.

  For one point, draw a light triangle from the list using a uniform distribution. Two points if you sample light triangles according to their emissive power `emission*area`. If you support texture-based emission, further points will be awarded. A material's emission value is determined by a scene's mtl file, given as a token Ke followed by three floats, the rgb value of the emission. Cornell and Conference

already have emissive materials that you can use without having to modify the mtl files.

- (1p+) Implement some informative visualizer for your path tracer. One point for a simple implementation that shows the path segments and some additional data for each vertex, eg. normals. More points will be awarded for more sophisticated visualizations, such as showing how the BVH traversal progresses step by step for each raycast.

  The starter code provides a simple framework for generating and rendering visualization data for traced paths. By pressing `Ctrl`, you can fire a number of debug paths originating from the current mouse position. For these paths, the `debugVis` flag will be enabled, and you can push visualization data into the `std::vector` the `tracePath` function receives as an argument. The `PathVisualizationNode` class has the following definition:

```
class PathVisualizationNode
{
        std::vector<PathVisualizationLabel> labels;
        std::vector<PathVisualizationLine> lines;
};
```

  Each node in the visualization vector can contain as many lines and labels as you want. Labels are strings that will get rendered on the screen in the given world-space position. The starter code shows the diffuse color of the hit position as a label. Lines will get rendered as line segments between two points, given as the start and stop vectors. You can also give lines customized colors that you can use to color-code different variables. If you push nodes into the visualization vector in the same order your code gets executed in, you can render step-by-step visualizations of the progress of your path or ray tracing algorithms. You can see the reference executable for an example of this.

3.2 Easy

- (3+p) Integrate your path tracer code to your radiosity solver from Assignment 2. Now, instead of iteratively computing subsequent bounces by using radiosity results from previous bounce like you did before, compute the vertex radiosities directly using your path tracer. That is, fire a number of full paths for each vertex in the scene and average f/p to get the vertex radiosity.
- (2+p) Support rendering multiple objects in a single scene. For more details see the round 1 handout.
- (3+p) Hierarchical light source importance sampling. Create a data structure for efficient light source sampling in the presence many light sources (some of which might be far away). Additional points for considering visibility information in addition to spatial proximity. Can be a separate structure or part of the BVH. Goes nicely with the emissive triangles extra.

Note: If you do this extra, you **must** include a scene with several light sources that demonstrates the improvement.

3.3 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 Multiple Importance Sampling (MIS) between the light sample and BRDF sample (continuation ray), as described in class. See Veach's paper for details. 2p for working diffuse materials (MIS won't help much with those) and another 2p with glossy materials (here you will see the most benefit!).
- (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!
- (6+p) Implement Gradient-Domain Path Tracing for diffuse scenes. This means that instead of estimating pixel brightnesses directly, you estimate also the *changes* between pixels, i.e., the image gradient, by using pairs of correlated paths. Using the throughput image (or the ordinary brightness image) and the generated gradient image, you then perform "screened Poisson reconstruction" by solving a special kind of linear system of equations to get an image that often has far better quality without having to do too much extra work! Note that for purely diffuse scenes you can always reconnect paths after the first bounce, which simplifies things a lot. You can either use the provided reconstruction algorithm or implement your own for an additional 5p. Another additional 5p if you support glossy materials and only reconnect the offset paths at diffuse path vertices.

3.4 Hard

- (5p) Implement progressive photon mapping (link). More points for the modern adaptive version (link).
- (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.

3.5 Medium-Very Hard: Metropolis Light Transport

There are numerous different variations of Metropolis light transport for solving global illumination even in very difficult scenes. Some are not too difficult implement, and improve convergence speeds dramatically. For the listed 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.

- (5p) Kelemen-style MLT for unidirectional path tracing. It is not overly hard to convert your standard path tracer to a Kelemen MLT version; the main difficulty lies in code organization related to the path sampler and PDF computation.
- (5p) Kelemen MLT for bidirectional path tracing. You should only attempt this if you have a working bidirectional path tracer at hand first, of course! As with the above, the conversion is not necessarily overly hard, but does require you to think things through carefully.
- (10p) Build Multiplexed Metropolis Light Transport on top of the previous bidir-Kelemen method.
- (20p) On top of Kelemen MLT (unidirectional suffices), implement the Anisotropic Gaussian Mutations for Metropolis Light Transport through Hessian-Hamiltonian Dynamics algorithm. It is, in terms of light transport computations, simpler to implement than Veach MLT. It relies on analytic derivatives of the path throughput function to derive very smart mutation strategy for Kelemen-style MLT. Computing the derivatives efficiently using automatic differentiation (AD) is the main challenge of this method. Be sure to watch the paper talk video on the webpage even if you don't attempt this.
- (15p) Veach-style MLT, including at least the bidirectional and lens mutators. You should only attempt this if you have a working bidirectional path tracer.
- (7p) Energy Redistribution Path Tracing – this is Veach-style MLT with many short chains and a special kind of seeding procedure.
- (10p) On top of Veach MLT, implement Gradient-domain Metropolis Light Transport (Lehtinen et al., 2013!). This is the (strangely enough, much more complex) predecessor to gradient-domain path tracing algorithm.
- (20p) On top of Veach MLT, implement Jakob's Manifold Exploration for supporting scenes with very hard specular interactions. We recommend watching the talk video on the project webpage regardless of whether you try implementing this algorithm or not.

4 Submission

- Make sure your code compiles and runs both in Release and Debug modes on Windows with Visual Studio 2022, preferably in the VDI VMs. You may develop on your own platform, but the TAs will grade the solutions on Windows.
- Comment out any functionality that is so buggy it would prevent us seeing the good parts. Use the `reference.exe` to your advantage: does your solution look the same? Remember that crashing code in Debug mode is a sign of problems.
- Check that your `README.txt` (which you hopefully have been updating throughout your work) accurately describes the final state of your code. Fill in whatever else is missing, including any feedback you want to share. We were not kidding when we said we prefer brutally honest feedback.
- Package all your code, project and solution files required to build your submission, `README.txt` and any screenshots, logs or other files you want to share into a ZIP archive. *Do not* include your `build` subdirectory or the reference executable. Beware of large hidden `.git`-folders if you use version control - those should not be included.
- Sanity check: look inside your ZIP archive. Are the files there? Test and see that it actually works: unpack the archive into another folder and see that you can still compile and run the code.
- If you experience a system failure that leaves you unable to submit your work to MyCourses on time, prepare the submission package as usual, then compute its MD5 hash, and email the hash to us at `cs-e5520@aalto.fi` before the deadline. Then make the package that matches the MD5 hash available to us e.g. using `filesender.funet.fi`. *This is only to be done when truly necessary; it's extra work for the TAs.*

Submit your archive in MyCourses in "Assignment 3" by May 11, 2025 at 23:59

5 Appendix: Reference images

These example images have been rendered with the example binary's filtering mode set to "Sobol with Gaussian". Each image was rendered for approximately 2.5 minutes on an Intel Core i7 quadcore CPU. Rendered with an older version of the reference that does not include the light in the geometry, hence it is not visible.