CS 480 Computer Graphics

Fall 2024

Assignment 3

Due Date: 11.08.2024, 11.59pm

Note: This is the first programming intensive assignment. It will take significantly longer to finish than the first two assignments. It's highly recommended to start early.

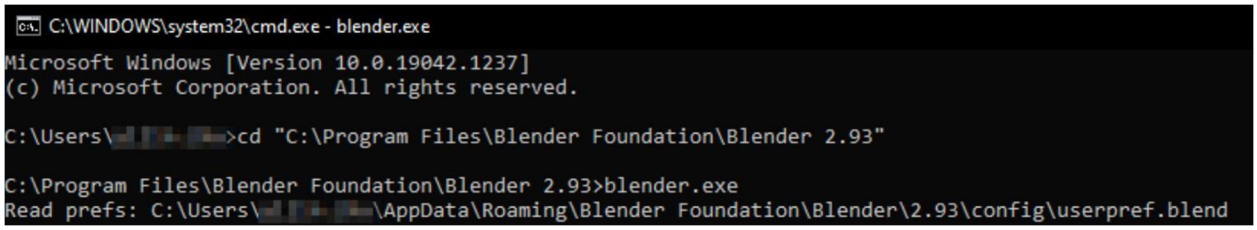
In this assignment, you will complete a ray tracer code in Python for interacting with the provided Blender scene. The scene and initial code setup, which leverages Blender's Python API, are included in `cs480-hw3.blend`. The current code lacks essential parts for rendering the scene through ray tracing, as indicated by an accompanying image outlining four critical steps. Your responsibility is to implement these missing parts, detailed as "Action: TODO" in the provided PDF (this PDF file). Notably, the essence of ray tracing—and where it surpasses scanline rendering in terms of physical accuracy—resides in the implementation of reflections and transmissions.

1. Setting up the Assignment

1.1 Starting Blender from the Command Line

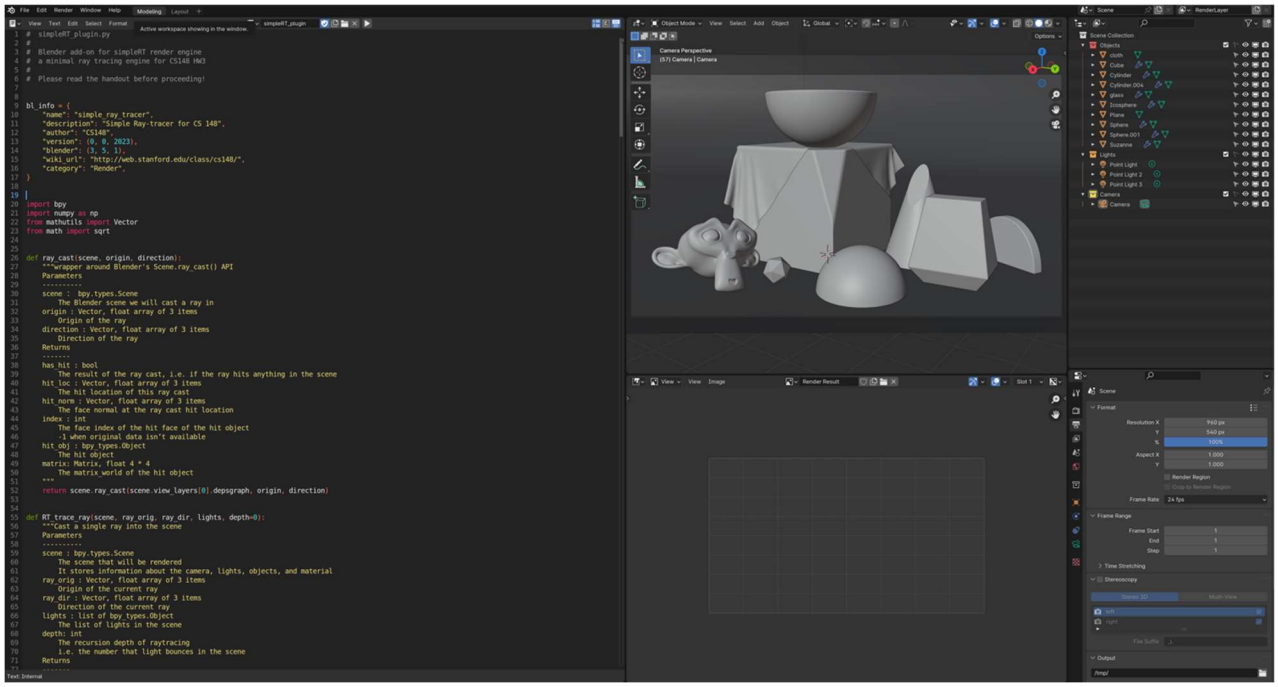
To conduct advanced Python development in Blender and enable debugging through the command line, follow these steps:

- Windows:
    1) Press Windows + R on your keyboard to open the "Run" dialog.
    2) Type `cmd` and press Enter to open the Command Prompt.
    3) Change the directory to your Blender installation folder. For example: by typing cd C:\Program Files\Blender Foundation\Blender 3.5 and pressing Enter. Adjust the path if your Blender installation is in a different location.
    4) In the Blender directory, type blender.exe and press Enter. This method allows Blender to output bugs or any other information that you print using the print() function directly to the command line.



- MacOS: Please refer to the official document linked here: https://docs.blender.org/manual/en/3.3/advanced/command_line/launch/macos.html
- Linux: Please refer to the official document linked here: https://docs.blender.org/manual/en/3.3/advanced/command_line/launch/linux.html

Now Blender will launch as usual. Open the provided `cs480-hw3.blend` file. You should see a workspace layout like the screenshot below.

