ECSE 446/546: Realistic/Advanced Image Synthesis

Assignment 1: Ray Tracer & Basic Shading

Due: Tuesday, October 3rd, 2023 at 11:59pm EST on myCourses

Final weight: **20%**

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`

For example, if your ID is **234567890**, your submission filename should be **234567890.py** and should include all the entirety of your solution submission for this assignment, according to the instructions below.

> ⓘ Every time you submit a new file on *myCourses*, your previous submission will be overwritten. We will only grade the **final submitted file**, so feel free to submit often as you progress through the assignment.

1.1 Late Policy, Collaboration & Plagiarism, Python Language and Library Usage Rules

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

2 Rays-Sphere Intersection

Consider a ray defined by its origin $\mathbf{o}$, direction $\mathbf{d}$ and parametric distance t. A point $\mathbf{x} = \mathbf{r}(t)$ along the ray can be expressed as

$$\mathbf{x} = \mathbf{r}(t) = \mathbf{o} + t\mathbf{d}\,.$$

We define a sphere by its center $\mathbf{c}$ and radius r, with an implicit inside-outside function for its surface as

$$f_{sphere}(\mathbf{x}) = ||\mathbf{x} - \mathbf{c}||_2^2 - r^2\,.$$

We can solve for intersection point(s) along a ray and on the surface by substituting the parametric equation of a ray for $\mathbf{x}$ and solving for t in $f_{sphere}(\mathbf{r}(t)) = 0$ as

$$||\mathbf{o} + t\mathbf{d} - \mathbf{c}||^2 - r^2 = 0\,.$$

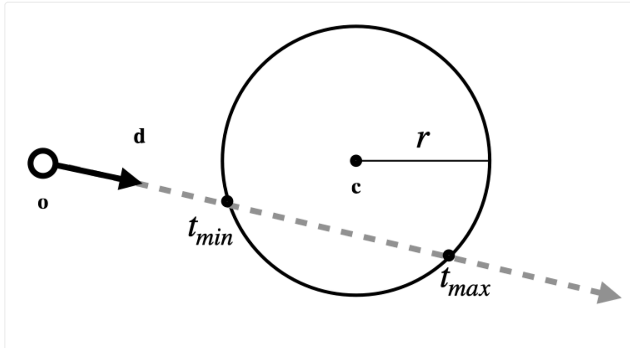
The solutions to a quadratic equation in t of the form $At^2 + Bt + C = 0$ are

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} = \frac{-B \pm \sqrt{\Delta}}{2A}$$

where $\Delta = B^2 - 4AC$ is the discriminant.

Depending on the value of the discriminant Δ, there are three possible outcomes:

1. if $\Delta < 0$, a real solution does not exist and a ray does not intersect the sphere; here, we set and return $t =$ `np.inf`,
2. if $\Delta = 0$, then the ray intersects the sphere at only one point, and we simply set and return the singular distance t, and
3. if $\Delta > 0$, then the ray intersects the sphere at two points; for rays that originate from the camera and point towards the scene, the intersection point we care about is the one that is closest to the camera, and so we set and return $t = t_{min}$ (the smallest intersection test greater than 0, *in world space*).



Ray-Sphere Intersection

> ☑ **Deliverable 1 [2.5 points]**
> Complete a vectorized `numpy` implementation of the `intersect` method in the `Sphere` class. The method takes as input a **bundle of rays** (`Rays`), intersects the rays with the (single) sphere, and returns a 1D `numpy.array` representing the appropriate intersection distances t for the rays in the bundle.

3 Eye Rays Generation

Eye rays all originate at the camera location and traverse in directions towards the *center* of pixels on an image plane located one unit away (in world space) from the camera.

In order to form the complete geometric setup necessary to compute the eye ray directions, we require an appropriate coordinate system, centered at the camera, that orients the **central viewing axis**, and the vertical and horizontal **extents** of the (clipped) image viewing plane. The diagrams below serve to complement our parameterization of this coordinate system.

