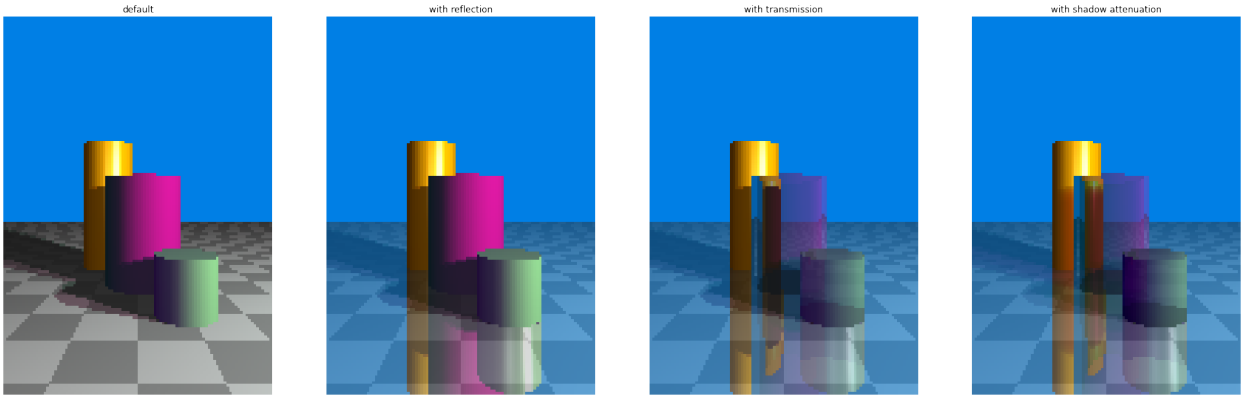
CS148 Homework 4

Homework Due: Jul 23rd at 11:59 PM PST
Quiz Date: Tuesday Jul 19th

1 Introduction

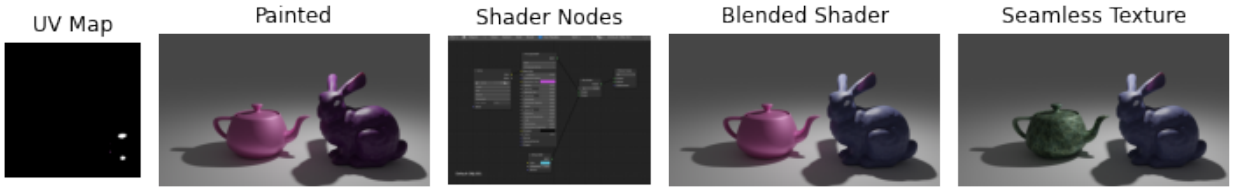
In this homework, you will play around with and implement recursive raytracing, i.e. upgrading a raycasting framework to a Whitted raytracer! Note that the work needed here is only 20-30 lines of code, but it requires understanding of raytracing as discussed in lecture and as you initially implemented in the previous assignment. We recommend starting the reading as early as possible (and as such, **the quiz contains questions that require you to take a look at the starter code**).
For the technical component of this homework, you will:

- Implement reflection and refraction logic for ray generation as discussed in lecture. (2 points)
- Add recursive functionality to the `raytrace()` starter code. (1 point)
- Add shadow ray attenuation to handle shadows for transmissive objects. (2 points)
- **[Extra Credit]** Add sphere geometry or depth of field to the raytracer. (1 point each)



For the Blender component of this homework, you will work with materials and textures in-depth, and:

- UV unwrap and paint a UV texture for 1 handcrafted model. (2 points)
- UV unwrap, scale, and assign a seamless texture for a second handcrafted model. (1 point)
- Make a complex shader by experimenting with the shader graph and shader nodes. (2 points)



2 Quiz Questions

- Read through and understand the second cell in the HW4 notebook. Given a cylinder **C**, a ray **r** that we know intersects the cylinder, and a single point light **L**, describe what helper functions you would need to call to create a shadow ray.
- Describe a visual phenomena that a distributed raytracer can easily render but a Whitted raytracer can't, and explain why.
- Describe explicitly how bidirectional raytracing can result in color bleeding.
- Describe a situation where we'd use UV coordinates with values larger than 1.0
- Explain how normal mapping works, and what the difference is between normal, bump, and displacement mapping.

