Ray Tracing in C explained

Mouad Sidqi Follow
Aug 29 · 12 min read

Parts :

Introduction :

Ray Tracing is a rendering technique that can produce photo-realistic computer generated imagery. It allows us to achieve realistic lighting effects and to recreate true to life images. (It can also be used to improve 3D audio, something that I'd love to write about in the future) It comes however at the cost of performance which made it not suitable for real-time applications(most notably games) for a long time. In this article I will explain and code a ray tracer in C as well as explain how I implemented illumination.

. . .

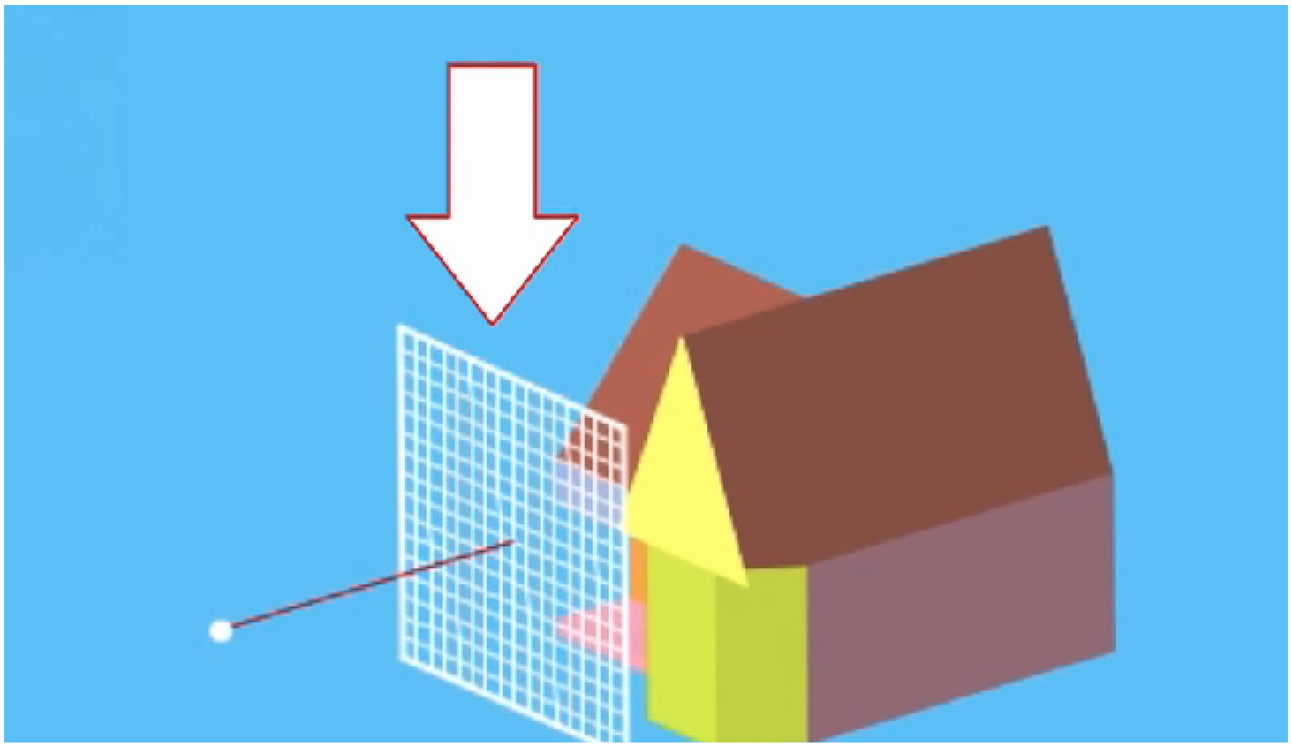
How does it work ?

To put it simply, it works by simulating what an eye in the real world would see in each "part" of a scene and then figuring out what the color of that part is depending on variables related to each part. For example, the variables could be the material of an object, the reflectance and/or transparency, the point is that the color of that "part" will look different depending on its variables and from where we observe it. Now let's put that in computer-graphics terms, RT works by sending or receiving(more on this later) rays *from each pixel* of our screen and following its path in a 3D space, then lighting the pixel with a color value depending on what the ray *intersected* with.

