ECSE 446/546: Realistic/Advanced Image SynthesisAssignment 5: Global Illumination

Due Monday, November 19th, 2018 at 11:59pm EST on myCourses
20% (ECSE 446 / COMP 598) *or* **15%** (ECSE 546)

You will build path tracers capable of rendering realistic images with surface global illumination effects in this assignment. Your implementations will account for both the direct and indirect illumination in a scene.

Before you begin, we recommend you review the course material, especially slides on implict and explicit path tracing.

Contents

Afterwards, start by copying your rendering loops as well as any code needed from previous assignments and **ensure that you initialize your random sampler** object with your student ID, *e.g.*, `Sampler sampler(123456789)`.

1 Offline Rendering (*80 pts*)

In this first part, you will extend your previous BRDF implementations to a more flexible mixture model. Then, you will implement both an implicit and explicit path tracer to render your first images with global illumination.

1.1 Diffuse and Phong BRDF Mixture Model (*10 pts*)

In earlier assignments you implemented separate diffuse and Phong BRDFs, including evaluation and sampling routines needed to integrate them into Monte Carlo-based integration algorithms. You might have noticed that for the Phong model, you were combining diffuse a glossy in your `eval()` function, but *not* in your sampling routine. In other words, you were importance sampling the glossy component only in `sample()`.

Another common BRDF is a *mixture model* that combines both diffuse and glossy (e.g., Phong) reflection effects, i.e., $f_r = \rho_d \, f_r^{\text{diffuse}} + \rho_s \, f_r^{\text{glossy}}$. These models are useful, allowing you to augment glossy reflections with a diffuse "base coat": any reflected light that falls sufficiently outside the glossy lobe/highlight will not result in an entirely black appearance.

There are two immediate ways of implementing such a mixture model, especially given your existing independent implementations of the diffuse and Phong BRDFs. The first is to explicitly treat the mixture BRDF as a sum of the two individual BRDFS. Here, you would not be able to perform perfect BRDF importance sampling of the mixture and, instead, you'd have to pick between either importance sampling the diffuse (cosine-weighted) or glossy (cosine-power) components of the distribution.

Alternatively, we can leverage a stochastic interpretation of the mixture: using Russian Roulette, you stochastically choose between performing *either* a diffuse *or* glossy reflection event at each BRDF sampling decision. Then, depending on your stochastic choice, apply the appropriate importance sampling scheme.

For example, for a mixture model that's 50% diffuse and 50% glossy, instead evaluating a BRDF that's a weighted sum of these lobes, you'll use a Russian Roulette decision that will treat the reflection as a purely (i.e., 100%) diffuse reflection 50% of the time, and as a purely (i.e., 100%) glossy reflection the other 50% of the time. Of course, when you choose to reflect diffusely, you can use cosine-weighted IS; otherwise, cosine-power IS for the glossy case.

We ask that you implement this second approach (although you can validate it against the first approach, if desired). Instead of always assuming a 50/50 split between diffuse and glossy reflection in the mixture, use the diffuse (ρ_d) and glossy (ρ_s) reflection coefficients as their respective weights. These weights are normalized, i.e., $\rho_d + \rho_s = 1$. Don't forget to normalize your estimate by the appropriate Russian Roulette probability. In other words, you need to rescale the 2D sample by the correct amount after determining the scattering type. This will allow you to reuse your sample across the Russian roulette decision and BRDF lobe importance sampling.

Implement a new BRDF, including its evaluation function `MixtureBSDF::eval()`, its sampling function in `MixtureBSDF::sample()` and its corresponding `MixtureBSDF::pdf()` methods, all in `src/bsdf/mixture.h`. Your Mixture PDF should implement a linear combination of glossy and diffuse BRDFs. When dividing by the sampling PDF (which will depend on your Russian Roulette), make sure that this division is well-defined, otherwise simply return zero.

