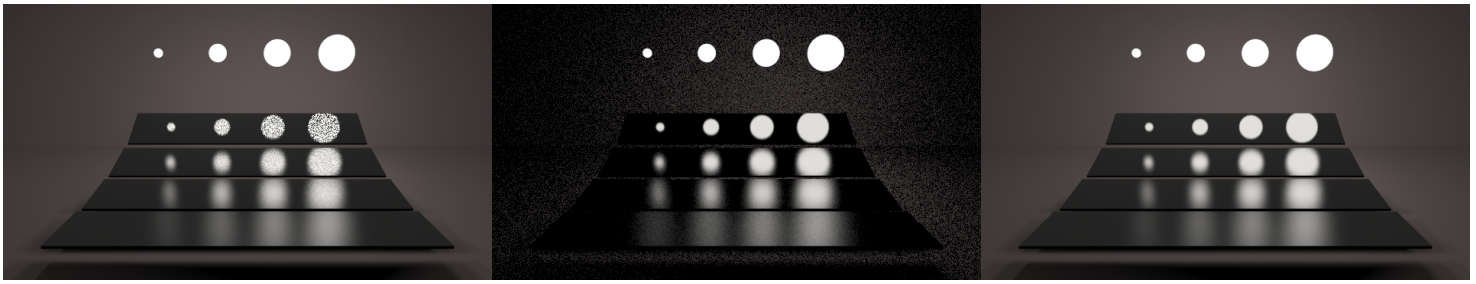
ECSE 446/546: Realistic/Advanced Image Synthesis

Assignment 4: MIS, Polygonal Lights and Control Variates

Due Sunday, November 10th, 2018 at 11:59pm EST on myCourses
18% (ECSE 446 / COMP 598) *or* **13%** (ECSE 546 / COMP 599)



This assignment begins by extending the direct illumination integrator from A3 using a multiple importance sampling (MIS) combination of both BRDF and emitter importance sampling.

Afterwards, you will implement a new integrator tailored to *polygonal* light sources composed of triangles. Here, you will first compute an analytic solution to the unshadowed direct illumination due to a uniformly emissive polygonal emitter before validating your implementation with a numerical surface area-based Monte Carlo estimator for the same integral. Finally, you apply your analytic solution as a *control variate* to reduce the variance of a complete (i.e., shadowed) Monte Carlo-based direct illumination estimator.

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(A4)`** comments that you can search for from your IDE.

1 Offline Rendering (*60 pts*)

1.1 Multiple Importance Sampling (*25 pts*)

In Assignment 3, you implemented BRDF sampling and spherical emitter sampling. You will now combine these two sampling schemes using multiple importance sampling (MIS), as introduced by Veach & Guibas (1995). Specifically, you will implement an integrator that combines BRDF and emitter importance sampling to render an image.

Recall that if two sampling distributions p_f and p_g can be applied to estimate an integral $I = \int_\Omega h(x)\,\mathrm{d}x$, we can write an expression for single Monte Carlo estimator that uses both distributions simultaneously using the MIS formulation, as

$$I \approx \frac{1}{n_f}\sum_{i=1}^{n_f}\frac{h(X_i)w_f(X_i)}{p_f(X_i)} + \frac{1}{n_g}\sum_{j=1}^{n_g}\frac{h(X_j)w_g(X_j)}{p_g(X_j)}, \tag{1}$$

where n_f is the number of samples drawn proportional to p_f, n_g the number of samples drawn proportional to p_g, and w_f & w_g are special normalization weighting functions chosen so as to not bias the estimator.

Many such weighting functions exist, and one provably good choice uses a *balance heuristic*, given by the following equation, where $s \in \{f, g\} = S$:

$$w_s(x) = \frac{n_s\, p_s(x)}{\sum_{i\in S} n_i\, p_i(x)}. \tag{2}$$

Your first task is to modify your `DirectIntegrator` integrator to support MIS with this heuristic. The balance heuristic is already implemented as part of the `DirectIntegrator` structure. Implementation-wise, your `DirectIntegrator::renderMIS()` function should have two Monte Carlo loops: one over `m_emitterSamples` and one over `m_bsdfSamples`. The former should be similar to your `DirectIntegrator::renderSolidAngle()`, except adapted to take the weighting function into account. Use the function `DirectIntegrator::balanceHeuristic()` and subtended solid angle emitter sampling for this part. The BRDF sampling part is similar: once you find an intersection `i`, you will draw a sample proportional to the BRDF with `BSDF::sample()` and weight it appropriately.