This assignment will only treat a *single eye ray through the* ***center*** *of each pixel*. In order to obtain the world-space origins and directions of each eye ray, we will perform a series of coordinate system transformations:

- Starting from *2D normalized device coordinates* (NDC) on the image plane, we specify the (continuous, per-pixel centered) pixel coordinates such that the corners of the NDC plane lie at $(\pm 1, \pm 1)_{ndc}$[1]. Do not forget to shift the 2D pixel coordinate to lie in the center of each stratum, using the width (`self.w`) and height (`self.h`) in the `Scene` class to appropriately discretize the NDC plane. The 2D coordinate axes, x_{ndc} and y_{ndc}, for the NDC plane are illustrated below on the left.
- A perspective camera is parameterized by its:
  - location (position) in world-space, eye_w (`self.eye` in `Scene`),
  - a point it is looking at in world-space (`self.at` in `Scene`),
  - a direction specifying where "up" is for the camera, up_w, also expressed in world-space (`self.up` in `Scene`), and
  - the camera's vertical field of view angle (fov) expressed in degrees, (`self.fov` in `Scene`); note that the horizontal fov can be treated using the aspect ratio (`self.w/self.h`).
- While, given the aforementioned camera parameters, one can proceed to directly obtain the world-space eye ray directions[2], it is often more convenient to first express these directions in a coordinate system centered at the camera, before finally transforming them into world-space. The right side of the diagram below illustrates this *camera coordinate system*, with the camera at the origin looking down the (positive) z-axis (z_c); here, the x- and y-axes in the coordinate system are defined as $x_c = up_w \times z_c$, with z_c as the (normalized) vector from the eye to look-at point, and $y_c = z_c \times x_c$. In this coordinate system, the image plane is one unit away from the eye along z_c.
  - It can be useful to think about how the same points/directions are expressed across these three coordinate systems, for example:
    - the center of the (2D) NDC plane coincides with the (3D) point one unit along the camera coordinate system's z-axis *and* with the (3D) point from the eye towards the look-at direction in world-space, i.e., $(0,0)_{ndc} = (0,0,1)_c = (eye_w + \texttt{self.at})_w$.
- In the camera coordinate system, you can use the trigonometric relationships formed by the vertical (and horizontal) fovs, and the unit distance between the origin and the center of the image plane, to transform the NDC-space (centered) pixel coordinates to camera-space eye ray points or directions.
- Finally, you can transform these points/directions from camera-space to world-space using, e.g.,

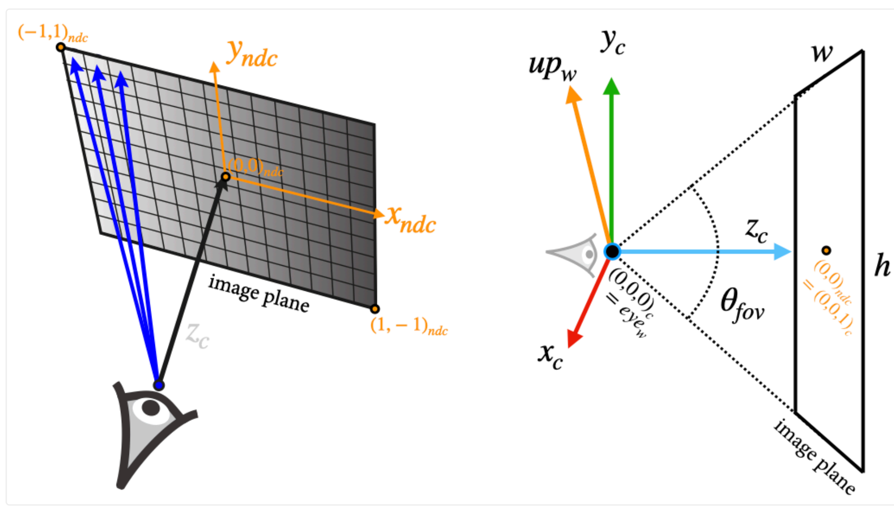
$$\begin{bmatrix} | & | & | & | \\ x_c & y_c & z_c & eye_w \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

> ⓘ Be mindful of how you treat homogeneous coordinates, especially for points post-transformation: their w component should be normalized to 1, and it's good practice to also normalize direction vectors.

