ECSE 446/546: Realistic/Advanced Image Synthesis

Assignment 2: Progressive Monte Carlo Estimation

Due: Wednesday, November 2nd, 2022 at 11:59pm EST on myCourses

Final weight: **25%**

Contents

1 Assignment Policies and Submission Process

Download and modify the standalone `Python` script we provide on *myCourses*, renaming the file according to your student ID as `YourStudentID.py`

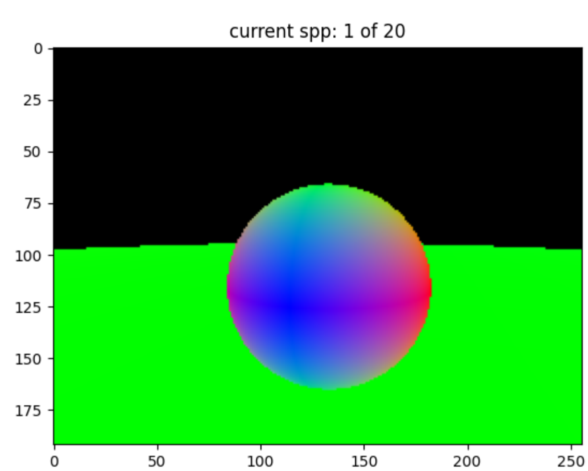
> ⓘ As usual, every new file you submit on *myCourses* will override the previous submission, and we will only grade the **final submitted file**.

1.1 Late Policy, Collaboration & Plagiarism, Python/Library Usage Rules

For late policy, collaboration & plagiarism, Python language and library usage rules, please refer to the Assignment 0 handout.

2 Pixel Anti-Aliasing and Progressive Renderer

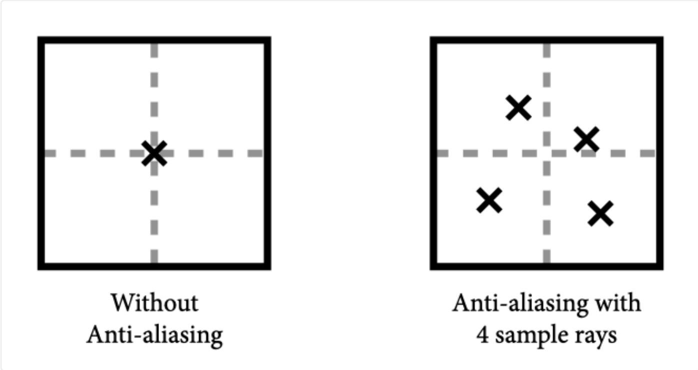
When generating your eye rays in Assignment 1, we exclusively sampled a viewing direction through the **center** of each square pixel. When the directly visible geometry varies spatially at a rate higher than our pixel grid resolution, our resulting image can suffer from so-called "jaggies" — aliasing artifacts that manifest themselves primarily at the silhouettes of visible objects:



Note the aliasing artifacts around object silhouettes.

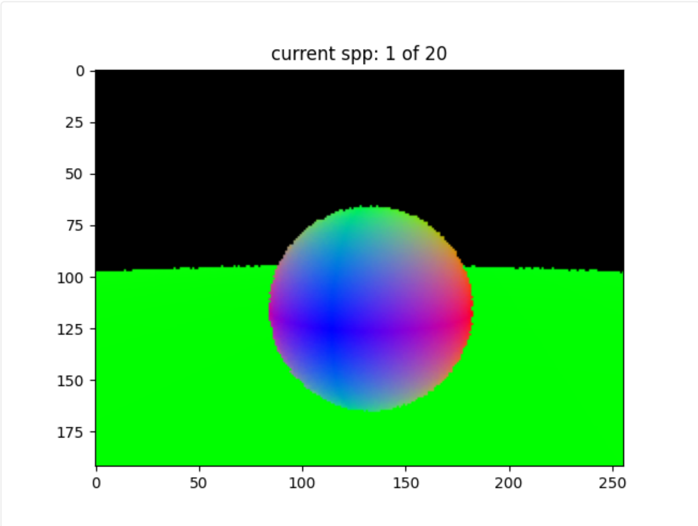
One simple strategy to eliminate these artifacts is to *anti-alias* (AA) our image, i.e., by super-sampling eye ray directions over each pixel's area.