3 Technical Component

The technical component of this homework will be done within the HW4 jupyter notebook in the cs148 anaconda environment. First, scan through the Jupyter notebook to see example usage of the data structures involved in this assignment. In particular, the existing code has logic somewhat similar to last week's assignment, but is factored out in a way that allows for re-usability and recursion. Note that helper functions are given in `HW4/raytracer_utils.py` and `HW4/debug_utils.py`. You should NOT edit any of these, and do not need to read through them carefully (though `raytracer_utils` would be valuable to read through if you're still not completely sure about how a raytracer works.

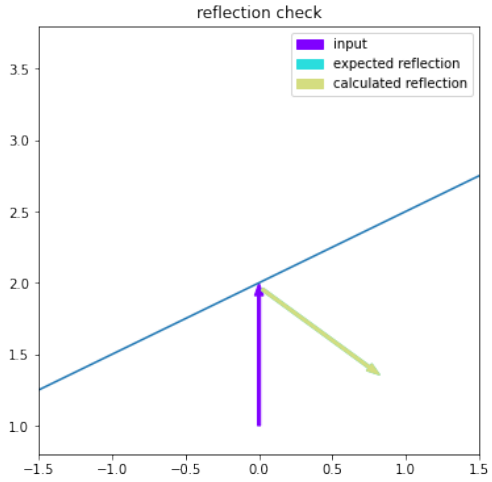
There are seven code blocks that you will need to fill in in order to implement the functionality of the recursive raytracer. They are all doable with 5-10 lines of code. Again, note that this and last week's homeworks are more concept heavy than implementation heavy; once you figure out what we are doing the rest should be easy!

TODO1. Reflection

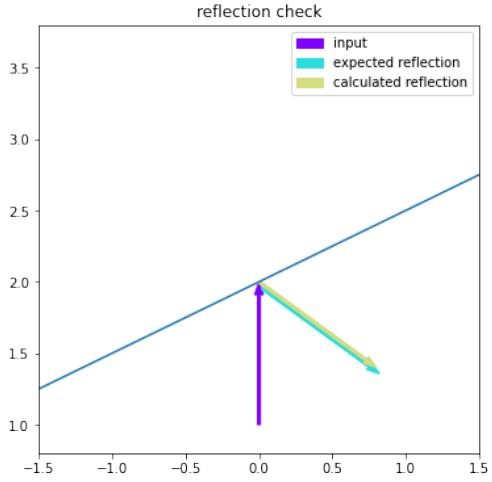
In the `reflectRay` function, you will need to generate a ray being reflected off of a surface. Given an existing ray and some intersection, where we know the intersection position and intersection normal, you need to create a new ray as mentioned in the lecture slides. Recall from lecture the equation for the reflected direction:

$$\hat{D}_{reflect} = \hat{D} - 2(\hat{N} \cdot \hat{D})\hat{N}$$

Action: Fill out this function. We've provided a sanity check for this component. If you implemented your method correctly, you should only see one line and two vectors (rays) like this when you run the cell containing your implementation:



Anything else may result in problems for the later sections. (Hint: if it's just slightly offset like the plot below, you may have not done the self-occlusion fix to the ray origin properly.)



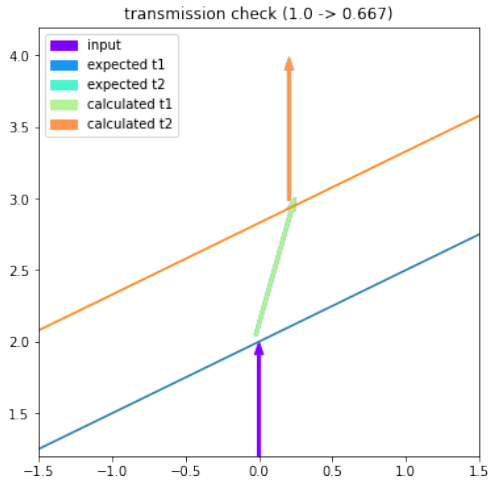
TODO2. Refraction

For the first part of this step, fill in the `computeTransmissionDirection` function to compute the transmitted direction. Additionally, detect and return `None` when total internal reflection occurs. Refer to slides for the derivation of the transmitted direction:

