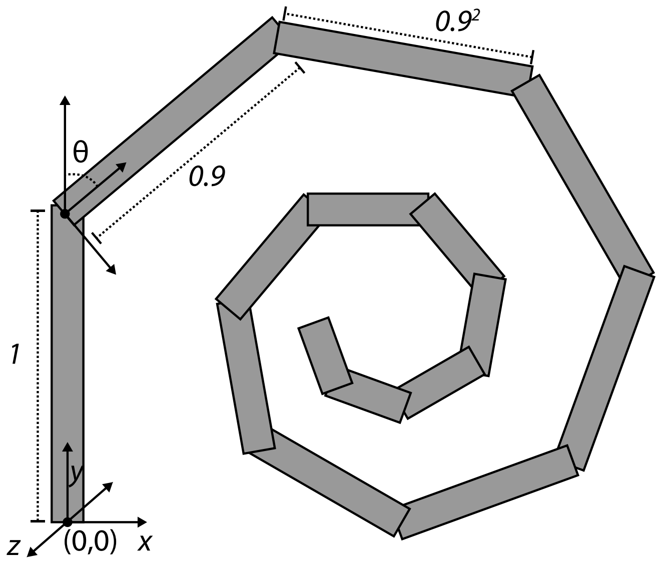
Stanford CS248: Interactive Computer Graphics
Participation Exercise 3

Problem 1: Ray Tracing A Funky Scene

Consider a scene contain 15 rectangular-boxes as shown below. The scene is formed by starting with a box of length=1.0, with it's center-bottom at the origin, and extending down the Y axis (the width and depth of the box do not matter in this problem). The next box begins at the end of the first box (at point (0,1,0)), but it has a length that is 0.9 times the first box, and rotated with an angle θ relative to the first box. Note that the rotation is about the Z axis, in the *clockwise direction* as viewed when looking down the -Z axis. This pattern of starting the next rectangle at the end of the previous, rotating by θ, and shrinking length by a factor of 0.9 repeats for a total of 15 boxes.



You are given a function `void ray_rect_isect(Ray r, float rect_length)` that computes the first hit of a ray r, with a box aligned with the Y axis of length `rect_length` (see code on next page) For example, the call `ray_rect_isect(r, 1.0)` would compute the intersection of a ray with the first box. Upon return `r.min_t` would reflect the distance to the closest hit. *If the original `r.min_t` of the ray passed into the function is smaller than the t computed from the intersection, then `r.min_t` is unchanged.*

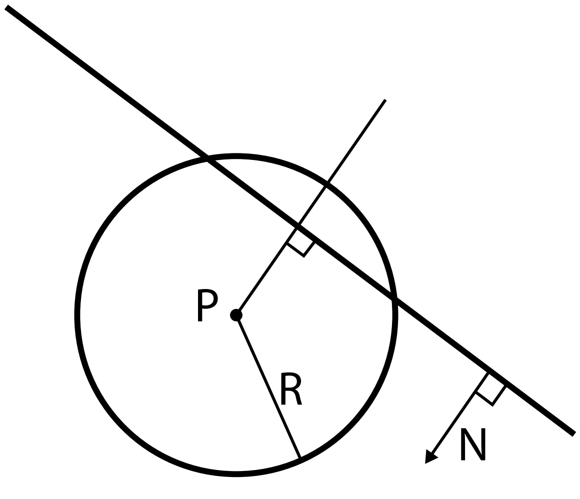
You are also given a function `rotatez` that takes a 3-vector v and rotates it θ degrees about the Z axis in the **counter-clockwise direction.**. Using only these routines, on the next page, please implement the function `isect_funky_scene(Ray& r)`, which will fill in `r.min_t` to contain the closest point on the ray. You should make no assumptions about the origin or direction of the ray in 3D, except that it does not originate from inside any of the boxes.

Hint: be careful about the direction of your rotations. The geometry rotations in the figure are clockwise, and the function `rotatez()` specifies a rotation in the counter-clockwise direction as was typical in class.

```
// students are free to assume that useful constructors, or common ops like
// addition, multiplication on vectors is available..
struct vec3 {
  float x, y, z;
};

struct Ray {
  vec3 o;
  vec3 d;
  float min_t;  // assume this is initialized to INFINITY
};

// rotate counter-clockwise about Z axis (counter clockwise defined when looking down -z)
vec3 rotatez(float theta, vec3 v);

// fills in r.min_t if intersection with rectangular box is closer than current r.min_t
void ray_rect_isect(Ray r, float rect_length);

// assume r.min_t is INFINITY at the start of the call
// result: fill in r.min_t as a result of intersecting r with the 15 segments
void isect_funky_scene(Ray& r) {









}
```

Problem 2: More Geometry Intersection

An interesting fact is that testing whether a plane given by $\mathbf{N}^T\mathbf{x} = C$ and a sphere with center point $P = (P_x, P_y, P_z)$ and radius r intersect can be boiled down to closest point on plane. Consider a ray moving from P toward the plane (in the direction -$\mathbf{N}$. (The intersection of this ray with the plane gives the closest point on the plane to the center of the sphere.)



