ECSE 446/546: Realistic/Advanced Image Synthesis

Assignment 5: Global Illumination

Due Sunday, December 1st, 2019 at 11:59pm EST on myCourses
20% (ECSE 446 / COMP 598) *or* **15%** (ECSE 546 / COMP 599)

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

Before starting, we recommend that you review the course material, especially slides on implict and explicit path tracing. Keep in mind that many of the approaches you have implemented and studied for *direct illumination* can be generalized or adapted to the global illumination setting.

Contents

Sync up with the Git repo using `git stash save && git pull && git stash pop`. See the TinyRender docs for more information. The base code now contains **`TODO(A5)`** comments that you can search for from your IDE.

1 Offline Rendering (*80 pts*)

1.1 Implicit Path Tracing (*40 pts*)

You will first implement an implict path tracer. Recall the hemispherical form of the rendering equation discussed in class:

$$L(\mathbf{x}, \omega) = L_e(\mathbf{x}, \omega) + \int_{\mathcal{H}^2} f_r(\mathbf{x}, \omega', \omega)\, L(r(\mathbf{x}, \omega'), -\omega') \cos\theta'\, \mathrm{d}\omega' \tag{1}$$

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 starts being employed

The `PathIntegrator::render()` function currently performs the first intersection hit from the eye for you; you need to implement the light transport estimator in the corresponding `renderX()` subroutine . You will first implement an *implicit path tracer* with **BRDF importance 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 avoid this bias.

Depending on how you implement your path tracing algorithms (*i.e.* whether recursively or with a loop), you may need to introduce additional bookkeeping logic and/or variables (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 implementation of the algorithm.

Test your implementation on `data/a5/cbox/cbox_path_implicit_X_bounces.toml` for various numbers of scattering events *X*. If your (potentially recursive) path construction is implemented correctly, your rendered image should look similar to the following images (each rendered with 128 spp):

