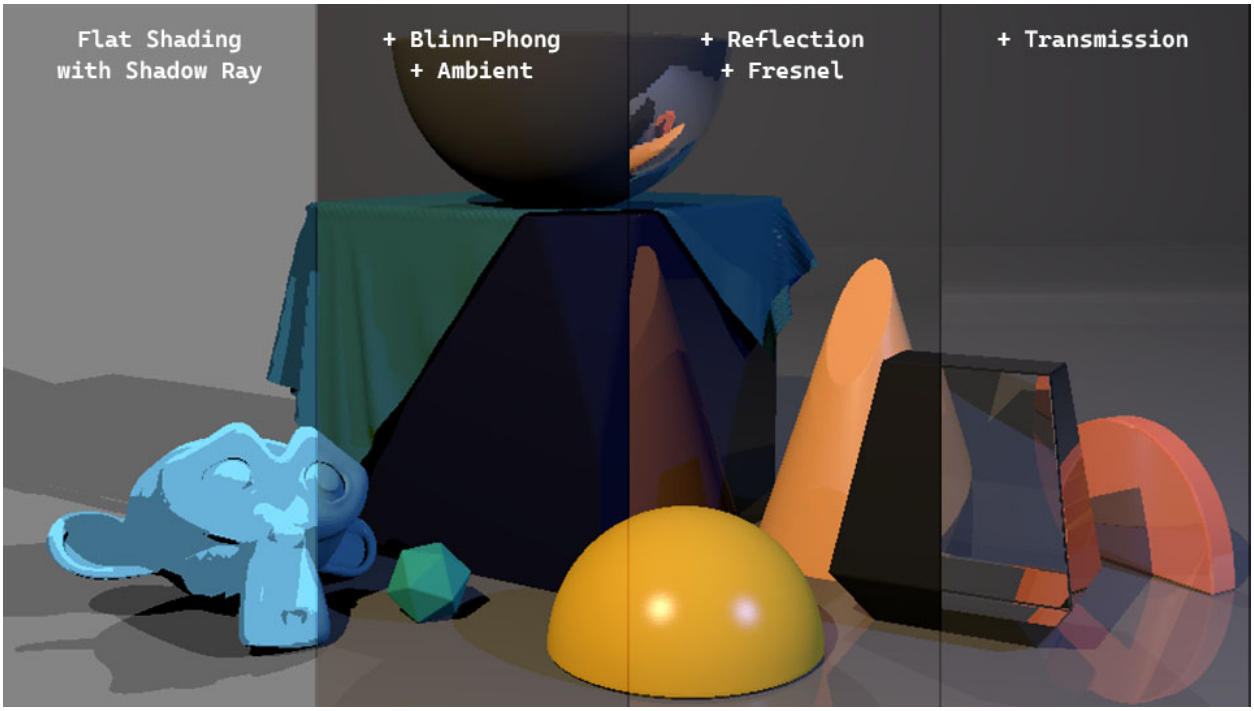
CS148 Homework 3 - Ray Tracing

Grading on Monday, Oct 12 – see the pinned posts on Ed for details!

0.1 Assignment Outline



The format of this assignment differs slightly from that of the previous assignment. Instead of splitting the assignment document into the TODOs first, then the instructional tutorials after, we will actually swap back and forth between them throughout the document. **We do this because this assignment is very coding intensive, so we want you to review the class material carefully as you do each TODO. Please start early!**

Your task is to finish the code for a Python ray tracer that interacts with the above scene in Blender. We've already created the scene for you in Blender and have set up the code infrastructure to access information from the scene using Blender's Python API. **If you are using Blender 4.3 or below, use:** this file. **If you are using Blender 4.4 or above, use:** this file.

The provided code is missing the key lines of code that are needed to properly render the scene as a ray traced image. The above image shows the four steps of ray tracing that are currently missing in the code. **Your job is to fill in the missing code for each of these steps, which are further elaborated as Action: TODO items in this PDF.** The power of ray tracing mostly lies in the last two, right-most steps, where reflections and transmissions are added to the objects. This is a feat that scanline renderers cannot do with physical correctness.

0.2 Collaboration Policy

These policies are the same as they were in HW2. See the HW2 document for details.

0.3 AI Policy

You MAY NOT use AI tools to generate anything for this HW! However, you may use AI tools to reinforce your understanding of the material. For instance, you can ask an AI chatbot to explain recursive ray tracing to you, as long as you don't ask it to write you the code.