Concretely, for each pixel, instead of considering a single ray through its center, we will average the contribution (e.g., the shading) across **many primary rays**. These jittered eye ray directions will instead be generated by picking a random location (uniformly over the area of the pixel) when generating your primary rays:



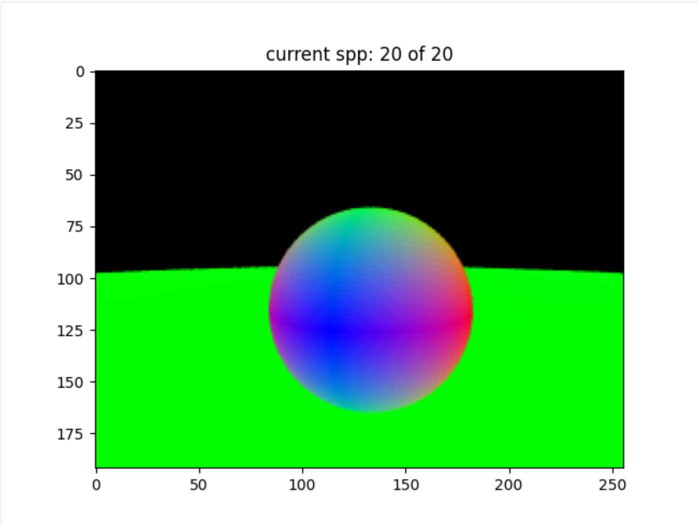
Pixel Anti-Aliasing

Tracing a single jittered eye ray into the scene will replace the structured aliasing with unstructured noise (below),



A single randomly-jittered eye ray substitutes aliasing with noise.

however, when averaging the shading result across many such jittered rays, the noise is eventually averaged away, and so too the aliasing:



Averaging 20 jittered eye rays through each pixel.

The base code provides a modified signature for the `Scene.generate_eye_rays` method, and you will augment your Assignment 1 implementation of the routine as follows: when the `jitter` is `True`, instead of generating a batch of eye rays where each ray goes through the center of each pixel, you will generate a batch where each ray goes through a random location in the area of each pixel.

Moreover, you will complete the implementation of a *progressive accumulation renderer* in `Scene.progressive_render_display` that handles the logic of iteratively computing, averaging and displaying the results of the `Scene.render` routine.

`Scene.render` generates a batch of `Scene.width * Scene.height` eye rays (with or without `jitter`ing, depending on the `Scene.progressive_render_display`'s `jitter` parameter) and computes shading at each of these shading points. For the first two deliverables, you can use the debug implementation of `Scene.render` that we furnish, which simply visualizes the primary shading points' normals.

Concretely, `Scene.progressive_render_display` expects the following arguments:

- `jitter`: `True` if the generated eye rays should be jittered at each progressive iteration (i.e., AA enabled), `False` otherwise (i.e., no AA),
- `total_spp`: the total number of samples per pixel[1] for the final rendered image,
- `spppp`: the number of *Monte Carlo integration samples* to trace **per progressive rendering iteration** (for Deliverables 3 and 4),
- `sampling_type`: a parameter that will also be passed to `Scene.render` to select between the importance sampling routines required in Deliverables 3 (for both ECSE 446 and 546) and 4 (only for ECSE 546).

[1] More precisely, the final image should have `np.ceil(total_spp / sppp)` samples, as in the base code.

> You cannot vectorize your `Scene.progressive_render_display` routine over rendering passes (since you need to display the result after each iteration); simply use a `for` loop over the individual passes.

Additional Notable Changes in the Assignment 2 Base Code

Unlike Assignment 1, the scene can now contain both sphere and mesh objects; as such, we have abstracted different geometry types using the `Geometry` class, with `Sphere` and `Mesh` subclasses.

One important distinction between how `Scene.intersect` and `Geometry.intersect` interact — compared to Assignment 1 — is that `Geometry.intersect` must now return **both** the intersection distance **and** hit point normal, for valid ray-geometry intersections.

