CS 174A (Graphics): Assignment 3

Weight: 15%, divided into 40 points

For project 3 you will use ray tracing to sample scenes, created according to arbitrary text files describing the location of balls and lights. You will first download an updated version of your familiar code template, with minor changes plus one new file that houses project 3: `ray-tracer.js` (a `Scene_Component` of about 180 lines) plus a file for test cases. Besides four new buttons, all the old controls (click below to expand) still apply.

Firefox is unable to properly keep our image current, so use Chrome this time. To play with a working solution of this project, use the blue button above. Your output should be qualitatively the same as the solution's.

The task:

Fill in blanks (marked with TODO comments) in four functions in `ray-tracer.js`, in this order:

`Ray_Tracer::get_dir():` Maps an (x,y) pixel to a corresponding xyz vector that reaches the near plane. Once you finish this function, everything under the "background effects" menu should start working.

`Ball::construct():` You are provided with a `Ball` class with some data members that automatically get filled in by `Ray_Tracer::parse_file()`. Using those for calculation, finish filling in data members. You may think of more to add in later. `Ray_Tracer` already stores an array of `Ball`s. Once this function is finished, functionality to raster the image (turn ray tracing off) will work.

`Ball::intersect():` Given a ray, check if this `Ball` is in its path. Recieves as an argument a record of the nearest intersection found so far (including a `Ball` pointer, a t distance value along the ray, and a normal), updates it if needed, and returns it. Only counts intersections that are at least a given distance ahead along the ray.

Tip: Once `intersect()` is done, call it in `trace()` as you loop through all the spheres until you've found the ray's nearest available intersection. Simply return a dummy color if the intersection tests positive. This will show the spheres' outlines, giving early proof that you did `intersect()` correctly.

`Ray_Tracer::trace():` Given a ray, return the color in that ray's path. The ray either originates from the camera itself or from a secondary reflection or refraction off of a ball. Call `Ball.prototype.intersect` on each ball to determine the nearest ball struck, if any, and perform vector math

(the Phong reflection formula) using the resulting intersection record to figure out the influence of light on that spot. Recurse for reflections and refractions until the final color is no longer significantly affected by more bounces.

Arguments besides the ray include `color_remaining`, the proportion of brightness this ray can contribute to the final pixel. Only if that's still significant, proceed with the current recursion, computing the Phong model's brightness of each color. When recursing, scale `color_remaining` down by `k_r` or `k_refract`, multiplied by the "complement" (`1-alpha`) of the Phong color this recursion. Use argument `is_primary` to indicate whether this is the original ray or a recursion. Use the argument `light_to_check` when a recursive call to `trace()` is for computing a shadow ray.

Your code already parses text lines and stores their values in classes: either `Ray_Tracer`, `Light`, or `Ball`. With each successive line, the text file could assign to one of the following fields:

1. NEAR, LEFT, RIGHT, BOTTOM, and TOP, specifying axis-aligned distances to the edges of the near plane, for a viewing frustum in front of the camera.
2. RES, specifying the ray sampling resolution. Only certain sizes are allowable. Your code works by drawing to a texture, and only power of two sizes can be stored in WebGL texture buffers.
3. SPHERE, using triples of floats to describe the position, size, and color of a new ball. Seven more floats describe the material: k_a (ambient glow from no particular source), k_d (diffusive reflectivity), k_s (shininess), n (specular exponent), k_r (reflectivity), k_refract (refractivity), and refract_index.
4. LIGHT, triples of floats that describe the position and color of point lights.
5. AMBIENT, the ambient lighting applied to special background effects, and BACK, the solid color of one of those effects.

Several samples of these scenes are provided in `Test_Cases.js`.

Grading scheme:

[5 points]	Coding Style (i.e., well designed, clean, commented code)
[10 points]	Ability to cast a ray and display the spheres properly
[5 points]	Local illumination
[10 points]	Shadows
[10 points]	Reflections and refractions

Notes:

- You'll need the matrix inverse. Place or cache it carefully to avoid a performance hit.
- Given a reasonable resolution (256x256), your program should take no more than a few seconds to run.
- A parametric position along a ray between 0 and 1 falls between the eye and the near plane, and hence is not a part of the view volume. Note that recursion rays do not have near planes, and require only 0.0001 of breathing room instead.
- If you create shadow rays as a vector directly from the closest hit point to the lights, then you are looking for any intersections with hit time between 0.0001 and 1. To deal with floating point roundoff, you should not consider an intersection at time 0 to be blocking the light from the object.
- A positive "NEAR" value represents the distance along the negative z-axis.
- Your code should handle hollow spheres which are "cut" by the near-plane.
- See the lecture notes for important slides on illumination formulas and on sphere intersection tests.
- You may use the following rough pseudocode of a local illumination model:

```
vec3 surface_color = k_a * sphere color + for each point light source (p) {
                       this.lights[p].color * (
                     k_d * (  N dot L,    positive only) * (the sphere's color) +
                     k_s * ( (N dot H)^n, positive only) *  white       ) }
```

and then:

```
vec3 pixel_color = surface_color + (white - surface_color) *
                ( k_r * trace(...).slice(0,3) + k_refract * trace(...).slice(0,3) )
```

- Be sure to clamp each component of surface_color, the above vec3, to at most 1.
- When multiplying two colors, do coefficient-wise products using mult_3_coeffs().
- Since this is javascript, operators are not defined; forgetting to convert just a single operator like * or - into a scale_vec, mult_vec, mult_3_coeffs, add, or subtract could lead to an undefined crash. This will happen a lot.
- Your intersect() function will come up with wrong answers if you dot quantities together when they are all still vec4s.
- The quadratic formula from the lecture slides is correct. Note that the definition of B includes the 2. That way, the two gets squared and factored out, and then cancels the coefficients of the 4ac and the 2a so you get the simpler formula shown on the slide.
- There are three possibilities if you intersect a sphere: The near plane could come before, between, or after the two intersection points. Choose the point you go with accordingly. If in the "between" case just mentioned, and you're seeing the inside of a sphere, you should make your normal

based on that inner surface. Rastered graphics, shown with ray tracing off, are incapable of this correction.

- Your recursive calls to trace each go in different directions with new rays. It's up to you to calculate those directions. When keeping track of "color remaining" to stop your recursion, rememeber how to calculate the next call's remaining potential to be lit up more:

(k_r or k_refract) * color_remaining * (white - surface_color)

- Collaboration: None. If you discuss this assignment with others you should submit their names along with the assignment material. Any students copying from an online, shared, or de-obfuscated solution will have submissions that are structurally close enough to declare plagiarized.
- Start working on this assignment early. You will not have time to do it well at the last minute.
- Submit this on CCLE in a .zip archive, and name the file your bruin ID. Include the whole template (all files) in your submission.