1.2 Implicit Path Tracing (*30 pts*)

Next, you will implement an implict path tracer. Recall the hemispherical form of the rendering equation discussed in class:

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + \int_{\Omega} f_r(\mathbf{x},\omega',\omega)\, L(r(\mathbf{x},\omega'),-\omega')\cos\theta'\,\mathrm{d}\omega' \tag{1}$$

where $r(\mathbf{x},\omega')$ is the ray tracing function that returns the closest visible point from $\mathbf{x}$, in direction ω'. We know that equation (1) can be approximated as a single-sample Monte Carlo integral estimate:

$$L(\mathbf{x},\omega) \approx L_e(\mathbf{x},\omega) + \frac{f_r(\mathbf{x},\omega',\omega)\, L(r(\mathbf{x},\omega'),-\omega')\cos\theta'}{p(\omega')}. \tag{2}$$

The basecode includes a new integrator `PathIntegrator` in `src/integrators/path.h`. This structure is where you will implement your various path tracing integrators, and it contains the following fields:

Field	Description
`m_isExplicit`	Implicit or explicit path tracer boolean toggle
`m_maxDepth`	Maximum path depth, meaning the total number of possible bounces for a path (*e.g.*, −1 = infinity, 0 = emitted light only, 1 = emission and direct illumination, 2 = emission and direct and 1-bounce indirect lighting, etc.)
`m_rrProb`	Russian roulette probability (*e.g.* 0.95 means a 95% chance of recursion)
`m_rrDepth`	Path depth at which Russian roulette path termination is employed

The function `PathIntegrator::render()` currently performs the first intersection hit for you; the recursion needs to be done in the corresponding subroutine `renderX()`. You will first implement an *implicit path tracer* with **BRDF sampling** in `PathIntegrator::renderImplicit()`. Your path tracer will truncate path lengths, meaning each path should bounce `m_maxDepth` times away from the eye. Clamping the maximum path length will introduce bias in your estimator. Later, we will use Russian roulette termination to solve this problem.

Depending how you implement your path tracing algorithms (*i.e.* whether recursively or with a loop), you may need extra logic/variables for bookkeeping (e.g., tracking the current number of path vertices, or the accumulated throughput, or the joint path PDF value). Feel free to add any additional methods to the integrator that you may need to structure your particular algorithm.

Test your implementation on `data/a5/cbox/cbox_path_implicit_X_bounces.toml` for a different number of scattering events *X*. If your (potentially recursive) path construction is implemented correctly, your rendered image should look something like the following (rendered with 128 spp):

Figure 1: *0 bounces (emission only)*

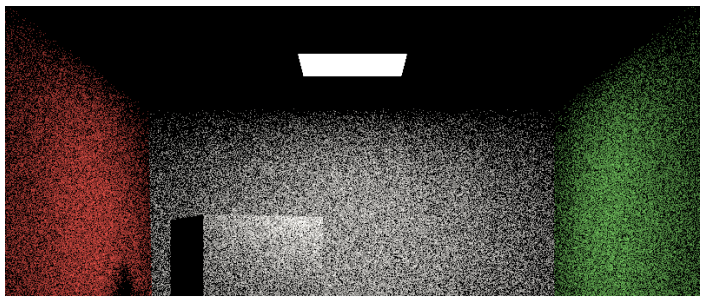

Figure 2: *1 bounce (emission and direct illumination)*

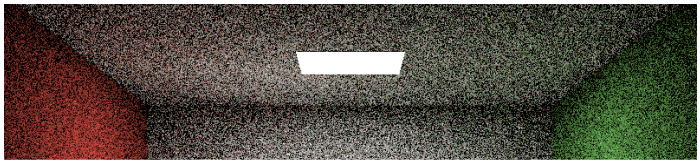

Figure 3: *2 bounces*

Figure 4: *4 bounces*

You can play around with the scene TOML files to modify the path lengths and the number of samples per pixel, when debugging. When you believe your implementation is complete, render the following evaluation scenes:

- `data/a5/livingroom/livingroom_path_implicit_1_bounce.toml`
- `data/a5/livingroom/livingroom_path_implicit_2_bounces.toml`
- `data/a5/livingroom/livingroom_path_implicit_4_bounces.toml`

1.3 Explicit Path Tracing (*40 pts*)

The images you just finished rendered were likely to be very noisy, due to the nature of implicit path tracing: blindly tracing paths with the hope of randomly hitting a light.

Given our knowledge of the scene (i.e., light geometry is explicitly provided as input to the renderer), we can take advantage of light importance sampling schemes from Assignment 4 to arrive at more effective path tracing estimators.

Implement explicit path tracing: now, every time light (really, importance) scatters at a surface point, we will split our outgoing radiance estimate according to the direct and indirect illumination contributions. Be careful to avoid double counting the *same transport contributions (as discussed in class)*:

$$L(\mathbf{x}, \omega) = L_e(\mathbf{x}, \omega) + L_{\text{dir}}(\mathbf{x}, \omega) + L_{\text{ind}}(\mathbf{x}, \omega). \tag{3}$$

This estimator performs *explicit direct illumination* estimation at every path vertex, and your next goal will be to implement this *explicit path tracing* algorithm.