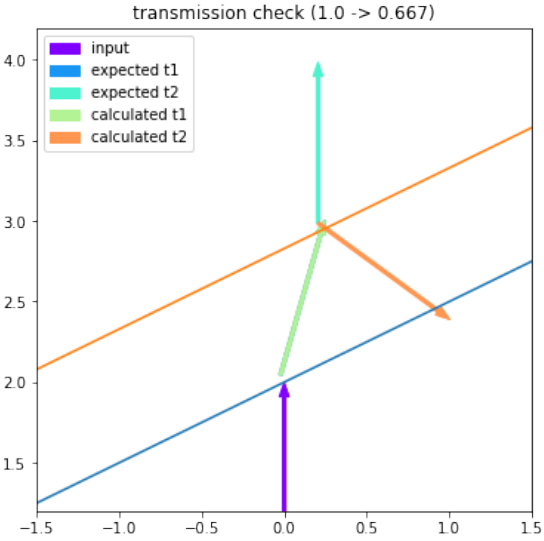
$$\hat{D}_{transmit} = \frac{n_1}{n_2}\hat{D} - \left(\frac{n_1}{n_2}\hat{N}\cdot\hat{D} + \sqrt{1-(\frac{n_1}{n_2})^2(1-(\hat{N}\cdot\hat{D})^2)}\right)\hat{N}$$

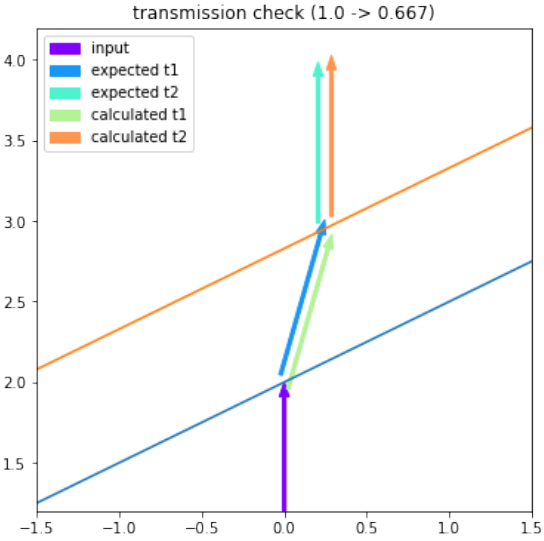
For the second part of this step, complete the `refractRay` function. Take into account the intersected object's index of refraction, and whether the ray is entering the object or exiting (denoted in `get_intersection_info()` by `from_inside_object` as mentioned in the second cell of the notebook. If `from_inside_object` is true, that mean the ray started from inside the object and thus is exiting the object at this current intersection point.
Also recall the need to push the ray's origin outwards or inwards from the object's surface by a small amount in order to avoid spurious self-intersections.
Action: Fill out these 2 functions. We've provided a sanity check for this component. If you

implemented your method correctly, you should only see 2 lines and three vectors (rays) like this when you run the cell containing your implementation:



Anything else may result in problems for the later sections. (Hint: the left plot below happens if you're not handling whether we're entering or exiting the object properly. The right plot happens if you have not done the self-occlusion fix properly.)





TODO3. Recursion

Once you have completed the implementation of `reflectRay` and `refractRay`, what remains is to

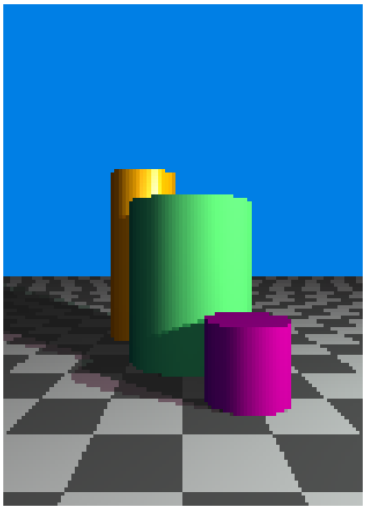
- Add the shadow ray creation logic from HW3 into HW4. The logic should be identical. Reach out to us if you are still struggling with this past the HW3 submission deadline (TODO 3.0)
- Recursively cast reflected and transmitted rays in the `rayTrace` function. (TODO 3.1 and 3.2)

Collect the colors from these recursive raycasts to be accumulated into the final color for the current intersection. Review the slides if you're unsure what to do here.