Quiz Questions (1 pt)

You will be randomly asked one of these questions during the grading session:

- Describe one of the ways we discussed in class on how to compute a ray-triangle intersection. You do not need to provide all the mathematical details. Just give enough of the high-level ideas to show that you understand the approach.
- Why is transforming a normal vector between object and world space different from transforming an object between those same spaces?
- Explain the idea of total internal reflection. Conceptually, when does this happen, and what effect does it have on the transmission of light through a material?
- What does Beer's Law say about light as it travels through a medium? How does the attenuation coefficient come into play with modifying the light intensity appropriately as the light travels through some distance?
  Fun fact: if we wanted to be fully correct with our ray tracer code, then we should be implementing Beer's Law for transmissive rays as they go through (e.g. glass) objects. However, we aren't doing it here for simplicity.

1 Setting up the Assignment

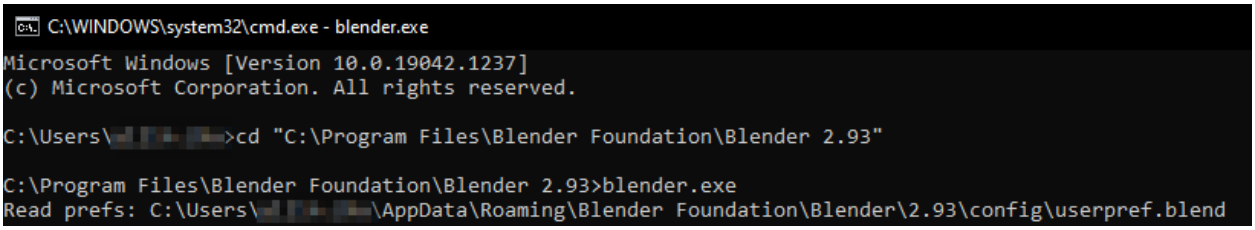
1.1 Starting Blender from the Command Line

Like before in HW2, since you will be doing Python development in Blender, you will want to launch Blender from the command line. Doing so will let Blender output any bugs or any information you explicitly print via the Python print() function to the command line. Here's a review of how to launch Blender from the command line:

- Windows: Press the `Windows` + `R` keys on your keyboard to open the "Run" box. Type `cmd` and hit enter to open the command prompt.

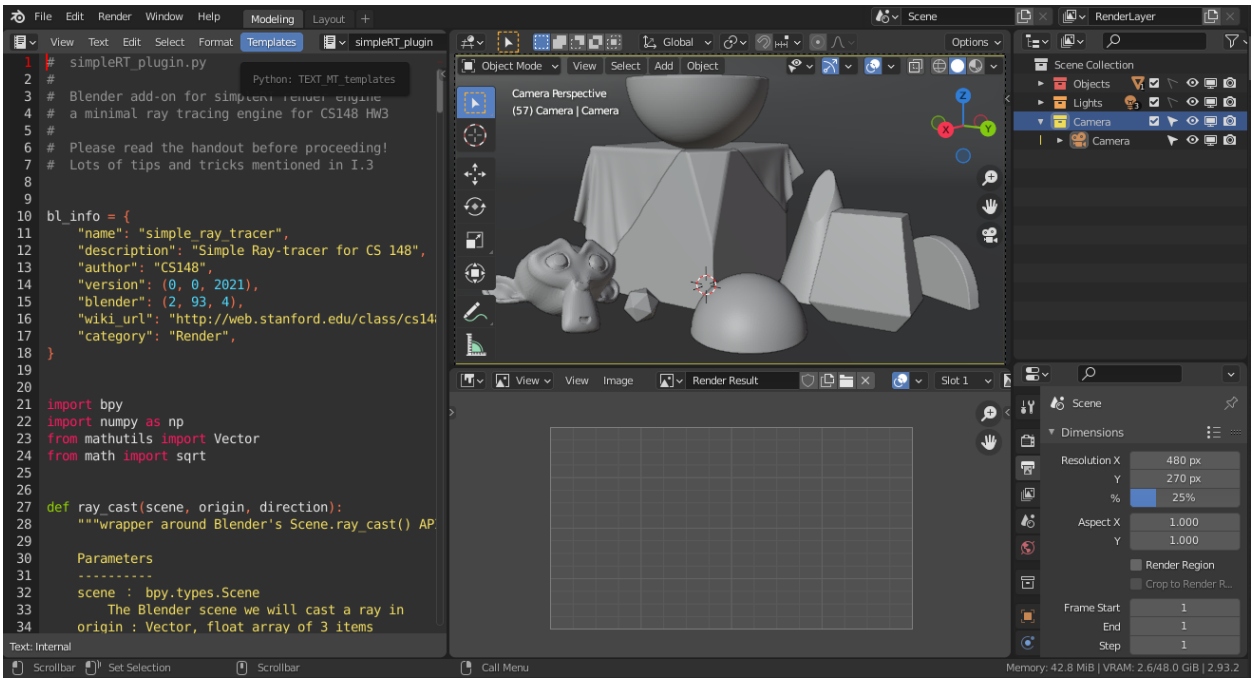
  `cd` to the Blender installation directory. The default location is `C:\Program Files\Blender Foundation\Blender 5.2`.

