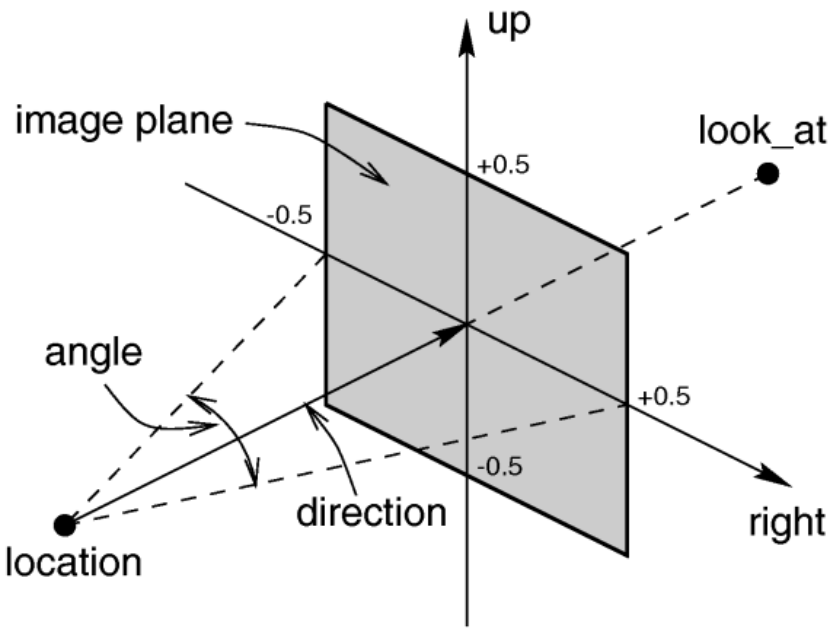
N. A Tiny Raytracer

In this problem, you're to implement a tiny *ray tracer* that renders a scene consisting of a single *point-light* and some triangle meshes.

The camera model

The camera in this problem is a perspective camera located at **camera_pos** (3D point) and looking at **camera_target** (3D vector) with an up vector **camera_up** (3D vector). Its horizontal field of view is **f** degrees, and can produce pictures of **W** x **H** (in pixels).



Don't worry if you could not understand the last paragraph. Just look at the picture above. Imagine a so-called image plane (it's actually a rectangle) in front of the camera. The final image (i.e. the output of this problem!) is the discrete version of the image plane.

Then **camera_up** is the y-axis of the picture, and the ray passing through **camera_pos** and **camera_target** is passing through the center of the image, note that we do not care about how far the target is; only the origin (i.e. **camera_pos**) and direction of the ray is relevant. The horizontal field of view is the angle between the left and right boundaries of the image.

To calculate the projected coordinate (i.e. the corresponding location of a 3D point in the image plane) of an arbitrary point **P** in 3D, simply make a ray from **camera_pos** to **P**, then the intersection of this ray with the image plane is what you want. It's not difficult to show that the position of the image plane does NOT change the projected coordinate of any point (given that the projected coordinate is normalized, as in the picture), so you can choose any plane to be the image plane, as long as its y-axis is **camera_up**, and the ray from **camera_pos** to **camera_target** passes through the center of the image plane. Note that every pixel is a square, so the width/height ratio of the image plane is always W:H.

Raytracing overview

Here is how ray tracing works: for each pixel in the final image, we define a ray that extends from the camera to the **center** of the pixel (please look at the picture above again). We follow this ray out into

the scene and as it bounces off of different objects. The final color of the ray (and therefore of the corresponding pixel) is given by the colors of the objects hit by the ray as it travels through the scene.

In the simplest case, all the objects are neither reflective nor transparent, so every time a ray hits an object, we follow a single new ray from the point of intersection *directly towards* the light source, to calculate the color of the ray (see below). If this new ray is blocked by an object, it is shadowed.