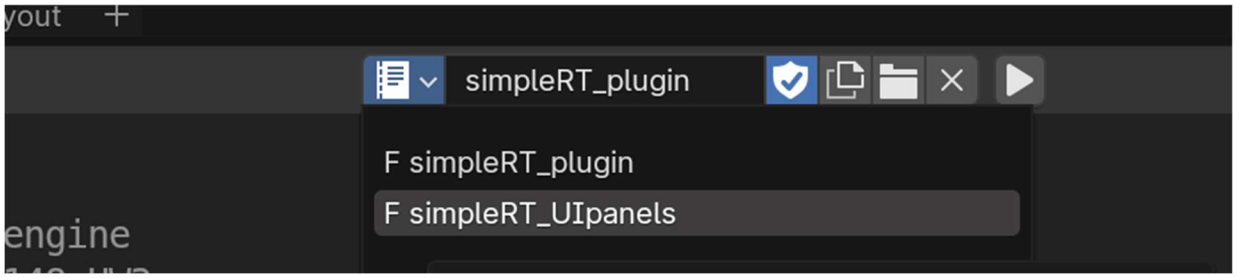


The UI is organized into three main sections:

- Left: Text Editor, where code can be edited and executed.
- Top right: 3D Viewport, for manipulating and previewing the scene.
- Bottom right: Image Editor, where the rendered image is displayed.

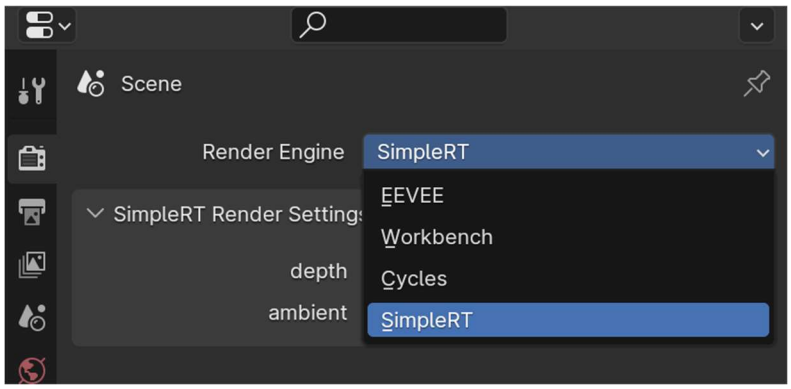
1.2 The Text Editor

Action: Switch the script to `simpleRT_UIpanels` using the dropdown button in the Text Editor and run the script.

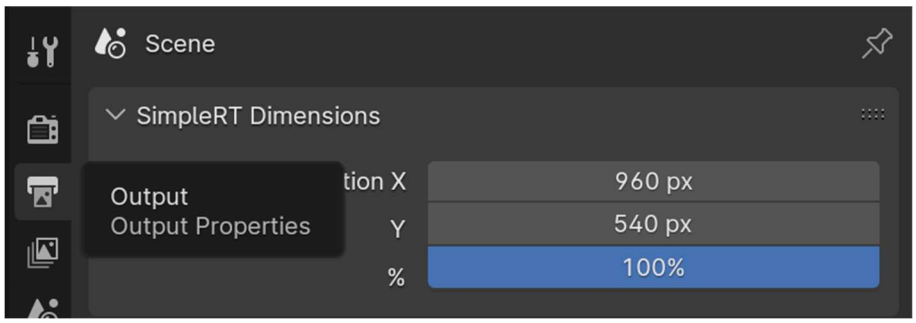


The `simpleRT_UIpanels` sets up custom properties for the renderer, including material properties and light parameters, and integrates these properties into the Blender UI through widgets for easy adjustments. If this explanation is unclear, simply remember to run the `simpleRT_UIpanels` script each time you open the `.blend` file for this homework before working with the main script.

Now switch back to `simpleRT_plugin` and run it. To check if the script works, change the Render Engine to SimpleRT via the Properties Editor, which is the new render engine added to Blender through the script execution.



Post-switch, the UI in the Properties Editor will update, revealing the "SimpleRT Render Settings" in the Render tab and the "SimpleRT Dimensions" panel in the Output tab.



If the UI still displays panels from other render engines, press F3 while in the 3D Viewport, enter "reload script", and hit Enter to refresh all plugins.

1.3 Editing and Debugging

Action: Your task involves completing the pending tasks in the `simpleRT_plugin` script. Blender offers a supportive development environment, though it may not fully match the capabilities of professional Python IDEs or text editors. Debugging primarily occurs through the command line from which Blender was launched.

As a practice, inputting nonsensical code like “asdfghjkl” into the `simpleRT_plugin` script and running it will result in an error, “NameError: name ‘asdfghjkl’ is not defined”, displayed in the command line during render attempts. Additionally, using print() statements in the script will output to the command line, aiding in debugging.

Useful Text Editor shortcuts for efficient coding in Blender include:

- Tab: Indent
- Shift + Tab: Unindent
- Ctrl + / or Cmd + /: Toggle comments

For a comprehensive list of shortcuts, refer to the Blender documentation here: https://docs.blender.org/manual/en/3.5/editors/text_editor.html

Blender includes an optional Python Console within its user interface, serving as an interactive Python shell, alongside the Info Editor, which is useful for displaying logs, warnings, and error messages. While mastering these tools can significantly ease development work in Blender, initially, relying on the command line and `print()` statements for debugging purposes is sufficiently effective. This approach allows you to familiarize yourself with Blender's environment and scripting capabilities without the need for advanced tools.