  In the Blender installation directory, type `Blender.exe` to launch Blender. The following image shows the above steps in action launching a past version of Blender.



- Mac: Please refer to the official document linked here.
- Linux: Please refer to the official document linked here.

Blender will launch as usual. Open the .blend file that came with this homework from the menu bar, i.e. **File → Open**. You should see a workspace layout similar to the following:

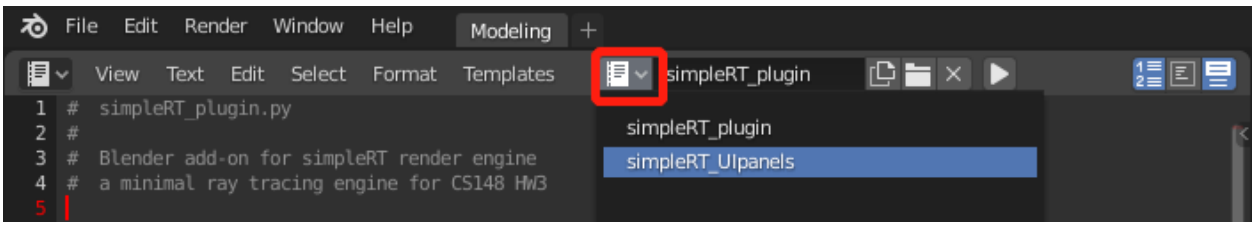


We've set up three main sections for you in the UI:

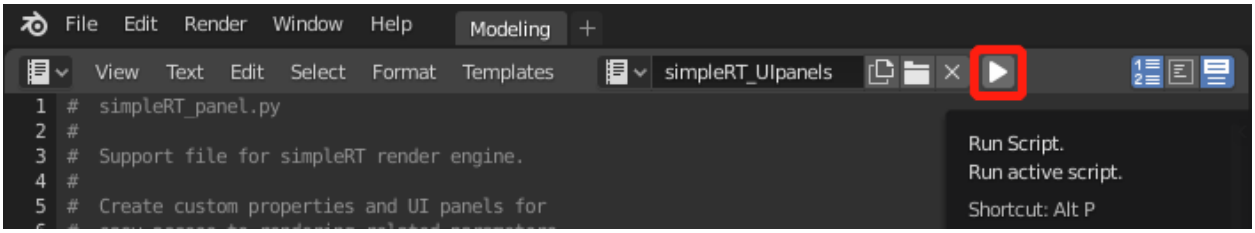
1. Left: Text Editor. We can edit and run the code in this space.
2. Top right: 3D Viewport. We can manipulate and preview the scene in this area.
3. Bottom right: Image Editor. We can view the render result as an image here.

1.2 The Text Editor

Action: In the Text Editor, you can switch between files using the dropdown button. Browse to a script called **simpleRT_UIpanels** and run it.



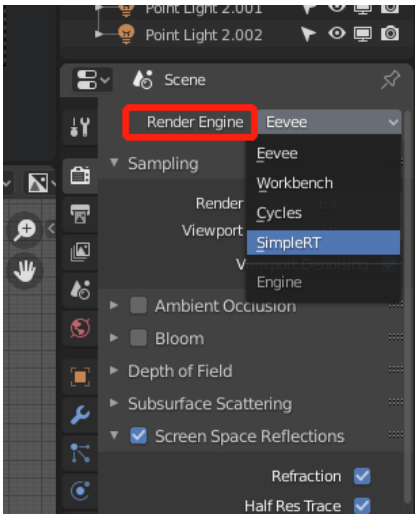
Remember that to run a script, press the "Run Script" button on the top bar of the Text Editor. Alternatively, if you do not see the button, then you can still run the script from the menu bar above the Text Editor with **Text → Run Script**. Nothing should appear in the command line if the script is executed successfully.