Action: Find the cell that contains

```
# Load objects and lights
objects = get_default_scene_objects()
lights = get_default_scene_lights()
# Set camera position and film plane
camera_aperture = np.array([0, 0, 1])
film_xlim=np.array([-0.5,0.5])
film_z = 0
# TODO: change this as you test. This should
raytrace_settings=RayTraceSettings(enable_reflection=False, enable_transmission=
    False, enable_shadow_attenuation=False)
# Set image resolution; feel free to adjust
# (note: starting with a small value is recommended for quick iteration)
screen_height = 140
screen_width = 100
rendered_image = generateImage(screen_height, screen_width,
                               camera_aperture, film_xlim, film_z,
                               raytrace_settings, objects,lights)
show_image(rendered_image, height = 8, width = 8)
```

If you run it as is with the correct shadow ray code, you should see something along the lines of this (note that the geometry is changed from the initial reference image):



If you change the `raytrace_settings` and enable reflections and refractions, you should see reflections and transmissive objects!

TODO4. Shadow Attenuation

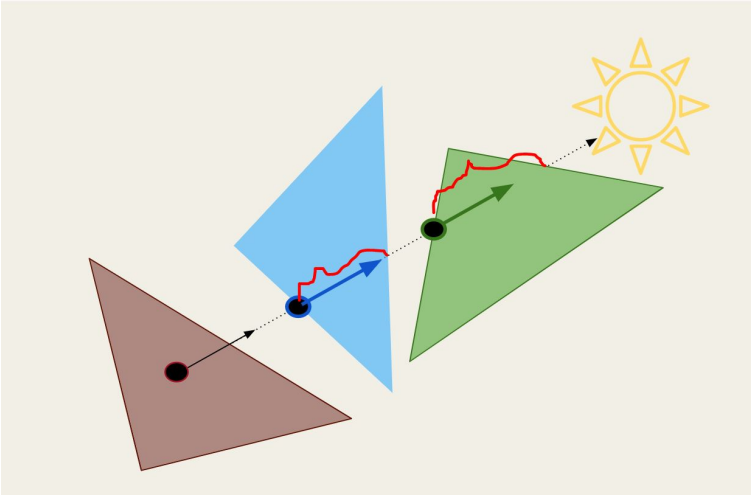
Finally, complete the implementation of the `computeVisibleLight` function to support attentuation of light due to a shadow ray passing through a transmissive object on its way to a light source. Note that since we're not doing bidirectional ray tracing, all you need to do is to trace a straight line to each light, then for each object that the shadow ray intersects, compute the length (t) that the ray travels within each object.

If any of the objects are non-transmissive (i.e. no light passes through), `computeVisibleLight` should return `None`.

For each transmissive object o in the shadow ray's path, attenuate the light with Beer's law using the material attenuation coefficient c_o and also multiply by the transmissivity factor x_o, i.e.

$$L = \Pi_o(x_o \cdot \text{BeersLaw}(c_o, t_o)) \cdot L_0$$

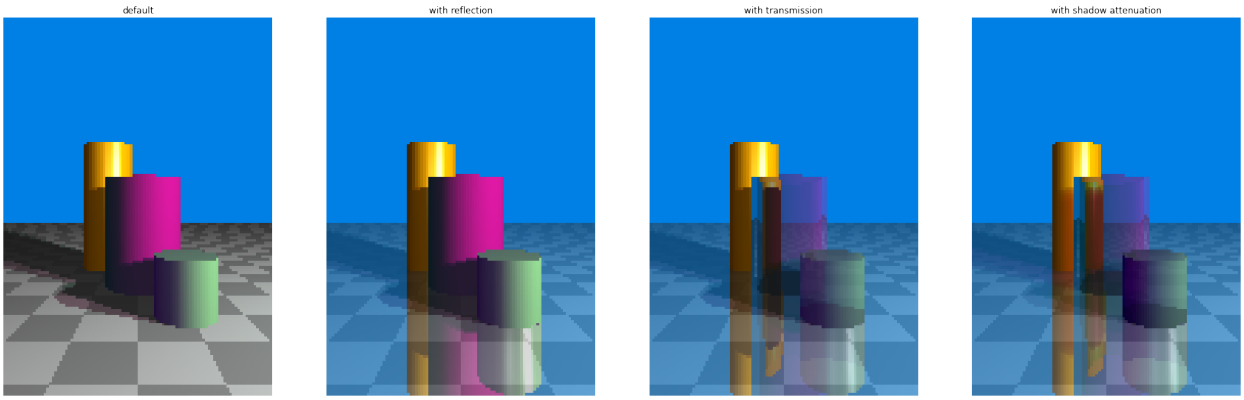
Hint: in order to calculate the path travelled within the object, you will need to create updated rays to intersect the object. A diagram is shown as follows:



(Also note that due to the simplicity of our objects – planes and cylinders – once a straight ray passes through the object it will not pass through that object again, which is not the case for more complex/arbitrary geometry. You may use this knowledge to your advantage!)
Action: Now enable shadow attenuation in the cell you modified to enable reflection/transmissions in the section above. You should now see tinted shadows next to/behind the transmissive objects.

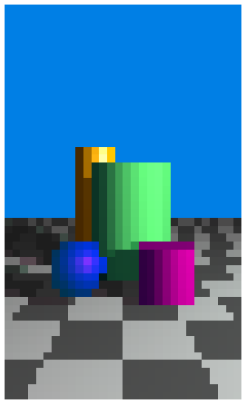
TODO5. Final comparison

Action: Run the cell right below the one you just modified, to generate and plot four images as shown. Note that your scene geometry is slightly different from this example. **Do not change this cell – just run it as is!**



3.1 Bonus Credit

As an easy extension (for 1 pt bonus credit), figure out what is needed to incorporate the spheres you defined in last week's assignment and add it in the last cell. After uncommenting the image generation code, you may see something like this (but more high resolution):



As an harder extension (for 1 pt bonus credit and a sense of accomplishment), figure out how to start upgrading this raytracer to a distributed raytracer via depth-of-field! Refer to the slides for the diagram of what you'd need to do to get this done. In particular, one thing that we didn't cover in lecture at all is how to actually do any of the random sample generation (which as we mentioned in lecture is a crucial component of Monte Carlo methods!) Read through https://mathworld.wolfram.com/DiskPointPicking.html and the starter code, and think about how you'd write a modified `generateImage` function that additionally takes in parameters for depth of field such as

```
#old
rendered_image = generateImage(screen_height, screen_width,
                               camera_aperture, film_xlim, film_z,
                               raytrace_settings, objects,lights)
#new
rendered_image_dof = generateImageDOF(screen_height, screen_width,
                               aperture_position, aperture_radius,
                               num_camera_rays_per_pixel, focal_plane_distance,
                               film_xlim, film_z,
                               raytrace_settings, objects,lights)
show_image(rendered_image_dof, height = 8, width = 8)
```

Feel free to reach out on Piazza/OH if you want to tackle this but aren't sure where to start.

3.2 Additional Resources

We've given you a super simple barebones raytracer written in Python, which is extremely inefficient and is not what you'd ever use in a production setting. That being said, the data structures and logic is fashioned in a way that will make it easy for you to pick up development in more advanced high-performance raytracers should you be interested in doing so! If you're interested in learning more about the complex work that goes on with Monte Carlo-based raytracing and pathtracing systems, we highly recommend looking at https://pbrt.org/ as well as https://gfxcourses.stanford.edu/cs348b/spring22.

Lastly, this research paper from Disney describes the BRDF (later extended to BSDF) that is the algorithm behind the Principled BSDF shader in Blender. Besides a further/deeper look at the technology behind the shader, the diagram on page 13 is useful for a more "principled" approach to playing around with textures and shaders!

4 Blender Component

For this week's Blender component, we're trying a slightly different format here. Lecture 8 really benefits from some hands-on demonstrations with textures, so the instructions/what you'll need to do are mostly contained within the lecture itself. As such, we recommend watching or rewatching the lecture 8 recording before starting. The instructions here are minimal, but we have demonstrations during class that will hopefully be quite easy to follow along. In particular, you will need to do the following:

1. Setup or load a blender scene that has at least two objects modeled by you (not including primitives such as a ground plane.) You should be rendering in Cycles for your submitted renders, with at least 600x800 resolution for the homework submissions, and should hopefully have moved to a nice lighting setup (not required for this assignment, but note that these Blender assignments are really designed to help you start building your final project.) We recommend iterating on Eevee and switching to high resolution Cycles for the submission.