1.4 Rendering and Saving

After editing the `simpleRT_plugin` script, re-run the script and initiate the rendering process by pressing F12. Alternatively, you can start the render by selecting `Render` → `Render Image` from the top menu bar. This workflow aligns with the approach used with the `Cycles` render engine, in contrast to the `Render Animation` employed with the `Workbench` render engine.

You should see the rendered image appear in the Image Editor and a progress bar in the bottom

status bar:

The `simpleRT_plugin` script includes functionality to print the elapsed and remaining time for the render in the command line. You can abort the rendering process at any moment by pressing the Esc key.

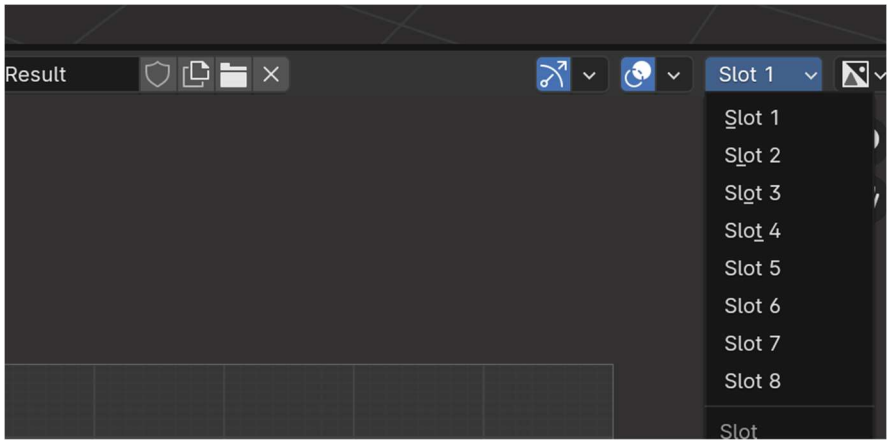
Upon completion of the render, you can save the image by navigating to `Image` → `Save Image` or by pressing Alt/Option + S. This allows for easy monitoring and management of the rendering process within Blender. With the starter code, the result is a completely black image.

1.5 Speeding up Rendering with Lower Resolution

To adjust the resolution quickly, alter the resolution percentage in the Properties Editor by navigating to the `SimpleRT Dimensions` panel under the Output tab. By setting the percentage to 25-50%, you can reduce the resolution while maintaining the aspect ratio, aiming to keep render times below 10 seconds, although this may vary based on your computer's specifications. This approach will produce an aliased (blurry) image, but it's sufficient to verify if your code functions. Once satisfied with the image, revert the percentage to 100% for a high-resolution version, which might take longer to render but allows for detailed inspection.

1.6 Comparing Render Results

You can store up to 8 rendered images in the Image Editors. You can toggle between them afterward using slots. It is very helpful in terms for comparing results from different version of code, or different material properties. To render the image to a certain slot, select that slot before rendering. See screenshot below for your reference.



You can switch between slots by using the number keys for direct access to the corresponding slot (ensure "Emulate Numpad" is disabled in Preferences) or use J to cycle forward and Alt + J to cycle backward through saved renders. For further information, consult the Blender documentation.

2. Assignment Checkpoints

2.1 Shadow Rays

A ray can be represented by defining two key components: an origin point and a direction vector. The origin point specifies where the ray starts in space, while the direction vector indicates the path the ray follows from its origin. This approach allows you to simulate how rays, including shadow rays, interact with objects in a scene for tasks like rendering or collision detection.

To represent the origin point “point a” at coordinates (1, 1, 1) in Cartesian space as a vector, you can denote it as $\vec{a} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.

To represent the vector direction geometrically as an arrow pointing from a start point $\vec{a}$ to an end point $\vec{b}$, you calculate the direction vector by subtracting the start point from the end point, i.e., $\vec{b}$-$\vec{a}$. This calculation gives you the directional vector from $\vec{a}$ to $\vec{b}$, indicating the direction and magnitude of the arrow from the start to the end point.

It's usually advisable to normalize direction vectors once they are calculated, which will clarify later in this assignment. This process transforms vectors into unit vectors, meaning they have a length of one. This is achieved by dividing the vector by its magnitude from a mathematical perspective . Python offers a handy function for this purpose `.normalized()`. As an example: `my_normalized_version_of_v = v. normalized()`.

In ray tracing, we begin by emitting a ray from the camera or eye, which serves as the ray's origin. The ray's direction is determined by a vector extending from the camera to a pixel (referred to as p) on the film plane. The ray proceeds in this direction, moving beyond the film plane into the scene, until it intersects with an object. Upon intersecting an object, calculations are performed to determine the color the ray should collect from the object. This color is then contributed to a cumulative total that represents the color of pixel p for the purpose of rendering.

For this part of the homework, you need to:

1. Construct the shadow ray(s).

   A shadow ray originates from the point where the primary ray tracing ray intersects an object, identified in the code as `hit_loc`. This point serves as the starting point for the shadow ray.

   The direction of the shadow ray is defined by a vector that extends from the intersection point `hit_loc` to a light source in the scene. Given that the code iterates over all lights, you can determine the position of the current light in the loop using `light.location`.