A. Give an algorithm for determining if the sphere and plane intersect. As part of this algorithm, give an expression for the closest point on the plane to P. Recall that the equation for points on a sphere of radius r centered at P is:

$$(x - P_x)^2 + (y - P_y)^2 + (z - P_z)^2 = r^2$$

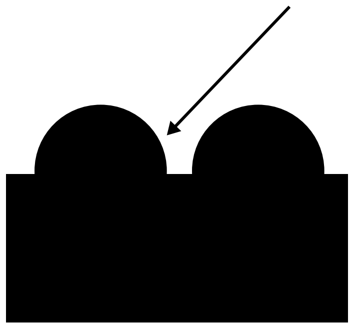
B. Use your algorithm from part A as a subroutine in an algorithm for computing **whether a triangle with vertices T_0, T_1, T_2 intersects a sphere with center point P and radius r.** In addition to results from part A, your solution may also use ray-sphere intersection and point-in-triangle tests as subroutines (see functions below). However, you should clearly state how you use the results of these subroutines. (e.g., how to you use the T-value returned on a hit.)

   Your solution can assume that the triangle is not entirely inside the sphere, but should make no other assumptions.

```
// returns true if intersection exists, sets r.min_t to the closest of the
// (potentially two) intersection points (including the case where the ray
// starting within the sphere).
bool ray_sphere_isect(Ray r, vec3 sphere_center, float sphere_radius);

// returns true if the point, assumed to be in the plane of the triangle,
// is within the triangle, false otherwise.
bool in_triangle(vec3 pt, vec3 t0, vec3 t1, vect3 t2);
```

Problem 3: Intersecting Solids

Consider the 2D shape given below, which is made up of the intersection of volumes contained by two circles and a rectangle.

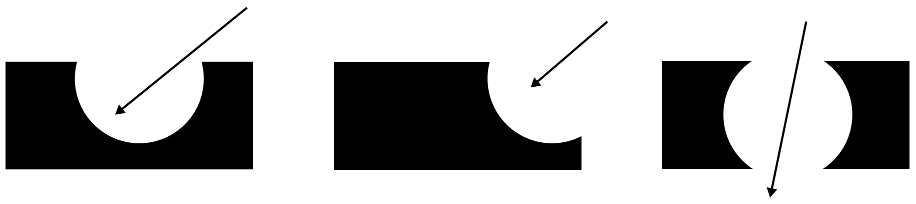


Assume you are given two functions for ray-shape intersection as given below. Both functions return true if there is an intersection (false otherwise), and fill in ray t values for up to two hits. If there is a single intersection point, such as when the ray grazes the shape, just assume that both t values are the same. If there is no intersection, assume the t's are set to a very large number.

```
bool rayBoxIsect(Box b, Ray r, float* t1, float* t2);
bool rayCircleIsect(Circle c, Ray r, float* t1, float* t2);
```

A. Assuming we call the circles c_1 and c_2, and the box b, give an algorithm for finding the closest intersection of a ray with the shape above.

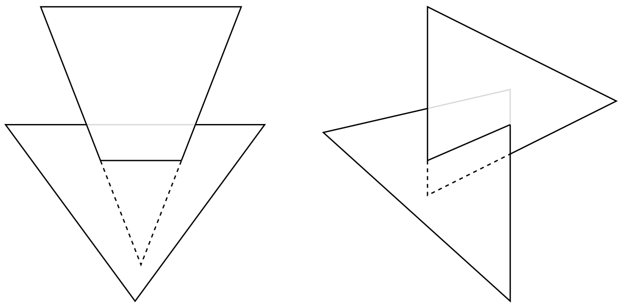
B. Now consider a new kind of shape, which is formed by *subtracting the region inside one circle from the region inside one box. (a few cases are given to help your understanding.)*



Give an algorithm for finding the closest intersection of a ray with this new type of shape. *Note, please make sure you handle the case where the shape is actually two disjoint pieces, and the case where there is no intersection! However, to make things simpler, you can assume the ray origin is outside the area of the box or the circle.*

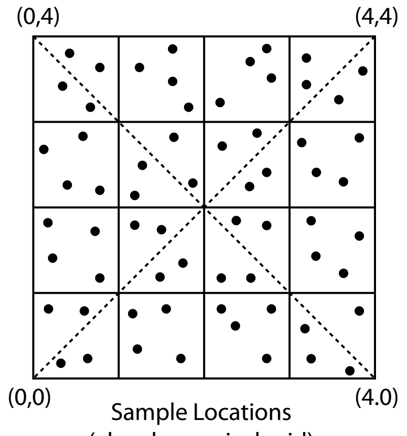
Problem 4: Intersecting Triangles (THIS PROBLEM IS OPTIONAL PRACTICE AND NOT GRADED)