[1] We will use the subscripts $\square_{ndc}$, $\square_c$ and $\square_w$ to distinguish between coordinates in the NDC-, camera- and world-spaces.

[2] All the eye rays have the same origin in world-space, eye_w (`self.eye` in `Scene`), for a perspective camera.

> ☑ **Deliverable 2 [10 points]**
> Complete a vectorized `numpy` implementation of the `generate_eye_rays` function in`Scene`, which has no input parameters but — instead — relies on `Scene` member variables (as detailed, above). This function should return a new instance of the `Rays` class, containing your eye ray bundle.



NDC and Camera Coordinate Systems.

4 Scene Intersection

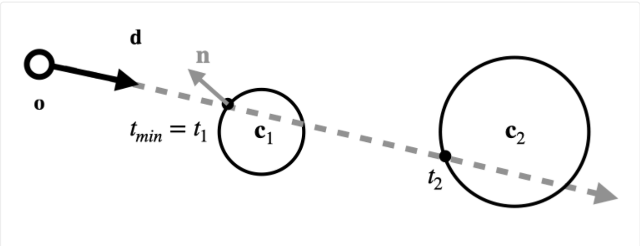
In addition to the camera parameters, the `Scene` class contains all of the scene geometry, their material properties, and light source information. For this assignment, your scene comprises a list of `Sphere` objects. When testing for ray (bundle) intersection on the entire scene, you will iteratively intersect your ray (bundle) with each sphere in the scene — here, given the few number of spheres, you can use a `for` loop if you don't want to vectorize this aspect of the computation.

When doing so, for each ray, we can obtain the nearest intersection distance using the `intersect` routine in `Sphere` (Deliverable 1), however we will compute and output additional information about the closest intersection point:

- `hit_distances`: the distance between the hit point location and the ray origin location,
- `hit_normals`: the direction of the unit normal at the hit point surface intersection, and
- `hit_ids`: the IDs (indices) of each sphere that was intersected by the ray bundle.

For rays that do not intersect any objects in the scene, their corresponding index in these outputted fields should be set to `np.inf`, `(np.inf, np.inf, np.inf)`, and `-1`, respectively.

> ⓘ Careful: in `Python`, `-1` is a *valid* array index, and so if you index your sphere list with the `hit_ids`, you will also get (erroneously) valid data for those ray indices in your bundle that *do not* hit any objects.

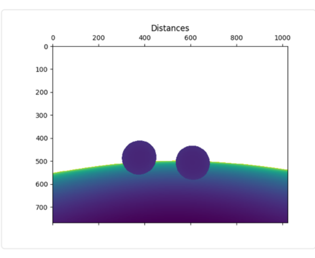


Ray Scene Intersection

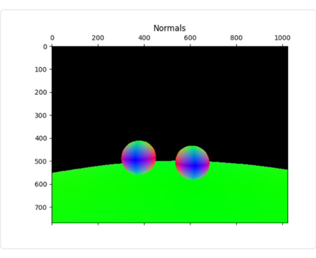
> ☑ **Deliverable 3 [2.5 points]**
> Complete a vectorized `numpy` implementation of the `intersect` method in the `Scene` class. The method takes in a bundle of rays (one `Rays` object), intersects the rays with the spheres in the scene, retains the nearest intersection (if any) and returns a tuple representing the hit distances (1D `numpy.array`), hit normals (2D `numpy.array`), and hit object IDs (1D numpy.array`).