Figure 1: *0 bounces (emission only)*

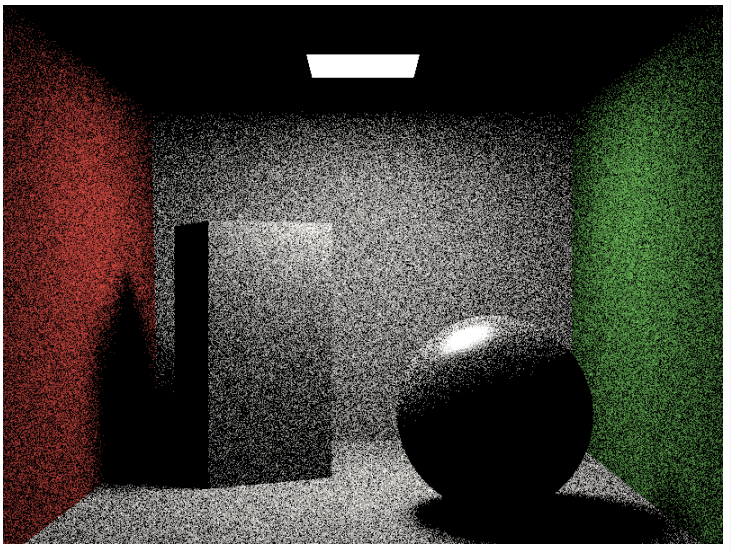

Figure 2: *1 bounce (emission and direct illumination)*

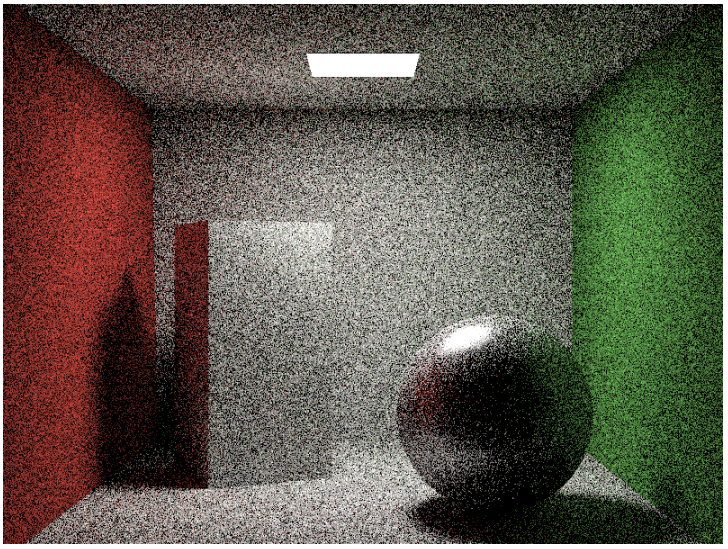

Figure 3: *2 bounces*

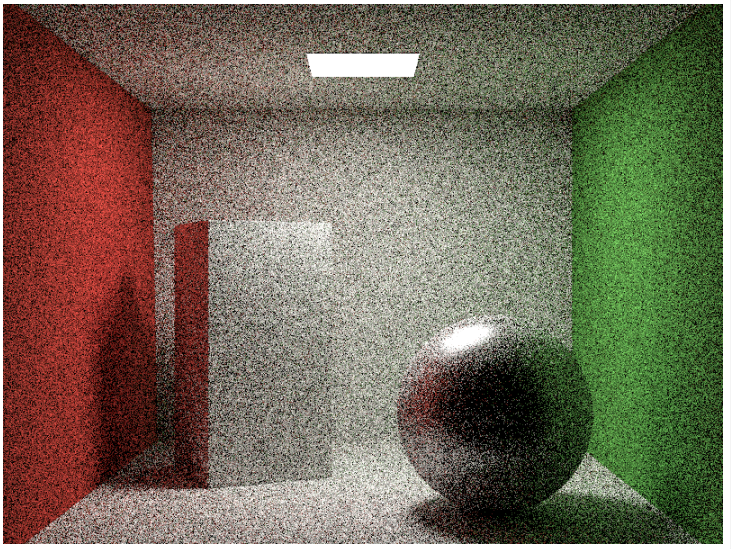

Figure 4: *4 bounces*

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`

1.2 Explicit Path Tracing (*40 pts*)

The images you just rendered were likely to be very noisy, especially at low sampling rates, due to the nature of implicit path tracing: blindly tracing paths with the hope of randomly hitting a light may not always be an efficient strategy.

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 3 and 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)*:

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

1.2.1 Explicit Path Tracing Integrator (*30 pts*)

The key difference between explicit and implicit path tracing is that direct and indirect lighting contributions are estimated separately, 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 at the scattering vertex).

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 function `sampleEmitterPosition()` to retrieve a point sampled uniformly according to the surface area on an arbitrary mesh emitter, along with its corresponding normal, sampling PDF and radiance. Do not forget to take the Jacobian of the area-to-solid-angle parameterizations into account in your integrator, if necessary.

Below are images generated with a varying number of indirect bounces, each 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)*