Before diving deeper, let's first define few concepts:
— Window, frame and the camera.
— Camera Ray, Shadow Ray.
— Lighting (Diffuse, Specular, Ambient).

Window , frame and the camera:

The window is the same window of our program in the monitor, while the frame is its representation in 3D space.



The following image shows a camera looking at a house through a frame. source: khan academy.

When I started working on this project it helped me a lot to visualize how our camera(eye) looks at a scene, see image on the left for an example. the camera so far is just a vertex in a 3D space and we need to represent what it sees in our 2D window. we do that by representing the 2D window in our 3D space by using a frame that the camera can look through.

Ray types (Backward / Forward RT):

I’ve mentioned earlier sending and receiving rays in a ray tracer, the difference is that a **forward ray tracer** sends rays *from light sources* and traces each of its reflections until it reaches the camera frame. this is a lot more computationally expensive due to all the light rays/photons that will be calculated but won’t be used due to not intersecting with the camera frame.

Forward ray tracing

Backwards ray tracer sends rays *from the camera* and after intersecting with an object, sends another ray towards the light source to check if there's an object blocking it.

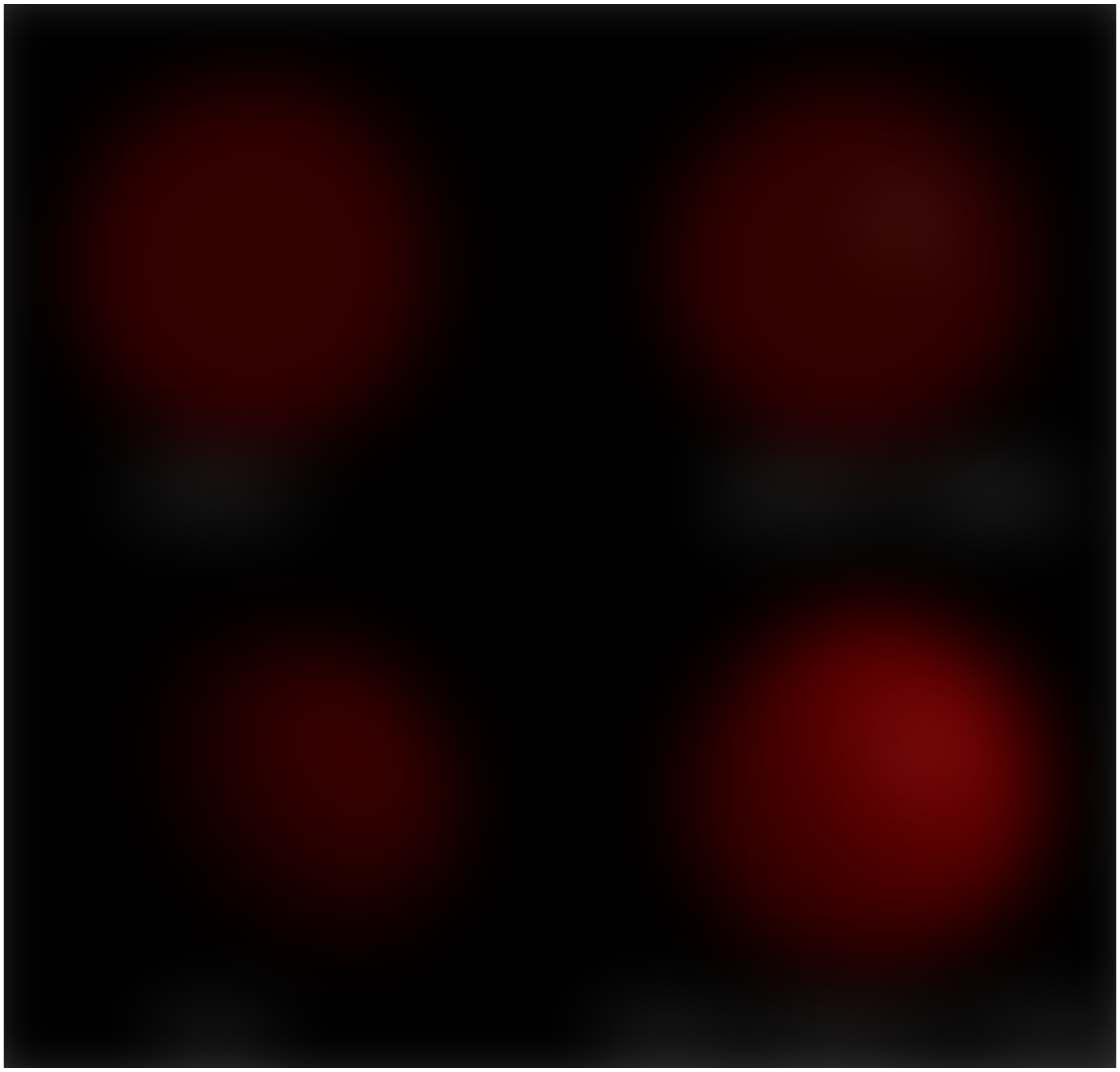
Camera rays are rays sent towards a scene from the camera while **Shadow rays** are sent towards a light source from intersection point.