1.3.1 Basic Explicit Path Tracer Integrator (*30 pts*)

You can use your `DirectIntegrator::renderEmitter()` as a starting point to implement `PathIntegrator::renderExplicit()`. The key difference between explicit and implicit path tracing is that direct and indirect lighting contributions are decoupled, meaning that they are (importance-)sampled separately. To avoid double counting, an indirect ray needs to be re-sampled if it intersects a light (and, so, contributes *directly* to transport along the path).

Implement `PathIntegrator::renderExplicit()` with **surface area emitter importance sampling** for the direct lighting contribution, and **BRDF importance sampling** for the indirect lighting contribution.

Note that light sources are not assumed to be spheres anymore: they can be attached to any shape in the scene. You can use the functions `sampleEmitterPosition()` to retrieve a uniformly sampled point on an arbitrary mesh emitter, along with its corresponding normal, sampling PDF and radiance. Don't forget to take the Jacobian of the area-to-solid-angle parameterizations into account in your integrator, if necessary.

Below are images for a different number of indirect bounces, rendered at 128 spp. These images correspond to the scenes `data/a5/cbox/cbox_path_explicit_X_bounces.toml` that you should use to test your implementation.

Figure 5: *0 bounce (emitter only)*

Figure 6: *1 bounce (direct illumination only)*

Figure 7: *2 bounces*

Figure 8: *4 bounces*

1.3.2 Russian Roulette Path Termination (*10 pts*)

Artificially truncating path lengths to a fixed depth introduces bias. To avoid this problem, you will modify your explicit path tracer to support Russian roulette termination that probabilistically terminates path construction in an unbiased manner.

Use `m_rrDepth` to determine whether to employ Russian roulette termination at your current path vertex, and use the `m_rrProb` as your RR termination probability. Render the Cornell box scene `data/a5/cbox/cbox_path_explicit_rr.toml` and compare it to a converged reference image below (rendered with 512 spp).

Figure 9: *Explicit path tracer with Russian roulette path termination. RR is employed starting at path depth* $d = 4$ *with path recursion probability* $p = 0.95$.

When you're done, render the following final scenes:

- `data/a5/livingroom/livingroom_explicit_1_bounce.toml`
- `data/a5/livingroom/livingroom_explicit_2_bounces.toml`
- `data/a5/livingroom/livingroom_explicit_rr.toml`

Debugging Your Path Tracer

Finding bugs in your path tracer can be difficult and time-consuming. Path tracing is by far the most computationally intensive algorithm you have had to implement in the course. As such, a single path traced rendering can take several minutes to complete, even on modern laptops/desktops. Below are a few tips to speed up computations during testing:

- Edit the TOML scene description file:
  - Decrease the resolution of the film (while preserving its aspect ratio)
  - Decrease the number of samples per pixels
- Debug one path bounce at a time: make sure 0 bounce = emitters only works before moving on to 1 bounce = direct illumination only, etc.
- Implement a multithreaded rendering loop: this will speed up your final rendering time, but won't help much with debugging

2 Real-time Rendering (*20 pts*)

You will implement your first *hybrid* rendering algorithm: one that uses data precomputed using a (costly) offline rendering preprocess, and a separate real-time reconstruction viewer.

2.1 Precomputed Global Illumination (*20 pts*)

Interactive global illumination remains one of the grand challenges in computer graphics, and is an area of ongoing research. That being said, under certain constraints, we already have solutions to this problem.

For example, assume that the geometry and lighting in a scene are **static** (i.e., they don't ever move), and that only the virtual camera is allowed to move around. One can imagine using a modified offline rendering algorithm to compute the entire outgoing radiance spherical distribution $L(\mathbf{x}, \omega)$

$$L(\mathbf{x}, \omega) = L_e(\mathbf{x}, \omega) + \int_{\mathcal{H}^2} f_r(\mathbf{x}, \omega', \omega)\, L(r(\mathbf{x}, \omega'), -\omega') \cos\theta'\, \mathrm{d}\omega' \tag{4}$$

at many points $\mathbf{x}$ in a scene, and then interpolating from these points and evaluating $L(\mathbf{x}, \omega_o)$ at outgoing directions ω_o corresponding to the current camera view.