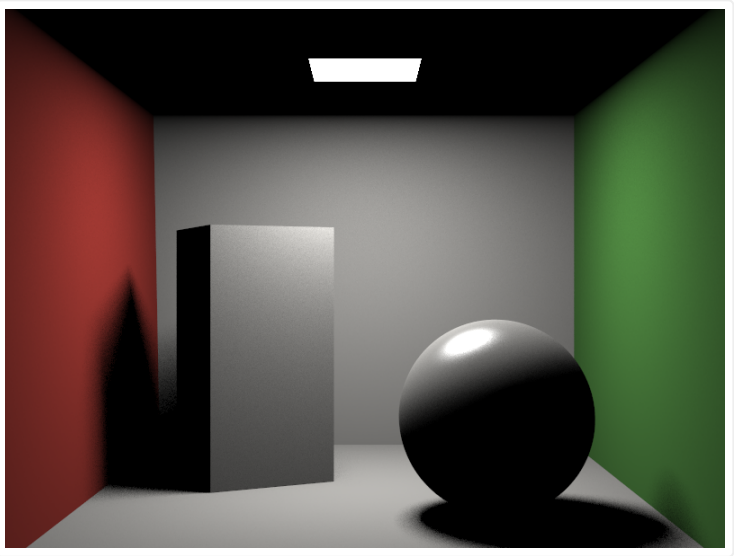

Figure 6: *1 bounce (direct illumination only)*

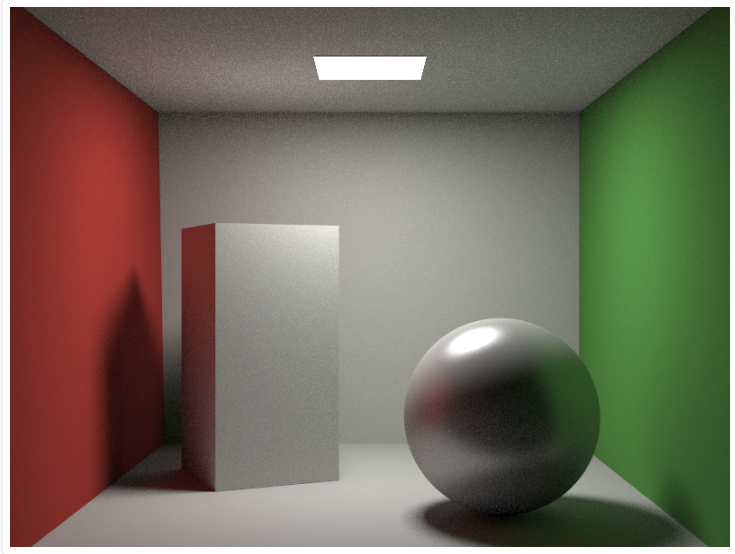

Figure 7: *2 bounces*

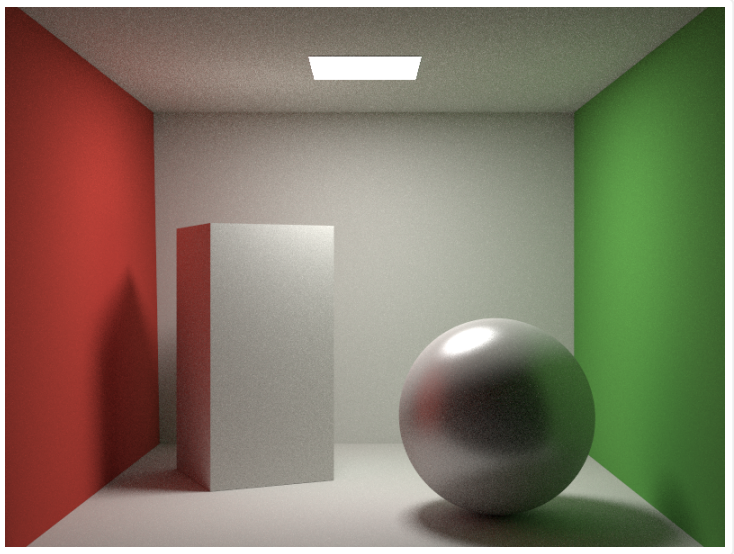

Figure 8: *4 bounces*

1.2.2 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). Here, you are employing a single, global RR termination probability, as opposed to one that is set more intelligently (e.g., according to the local scattering properties at the shading point).