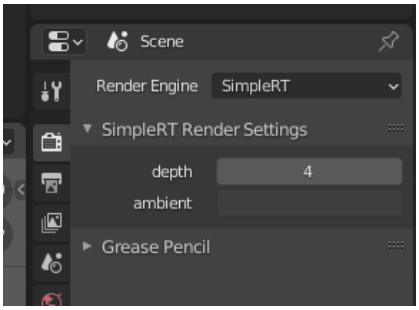


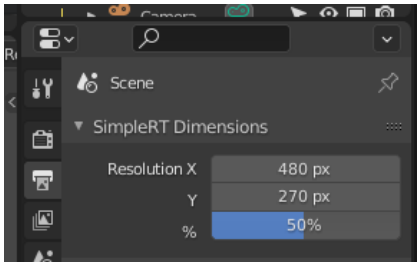
The `simpleRT_UIpanels` script creates custom properties that will be used for our renderer such as material properties, light parameters, etc. It then exposes these properties through the Blender UI so that we can change them easily from user interface widgets. If none of that made sense to you, then don't worry about it. Just know that **we need to run the `simpleRT_UIpanels` script every time we launch the .blend file for this HW before we can work with the main script!**

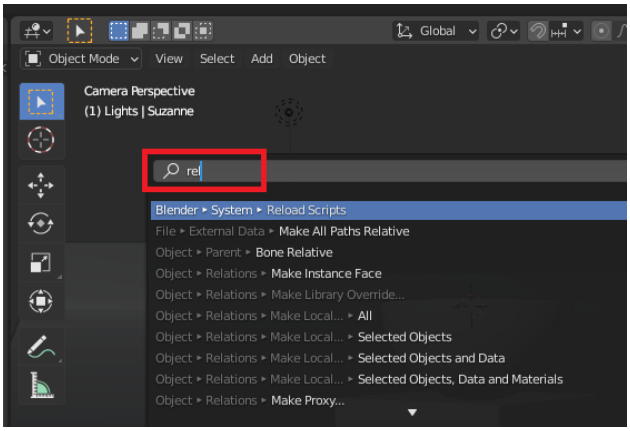
Switch back to the `simpleRT_plugin` and run it. To check if everything is working, switch the Render Engine to SimpleRT from the Properties Editor. This is the new render engine that we just added to Blender by running the script. We will work on this render engine in this homework.

The UI in the Properties Editor should be refreshed. You should see the `SimpleRT Render Settings` panel in the Render tab and the `SimpleRT Dimensions` panel in the Output tab. If you still see the UI panels from other render engines, press F3 with your cursor in the 3D Viewport, and type `reload script`; hit `Enter` to reload all the plugins.









1.3 Editing and Debugging

Action: Your task is to edit the `simpleRT_plugin` to finish all the TODOs. Blender has a very friendly development environment compared to other 3D software, but it is still not on par with professional Python IDEs or professional text editors. **Most debugging information will be displayed in the command line window from which you launched Blender.** As an example, try typing some gibberish (e.g. "asdfghjkl") on a new line in the `simpleRT_plugin` script. If you then run the script, then try to render (see next subsection), then you should get an error thrown in the command line that says: `NameError: name 'asdfghjkl' is not defined` Furthermore, you can write `print()` statements in the `simpleRT_plugin` script and expect to see the print out in the command line.

There are some handy shortcuts in the Text Editor. You'll likely use the formatting operations the most:

- `Tab` for indent
- `Shift` + `Tab` for unindent
- `Ctrl` + `/` or `Cmd` + `/` to toggle comments

For more shortcuts in the Text Editor, please see the Blender documentation.

There is optionally a Python Console in the Blender user interface. You can see the documentation here and can use it as an interactive Python shell. Additionally, the Info Editor can be useful to display logs, warnings, and error messages. Both of these tools can make your development life easier in Blender if you master them, but when we're just starting out, simply working with the command line and `print()` statements should be enough for debugging.

1.4 Rendering and Saving

After you make edits to the `simpleRT_plugin` script, run the script again, and hit F12 to start the render. Alternatively, you can start the render by going to `Render` → `Render Image` in the top menu bar. Note that we are using `Render Image` with this makeshift render engine like we do with `Cycles`, as opposed to `Render Animation` for the `Workbench` render engine.