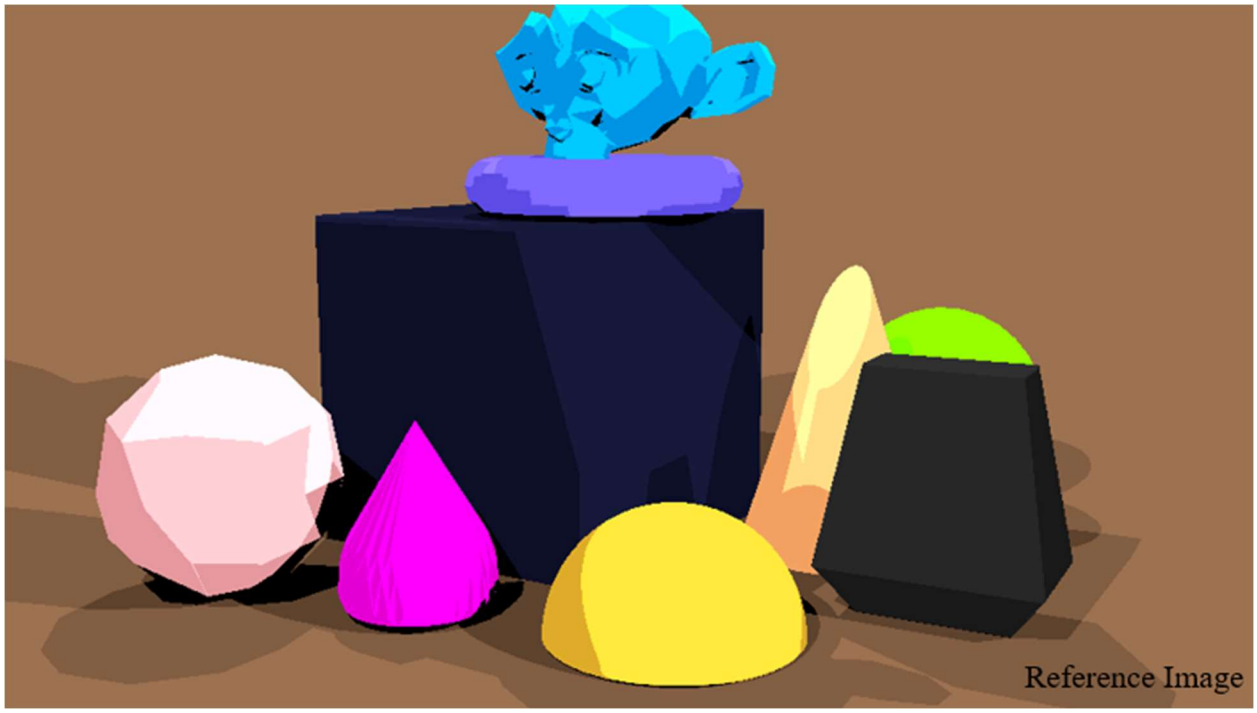
   Therefore, to compute the shadow ray's origin and direction, you need to use `hit_loc` as the origin and calculate the direction vector by subtracting `hit_loc` from `light.location`.

2. Account for spurious self-occlusion.

To mitigate spurious self-occlusion, you should adjust the shadow ray's origin slightly. This adjustment is necessary to prevent the ray from incorrectly intersecting the surface from which it originates. The normal direction at the point of intersection is provided as `hit_norm`, and a small epsilon value, `eps`, is given to represent a minor offset (specifically 10^{-3}, a small number). There are two main methods to handle self-occlusion. Both methods should work here for shadow rays, but they can vary in their image results by a few pixels, and that is fine. It should still be obvious whether the image is correct.

Action: Save the image render of this checkpoint at 960x540 resolution for grading.

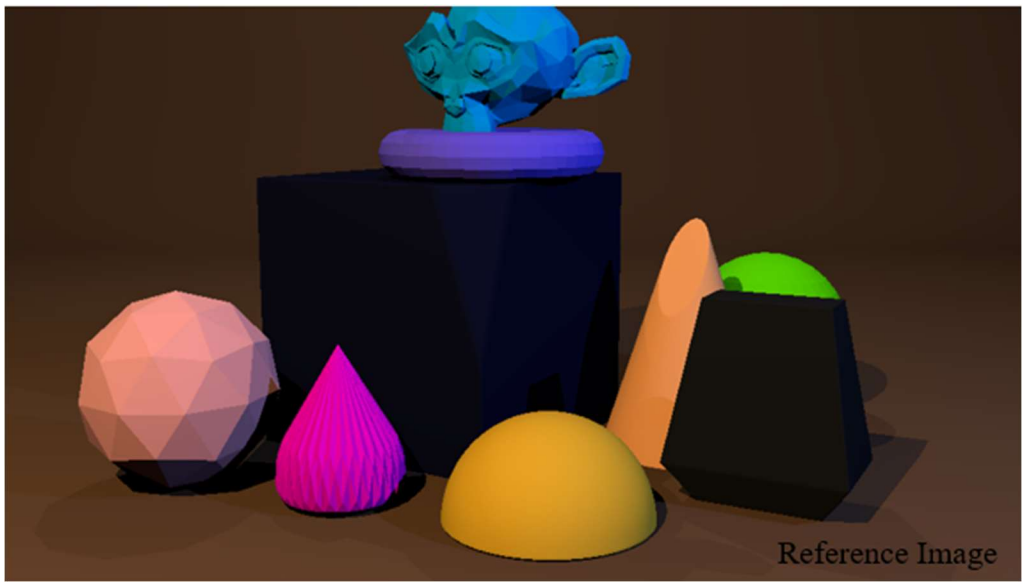
If you've coded everything correctly, you should get something close to the following image.