In the test code, we visualize the intersection distances, normals and IDs for a simple test scene. The reference images are as follows:

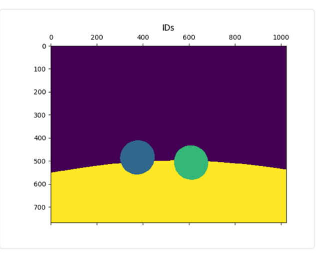
Intersection Distances



Intersection Normals



Intersection IDs



Congratulations: you've solved the primary visibility problem using ray tracing!

5 Shading

Fresh off your victory over the primary visibility problem, you're ready to do some basic shading. The routine you implemented in Deliverable 3 either directly returns — or can be used to compute — hit point locations $\mathbf{x}$ in the scene (for your bundle of eye rays), the hit distances t from the eye, the hit normals $\mathbf{n}$, and the corresponding scene object IDs: this is everything you'll need to implement basic shading using a diffuse material.

It's wise to proceed in two stages: first, computing the unshadowed shading, and then the shadowed shading.

5.1 Unshadowed Shading

This assignment assumes every sphere is diffuse with an albedo ρ. This property is stored in the first three elements of the four-element `self.brdf_params` property of the `Sphere` class; you can safely ignore the fourth element of `self.brdf_params` as it'll be used in later assignments.

When shading a scene with (potentially many) directional light sources, each with light direction $\mathbf{l}$ and intensity Φ, recall that the diffusely reflected light intensity is

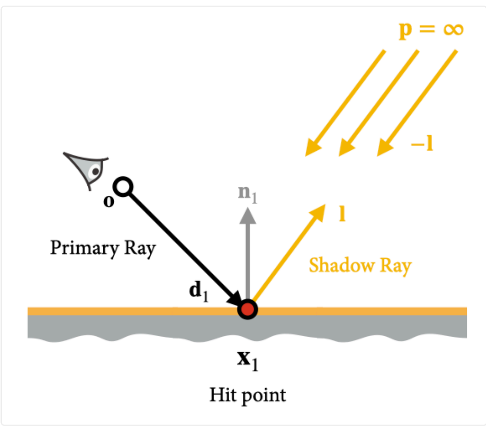
$$L_d = \frac{\rho}{\pi}\Phi \max(0, \mathbf{n} \cdot \mathbf{l})\,.$$

The total reflected light intensity is just the sum of the reflected light intensities due to each light. You can access light source parameters in the `Scene` instance. Note that, as with intersecting individual sphere objects, you are permitted to use `for` loops over the lights if you don't want to vectorize that component of the code.

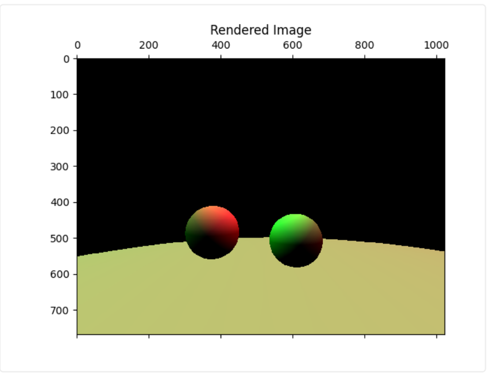
> ☑ **Deliverable 4 - part 1**
> Complete the first step of the `shade` method in the `Scene` class - unshadowed shading. The method takes a bundle of eye rays (`Rays`), intersects the rays with the scene (i.e., uses Deliverable 3), and computes the output image intensity by applying and accumulating the L_d formula for every light. Your routine should return the rendered scene as `numpy.array` of shape `(height, width, 3)`.

We provide a test scene and its unshadowed rendering for reference, below.