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

Furthermore, the provided code in the `simpleRT_plugin` script should also be printing the elapsed and remaining time for the render in the command line. You can press the `Esc` keyboard key at any time to stop the render.

When the rendering is done, you can use `Image` → `Save Image` or `Alt/Option` + `S` from the keyboard to save the result. **With the starter code, the image should render a completely black image.**

1.5 Speeding up Rendering with Lower Resolution

To speed things up while you're debugging, you should keep the image resolution low to make the renders fast (since the speed is based on iterations over the number of pixels). After the low-resolution image looks correct, crank up the resolution and render a higher resolution image for submission.

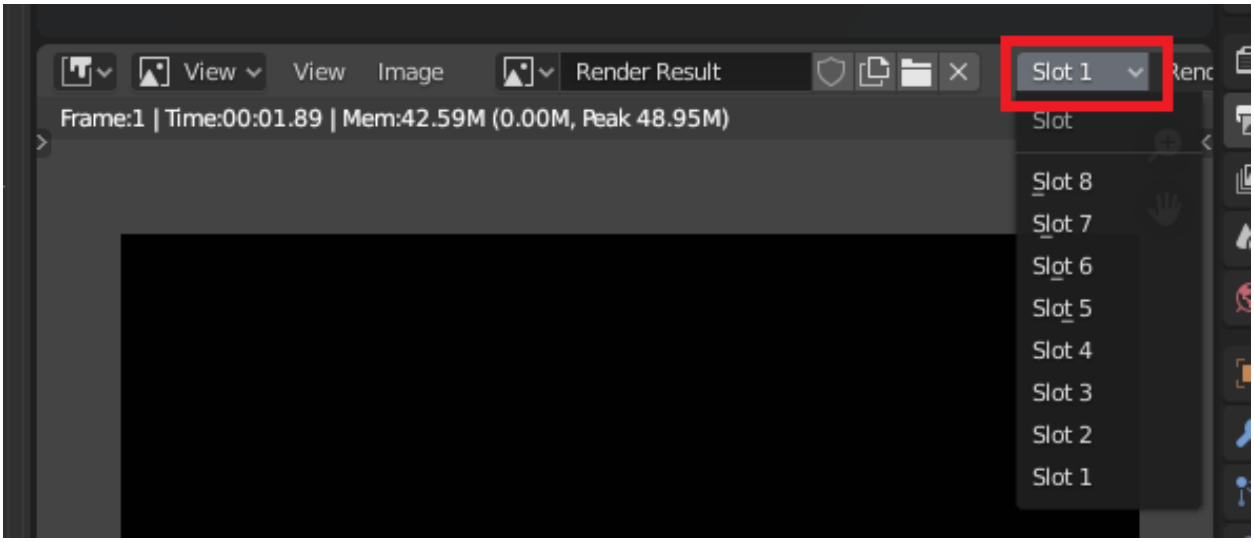
A quick and easy way to change the resolution is to change the resolution percentage from the Properties Editor. Go to the Output tab for the **SimpleRT Dimensions** panel. You can modify the **%** option to shrink the image resolution without changing the aspect ratio. Usually 25-50% is a good place to start and should hopefully keep your render time under 10 seconds (though it can depend on your computer specs). You will of course get a very aliased (blurry) result, but you should be able to tell if your code is working from this image.

After the image looks right to you, change the percentage back to 100 for a higher resolution image. The render will likely take a few minutes, but with this resolution, you should be able to check all the details.

1.6 Comparing Render Results

In the Image Editor, you can store up to 8 rendered images in slots and toggle between them afterward. This could be helpful when comparing the results from different versions of code, or different material properties.

To render the image to a certain slot, select that slot before rendering:



To toggle between the slots, you can use the number keys to go to the slot with the corresponding number (turn off **Emulate Numpad** from the **Preferences**) or press **J** and **Alt** + **J** to cycle forwards and backwards through saved renders respectively. You can see the Blender documentation for more details.

2 Assignment Checkpoints

All the TODOs are listed as comments in the **simpleRT_plugin** script. We recommend first reading the TODO descriptions in this PDF document first, then go to the sections in the code. The sample images in this PDF are rendered at 200%. Expect to see more aliasing in your images.