Deliverable 1 [10 points]

- augment your implementation of the `Sphere.intersect` from Assignment 1 to now return the hit distances *and* hit point normals,
- augment your implementation of `Scene.intersect` from Assignment 1 to treat *arbitrary* geometry types, by relying on `Geometry.intersect`,
- augment your implementation of the `Scene.generate_eye_rays` from Assignment 1 to generate a jittered bundle of eye rays, when `jitter` is `True` (and the originally unjittered eye rays if `False`),
- complete the implementation of `Scene.progressive_render_display` to iteratively generate eye rays, passing them to `Scene.render` and displaying a running average of the `Scene.render` output; each call of `Scene.render` will (eventually, in Deliverables 3 and 4) rely on `sppp` when computing Monte Carlo estimates of the ambient occlusion. `Scene.progressive_render_display` does not return any values.

3 Phong Normal Interpolation

When shading triangle meshes, we reviewed three spatial shading strategies in class: flat, Gouraud, and Phong shading.

The base code's implementation of `Mesh.intersect` relies on the `gpytoolbox` library[2] to compute an ensemble of ray-mesh intersections, given a ray bundle and a loaded mesh instance. Moreover, for those rays with valid intersections, `gpytoolbox.ray_mesh_intersect` returns three numpy arrays of outputs:

1. `hit_distances`: an array with shape `(num_rays,)` of floating point ray intersection distance parameters (or `np.inf` for those rays that did not intersect the mesh),
2. `triangle_hit_ids`: an integer numpy array of size `(num_rays,)` with indices for the triangle in the mesh that was intersected (and −1 for those rays that did not intersect the mesh), and

3. barys: a numpy array of size (num_rays,3) of the Barycentric coordinates of the ray-triangle intersection (or [0,0,0] for those rays that did not intersect the mesh).

[2] You can install this library using `pip install gpytoolbox` in your Python environment.

The implementation that we furnish to you simply returns the triangle face normals (i.e., flat shading) for valid ray-mesh intersections.

You will be responsible for implementing Phong normal interpolation, relying on the intersected triangle face geometry and the Barycentric coordinates.

More precisely, given a ray that intersects a mesh, Barycentric coordinates can be used to linearly interpolate any scalar- or vector-valued signal — defined at the vertices of the triangle — over points on the face of the triangle.

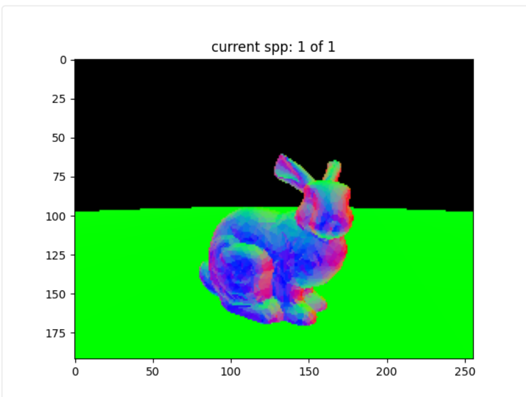
For example, given a single Barycentric coordinate $[\alpha, \beta, \gamma]$ (recalling, of course, that the vectorized output of `gpytoolbox.ray_mesh_intersect` provides an *array* of such coordinates), we can, e.g., obtain the intersection point $\mathbf{p}$ on the face of the triangle as a linear combination of the triangle's vertices, as $\mathbf{p} = \alpha\,\mathbf{v}_0 + \beta\,\mathbf{v}_1 + \gamma\,\mathbf{v}_2$.