Unshadowed Directional Light



Rendered Unshadowed Directional Light



ECSE 546 Students Only

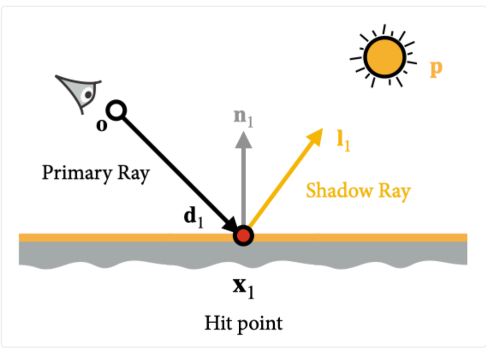
In addition to directional lights, ECSE 546 students will implement shading for point lights. Here, lights are parameterized by the light **position**, not its direction. Recall that, in this setting, light intensity falls-off according to distance d between the hit point and the light source location, as

$$L_d = \frac{\rho}{\pi}\left(\frac{\Phi}{4\pi d^2}\right)\max(0, \mathbf{n} \cdot \mathbf{l})\,.$$

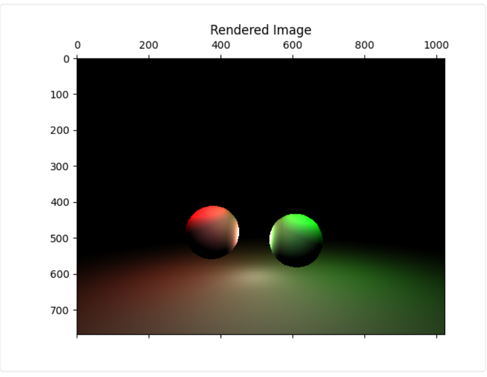
> ☑ **Deliverable 5 - part 1 (ECSE 546 only!)**
> Augment the `shade` method in the `Scene` class to implement unshadowed shading for point lights. As before, the method takes in a bundle of eye rays (`Rays`), intersects the rays with the scene, and computes the image intensity by applying and accumulating the modified L_d formula for every light. Your routine should return the rendered scene as `numpy.array` of shape `(height, width, 3)`.

Our mainline provides a (commented out) test scene that ECSE 546 students should uncomment and use. Its expected unshadowed output is included for your reference, below.

Unshadowed Point Light



Rendered Unshadowed Point Light



5.2 Shadowed Shading

To finalize this deliverable, you will render shadows by tracing an additional shadow ray bundle.

Summarizing the lecture content on this task, you will construct and trace a `Rays` bundle with your shadow rays, from each (valid) hit point and towards a singular light source (remember, compute shadowed shading for a single light source, then accumulate shading results over all the light sources). For directional lights, if the shadow ray intersects an object, then the hit point that we are shading is in the shadow, and so we set its contribution to 0.

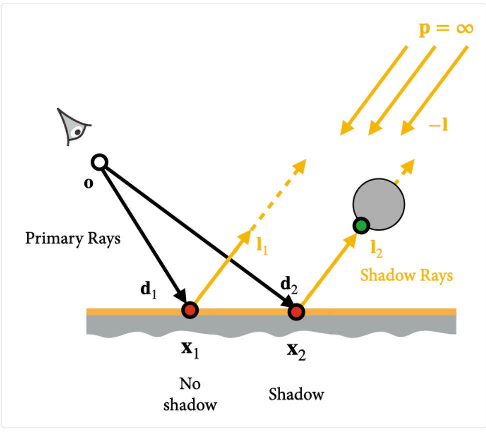
> ⓘ **Avoid Shadow Acne**: shadow acne can cause unexpected noise in your renderings due to numerical precision. You will implement two measures to avoid this:
>
> 1. when intersecting a bundle of rays with a sphere, we only register a ray hit if the hit distance is greater than a small threshold (`Sphere.EPSILON_SPHERE = 1e-4`), and
> 2. when constructing your shadow rays, we offset the shadow ray origins by a small distance (`shadow_ray_o_offset = 1e-6`) along the hitpoint normal direction.

Note again that, while you are allowed to use `for` loops over lights, your shadow ray **construction** and **tracing** should be vectorized.