Assume that you are given two triangles, A and B, defined by vertices $(\mathbf{p_{a0}}, \mathbf{p_{a1}}, \mathbf{p_{a2}})$ and $(\mathbf{p_{b0}}, \mathbf{p_{b1}}, \mathbf{p_{b2}})$ and function `(bool, float) ray_tri_isect(Ray r, Triangle tri)` which returns true if an intersection exists and, if so, the value of the ray's t at which a ray intersects the triangle. Given these methods, describe an algorithm that computes whether triangle A intersects with triangle B. (Rough pseudocode is fine, it need not be compilable, but be precise about how you will use the results of `ray_tri_isect()`.) To keep things simple, you do not need to worry about the case where triangles are co-planar and self-contained.

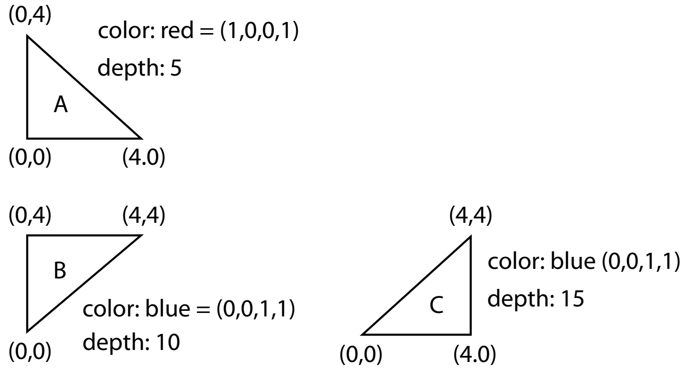


Problem 5: Rasterizing Triangles (THIS PROBLEM IS OPTIONAL PRACTICE AND NOT GRADED)

Consider rasterizing the three triangles (A, B, C) given below (at right) to a 4×4 pixel image with 4× supersampling shown at left.. The coordinates of image space are given on the figure (We're using the convention that (0,0) is in the *bottom left*.) Note that unlike assignment 1, the four sample positions per pixel are now placed at **random locations** in each pixel.

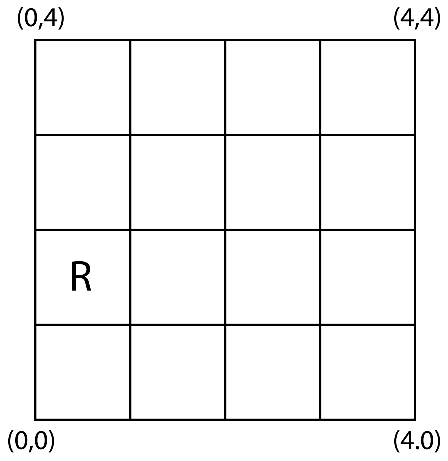


Sample Locations
(also shows pixel grid)



Triangles to Render

A. On the grid below, draw the final rendered output assuming that coverage and depth testing are performed at the provided sample locations, and that the supersample buffer is resampled to a final image by means of **convolving the supersample buffer with a 1-pixel wide box filter**. For simplicity, define the values of several color RGBA variables and just write the variable name in each pixel. (e.g., let R = (1,0,0,1), and one red pixel has been marked for you in the figure.)



B. How would the results in part A change if the resampling filter in part A was replaced with a 3-pixel wide box filter? You only need to answer in words – you do not need to illustrate a result. (You many ignore boundary conditions as well.)

C. Now assume triangle A's color is changed to be 75% opaque red. Recall that in a **non-multiplied alpha representation** this is $C = (1.0, 0, 0, 0.75)$.

Now, assume the renderer is changed to work in the following way... Instead of updating ALL SAMPLES COVERED BY A TRIANGLE that pass the coverage and depth tests, and doing so using the alpha blending equation $C_{\text{new}} = \alpha C_{\text{tri}} + (1 - \alpha) C_{\text{old}}$, the renderer discards a fraction of the triangle's covered samples according to α. Specifically, for a triangle with opacity 75%, the renderer **randomly discards** 1-.75=25% of the samples covered by the triangle, and for all other covered samples, treats the triangle as if it is fully opaque (e.g., has color (1.0, 0, 0, 1.0)).

Assuming the supersample buffer is resolved to a single sample per pixel using a 1-pixel box filter (as was done in Part A), **describe why the new rendering scheme results in the same answer as if alpha blending was used on all covered samples.**

Hint: To keep things simple, your answer need only consider the case where the transparent triangle covers a entire pixel. A clear answer will describe why proposed algorithm gives the same result as the equation above, showing that the math works out the same.

**** For extra credit, give (1) a clear explanation why, in expectation, this approach can rendering transparent surfaces in any order using regular depth testing and no alpha blending. (2) consider why it might not work so well with only a small number of samples per pixel.*