Backward ray tracing

Lighting (diffuse, specular, ambient):

In this project I used Phong reflection model which is composed of 3 elements added together:

Ambient lighting is light that takes effect regardless of whether the light hit an object or not.
Diffuse reflection is light scattered around after hitting an object, it depends on where the ray hit and the angle at which we sent the Shadow ray.
Specular highlight is the bright spot on objects, it depends on type of material the object is made of.



Phong example with a sphere

. . .

Pseudo code:

Now that we've gotten that out of the way, lets start coding! first we'll make a pseudo-code:

so far we know that a camera in a Ray Tracer is defined by an eye and a frame, the eye is the origin from which we send rays, through the frame, into the 3D space.
this means that we should :
— For each pixel in the window, map the window to the frame, create a direction vector (by subtracting the mapped vertex from the camera position).
— This gets us a **Ray** defined by an **Origin**(camera's position vector), **Direction**(normalized vector after subtraction) and **t**(Scalar value which defines how far the intersected object is from the Origin).
If P is the intersection point, then P = Origin + t * Dir.

```
typedef struct s_ray
{
     t_vec4 origin;
     t_vec4 dir;
     double t;
}             t_ray;
```

— Loop through all objects, get the closest intersection point.
— Loop through all lights, add contribution of each light to the final color of that pixel.

```
create camera;
create scene;
foreach(pixel in window)
{
  map window pixels to the frame;
  send a camera_ray;
  foreach(object in scene)
  {
    if (ray intersects with object
      && intersection point is closer than previous ones)
    {
      foreach(light source)
      {
        send a shadow_ray;
        if (ray reached the light source)
        {
          Total Ld += diffuse(Ld) value for this light source;
          Total Ls += specular(Ls) value for this light source;
        }
        else
          This light won't contribute to Total Ld and Ls;
      }
     Pixel value = Ld + Ls + La;
    }
  }
}
```

N.B: La (ambient) is *outside the light loop* because it independent from light sources.

This is the general structure of our code, it should give you an idea of the steps I'll be explaining from here forward starting with the camera.

. . .

Implementing the camera :

What we need as **input** for our camera are 2 position vectors (look_at position and camera position) and 1 double/float (focal_length).
the **focal_length** is the distance between our camera position and the frame. changing this value will result in a different **field of view**. as for the 3 other vectors (up, forward, left) they will be used to describe the orientation of our camera.

```
typedef struct s_camera
{
     t_vec4 lookat;
     t_vec4 position;
     t_vec4 up;
     t_vec4 forward;
     t_vec4 left;
     double focal_length;
}             t_camera;
```

Next we will need information about the frame, notably how big it is in the 3D space. for simplicity we'll give it 1 in width and 1 in height, what this means is that we'll map our pixels from the window into a 1 by 1 frame in 3D space.

```
typedef struct s_camera
{
     t_vec4 lookat;
     t_vec4 position;
     t_vec4 up;
     t_vec4 forward;
     t_vec4 left;
     double focal_length;
     double frame_width;
     double frame_height;
}              t_camera;
```

Now lets make our camera function: it's a function that takes 3 inputs and returns a camera : we start by defining the orientation of the camera, first by calculating the **forward vector** (lookat -position).

we know that cross product of two vectors is another vector that is perpendicular to *both of them.* so **left vector** is the cross product of ((0,1,0) and forward). and **up vector** is the cross product of (forward and left).