In realistic scenes with like glass and water, we need to consider multiple bounces from objects to handle reflections and refractions. If an object is reflective we simply trace a new *reflected ray* from the point of intersection towards the direction of reflection. The reflected ray is the mirror image of the original ray, pointing away from the surface. If the object is to some extent transparent, then we also trace a *refracted ray* into the surface. If the materials on either side of the surface have different indices of refraction, such as air on one side and water on the other, then the refracted ray will be bent, like the image of a straw in a glass of water. If the same medium exists on both sides of the surface then the refracted ray travels in the same direction as the original ray and is not bent. The exact behavior of refraction is captured by *Snell's law*: the ratio of sines of the angles of incidence and refraction is equivalent to the opposite ratio of the indices of refraction:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}$$

Note that when *total internal reflection* occurs, you should simply stop tracing that ray. Don't spawn a reflected ray for that!

Naturally, reflected rays and refracted rays themselves can spawn other ways, so we're actually defining a **tree of rays**. In order to prevent the tiny render from being too slow, we need to restrict the height of the tree, so each leaf of the tree is either a ray hitting a non-reflective and non-transparent object (or missing everything!), or the tree depth reached a maximum. If the maximum depth is set to zero, we're disabling reflections and refractions (but direct lighting is still working). In this problem, the maximum depth is always 4. Here is a pseudo-code for this recursive procedure:

```
Color trace_ray(int depth, Ray ray) {
  Color point_color = BLACK, reflect_color = BLACK, refract_color = BLACK;
  Intersection i = get_first_intersection(ray);

  if(i.objID >= 0) { // intersection exists
    double refl = scene.obj[i.objID].refl;
    double refr = scene.obj[i.objID].refr;
    point_color = get_point_color(i) * (1 - refl - refr);
    if(depth < maxdepth && refl > 0)
      reflect_color = trace_ray(depth+1, get_reflected_ray(ray, i)) * refl;
    if(depth < maxdepth && refr > 0)
      refract_color = trace_ray(depth+1, get_refracted_ray(ray, i)) * refr;
  }
  return point_color + reflect_color + refract_color;
}
```

When the depth does not reach the maximum, always trace the reflected ray as long as the object's *reflectiveness* if positive, no matter how small it is. The same is true for the *refractiveness*. The addition of two colors will be defined shortly.

Shading

There are many different ways to determine color at a point of intersection (i.e. the "get_point_color" function above). In this problem, we use *Lambertian shading* (also known as cosine shading), which determines the brightness of a point based on the normal vector at the point and the vector from the

point to the light source (recall that there is only one point light). If the two coincide (the surface directly faces the light source) then the point is at full intensity. As the angle between the two vectors increases, as when the surface is tilted away from the light, then the brightness diminishes. This model is known as cosine shading because that mathematical function easily implements the above effect: it returns the value 1 when given an angle of zero, and returns zero when given a ninety degree angle (when the surface and light source are perpendicular).

In case the result of the dot product is zero, we still may not want that part of the object to be pitch-black. After all, even when an object is completely blocked from a light source, there is still light bouncing around that illuminates it to some extent. For this reason we make use of **ambient_coefficient** and **diffuse_coefficient**, which is just (1-ambient_coefficient), in the following pseudo-code. Note that the triangles are two-sides, so we take the absolute value of the dot product. The variable *object_color* is an attribute of the object (just like **reflectiveness**, **refractiveness** or **index**), which is the color of the object when fully illuminated.

```
double shade;
if(is_shadowed(i)) // check whether the intersection point i is shadowed
  shade = 0;
else
  shade = fabs(Dot(light_vector, normal_vector)); // both vectors are normalized
return object_color * light_color * (ambient_coeff + diffuse_coeff*shade);
```

Recall that the intersection point is shadowed if and only if the segment connecting that point and the light source is blocked by an object (even if the object is reflective or refractive!). This can lead to incorrect shadows, but this is a limitation of this simplified problem, so just ignore this "bug".

In this problem, we use a triple (r, g, b) to represent an RGB color, where 0<=r,g,b<=1. To add two colors, multiply two colors or multiply a color with a constant, just treat the color as a 3D vector. It's not difficult to see that if you strictly follow the rules above, the result of adding two colors will always be a valid color (i.e. each component is between 0 and 1).

Input

There will be at most 20 test cases. Each case begins with one integer n (n<=20), the number of objects, followed by the descriptions of each object.