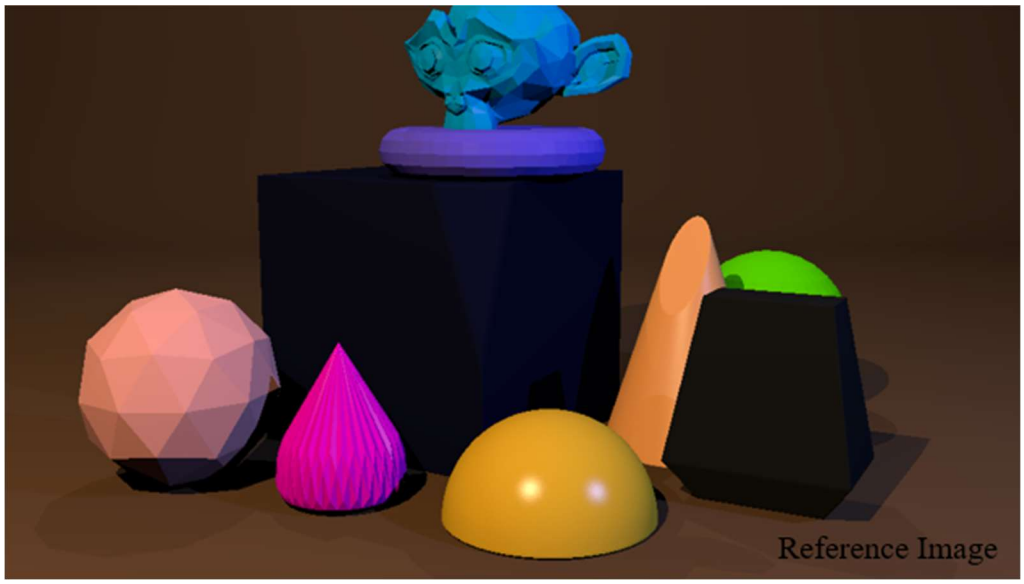
2.2 Diffuse and Specular BRDF Components

Carefully review the instructions above `TODO 2` in the script regarding calculating the diffuse color as explained in the lecture. Following this, adapt your code to apply the same process to the specular color. It might be necessary to transform both diffuse and specular color variables into Python arrays using `np.array` to prevent a type mismatch during their multiplication with the light intensity. This step is needed since the color variables are initially set as Python vectors but must be changed to Python arrays for a proper element-wise multiplication with another Python array. To compute the dot product of two vectors in Python, you can use the `.dot()` function. For instance, the dot product between vectors $\vec{v}_1$and $\vec{v}_2$ would be `v1.dot(v2)`.

If you've coded everything correctly for the diffuse color, you should get something close to the following image.



If you've also coded everything correctly for the specular color, you should get something close to the following image. Notice the shiny highlights on the yellow half sphere.

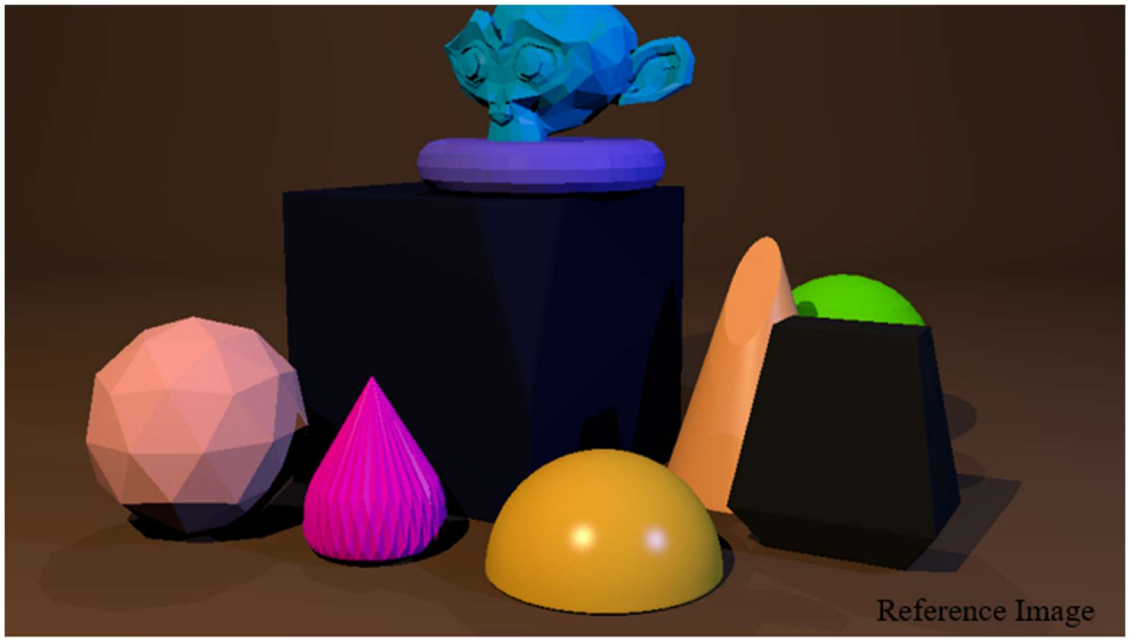


Action: Save the image render of this checkpoint at 960x540 resolution for grading.

