Assignment 3: Path Tracing

In this assignment, you will implement path tracing to create photorealistic renderings of 3D scenes with global illumination. No starter code will be provided. Like in Assignment 1, you will compute the rendered image yourself on the CPU and then display it — only this time, you will do it using path tracing instead of rasterization.

Part 1

1. Start by implementing a very basic ray tracer. For now, define a scene to be a list of spheres with specified centers and radii (we will change this later). Assume the camera is located at the origin and pointed along the –*z* direction with a fixed vertical angle of view (choose something between 40° and 90°), so that you can compute the ray **o** + *t* **d** for any pixel location. Intersect the ray with the scene, find the normal **n** of the nearest hit point, and set the pixel colour to be ½ (**n** + (1,1,1)), or (0,0,0) if nothing was hit. Even without doing any lighting calculations, this will allow you to see the shape of the objects in the scene. (Once you've computed the entire image, you will of course have to open a window to display it, and/or save it to a file.)

    - It is convenient to make the ray intersection test accept parameters for the minimum and maximum values of *t* within which intersections will be considered. To avoid shadow acne, for example, you will want to set $t_{\min}$ to some very small positive value.
    - You are not required to implement any spatial data structure like a BVH or a kd-tree.
    - Test your ray tracer on a scene with a unit sphere centered at (0, 0, –2) and a sphere of radius 100 centered at (0, –101, –2). Check that the result looks like a perfectly round sphere resting on an approximately flat ground.

2. Modify your scene representation to allow other shapes to be included, namely, infinite planes and axis-aligned boxes. (You can also implement other shapes if you want.) A natural way to do this is to make all these shapes subclasses of an abstract "shape" class. Each concrete shape should provide its own ray intersection function that returns the nearest valid intersection of the given ray with it. Replace the second sphere with the plane *y* = –1 and add some boxes to the scene to verify that your new shapes are working correctly.

3. Add some lighting. Modify the scene representation to also have a list of light sources (assume point sources with specified intensity in RGB for now), and give each shape a

diffuse albedo as an RGB triple. For each pixel, set the colour to be the radiance going towards the camera based on the Lambertian model, adding up the contribution of each visible light source.

- The light intensity can be arbitrarily large, but the diffuse albedo must be between 0 and 1.
- We have discussed in class how to compute the irradiance at any surface location due to a point light source. The Lambertian BRDF then tells us how to compute the exitant radiance from the irradiance.
- Trace shadow rays towards each light source to determine if it is visible from the surface point. Remember to use a small bias value to avoid shadow acne, and to ignore any surfaces hit by the shadow ray *after* passing the light source (this is what the $t_{\min}$ and $t_{\max}$ parameters are for)!

4. ~~Optional:~~ (**Correction:** not optional) Add a viewing transformation that allows you to move and rotate the camera. As discussed in one of the earlier lectures, you can do this by specifying the center of projection, the view vector, and the up vector, and using them to construct an affine transformation matrix. Also add affine transformation matrices to each shape so that you can move, rotate, and scale them.

   - To intersect a ray with the transformed shape, simply transform the ray with the inverse matrix and then intersect it with the original shape. But remember that the normal should then be transformed with the *inverse transpose* of the transformation matrix, as discussed in class.