Rendering the Veach Scene

Implement the MIS integrator in `DirectIntegrator::renderMIS()` and test your code against the test scenes in `data/a4/veach` based on those originally proposed by Eric Veach. Below are some reference images. Note that a total of five (5) lights are present in the scene (there is one ambient light not directly visible in the images).

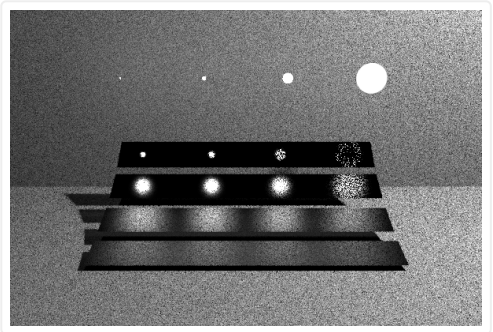


Figure 1: *Emitter sampling with 1 spp*

Figure 2: *BSDF sampling with 1 spp*

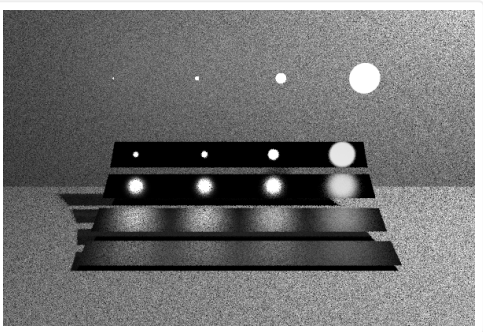


Figure 3: *MIS with 1+1 emitter+BSDF samples per pixel*

The final render for this part of the assignment is the same scene, except the lights have different colors.

> Note that, to earn full marks, you need a *fully functioning implementation of MIS*. You will obtain a score of zero if you only have BSDF and/or emitter sampling working separately. The entire challenge of implementing MIS stems from properly resolving the interplay between emitter and BSDF samples.

Tips & Tricks

Implementing MIS can be tricky and you might find many subtle bugs in your previous implementations that did not surface until now. Here are a few debugging tips:

- Begin by testing with diffuse-only materials. Only move on to MIS tests with Phong BRDFs once you are convinced your integrator works properly with diffuse surfaces.
- Test your MIS estimator with *emitter samples only* (i.e., by setting the number of BSDF samples to zero) and make sure the output image is the same as your original BRDF importance sampling estimator from A3.
- Afterwards, similarly test your MIS estimator with *BSDF samples only* (i.e., setting the number of emitter samples to zero in the TOML file). The output image should obviously have more noise, but should also eventually converge with more samples.
- Finally, *combine the two* and compare the rendered image on the provided test scene.
- When in doubt, recall that a naive uniform sampling MC estimator will (eventually) generate the correct answer: this is one way to isolate bugs in your PDF sampling and evaluation code.

1.2 Analytic Shading for Polygonal Emitters (*15 pts*)

Your second task is to implement direct illumination integrators tailored to polygonal light sources. Recall that in Assignment 2 you implemented a shading algorithm to compute the direct illumination from isotropic point light sources. In this case, we were able to express the incoming radiance due to the emitter analytically and, so, no

numerical integration was needed to estimate the diffuse outgoing radiance. Polygonal lights admit a closed-form solution for their incoming irradiance: your goal is to use leverage this expression for direct illumination. To simplify our problem, we will assume only diffuse reflection and only treat emitters that are entirely in the upper hemisphere of every shading point (otherwise you would have to clip the polygonal source at potentially every shading point).

Consider the following geometric problem statement. Suppose $P \subset \mathbb{R}^3$ is a simple planar polygon with m vertices $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m$. If P is an emitter with uniform irradiance $E = \Phi / \text{Area}(P)$ then the incident irradiance from the emitter at a shading point $\mathbf{x}$ is given by the following closed-form expression:

$$E_i^{\text{poly}}(\mathbf{x}) = \frac{E}{2\pi} \sum_{i=1}^{m} \Theta_i(\mathbf{x}) \big(\Gamma_i(\mathbf{x}) \cdot N(\mathbf{x}) \big), \tag{3}$$

where Θ_i is the angle subtended by the edge from vertices $\mathbf{v}_i$ to $\mathbf{v}_{i+1}$, Γ_i is the edge normal, and $N(\mathbf{x})$ is the surface shading normal at $\mathbf{x}$. More precisely,