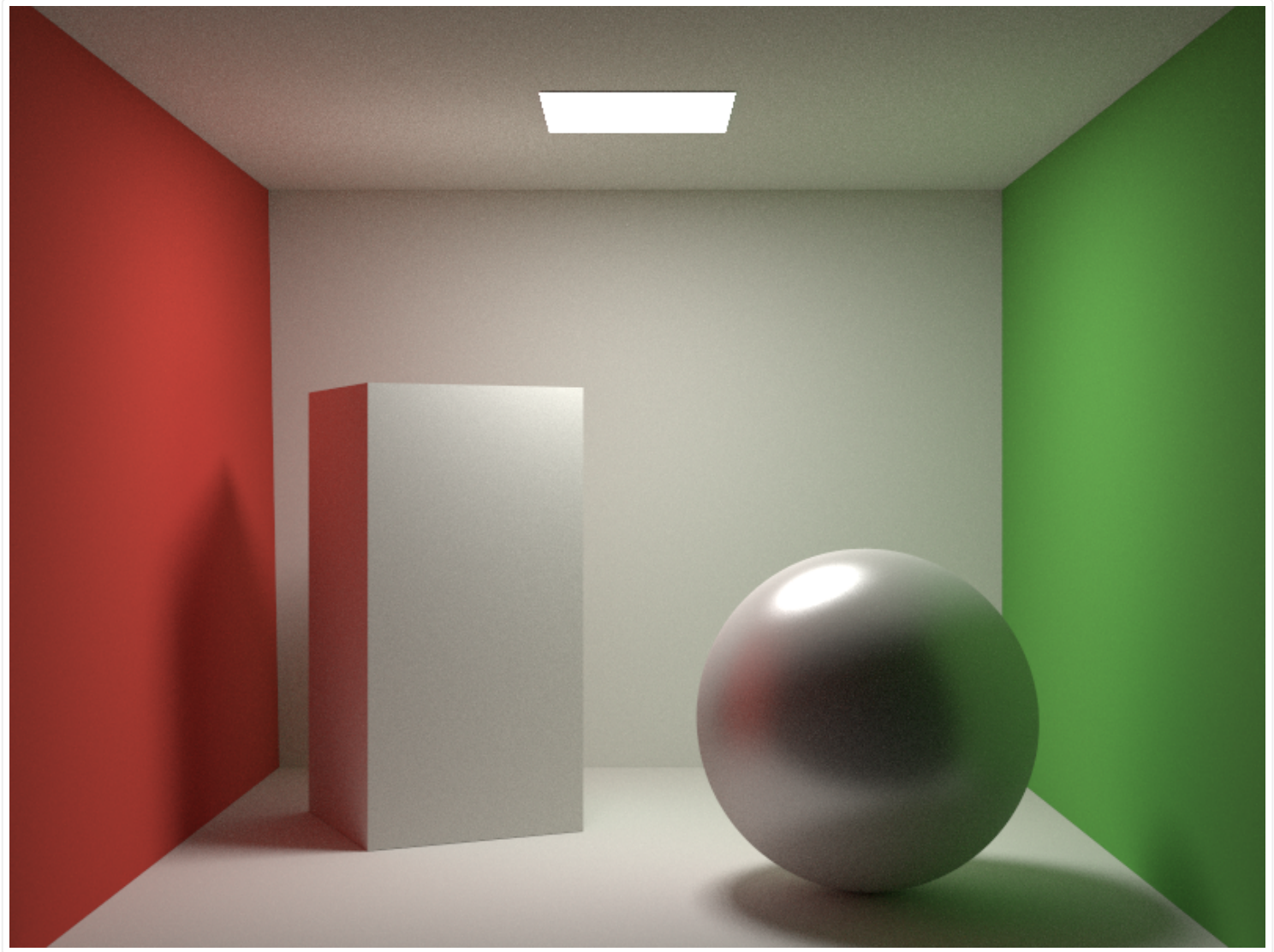

Figure 9: *Explicit path tracer with Russian roulette path termination. RR is employed starting at path depth* $d = 4$ *with path recursion probability* $p = 0.95$.

When complete, render the following final evaluation 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 will 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 iteration time 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), etc.
- Use the multithreaded rendering loop carefully: this will speed up your final rendering time, but may complicate your debugging process

2 Real-time Rendering (*20 pts*)

You will implement another hybrid rendering algorithm, combining data precomputed during a (costly) offline rendering preprocess with a separate real-time 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: here, 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, then interpolating from these points and evaluating $L(\mathbf{x}, \omega_o)$ at outgoing directions ω_o corresponding to the viewing direction at the shade point.

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

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

We can, of course, only store this diffuse outgoing radiance $L(\mathbf{x})$ at a discrete and finite set of points $\mathbf{x}$ in our scene. In this task, we will precompute this diffuse outgoing radiance and store it at every vertex of the triangulated scene geometry; then, at run-time, we will interpolate the precomputed 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*, *light baking* or *precomputed lighting* methods.

2.1.1 Precomputing & Storing Radiance in VBOs (*10 pts*)

To compute the radiance at each triangle vertex, you will rely on your *path tracer*: instead of computing the outgoing radiance at shading point traced from the eye, you will set the shading point directly to the scene vertex locations we want to precompute the value at (here, you can pass an arbitrary "view direction" to the path tracer, since we're computing diffuse outgoing radiance; see Figure 10).