2. **UV unwrap and paint a UV texture for 1 handcrafted model. (2 points)**

   (a) First, select an object and look at the **Material Properties** tab in the properties editor (the icon that is a circle with 4 quadrants in the lower right). Use the Principled BRDF material (check HW2 if you've forgotten how to do this), and click on the little yellow dot next to **Base Color**.

   (b) In the panel that pops up, select **Image Texture**. Create a new window partition in your user interface and select **UV Editor.**

   (c) Now go back to your viewport, click on the object that you want to paint, and go into Edit mode. Select **UV → UV Unwrap** on the top bar in the viewport, and you should see triangles show up in the **UV Editor**.

   (d) Optional, but highly recommended: try iterating with selecting a string of vertices (CTRL+left click) and using **UV → Mark Seams**, as well as smooth falloff UV coordinate manipulation in the UV editor, to get a UV map you're happy with. Triangles that you want more detail in should have more area in the UV space (Watch the lecture for details.)

   (e) Once you're happy with the UV coordinates, switch the viewport to **Texture Paint** mode and switch the UV editor to **Image Editor**. Paint your model by going between the two modes. You may find the masking tool mentioned in class helpful as well. (Summary from lecture: Select a closed face in Edit Mode in the viewport, hit Ctrl+L to select all enclosed by a seam, then go back to Texture Paint Mode.)

   (f) **Action:** Save your image texture at **HW4/images/painted_texture_map.png** by clicking **Image→Save** in the Image Editor.

   (g) **Action:** Save your render (at least two objects, one with painted texture at this point) at **HW4/images/painted_texture_render.png**

   (h) Make sure the first Blender cell in the jupyter notebook displays your render successfully.

3. **UV unwrap, scale, and assign a seamless texture for a second handcrafted model. (1 point)**

   (a) For a second object that is not the one you worked with in the steps above, repeat 2(a).

(b) Click on the folder icon below the Base Color property, and load in a seamless texture (you may want to pick one from https://renderman.pixar.com/pixar-one-thirty or find your own.)

(c) Open up the **UV Editor** and repeat 2(c).

(d) In the UV editor, select all uv coordinates (Ctrl+A) and scale them until the texture scale looks good in the viewport.

(e) **Action:** Save your render (at least two objects, one with painted texture, one with seamless texture at this point) at **HW4/images/seamless_texture_render.png**

(f) Make sure the second Blender cell in the jupyter notebook displays your render successfully.

4. **Make a blended shader by experimenting with the shader nodes. (2 points)**

(a) For this part, we strongly recommend watching the relevant portion of the lecture.

(b) Click on the object you want to create a blended material for, and switch the UV Editor to a **Shader Editor**. Play around with the Nodes available, and go between the viewport and the render view until you're happy with the results. You must have more than the default "Image Texture Node+ BRDF + Output Node" at the end. The example we showed in class, with two principled BRDFs and a single mixer node, is acceptable, but we encourage you to experiment. In particular, consider the following tutorials as some good places to start:

- Neon light material
  (2 minute video mixing emission and principled BSDF)
- Normal mapping, roughness, and albedo
  (6 minute video on the basics that we covered in class)
- Procedural shading
  (13 minute video on playing with gradients and random noise generators)

Please share any interesting shader tutorials in the homework Piazza post!

(c) **Action:** Screenshot your shader graph and save to **HW4/images/shader_graph.png**

(d) **Action:** Save your render (at least two objects, one with painted texture, one with seamless texture, at least one having some complex shader node graph at this point) at **HW4/images/shader_graph_render.png**

(e) Make sure the third Blender cell in the jupyter notebook displays your render successfully.

(f) **Action:** Fill in the details of what you worked on for the shader graph in the final cell.

Additional Resources

The other thing we mentioned in lecture is environment mapping and skyboxes. This is not required for the assignment, but we also recommend looking at this example of how to set up environment lighting. Have fun!