For those curious, the ray-object intersection logic is taken care of by the **scene.ray_cast** function from Blender. Our focus in this homework will be on how to use the information from the intersection to figure out the correct color calculations and light bounces.

2.1 TODO 1 (0.5 pt): Shadow Rays

Action: Save the image render of this checkpoint at 960 x 540 resolution for grading!

We discussed how to represent shadow rays and rays in general mathematically in lecture, but we need to think of these concepts from a different angle when representing them in code. Consider what defines a ray – you need an **origin point** and a **vector direction**.

The origin point is simply an xyz-coordinate in Cartesian space written as a vector. For instance, if we want to call the point at $(1, 1, 1)$ "point a", then we can write $\vec{a} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.

A vector direction is geometrically an arrow pointing from a start point to an end point. Call the start point $\vec{a}$ and the end point $\vec{b}$. Then the vector direction from $\vec{a}$ to $\vec{b}$ is computed as the difference $\vec{b} - \vec{a}$, i.e. the end point minus the start point.

For purposes that will become clearer later in this homework, it is often good practice to **normalize** direction vectors after you compute them. Normalizing vectors means making them unit vectors, i.e. giving them unit length. Mathematically, we do this by dividing the vector by its magnitude or size. In Python, there is a convenient `.normalized()` function that you can use. As an example: `my_normalized_version_of_v = v.normalized()`.

When we ray trace, we start by shooting a ray that starts from our camera or eye. That is the origin point of the ray. The direction of the ray is the vector direction starting from the camera and ending at a pixel (call it p) on our film plane. We continue tracing along this direction past our film plane into our scene until we hit aka intersect an object. When we finally intersect an object, we start doing operations to compute the color that the ray should gather from the object. This color is then added to a running sum keeping track of the color of the pixel p for rendering.

For this part of the homework, you need to:

1. Construct the shadow ray(s).

   Conceptually, a shadow ray is a ray that starts from the intersection of the ray that we are ray tracing and the object it hits. In the code this intersection is given to you as `hit_loc`. That is the origin of the shadow ray.

   The vector direction of a shadow ray is the vector starting from the intersection point and ending at a light. Notice how the TODO in the code is within a loop over all the lights. You can access the position of the light in the current iteration of the loop with `light.location`.

   You need to compute both the shadow ray origin and the shadow ray direction.

2. Account for spurious self-occlusion.

   You need to modify the shadow ray origin slightly to account for spurious self-occlusion as discussed in lecture. The normal direction at the intersection has been provided for you as `hit_norm`, and a small ϵ value has been given to you as `eps` (it's just simply 10^{-3}, i.e. some small number). Note that we talked about two ways to handle self-occlusion in lecture – 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.

You should get close to the following Checkpoint 1 image if you've coded everything correctly:

2.2 TODO 2 (0.5 pt): Diffuse and Specular BRDF Components

Action: Save the image render of this checkpoint at 960 x 540 resolution for grading!

Read the comments above the TODO 2 in the script carefully on how to compute the diffuse color as discussed in lecture. Then, pattern match your code to do the same for the specular color (which will be discussed in a later lecture, but is necessary here for the full visual effect). Note that you may need to convert the diffuse and specular color variables to Python arrays with `np.array` to avoid a type mismatch when multiplying it with the light intensity. This is because the color variables are declared as Python vectors, but need to be converted to Python arrays for an 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)`.

You should get this Checkpoint 2a image if you've coded everything for the diffuse color correctly:

You should then get the following Checkpoint 2b image, with shiny highlights on the yellow half sphere, if you've coded everything for the specular color correctly:

2.3 TODO 3 (0.5 pt): Ambient BRDF Component

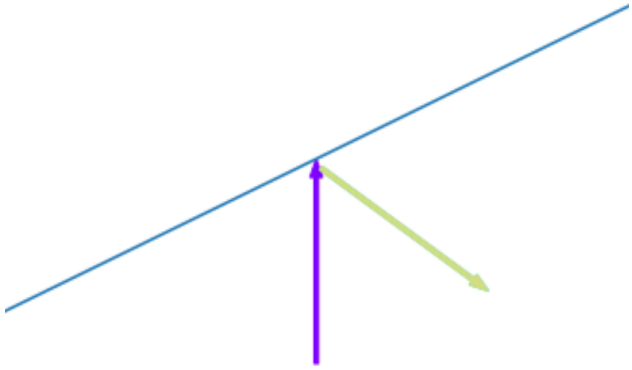
Action: Save the image render of this checkpoint at 960 x 540 resolution for grading!