$$\Theta_i(\mathbf{x}) = \arccos\left(\frac{\mathbf{v}_i - \mathbf{x}}{\|\mathbf{v}_i - \mathbf{x}\|} \cdot \frac{\mathbf{v}_{i+1} - \mathbf{x}}{\|\mathbf{v}_{i+1} - \mathbf{x}\|} \right)$$

and

$$\Gamma_i(\mathbf{x}) = \frac{(\mathbf{v}_{i+1} - \mathbf{x}) \times (\mathbf{v}_i - \mathbf{x})}{\|(\mathbf{v}_{i+1} - \mathbf{x}) \times (\mathbf{v}_i - \mathbf{x})\|}$$

Here, the vertices are processed in order so that $\mathbf{v}_{m+1} = \mathbf{v}_1$. This scalar expression was first derived by Lambert in the 18th century and later used by Arvo in 1994 in the context of rendering.

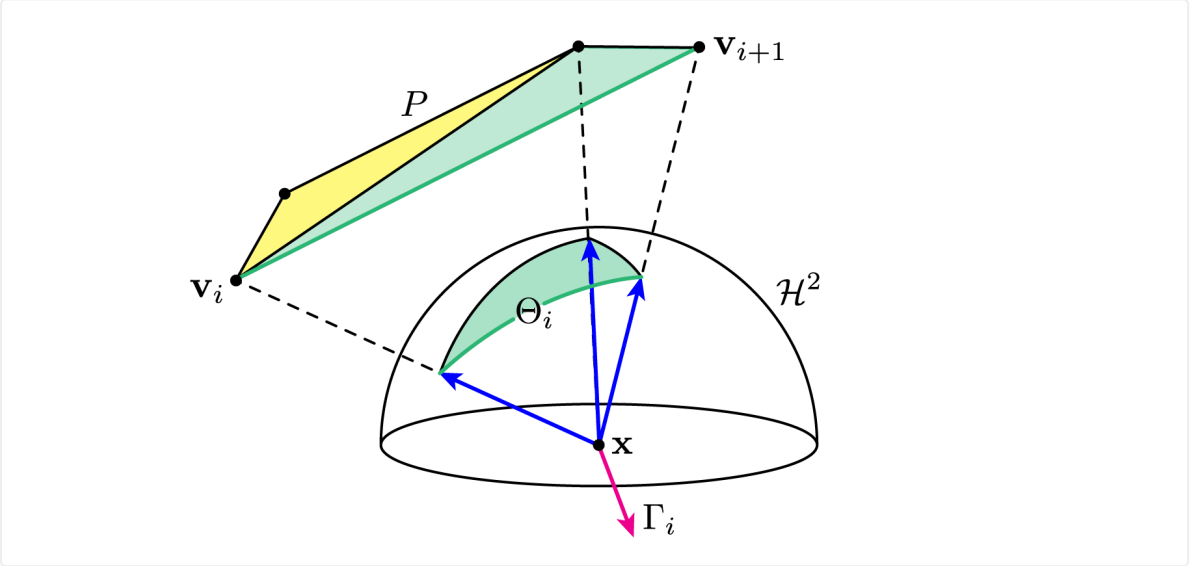


Figure 4: *Projection of polygonal light onto a unit hemisphere about the shading point* $\mathbf{x}$.

Equation (3) describes incoming radiance, and so the *unshadowed* diffuse outgoing radiance is

$$L_o(\mathbf{x}, \omega_o) = \frac{\rho}{\pi} E_i^{\text{poly}}(\mathbf{x}), \tag{4}$$

where ρ is the diffuse albedo at $\mathbf{x}$.

Constructor

First, you will need to retreive and organize the vertices of the emitter in the scene in a way that will allow you easily access them later. Since the emitter is not dynamic, you can do this once in the polygonal integrator's constructor.

Start by retrieving the number of emitter triangles (assuming 0-indexing). You can get a pointer to the corresponding shape as follows:

```
size_t shapeID = scene.getFirstLight();
auto shape = scene.worldData.shapes[shapeID];
```

Then, use `shape.mesh.indices.size()` to obtain the total number of vertices in the emitter mesh.