If we furthermore assume that our (static) objects only have **diffuse BRDFs**, then we need only precomptue and cache a *single* outgoing radiance *value* $L(\mathbf{x}, \omega) \equiv L(\mathbf{x})$, and not an entire spherical distribution, at different scene points $\mathbf{x}$:

$$L(\mathbf{x}) = \frac{\rho}{\pi} \int_{\mathcal{H}^2} L(r(\mathbf{x}, \omega'), -\omega') \cos\theta' \, \mathrm{d}\omega' \tag{5}$$

We can, of course, only store this diffuse outgoing radiance $L(\mathbf{x})$ at a discrete set of points $\mathbf{x}$ in our scene. In this task, we'll precompute this radiance and store it at every vertex of the scene geometry; then, when using this precomputed per-vertex data for shading, we'll simply interpolate the per-vertex shading over triangle faces (i.e., Gouraud shading) in order to approximate the final shading at all possible $\mathbf{x}$s. The finer the scene tesselation, the more accurate the approximation.

We will implement this interactive global illumination solution in two steps:

1. A precomputation pass: compute the (diffuse) outgoing radiance at every vertex and store these values in your scene's VBOs (Vertex Buffer Objects).
2. A runtime pass: render the scene using the VBOs, looking up the per-vertex data and interpolating it across triangle faces using rasterization.

> Variants of this technique are very commonly used in video games and are referred to *light mapping* or *light baking* methods.

2.1.1 Precomputation Pass: Computing and Storing Radiance in VBOs (*10 pts*)

To compute the radiance at each vertex, you will call your *path tracer*: instead of passing in a shading point traced from the eye, you will set the shading point at scene vertex locations (you can set the "view direction" arbitrarily, since we're computing diffuse outgoing radiance; see Figure 10).

Figure 10: *Computing per vertex radiance*

For each sample per "pixel" (really, per-vertex), you need to manually populate a `SurfaceInteraction` object to pass to your integrator in order to force it to compute a global illumination estimate at a specific vertex location:

1. set an arbitrary ray direction (ω_o) and shift the shading point position by ϵ along the normal to avoid self-intersections during integration.
2. Create a `SurfaceInteraction` and manually populate its attributes.
3. Call your `PathTracerIntegrator->renderExplicit()` with the incoming "ray" and the `SurfaceInteraction` references.

You will need to use the functions `getPrimitiveID()` and `getMaterialID()` to properly initialize some of the the `SurfaceInteraction` attributes.

Up until now, we were providing you with the VBOs needed for your interactive rendering tasks, and you were expected to employ them appropriately during rendering.

For this task you will need to complete an implementation of the `buildVBO()` function called from `init()`. Note that there is one VBO per scene object and each object is comprised of potentially many vertices.

In TinyRender, you can access a given object's vertex data using `scene.getObjectVertexPosition()` and `scene.getObjectVertexNormal()`. A VBO is the data buffer containing all the vertex attributes for a given object. In this case, the vertex attributes we need are `position` (x, y, z) and `precomputed color` (R, G, B). In TinyRender, the `GLObject` in `src/core/renderpass.h` provides a wrapper around pure OpenGL VBO objects.

```
struct GLObject {
   GLuint vao{0};
   GLuint vbo{0};

   int nVerts{0};
   std::vector<glfloat> vertices;
};
```

Vertices need to contain the raw data of the vertex attributes stored one after the other:

```
vertices[0 to 5] = x,y,z,R,G,B (for vertex 0)
vertices[6 to 12] = X,Y,Z,R,G,B (for vertex 1)
// ... for all remaining vertices
```

2.1.2 Runtime Pass: Precomputed Radiance Lookup (*10 pts*)

We provide you with vertex and fragment shaders (`src/shaders/gi.vs` and `src/shaders/gi.fs`) for this task, but you need to implement the `render()` function in `src/renderpasses/gi.h` that renders your scene with them.

The `render()` function is very similar to the one you implemented in `src/renderpasses.normal.h` (A1), but the attributes in the VBOs are *position* and *color* instead of position and normal.

To test your implementation, render the `cbox_gi_high_realtime.toml` scene and compare your output images with the ones rendered using your path tracer. Remember, since this is an approximation, you will *not* obtain the same image!

Figure 11: *Precomputed Global Illumination*

Figure 12: *Path tracing*

Also compare the results you obtain with varying scene discretizations: your should notice that using a finer scene discretization results in a more accurate approximation.

