Stanford CS248A: Computer Graphics
Participation Exercise 7

This exercise is graded for CREDIT ONLY. Serious attempts to answer the problems will be given full credit, even if the answers are incorrect.

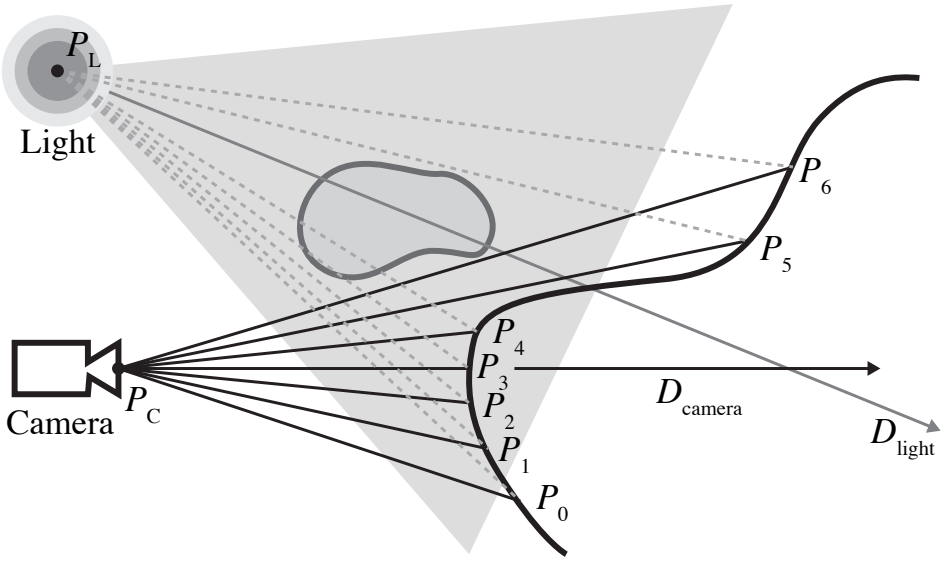
Problem 1: A Highly Irregular Rasterizer

Imagine that you have a special kind of rasterizer which doesn't evaluate depth/coverage at uniformly spaced screen sample points, instead it evaluates depth/coverage **at a list of arbitrary 2D screen sample points provided by the application**. An example of using this rasterizer is given below. In this problem you should assume that depths returned by `fancyRasterize` are in WORLD SPACE UNITS.

```
vector<Point2D>  myPoints;  // list of 2D coverage sample points: in [-1,1]^2
vector<Triangle> geometry;  // list of scene triangles in WORLD SPACE
Transform worldToCam;       // 4x4 world space to camera space transform
Transform worldToLight;     // 4x4 world space to light space transform
Transform perspProj;        // 4x4 perspective projection transform


// this call returns the distance to the closest scene element from the camera
// for all points in myPoints (assume infinity if no coverage)
vector<float> depths = fancyRasterize(geometry, myPoints, worldToCam, perspProj);
```

You are now going to use `FancyRasterize` to render images with shadows. Consider the setup of a camera, scene objects, and a light source as illustrated below.



A. Assume you use a traditional rasterizer to compute the depth of the closest scene element at each screen sample point. In the figure, the closest point visible under each sample when the camera is placed at position P_C and looking in the direction D_{camera} is given by P_i. All points in the figure are given in **world space!**

Assume you are given a *world space to light space* transform `worldToLight`. (Light space is the coordinate space whose origin in world space is P_L and whose -Z axis is in the direction D_{light}.) Describe an algorithm that computes, for each point P_i, if the point is in shadow from a point light source located at P_L. Your algorithm accepts as input an array of world space points P_i, world space points P_C, P_l, and has access to all variables listed in the example code. The algorithm should call `fancyRasterize` only once. (No, you are not allowed to just implement a ray tracer from scratch.)

Hint: Be careful, `fancyRasterize` wants points in 2D (represented in a space defined by the $[-1, 1]^2$ "image plane") so your solution will need to describe how it converts points in world space to a list of 2D sample points in this plane. **This involves transformation, perspective projection via `perspProj`, then convert from a homogeneous 3D representation to 2D.**

B. Does the algorithm you gave above generate "hard" or "soft" shadows? Why? (You can answer this question even if you did not correctly answer part A—just assume a solution that does what was asked in part A exists.)

C. Prof. Kayvon quickly looks at the algorithm you devised above and waves his hand dismissively. He says, "remember I told you in class that shadow mapping is such a hack", it only yields an approximation to ray traced shadows. The CAs jump in and shout, "Wait a minute here, this algorithm seems to compute the same solution a ray tracer would to me!" Who is correct? Why?

Problem 2: Diving into a Path Tracer

Consider the following pseudocode for a path tracer. It is the same code shown in lecture. It employs Russian Roulette to terminate paths with probability q.

```
// return radiance along ray 'ray'
Spectrum PathLo(Ray ray) {
  Spectrum Lo = 0, beta = 1;
  while (true) {
    Intersection isect = scene->Intersect(ray); // find scene intersection
    Vector3f wo = -ray.d;
    if (depth == 0)         // *** QUESTION IS ABOUT THIS LINE ***
      Lo += isect.Le(wo);

    BSDF bsdf = isect.GetBSDF();

    // accumulate reflectance due to direct lighting
    Lo += beta * ReflFromDirectLighting(bsdf, wo);

    // generate new ray direction wi, and evaluate BSDF given wo and wi.
    Spectrum fr = bsdf.Sample_f(wo, &wi, &pdf);

    // update path throughput before next step along path
    beta *= fr * Dot(wi, isect.N) / pdf;

    float q = 0.25;
    if (randomFloat() < q)
       break;         // terminate path
    else
       beta /= (1-q); // update path throughput
  }

  return Lo;
}
```

A. Notice the line of code with the comment `*** QUESTION IS ABOUT THIS LINE ***`. Please explain why the algorithm only accumulates surface emission into the path's output radiance if the path depth is 0. Note that path depth = 0 means this is a camera ray.

B. In your own words, why is it generally a good idea to special case the sampling of direct lighting (like in the code above), rather than implement the simple form of path tracing psuedocoded on slice 43? (Hint: how is this a form of importance sampling?)