Once you have the number of *triangles* that make up the polygonal light, you will store each triangle in a vector. In C++, vectors are dynamic arrays that can be filled sequentially. The vector `m_triangles` is already created for you but is empty. You must add elements to a vector using the `push_back` routine. Below is toy example on how to use this data structure for this assignment:

```
std::vector<v3f> tri(3);    // Create empty triangle (3 vertices / triangle)
tri[0] = v3f(0,0,0));       // Add vertices to triangle
tri[1] = v3f(0,0,1));
tri[2] = v3f(0,1,0));
m_triangles.push_back(tri); // Append triangle vector to vector
```

You will need to get the actual position of the vertices before appending them to each triangle. The utility function `scene.getObjectVertexPosition()` can be used to achieve this: it takes the shape ID and the index of the vertex within this shape as input, and ouputs the world-space coordinates of this vertex. Later, once you have all triangles stored in a vector, you will loop over each triangle and call the `PolygonalIntegrator::getEdgeContrib()` function on each consecutive pairs of vertices to compute the edge contribution from Equation (3).

Integrator

Next, continue by implementing `PolygonalIntegrator::renderAnalytic()` in `src/integrators/polygonal.h`. At a high level, your algorithm should follow the structure below:

1. intersect your eye rays with the scene geometry,
2. if there's an intersection `i`, loop through the vector of triangles and compute Equation (3) by summing the contribution of each subtended edge (implement the `getEdgeContrib()` function)
3. retrieve the albedo at the surface point and compute the light's incident radiance, and
4. compute and return the diffuse outgoing radiance at $\mathbf{x}$.

Render the test scene `grid_analytic_offline.toml` in `data/a4/grid/tinyrender`. As usual, make sure to compare with the reference to validate your implementation:

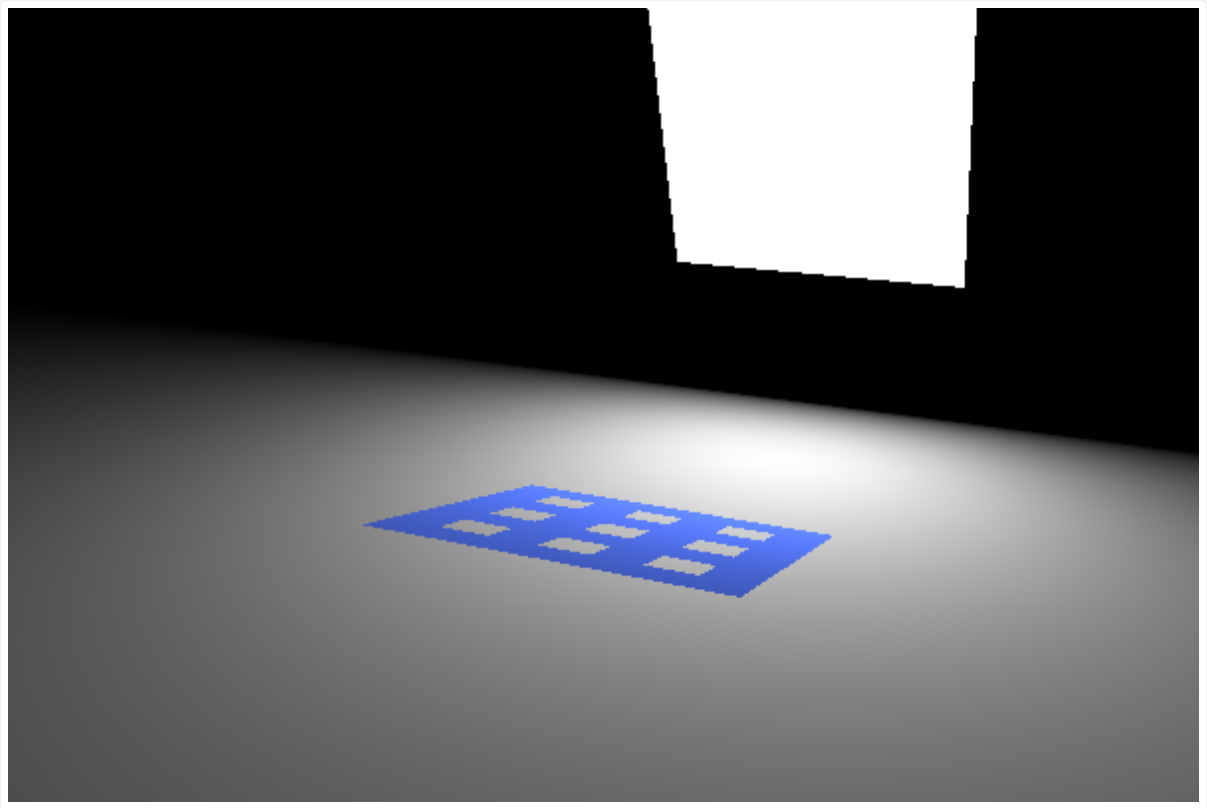


Figure 5: *Analytic (unshadowed) direct illumination*

1.3 Monte Carlo surface area shading (*5 pts*)

Next, implement a Monte Carlo-based direct illumination integrator for *general mesh lights* in `PoylgonalIntegrator::renderArea()`. This function will resemble your previous `DirectIntegrator::renderArea()`, except now you will be sampling points uniformly over an entire triangle mesh. The function `sampleEmitterPosition()` is provided for you. To sample a point, you must declare a few output variables which will then be passed (by reference) and populated by the sampling function:

```
v3f pe;    // Point on emitter
v3f ne;    // Surface normal at point
float pdf; // PDF of choosing point
sampleEmitterPosition(sampler, em, ne, pe, pdf); // Sample mesh uniformly
```

The remaining of this integrator should work exactly as before: evaluate the BRDF, convert the measure from area to solid angle before computing and returning the outgoing radiance estimate. Query the `m_traceShadows` variable to determine whether you should compute *unshadowed* or *shadowed* direct illumination (i.e., and, so, whether shadow rays need to be traced).

After completing your implementation, render the test scene `grid_area_offline.toml` in `data/a4/grid/tinyrender` and compare against the reference images:

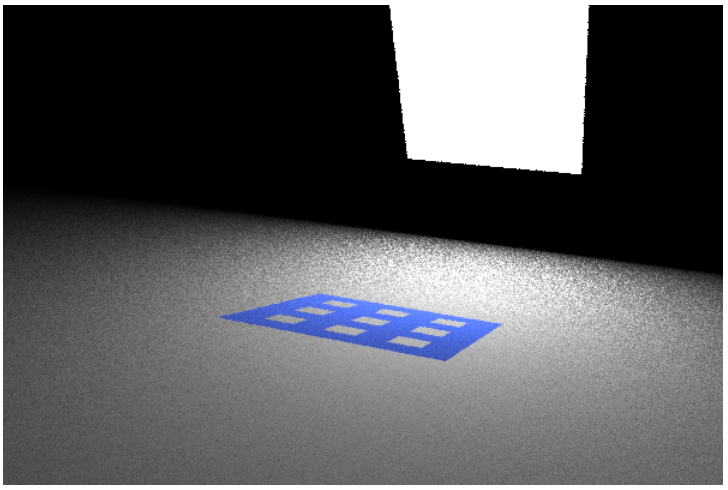


Figure 6: *Area direct illumination without shadows*

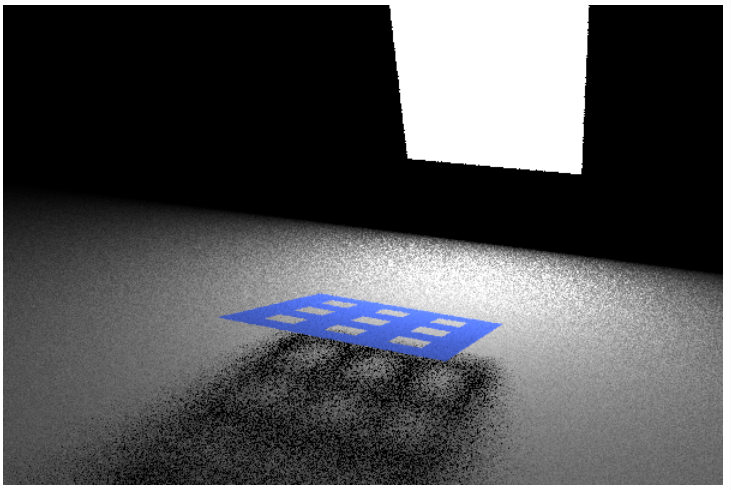


Figure 7: *Area direct illumination with shadows*

1.4 Shading with Control Variates (*15 pts*)

You will now implement a Monte Carlo estimator that combines your analytic (unshadowed) diffuse integral formulation with your stochastic direct illumination estimator. The closed-form expression you implemented in Part 1.2 provides an exact solution for unshadowed diffuse surfaces lit by polygonal light sources. Your MC estimator from Part 1.3 resolves shadows but can exhibit high variance. Your goal is to use the analytic shading as a control variate to reduce variance in your final estimator.