We will compute the ambient contribution as the product of the material diffuse color and the material ambient color here. Like in the last step, you may need to convert the diffuse color to a Python array with `np.array` to avoid a type mismatch.

You should get the following Checkpoint 3 image if you've coded everything correctly. Notice how the shadow of the black block on the orange half disc has gotten lighter.

2.4 TODO 4 (1 pt): Recursion and Reflection

Action: Save the image render of this checkpoint at 960 x 540 resolution for grading!



Suppose our ray is initially traced as the purple ray and intersects the blue edge of a reflective object. The ray gets reflected by the object to then continue in a different direction from which it began. In code, we represent the reflected ray as an entirely new ray object. This new reflected ray is represented by the light-green ray in the diagram pointing to the lower-right.

To represent a new ray object, we need the new ray origin and the new ray direction. Once we have these two characteristics to define the reflected ray, we need to trace the new ray. This is done via recursion in code by calling the `RT_trace_ray` function within itself.

The 2nd and 3rd arguments to the `RT_trace_ray` function ask for a ray origin and ray direction respectively. To trace the new, reflected ray, we simply just need to call the `RT_trace_ray` function with the reflected ray's new ray origin and the reflected ray's new ray direction.

1. First, you need to compute the direction of the reflected ray as covered in lecture.

   Note the purpose of the dot product in the equation for $D_{reflect}$. The dot product between two unit, normalized vectors is the cosine of the angle between them. So the two vectors involved in this dot product should be normalized beforehand. These two vectors are the original ray direction and the normal vector to the surface that the original ray intersected.

2. After you have the reflected ray direction, you need to compute the new origin of the reflected ray. Naively, you may think this is simply the intersection point, i.e. `hit_loc` in the code. However, you need to account for spurious self-occlusion as discussed in lecture and offset this intersection point in the normal direction by some small ϵ value.

3. Once you have the new origin and the new direction of the reflected ray, you need to recursively trace the ray. This is done by calling the function `RT_trace_ray` with the ray origin and ray direction arguments replaced with the new origin and new direction of your reflected ray.

   Note that throughout the `RT_trace_ray` function, you use the passed in ray direction argument for various dot products, which all want normalized vectors. So as a good general practice, you want to normalize the reflected ray direction when passing it into the function.

   You should also pass in `depth - 1`. We use the `depth` variable to keep track of how many recursive steps we take before terminating.

4. The function `RT_trace_ray` returns a color. This is the color gathered by the reflected ray when it finally terminates. We call this the reflection color aka $L_{reflect}$.

   $L_{reflect}$ is then weighted by the object material's reflectivity value, usually written as k_r. It is this weighted color that is then added to a running sum of the color that we are computing for the pixel.

You should get the following Checkpoint 4 image if you've coded everything correctly. Notice how the bowl at the top as well as the blue cube now reflect objects in the scene.

2.5 TODO 5 (0.5 pt): Fresnel

Action: Save the image render of this checkpoint at 960 x 540 resolution for grading!

The Fresnel equations model the change in the strength of a reflection based on the angle that our ray makes with the object it intersects (aka the angle of incidence). The equations can be simplified with Schlick's Approximation as discussed in lecture to compute a reflectivity value $R(\theta)$, also commonly written as k_r.

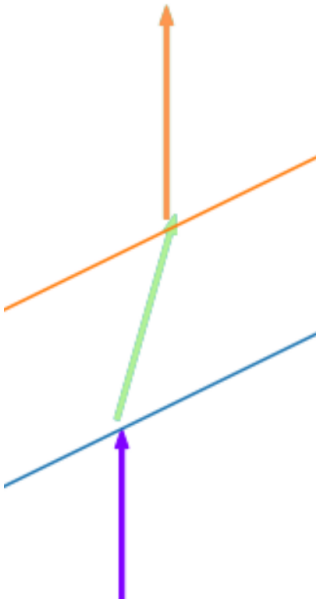
When computing the R_0 ratio, remember that n_1 is always the index of refraction (IOR) of the medium in which the ray starts. n_2 is always the IOR of the medium that the ray intersects. In the scene for the homework, all reflections occur from a ray originating from the air and bouncing off an object. So, n_1 is always the IOR of air, which is 1, and n_2 should be the IOR for the object, which is given to you as `mat.ior`.