Of more immediate use to you, we can similarly obtain the Phong-interpolated normals for a point on the face of the triangle (i.e., the ray-mesh intersection point, with associated Barycentric coordinates $[\alpha, \beta, \gamma]$), as

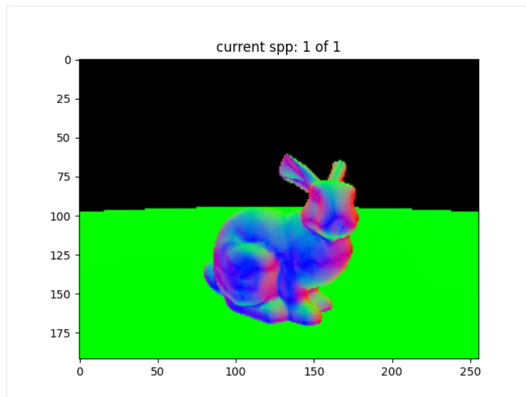
$$\mathbf{n} = \alpha\,\mathbf{n}_0 + \beta\,\mathbf{n}_1 + \gamma\,\mathbf{n}_2 \ ,$$

where $\mathbf{n}_i$ is the vertex normal at $\mathbf{v}_i$. The `gpytoolbox.per_vertex_normals` routine can be called at mesh initialization to precompute the per-vertex normals on the mesh.

Per-Face Normals



Per-Vertex Normals



> **Deliverable 2 [10 points]**
>
> Modify the `Mesh.__init__` and `Mesh.intersect` routines to compute and return Phong-interpolated normals for those rays that intersect the mesh.

4 Ambient Occlusion and Importance Sampling

Ambient occlusion (AO) is a special case of direct illumination. You will modify the `Scene.render` routine to compute a Monte Carlo integral estimate of AO with `num_samples` samples, and using the appropriate sampling strategy (`sampling_type`).

ECSE 446 students will only implement uniform sampling (i.e., `sampling_type = UNIFORM_SAMPLING`), whereas ECSE 546 students will additionally implement cosine importance sampling (i.e., `sampling_type = COSINE_SAMPLING`).

Recall from the lectures that the general N-sample Monte Carlo estimator for the AO reflection equation is

$$L_r(\mathbf{x}) \approx \frac{\rho}{\pi N} \sum_{j=0}^{N} \frac{V(\mathbf{x}, \omega_j) \max\left(0, \mathbf{n} \cdot \omega_j\right)}{p(\omega_j)} \quad \text{with} \quad \omega_j \sim p(\omega) \ ,$$

where visibility V can be evaluated by tracing a shadow ray, and the diffuse albedo ρ is encoded in the `Geometry.brdf_params` property.

4.1 Uniform Spherical Sampling

Implement a uniform MC sampler, with your choice of hemispherical or spherical sampling (and their associated PDF and sampling routines), to estimate the AO integral.

Recall from the lecture that you can generate a uniformly-sample ray direction $\omega = (\omega_x, \omega_y, \omega_z)$ on the sphere using two canonical random variables ξ_1, ξ_2 — computed using `np.random.rand` — and the following transformation:

$$\omega_z = 2\xi_1 - 1 \qquad r = \sqrt{1 - \omega_z^2} \qquad \phi = 2\pi\xi_2$$

$$\omega_x = r\cos\phi \qquad \omega_y = r\sin\phi$$

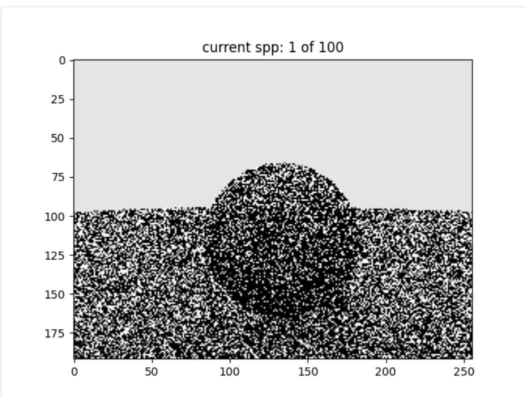
`Scene.render`'s `sampling_strategy` parameter denotes the sampling strategy, and the number of samples N as `num_samples`.