Recall the concept of control variates: say we want to integrate $I = \int_\Omega h(x)\,\mathrm{d}x$ using Monte Carlo,

$$I \approx \frac{1}{N}\sum_{i=1}^{N}\frac{h(X_i)}{p(X_i)}, \tag{5}$$

and we have access to a function $g(x)$ that resembles h and has an efficient (e.g., closed-form) method to evaluate its integral $G(\Omega)$. We can refer to $g(x)$ as the *control variate* in the following modified estimator of I:

$$I \approx \alpha G + \langle H - \alpha G \rangle = \alpha G(\Omega) + \frac{1}{N} \sum_{i=1}^{N} \frac{h(X_i) - \alpha g(X_i)}{p(X_i)}. \tag{6}$$

If $g(x)$ is correlated with the integrand $h(x)$ then this new Monte Carlo estimator has less variance compared to the original MC estimator. Here, the parameter $\alpha \in [0, 1]$ can allow us to modify the effect of the control variate.

Complete the implementation of `PolygonalIntegrator::renderControlVariates()` in `src/integrators/polygonal.h`. Specifically for this example, G is the analytic diffuse unshadowed direct illumination integral we previously derived and implemented, and h and g are the *shadowed* and *unshadowed* direct illumination integrands. Computing G will be identical to how you compute the shading in `PolygonalIntegrator::renderAnalytic()`. Implement `PolygonalIntegrator::estimateVisDiff()` to compute the estimator $\langle H - \alpha G \rangle$. Query the `m_alpha` variable to modulate the strength of the control variate appropriately, and the `m_visSamples` variable to set the number N of numerical MC samples to draw. You should continue to draw your samples uniformly according to the surface area of the emitter.

Once your implementation of these two functions is complete, render the test scene `grid_cv_offline.toml` in `data/a4/grid/tinyrender` and compare with the reference below.