2.3 Ambient BRDF Component

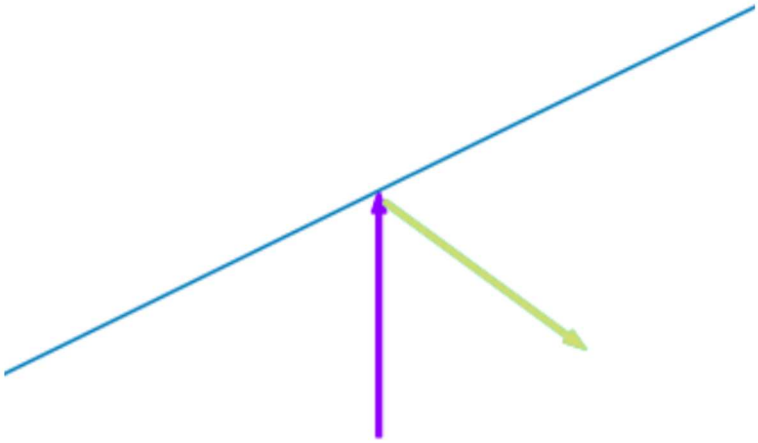
The ambient contribution is calculated by multiplying the material's diffuse color with its ambient color. Similar to the previous step, you might have to convert the diffuse color into a Python array using `np.array` to prevent a type mismatch.

If you've coded everything correctly, you should get something close to the following image.



Action: Save the image render of this checkpoint at 960x540 resolution for grading.

2.4 Recursion and Reflection

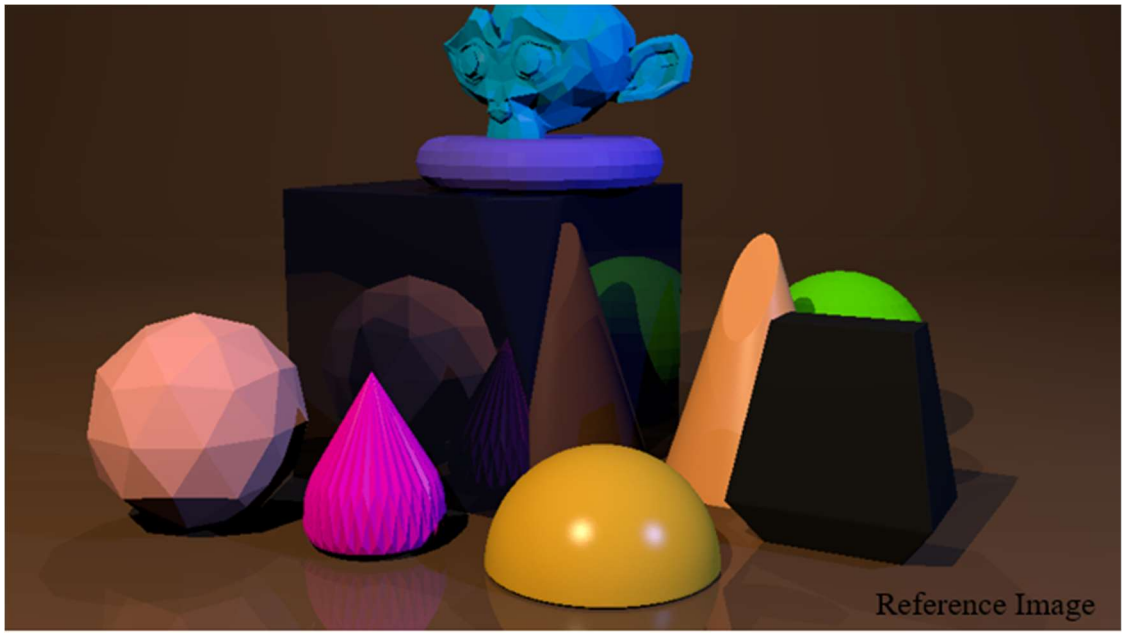


Imagine our ray starts off as a purple ray and intersects the reflective object's blue edge. Upon intersection, the object reflects the ray, causing it to diverge from its initial path. In programming terms, this reflection is modeled by creating a new ray object. This new ray, depicted as light-green in the diagram, extends towards the lower-right.

To model a new ray object, we must determine its origin and direction. These two attributes are essential for defining the reflected ray. To trace this new ray, we employ recursion, invoking the tracing function, `RT_trace_ray`, within itself. The function requires the ray's origin and direction as its 2nd and 3rd arguments. To trace the newly reflected ray, we call the `RT_trace_ray` function, specifying the new ray's origin and direction.

1. First, calculate the reflected ray's direction.
   Remember, the dot product in the equation for $D_{reflect}$ signifies the cosine of the angle between two unit, normalized vectors. Therefore, both the original ray direction and the surface normal vector, which the original ray intersects, should be normalized before using them in this dot product.
2. Next, determine the new origin of the reflected ray.
   While it might seem straightforward to use the intersection point, or `hit_loc` in the code, you must adjust for potential spurious self-occlusion issues. This adjustment involves moving the intersection point slightly along the normal direction by a small ε value.
3. Once the new origin and direction established for the reflected ray, proceed to trace it recursively.
   This involves invoking the `RT_trace_ray` function with the original ray origin and direction parameters replaced with the reflected ray's new origin and direction. It's essential to normalize the reflected ray direction before passing it to the function, as it uses the ray direction for various dot products requiring normalized vectors. Additionally, decrement the depth by 1(`depth – 1`) to manage the recursion depth(how many recursive steps we take before terminating).
4. The `RT_trace_ray` function will return a color, representing the color captured by the reflected ray at its termination point, referred to as the reflection color or $L_{reflect}$.
   This reflection color is then scaled by the material's reflectivity coefficient, often denoted as k_r. The scaled reflection color is then added to the cumulative color being calculated for the pixel.