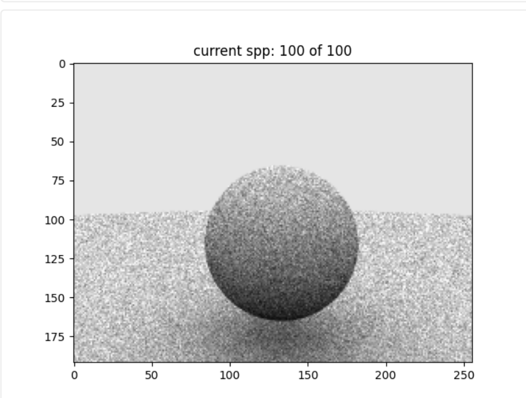
In `Scene.progressive_render_display`, you will pass `spppp` (*samples per pixel per pass*) as the `num_samples` for each `Scene.render` call in the progressive rendering loop.

Deliverable 3 [10 points]

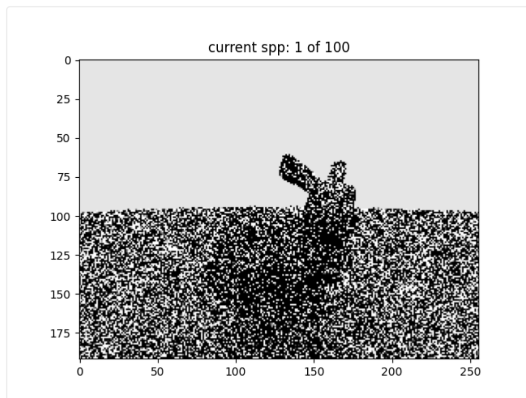
- Complete the implementation of the `Scene.render` function to support the `sampling_type == UNIFORM_SAMPLING` scenario: compute a `num_samples`-sample Monte Carlo estimator of the AO reflection equation using uniform sampling.
- Ensure that your `Scene.progressive_render_display` routine properly updates the render view, displaying progressively improving integral estimates (i.e., with progressively-increasing total sample counts.)

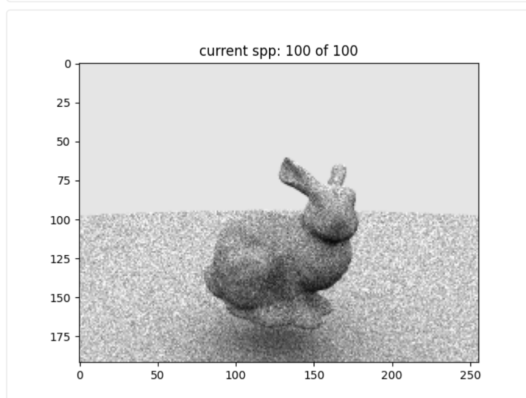
Uniform AO Estimator (sphere scene)





Uniform AO Estimator (bunny scene)





ECSE 546 Students Only

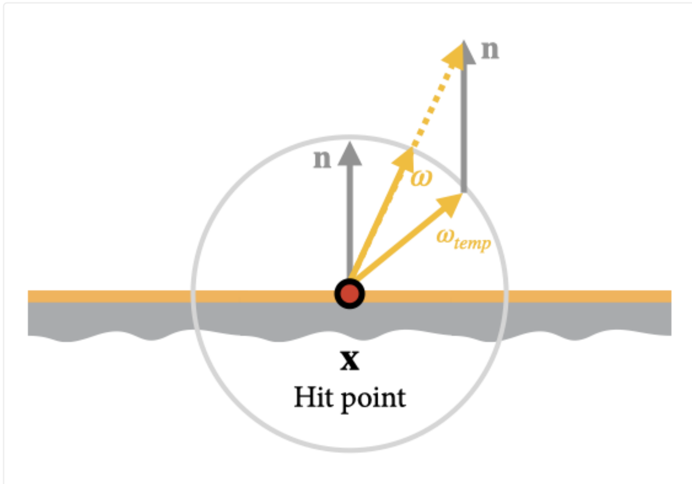
4.2 Cosine Importance Sampling

In addition to the uniform Monte Carlo estimator, ECSE 546 students will implement cosine importance sampling.