To compute the cosine term for Schlick's Approximation, remember that the dot product between two unit, normalized vectors is the cosine of the angle between them. The angle we are examining is the angle between the ray direction vector and the object's surface, which is represented by the normal vector at the intersection.

You should get the following Checkpoint 5 image if you've coded everything correctly. Notice the reflections of the ground and orange half-disc on the black block.

2.6 TODO 6 (1 pt): Transmission

Action: Save the image render of this checkpoint at 960 x 540 resolution for grading!



Consider the diagram on the right. Suppose our ray is initially traced as the purple ray and intersects the blue edge of a transparent object. Due to the material properties of the object, the purple ray is refracted and then transmitted through the object. In code, we represent the transmitted ray as an entirely new ray object. This new transmitted ray is represented by the light-green (middle) ray in the diagram.

Now, suppose instead of the purple ray, the ray that we are examining in our recursion is actually the light-green (middle) ray in the diagram. This ray originated inside an object and is traced until it intersects the orange edge of the object. The ray is then refracted and transmitted out of the object into the air. In code, we represent the transmitted ray as an entirely new ray object. In the diagram, the transmitted ray is represented by the orange ray.

In both cases, we represent the transmitted ray as an entirely new ray object. As you should know by now, to represent a new ray object, we need both the new ray origin and the new ray direction. Furthermore, we need to recursively trace the new ray by calling the `RT_trace_ray` function again with the new ray origin and the new ray direction.

1. First, you need to compute the index of refraction (IOR) ratio: $\frac{n_1}{n_2}$. The tricky part here is that n_1 and n_2 change depending on where the ray starts. n_1 is always the IOR of the medium in which the ray starts for the transmission. n_2 is always the IOR of the medium in which the ray ends up for the transmission.

   So if the object is starting in the air and is going through a transparent object to end up inside the object, then n_1 should be the IOR for air, which is 1, and n_2 should be the IOR for the object, which is given to you as `mat.ior`.

On the flip side, if the ray starts inside the transparent object and is going through it to end up in the air, then the values flip. You can check if the ray starts inside the object with the `ray_inside_object` boolean.

2. Once you have the IOR ratio, then you can compute the direction of the transmitted ray as discussed in lecture. One thing to note is that you need to account for total internal reflection.

   You only proceed with computing $D_{transmit}$ if there is no total internal reflection.

3. Similar to the previous steps, you need to recursively trace the ray. Again, this is done by calling the function `RT_trace_ray` again with the ray origin and ray direction arguments replaced with the new origin and new direction of your transmitted ray.

   The origin of your transmitted ray is the intersection point, aka `hit_loc`. But also remember that you need to account for spurious self-occlusion. 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 also pass in `depth - 1`.

4. As with the previous steps, the `RT_trace_ray` function returns a color. In this case, it's the transmission color, aka the color gathered by the transmitted ray when it finally terminates: $L_{transmit}$. The transmissiveness and reflectivity (k_r) of an object are theoretically supposed to add up to 1, so we weight the returned color by $(1 - k_r)$ as well as the object material's transmission properties given in `mat.transmission`. After weighting the returned color with these values, we add it to running sum of the pixel color like we did in the previous TODOs.

   You should get the following Checkpoint 6 image if you've coded everything correctly. Notice how the black block has turned into glass, allowing you to see the orange cone through it.

NOTE: The above is the expected output for those working on Blender versions 3.5 and earlier. Starting in Blender 3.6, a slight update in the software now results in the following variation in the transmission of the cone through the glass block:

This variation is fine and is considered correct.

3 What to Demo

During the grading session, please screenshare with your CA and:

- (0.5 pt) Show a 960 x 540 saved image of Checkpoint 1 (TODO 1: Shadow Rays).
- (0.5 pt) Show a 960 x 540 saved image of Checkpoint 2b (TODO 2: Diffuse and Specular BRDF Components).
- (0.5 pt) Show a 960 x 540 saved image of Checkpoint 3 (TODO 3: Ambient BRDF Component).
- (1 pt) Show a 960 x 540 saved image of Checkpoint 4 (TODO 4: Recursion and Reflection).
- (0.5 pt) Show a 960 x 540 saved image of Checkpoint 5 (TODO 5: Fresnel).
- (1 pt) Show a 960 x 540 saved image of Checkpoint 6 (TODO 6: Transmission) AND open your finished ray tracing script on Blender and render Checkpoint 6 live.
- (1 pt) Answer the quiz question given by the CA correctly.