If you've coded everything correctly, you should get something close to the following image. Notice how the "bowl" and the "blue cube" now reflects objects in the scene.



Action: Save the image render of this checkpoint at 960x540 resolution for grading.

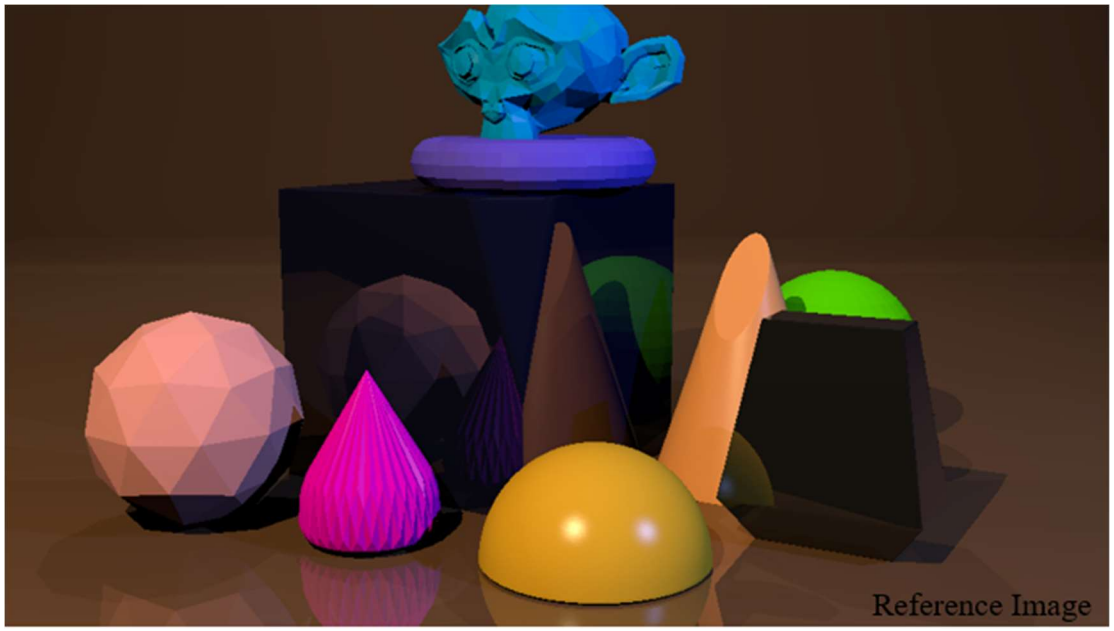
2.5 Fresnel

The Fresnel equations describe how reflection strength varies with the angle that our ray makes with the object it intersects (the angle of incidence). For simplicity, Schlick's Approximation , is used to calculate a reflectivity value, R(θ), often also referred to as k_r.

When determining the R_0 ratio, it's important to note that n_1 is the index of refraction (IOR) of the medium from which the ray originates, while n_2 is the IOR of the medium the ray intersects. In the context of this assignment, since all reflections are from a ray starting in air and reflecting off an object, n_1 is always the IOR of air, which is 1, and n_2 is the object's IOR, provided as `mat.ior`.

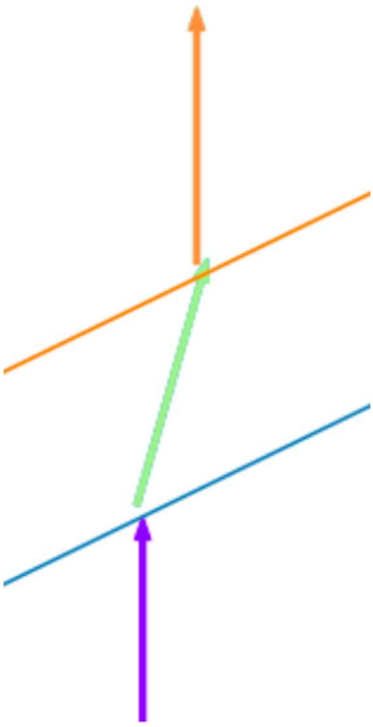
For calculating the cosine term in Schlick's Approximation's, remember the cosine of the angle between two unit, normalized vectors is found using their dot product. The angle we are examining here is between the ray direction vector and the object's surface normal vector at the point of intersection.

If you've coded everything correctly, you should get something close to the following image. Notice how the "black block" reflects the "half orange disk" on its side.



Action: Save the image render of this checkpoint at 960x540 resolution for grading.

2.6 Transmission



- Refer to the image on the left. Suppose that the ray we start with is marked in purple and it meets the blue boundary of a see-through object. Because of the object's material characteristics, the purple ray changes direction and passes through the object. In the programming, this process results in the creation of a new ray entity for the transmitted ray. This freshly transmitted ray is indicated by the light-green (middle) ray shown in the illustration.
- Now, consider that the ray under examination in our recursive process is the light-green (middle) ray from the diagram, not the purple one. This ray begins its journey within an object and is followed until it hits the object's orange boundary. At this point, the ray is refracted and exits the object, entering the air. Programmatically, this exiting ray is depicted as a completely new ray entity. The illustration shows this ray as the orange one.

In each case, the transmitted ray is characterized as a brand-new ray entity. It's essential to understand that creating a new ray object requires specifying its new origin and new direction.