Figure 13: *Low discretization* **Figure 14:** *Medium discretization* **Figure 15:** *High discretization*

When you're done, render the final scene `cubebox_gi_realtime.toml`. This might take a while...

Debugging Tricks

- First of all, make sure that (at least at the start) you are using very low SPP.
- To verify that you are creating the VBO correctly, initialize values with a fixed color (instead of the precomputed radiance) and visualize your results.
- Once your VBO seems to be created correctly, go ahead and change the fixed color to the computed radiance for direct illumination (set the path tracer's max depth to 1), before moving on to full global illumination.

3 Bonus: Extending Your Path Tracer (*up to 70 pts*)

At this point, you should be comfortable with the TinyRender codebase. For many of your, this is the last assignment, and there are still many hacker points available. Below are a few possible extensions to add to your offline renderer for bonus points, with a **maximum** of 70 additional points up for grabs.

3.1 Path Tracing Extensions

- **Path tracer with MIS (*10 pts*).** Modify your explicit path tracer to use multiple importance sampling with a balance heuristic for direct illumination. Edit the A4 Veach scene file to use your path tracer and render this new scene with a reasonable amount of samples per pixel.

- **Light tracer integrator (*20 pts*).** In path tracing, rays start from the eye and eventually connect to a light when progressively constructing a path. Light tracing is the opposite process of starting from a light source and building paths to the eye. Since we are dealing with pinhole cameras, only explicit light sampled connections will work. Implement such an explicit light tracer and re-render the Cornell box to validate your implementation.

- **Bidirectional path tracer (*60 pts*).** Combining path tracing and light tracing path sampling strategies results in a *bidirectional* path tracing (BDPT). A complete implementation of BDPT requires careful engineering: specifically, you will need new structures to store the vertices of your subpaths, and a function to connect your two subpaths in nondegenerate ways. You will also need to correctly evaluate the PDF at each scattering event, and for each fully-formed path. Most modern implementations of BDPT use an abstract tracer that can be used in both eye and light directions.

Implement bidirectional path tracing and render the modified Cornell box scene `data/a5/bonus_bdpt` to demonstrate how BDPT excels at rendering scenes with difficult-to-reach emitter profiles.

In all cases, include a new line to your `config.json` file with an appropriate scene title, *e.g.*

```
{ "scene": "Bonus: Cornell box rendered with BDPT",
  "render": "offline/cbox_bdpt.exr"
}
```

3.2 More Complex BRDFs

TinyRender only handles diffuse and Phong-based BRDFs, but there are many other reflectance models such as

- **Mirror BRDF (*10 pts*)**
- **Dielectric (glass) BRDF (*20 pts*)**
- **Microfacet model (*30 pts*)**

Adding a new BRDF to the framework can be lengthy: first, you will need to modify the material file parser in `Scene::load()` (in `src/core/renderer.cpp`) to permit the use of your new BRDF object. The ID you give to your new material corresponds to the `illum` tag in the material file of your scene, and any other material-specific parameters will need to be passed via a valid `mtllib` tag. Then, you need to create a BRDF structure for your new material and correctly implement all three functions `eval()`, `sample()` and `pdf()`.

Each of the proposed BRDFs are discussed in detail in PBRTe3. To demonstrate your implementation, re-render the Cornell box scene with your *explicit path tracer* and your new materials assigned to `leftBox` and/or `rightSphere`. Use a reasonable number of samples per pixel. Include a new line to your `config.json` file with an appropriate scene title:

```
{ "scene": "Cornell Box with mirror and dielectric BRDFs",
  "render": "offline/cbox_path_mirror_glass.exr"
}
```

What to Submit

Render all the scenes in `data/a5/livingroom` and `data/a5/cubebox`. Also edit your `config.json` to include your credentials. You should only be modifying the **lines 2–4** of your configuration file.

Submit this file with all your code in a `.zip` or `.tar` archive file. Include your raw `.exr` files in separated folders (see structure below).

```
a4_first_last.zip
   config.json
   src
   offline/
      livingroom_path_implicit_1_bounce.exr
      livingroom_path_implicit_2_bounces.exr
      livingroom_path_implicit_4_bounces.exr
      livingroom_path_explicit_1_bounce.exr
      livingroom_path_explicit_2_bounces.exr
      livingroom_path_explicit_rr.exr
    realtime/
      cubebox_gi.exr
```

Make sure your code compiles and runs before submitting! **You will obtain a score of zero if your submission does not follow this exact structure.** You can use the `tree` command to verify it.