Recall that cosine importance sampling can be more effective than uniform sampling, in the AO setting. Here, the sampling PDF is $p(\omega) = (\cos\theta)\big/\pi$, and the specialized Monte Carlo estimator simplifies expression simplifies to

$$L_r(\mathbf{x}) \approx \frac{\rho}{N} \sum_{j=0}^{N} V(\mathbf{x}, \omega_j) \;\; \text{with} \;\; \omega_j \sim p(\omega) \,.$$

There are many strategies for sampling cosine-distributed points (about the coordinate system defined by the surface normal $\mathbf{n}$ at $\mathbf{x}$). Perhaps the simplest such approach is to first generate a uniform spherical sample (ω_{temp}), then add it to the surface normal, before finally normalizing the result as ω:



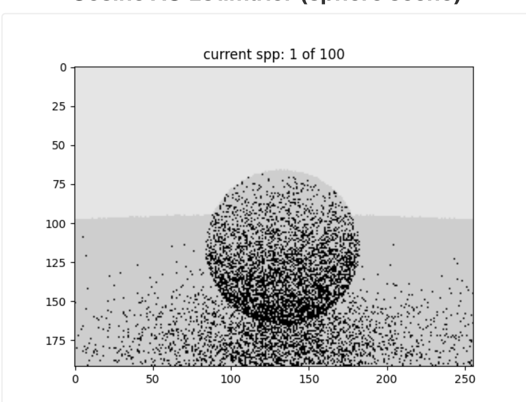
A Voodoo Trick for Sampling Directions with Cosine-density in the Shading Coordinate System.

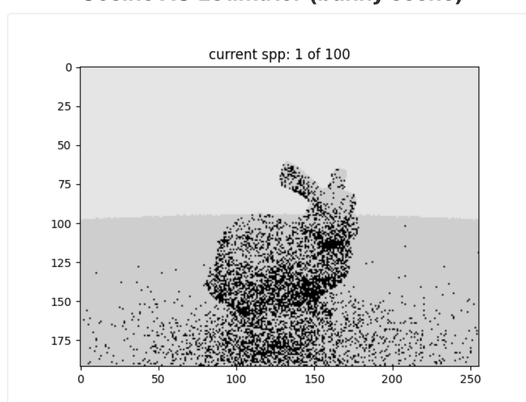
The resulting direction will have cosine density aligned about the appropriate, aforementioned coordinate system, i.e., $\omega \sim p(\omega)$.

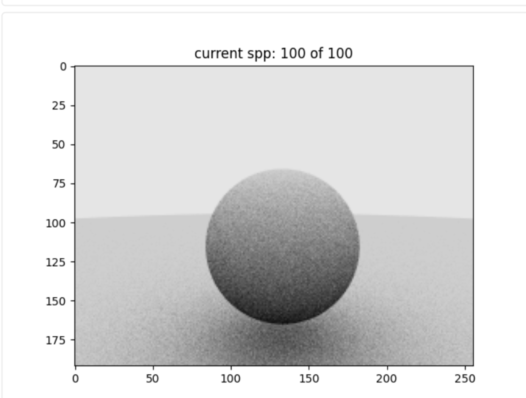
> **Deliverable 4 [10 points]**
> Augment the `Scene.render` function to treat the additional `sampling_type == COSINE_SAMPLING` case, computing a `num_samples`-sample Monte Carlo estimator of the AO reflection equation using cosine importance sampling.

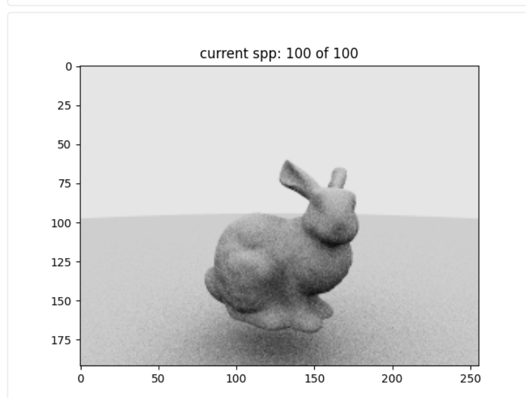
Cosine AO Estimator (sphere scene)

Cosine AO Estimator (bunny scene)









5 You're Done!

Congratulations, you've completed the 3rd assignment. Review the submission procedures and guidelines at the start of the Assignment 0 handout before submitting the `Python` script file with your assignment solution.