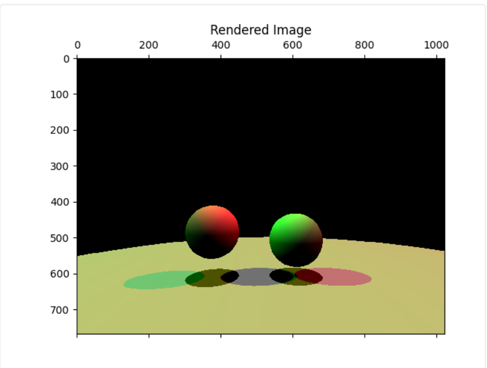
> ☑ **Deliverable 4 - part 2 [10 points]**
> Complete the `shade` method by adding shadowed shading functionality. Also, update the `intersect` method in the `Sphere` class to avoid shadow acne problems.

Our test scene reference shading output is included below.

Shadowed Directional Light



Rendered Shadowed Directional Light

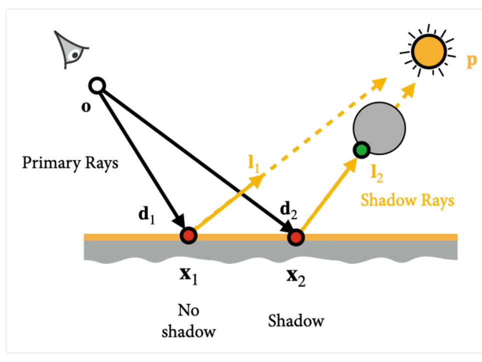


ECSE 546 Students Only

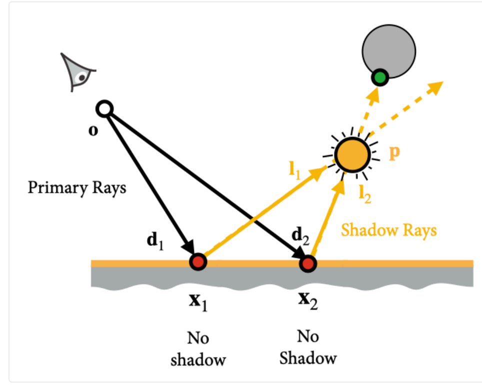
For point lights, determining whether a point is in the shadow requires some extra work compared to directional lights. After intersecting shadow rays in the scene, we need to additionally test whether the distance between the shading hit point and any shadow hit point is *less* than the distance from the shading hit point to the light.

The diagram below exhaustively itemizes the various shadowed vs. unshadowed scenarios.

Shadowed Point Light

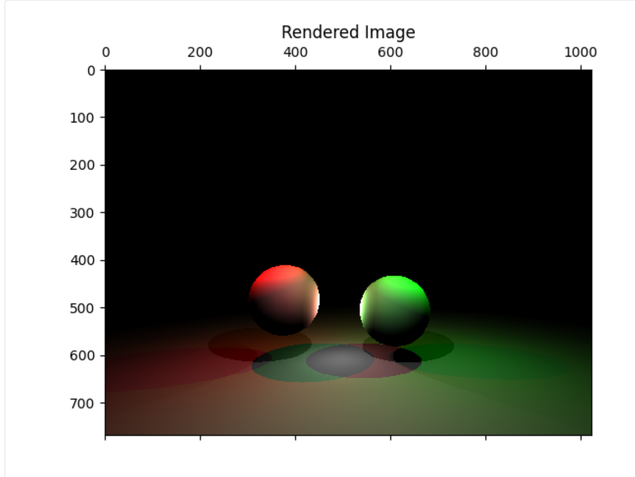


Unshadowed Point Light but Shadow Ray Hit



> ☑ **Deliverable 5 - part 2 (ECSE 546 only!) [10 points]**
> Augment the `shade` method to implement shadowed shading for point lights, returning the rendered scene as a `numpy.array` of shape `(height, width, 3)`.

As before, we include a reference shadowed rendering of our test scene, below.



Rendered Shadowed Point Light

---

6 You're Done!

Congratulations, you've completed the 1st (real) 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.