Each object begins with an integers p (3<=p<=25), the number of vertices. Each of the next p lines contains 3 real numbers, the coordinates of each vertex. The next line is an integer t (1<=t<=50), the number of triangles. Each of the next t lines contains 3 integers, the vertex indices of each triangle. Vertices and triangles are both numbered sequentially from 0. The last line of the object description contains 6 real numbers r, g, b, refl, refr and idx, where r, g, b (0<=r,g,b<=1) describe the object's color (for example, 1.0 1.0 1.0 means white, 0.0 0.0 0.0 mean black), refl and refr are the reflectiveness, refractiveness and media index of this object (refl,refr>=0, refl+refr<=1). Note that the index of vacuum is 1. Each triangle is two-sided, so the vertices are arbitrarily ordered. The next line contains real numbers *x*, *y*, *z*, a real number *amb* and a color *c*, that means the point light is located at (x,y,z), with ambient coefficient *amb* and color *c*. All the objects are sane (they have a reasonable structure, not just random triangles) and they do not overlap. As a result, when refraction occurs, the ray is either shooting from the vacuum or to the vacuum, but never from one object directly into another object.

The next line contains an integer q, the number of images to render. Each of the next q lines contains 10 real numbers and 2 integers. The first nine numbers are *camera_pos*, *camera_target* and *camera_up*, the last three numbers are *f*, *W* and *H* (10.286 <= f <= 100.389, 10 <= W <= 200, 10<=H<=200). *camera_up* is guaranteed to be perpendicular to camera's looking direction. The last test case is followed by a line with n=0, which should not be processed.

Output

For each test case, output two integer W and H (same as the input), and then H lines, with W colors formatted as RRGGBB (R, G, B values in hexadecimal. Letters can be either lowercase or uppercase) in each line. You can convert the output into a JPG file using the script available if the gift package, downloadable on the contest website. To reduce the impact of floating-point errors, the percentage of incorrect pixels in each image can be up to 1%. By "incorrect" we mean either r, g, b value differ from the standard output by at least 2.

Sample Input

```
7
4
552.8 0.0   0.0
0.0 0.0   0.0
0.0 0.0 559.2
549.6 0.0 559.2
2
0 1 2
2 3 0
1.0 1.0 1.0 0 0 0
4
556.0 548.8 0.0
556.0 548.8 559.2
0.0 548.8 559.2
0.0 548.8   0.0
2
0 1 2
2 3 0
1.0 1.0 1.0 0 0 0
4
549.6   0.0 559.2
0.0   0.0 559.2
0.0 548.8 559.2
556.0 548.8 559.2
2
0 1 2
2 3 0
1.0 1.0 1.0 0 0 0
4
0.0   0.0 559.2
0.0   0.0   0.0
0.0 548.8   0.0
0.0 548.8 559.2
2
0 1 2
2 3 0
0.0 1.0 0.0 0 0 0
4
552.8   0.0   0.0
549.6   0.0 559.2
556.0 548.8 559.2
556.0 548.8   0.0
2
0 1 2
2 3 0
```

```
1.0 0.0 0.0 0 0 0
8
130.0 165.0  65.0
82.0 165.0 225.0
240.0 165.0 272.0
290.0 165.0 114.0
290.0   0.0 114.0
240.0   0.0 272.0
130.0   0.0  65.0
82.0   0.0 225.0
12
0 6 3
6 4 3
2 0 3
0 2 1
7 6 0
7 0 1
4 5 3
5 2 3
6 5 4
7 5 6
2 5 1
5 7 1
1.0 1.0 1.0 0 0 0
8
423.0 330.0 247.0
265.0 330.0 296.0
314.0 330.0 456.0
472.0 330.0 406.0
423.0   0.0 247.0
472.0   0.0 406.0
314.0   0.0 456.0
265.0   0.0 296.0
12
2 0 3
0 2 1
0 5 3
5 0 4
6 2 3
5 6 3
7 0 1
0 7 4
2 7 1
6 7 2
7 5 4
7 6 5
1.0 1.0 1.0 0 0 0
278.0 548.0 79.5   0.1   1 1 1
1
278 273 -800    278 273 0   0 1 0   54.432   80 60
0
```

Output for Sample Input