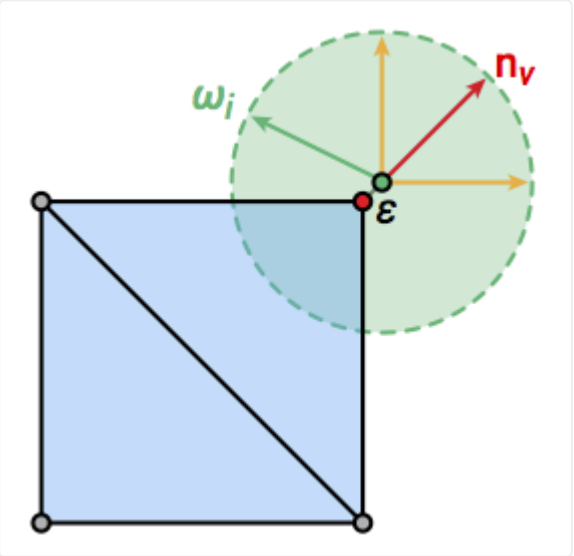

Figure 10: *Computing per-vertex outgoing radiance*

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 `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 composed 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 will need at run-time 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 (vertex 0 data)
vertices[6 to 12] = X,Y,Z,R,G,B (vertex 1 data)
// ... all remaining vertices' data
```

2.1.2 Rendering with Precomputed Lookups (*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!

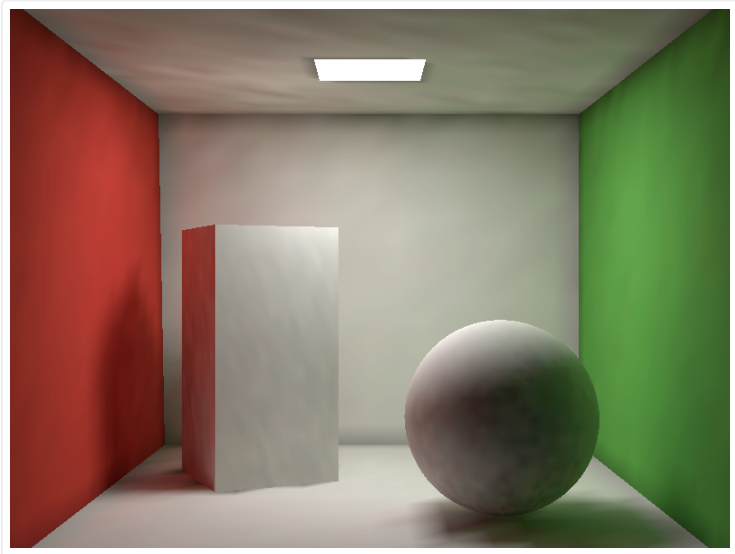

Figure 11: *Precomputed Global Illumination*

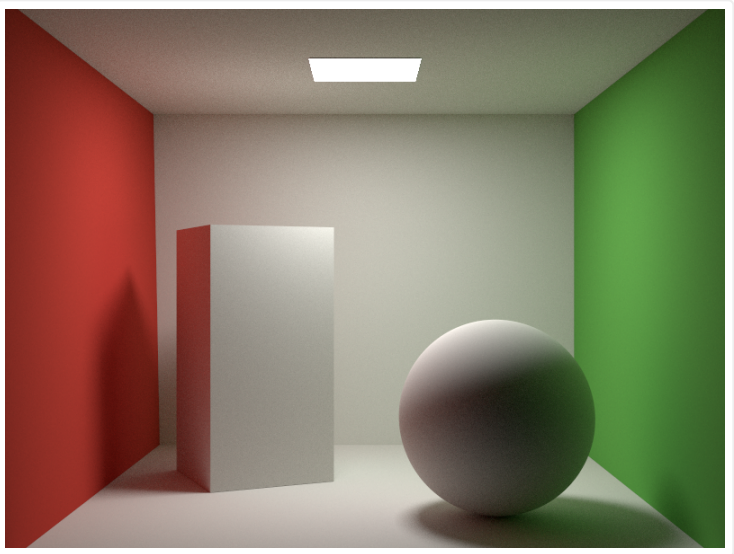

Figure 12: *Path tracing*

Also compare the results you obtain with varying scene discretizations: your should notice that using a finer scene discretization (i.e., more triangles/vertices) results in a more accurate approximation.

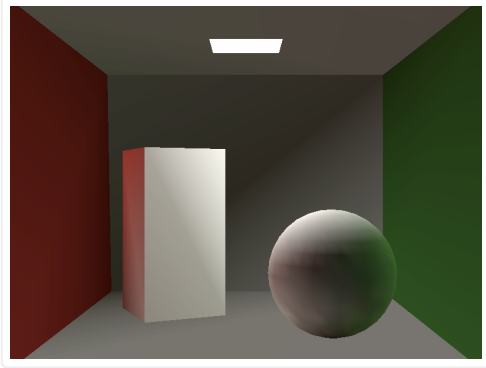

Figure 13: *Low discretization*

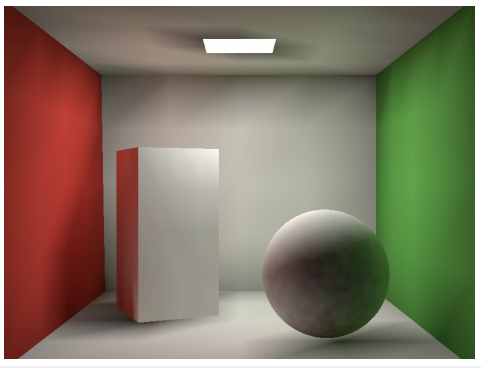

Figure 14: *Medium discretization*

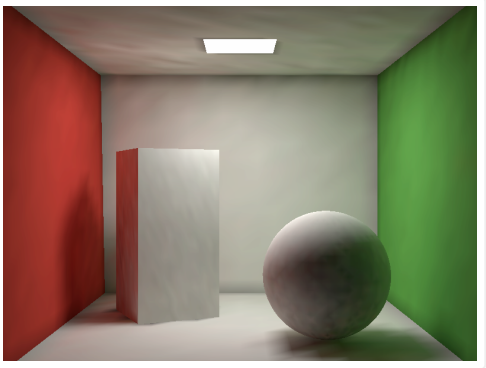

Figure 15: *High discretization*

When you're done, render the final scene `cubebox_gi_realtime.toml`. This may take a while, and so **plan ahead** to account for the rendering time needed to generate your final results before the submission deadline.

Debugging Tricks

- 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 (i.e., set the path tracer's max depth to 1), before moving on to full global illumination.

3 Bonus: Extending Your Path Tracer (*up to 50 pts*)

At this point, you should be comfortable with the TinyRender codebase. Below are a few possible extensions to add to your offline renderer for bonus points, with a **maximum** of 50 additional points up for grabs. We are also open to suggestions: if you've found a cool blog post or paper that you'd like to implement, speak to the Professor after class and he can assess:

1. whether it's suitable as a fun, manageable bonus task, and
2. how much it would be worth.

3.1 Examples of Possible Path Tracing Extensions

- **Path tracer with multiple importance sampling (*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 to demonstrate the improvements due to your MIS.

- **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, 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 (*30 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. Implementing BDPT subsumes the Light tracer task above, meaning you can't get credit for both.

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 Appearance Models

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

- **Mirror BRDF (*5 pts*)**
- **Dielectric (glass) BRDF (*10 pts*)**
- **Microfacet model (*20 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 in 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 the `data/a5` directory. When complete, edit the `config.json` file with your credentials and submit all your code in a `.zip` archive file. Please create a *separate* directory with only the files you need to submit, and compress this directory — this is different than selecting all files to submit and right-clicking compress. Include your raw `.exr` files in separate folders (see the 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
      // Other images to demonstrate bonus, optional
    realtime/
      cubebox_gi.exr
```

Make sure your code compiles and runs before submitting! **Code that does not compile or does not follow this exact tree structure may result in a score of zero.** If your assignment does not compile at the moment of submission, but you do have some correct images from previous iterations, please submit these. You can use the `tree` command to verify this structure.

formatted by Markdeep 1.07