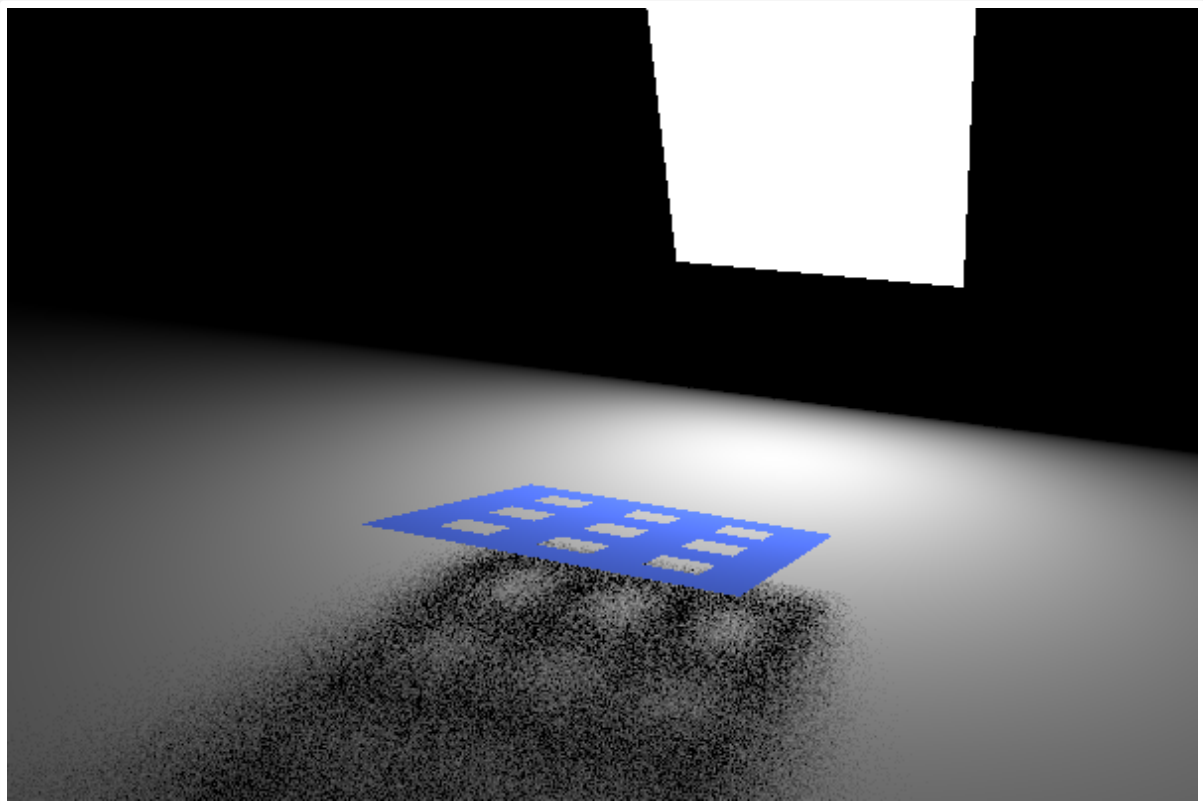


Figure 8: *Direct illumination using control variates*

Note how control variates significantly improve the (mostly) unshadowed regions of rendered image.

2 Real-time Rendering (*40 pts*)

You will implement a *hybrid* renderpass that combines both real-time and offline techniques. To complete the following tasks, you should first implement all the offline tasks in Part 1.

Concretely, you will implement a real-time loop (i.e., with the appropriate shaders) that computes the analytic illumination from a polygonal light, G, at interactive rates. Then, when the user presses the spacebar, the renderpass will run an offline pass (with 1 SPP) to estimate visibility and apply the difference $\langle H - \alpha G \rangle$ to αG. In other words, your shader will first compute unshadowed analytic direct illumination and then, when queried, it will composite in a coarse ray-traced soft shadows atop the real-time rendered shader result (using control variates).

To perform the compositing, you will get experience populating OpenGL texture buffers and implementing a *composition post-process pass* to add the appropriate compensation value to αG in a fragment shader. Each subsequent key press of the spacebar will perform an additional offline pass (with 1 SPP again) and the estimator for the visibility compensation will be updated (i.e., with a moving average) before appropriately recompositing the updated result. This will allow a user to refine the shadowing estimate each time they press the spacebar. When the camera moves, the compensation estimator will be reset to zero, removing all shadows from the (new view of the) scene.

In order to detect when the spacebar is pressed, you first need to modify your real-time rendering loop from A1 ub `src/core/renderer.cpp`:

```
renderpass->updateCamera(e); // Old line
renderpass->handleEvents(e); // New line
```

The function `renderpass->handleEvents(e)` calls `renderpass->updateCamera(e)`, so everything from past assignments will continue to work. Here, you will implement your own event handler to detect the spacebar press and add the shadowing functionality detailed above.

2.1 Unshadowed Diffuse Polygonal Shading Pass (*30 pts*)

You must complete the implementation of several functions in `src/renderpasses/polygonal.h`, each of which has `TODO`s and comments incorporated to help guide your implementation. We detail the steps, below.

Renderpass Initialization

Begin by implementing the `PolygonalPass::init()` function which requires that you initialize several variables for the renderpass. During initialization, iterate over the triangles of the emitter and store its vertices into the corresponding vector.

Keyboard Event Handling

Next, implement `PolygonalPass::handleEvents()` to handle various keyboard events. This function will rely on the SDL interface to detect when the camera moves or when the spacebar is pressed. When the point of view changes, clear the shadow buffer using `PolygonalPass::clearShadows()`. When the spacebar is pressed, add the ray-traced shadows using `PolygonalPass::addShadows()`;

Shadow Computation

The function `PolygonalPass::addShadows()` forms the core of your polygonal renderpass. You will compute the difference estimate for the control variate for each pixel by constructing a ray from the camera, intersecting it with the scene, and then calling the `polygonalIntegrator->estimateVisDiffRealTime()` routine. This last function will rely on the methods you developed for the offline control variate implementation, and it returns the per-pixel difference estimate. You must store each component of this vector in the corresponding data structure, `cvTermData`.

If it is not the first time that the spacebar was pressed (without moving the camera), `cvTermData` should be updated so that the contributions calculated at every press of the spacebar are averaged together: implement a moving average using the `samplesAccumulated` variable.

> The difference estimate $\langle H - \alpha G \rangle$ is stored as a texture buffer so it can be accessed/used in your fragment shader. Texture data **cannot be negative** but all your computed values will be negative. Therefore, you should store the *negative* of the estimate in the array. Remember to also take this into account when updating the moving average.

