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.

Part 2

5. If you have been directly setting the RGB colour of the pixel to the computed radiance values, this is incorrect! The computed radiance is in “linear RGB” i.e. proportional to the amount of light energy. To encode it in values which will be displayed correctly, use the sRGB gamma-correction formula discussed in lecture 21.

6. Extend your ray tracer to support specular reflection and refraction, from perfectly smooth metallic and transparent surfaces respectively. You may want to implement an abstract “material” class of which all these are subclasses. Each time you hit such a surface, shoot one or two secondary ray(s) to get the incident radiance from the reflection/transmission direction (again with bias of course). Choose a maximum recursion depth so your renderer doesn’t get stuck in an infinite loop, and a sky colour for rays that go to infinity.

   - The Fresnel coefficients affect both metallic and transparent materials. Use Schlick’s approximation $F = F_0 + (1 - F_0)(1 - \cos\theta)^5$ to compute the reflectance coefficient.

- For metals, store the value F_0 directly as a colour (e.g. copper has a reddish-orange F_0).
- For transparent materials, also called dielectrics, store the refractive index η (e.g. 1.333 for water, 1.5–1.7 for glass, 2.417 for diamond). Then $F_0 = ((\eta_1 - \eta_2)/(\eta_1 + \eta_2))^2$, where the η's are the refractive indices on the two sides of the surface. The outgoing radiance should be $FL_{\text{reflected}} + (1 - F)L_{\text{transmitted}}$, where F is computed using the larger of θ_i and θ_o. Don't forget to consider the case of total internal reflection!

7. Finally, add path tracing to your renderer! You are only required to do the simpler version without direct/indirect splitting; just shoot at most one ray from each bounce to sample the incident radiance, and forget about light sampling or shadow rays. Add an outer loop where you trace multiple paths for each pixel and average the results.

   - Point lights don't work very well with path tracing, so replace each light with a finite-size shape (a sphere or a box) having a purely emissive material that emits a constant radiance.
   - Use Russian roulette at each bounce to get an unbiased result. You should no longer have any explicit cap on the recursion depth. Instead, if you want *n* bounces on average, set the continuation probability to 1 – 1/*n*.
   - Use importance sampling whenever choosing which ray to sample. This includes cosine-weighted sampling of incident radiance at diffuse surfaces, and Fresnel-weighted sampling of reflected vs. transmitted rays at transparent surfaces.
   - Path tracing takes a large number of samples to produce a low-noise image. So if you want to see what's going on in between, you may want to organize your code to save the intermediate image after *n*, 2*n*, 3*n*, … samples per pixel.

8. **Optional:** Add direct/indirect splitting of the incident radiance, as discussed in class. This should make diffuse surfaces much less noisy. However, you will have to figure out how to turn it off for specular (metallic and transparent) surfaces, otherwise reflections and refractions of light sources may no longer work!

9. Set up a scene that shows off the features of your path tracer. It should demonstrate all three types of shapes, affine transformations, soft shadows, recursive reflection and refraction, as well as indirect illumination and caustics (which come for free with a path tracer). You should also put a little effort into the choice of colour scheme, lighting, and viewpoint to make it aesthetically pleasing.

Submission

Submit your assignment on Moodle before midnight on the due date. Your submission should be a zip file that contains your entire source code for the assignment, not including any binary object files or executables. The zip file should also contain a PDF report that includes various images produced by your renderer:

1. The test scene in step 1, rendered in "normals visualization mode".
2. The same as (1) but with a ground plane and some boxes (step 2).
3. The same as (2) but with diffuse lighting from at least 2 light sources, one far away so it approximates a directional light.
4. A scene with several rotated boxes and stretched spheres (i.e. ellipsoids), showing that normals are transformed correctly.
5. Any lit scene of your choice, one image rendered without gamma correction and one with.
6. A scene with at least one coloured metallic sphere and one transparent glass sphere.
7. A scene like the Cornell box, showing diffuse interreflection between walls of different colours.
8. If you implemented part 8, any lit scene of your choice, one image rendered without direct/indirect splitting and one with.
9. The aesthetically pleasing scene of your own design, demonstrating various light transport effects.

If there are any components of the assignment you could not fully implement, in the PDF you should also explain which ones they are and what problems you faced in doing them.

Separately, each of you will *individually* submit a short response in a separate form regarding how much you and your groupmate(s) contributed to the work in this assignment.