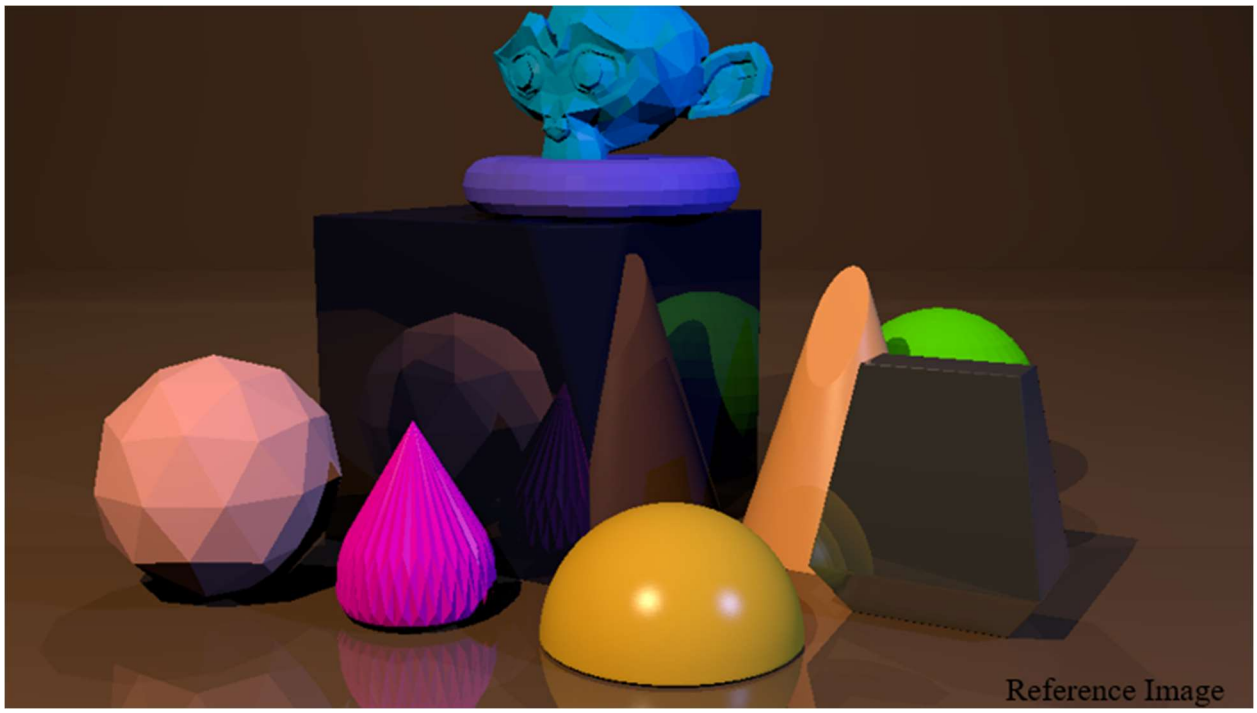
Additionally, we need to trace this new ray recursively by calling the `RT_trace_ray` function again with the new ray's origin and direction.

1. First, you need to compute the index of refraction (IOR) ratio: $\frac{n_1}{n_2}$

   The complexity arises because IOR for n_1 and $n_2$vary based on the ray's initial location: n_1 corresponds to the IOR of the medium where the ray originates for the transmission, while n_2 matches the IOR of the medium where the ray concludes its transmission.
   In other words, if the ray starts in air and travels through a transparent object, ending up inside the object, then n_1 would be the Index of Refraction (IOR) for air, which is 1. Meanwhile, n_2 should be the IOR for the object, indicated by the value provided as `mat.ior`. Conversely, if the ray begins inside the transparent object and exits into the air, the values for n_1 and n_2 are reversed. To determine if the ray starts inside the object, you can use the 'ray_inside_object' boolean flag.
2. Once you have IOR ratio, compute the direction of transmitted ray.
   It's important to remember to account for total internal reflection. You should only calculate the transmitted direction, $D_{transmit}$, if total internal reflection does not occur.
3. Similar to the previous steps, recursively trace the ray. This process is accomplished by invoking the `RT_trace_ray` function again, substituting the original ray origin and direction arguments with the new origin and direction of your transmitted ray.
   To be more specific, the origin of your transmitted ray is the point of intersection, referred to as `hit_loc`. It's a bit trickier in this case because transmitted rays are going through the object (as opposed to reflected rays which are bouncing off the object). This means you'll have to handle the ε offset in the normal direction slightly different from what you did in the reflection case.
   The direction of the transmitted ray is your computed $D_{transmit}$. Remember to normalize it, because it will be used in the dot products that you wrote for the previous steps throughout the recursion.
   Remember to pass in `depth-1`.
4. As with the previous steps, the `RT_trace_ray` function yields a color, specifically the transmission color or the color captured by the transmitted ray at its endpoint, denoted as$L_{transmit}$. The sum of an object's transmissiveness and reflectivity (k_r) is theoretically equal to 1, which leads us to weight the returned color by multiplying it by ($1 - k_r$) and

considering the material's transmission characteristics as specified in `mat.transmission`. After weighting the returned color with these values, we incorporate it into the sum of pixel color, as performed in earlier tasks.

If you've coded everything correctly, you should get something close to the following image. Notice how the "black block" now turned into glass, allowing you see the "orange cone" behind through the "black block".



Action: Save the image render of this checkpoint at 960x540 resolution for grading.

How to Submit

As before, you are expected to submit your demo video together with the screenshots of each task specified above. In this demo, you will show and also explain how you ran all the steps written above.

Late penalty: 10% for each late day and submission is not allowed after 3 late days.