Finally, bind the texture and fill/update the buffer with the data with the difference estimate you computed. Refer to `PolygonalPass::clearShadows()` for an example of how this process works.

Hybrid Rendering

Finally, `PolygonalPass::render()` requires you to check whether each object is an emitter or diffuse, and then to bind the appropriate uniforms for each shader. This will be very similar to how the uniforms in Assignment 2 were handled. Note that all of the variables you should need for the uniforms are *already declared* in the `src/renderpasses/polygonal.h` constructor. Additionally, you must use the uniforms that are created for you in `object` for the diffuse albedo.

2.2 Fragment Shaders (*10 pts*)

The vertex shader `src/shaders/polygonal.vs` is provided to you and you do not have to modify anything in this file.

The fragment shader `src/shaders/polygonal.fs` has all uniforms and variables needed to proceed. The computation for the αG term will resemble your implementation of it in the offline task. Iterate through the `emitterVertices` uniform and reconstruct the vertices for each triangle before calling `getEdgeContrib(v1, v2, pos)`. The ordering of the vertices will match how you set up `emitterVertexData` in your polyonal renderpass.

Textures in OpenGL are accessed through the built in `texture(tex, texCoords)` function, where `tex` has been declared as a `sampler2D` in the fragment shader. Textures are indexed from the *bottom left corner* and `texCoords.x` and `texCoords.y` should both be in the range $[0, 1]$. Values outside this range can be used if tiling of the texture if desired. To get the coordinates of the texture to sample, use `gl_FragCoord` which contains the screen space coordinates of the fragment to be shaded. The variables `gl_FragCoord.x` and `gl_FragCoord.y` are integers in the range $[0, W-1]$ and $[0, H-1]$, where W, H are the screen width and height, respectively. You must map them to the correct range when performing texture lookups. Recall that you would have stored the *negative* value of your difference estimate in the texture.

When completed correctly, your unshadowed real-time rendering for the grid scene should look identical to your offline rendering. When the spacebar is pressed, shadows should appear "one sample at a time, per pixel". More presses should improve the quality of your shadows. You can move the camera with the `WASD` keys on your keyboard to navigate the scene.

3 Bonus: Analytic Unshadowed Phong Shading (*10 pts*)

The bonus tasks involve extending the unshadowed analytic shading routines to support *Phong* BRDFs. Specifically, you will implement an analytic solution to the unshadowed Phong-only direct illumination due to an area light source with uniform emission:

$$L_o(\mathbf{x}, \omega_o) = \frac{\rho_s (n+2)}{2\pi} \int_{\mathcal{H}^2} L_e(\mathbf{x}, \omega_i) \, \lfloor \omega_r \cdot \omega_i \rfloor^n \, \cos\theta_i \, d\omega_i = \frac{\rho_s (n+2)}{2\pi} \int_{\Omega_e} \lfloor \omega_r \cdot \omega_i \rfloor^n \, \cos\theta_i \, d\omega_i, \quad (7)$$

where ρ_s is the specular albedo at $\mathbf{x}$, ω_r is the mirror reflection (about the surface normal at $\mathbf{x}$) of the view direction ω_o, and n is the Phong specular exponent. A generalization to cosine-power lobes of the analytic diffuse expression in Equation (3) is derived in an elegant paper: the necessary background, mathematical derivation, and algorithmic details are provided in Sections 6.2, 4.1 and 5.3 of the paper.

3.1 Offline Bonus Task (*10 pts*)

Similarly to Task 1.2, implement an offline integrator to compute the unshadowed direct illumination from an area emitter and a Phong BRDF, according to the formulation presented by Arvo.

You can use both `grid_cv_bonus.toml` and `mcgill_cv_bonus.toml` to test out your bonus code. Remember that this code sets the `scene.config.bonus` to true. Your code should perform the appropriate shading when this `bonus` flag is enable in either of these TOML files, reverting to the standard shading routines otherwise. Please render `mcigll_cv_bonus.toml` for submission.

What to submit

Render all the scenes in the `data/a4` 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/
      veach_color_bsdf_offline.exr
      veach_color_area_offline.exr
      veach_color_mis_offline.exr
      mcgill_analytic_offline.exr
      mcgill_area_offline.exr
      mcgill_cv_offline.exr
      mcgill_cv_bonus.exr // Optional
   realtime/
      star_analytic_realtime.exr
      mcgill_hybrid_realtime.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.