N.B : changing (0,1,0) produces a different the orientation for the camera. N.B2: If you want to use right hand rule, you will have to change the order of the cross product.

```
t_camera create_camera(t_vec4 position, t_vec4 lookat, double fl)
{
 t_camera camera;
 t_vec4 forward;
 t_vec4 world_up;

camera.position = position;
 camera.lookat = lookat;
 world_up = create_vector(0, 1, 0);

forward = normalize( vector_sub(camera.lookat, camera.position) );
camera.left = normalize( cross_product(world_up, forward) );
camera.up = normalize( cross_product(forward, camera.left) );
camera.forward = forward;

camera.focal_length = fl;
 camera.frame_height = 1;
 camera.frame_width = 1;
 return (camera);
 }
```

. . .

Creating a Ray and Mapping the pixels :

I explained earlier that our ray is defined by Origin, Direction and t. this is the ray's struct that we'll be using:

```
typedef struct s_ray
{
     t_vec4 origin; // same as camera position
     t_vec4 dir;
     double t;
     /*
        t is initialized to FAR where FAR is a constant equal to
        the maximum distance we want our camera to render/see
        (FAR == 1e+6 for my example)
     */
}              t_ray;
```

Now to create a ray we first need to map the pixels (x, y) to vertexes (x, y, z) in the frame, then calculate the ray's Direction (vertex -camera position).
Here's our simple map function :

```
double map(int x_or_y, double size, int width_or_height)
{
     return (x_or_y * size / width_or_height – (size / 2));
}
```

To explain this part, let's take the first pixel of our window(top left one) as an example. it will be mapped to the top-left part of the frame, to do that we **multiply the left vector by map(x)'s return value, add the vector (camera.up * map(y)), then add the resulting vector to (forward * focal_length)**. which gives us the first ray's **direction** after we normalize it.
For camera rays ray **origin** is the same as camera position and **t** is equal to FAR.

```
void camera_ray(t_ray *ray, t_camera *camera, int x, int y)
{
 t_vec4 new_left;
```

```
    t_vec4 new_up;
    t_vec4 new_forw;
    t_vec4 left_up;

   new_left = vector_scalar(camera->left,
                      map(x, camera->frame_width, WINDOW_WIDTH));
    new_up = vector_scalar(camera->up,
                      map(y, camera->frame_height, WINDOW_HEIGHT));
    new_forw = vector_scalar(camera->forward, camera->focal_length);
    left_up = vector_add(new_up, new_left);

    ray->dir =  normalize(vec4_add(left_up, new_forw));
    ray->origin = create_vector(camera->pos.x, camera->pos.y,
                                                    camera->pos.z);
    ray->t = FAR;
   }
```

Repeat for each pixel and send a ray through each one.

. . .

Objects loop and intersections :

For this part I decided to store all objects in a linked list, each node with two parameters, a void pointer to the object and a second parameter identify it.
If the object is a sphere for example, we call the appropriate intersection function which returns 1 when the intersection occurs and 0 otherwise.

Here's our sphere structure :

```
typedef struct s_sphere
{
    double specular; // (value between 0-1)
    t_vec3 diffuse; // (3 values between 0-1, for R-G-B)
    t_vec4 center;
    double radius;
    int color;
}              t_sphere;
```

Before explaining the equation for a sphere intersection, it is important to note that it is necessary to consider:
— Only the **positive t solutions** as they're the ones INFRONT of the camera.

— The second thing to note is to **only consider t solution if it is less than ray->t** so we don't intersect with an object that's behind an object we previously intersected with.
— Lastly, NEAR is a constant equal to 1e-6, it is used here for two reasons:
reason one is to check that the t value is positive(avoiding margins of error) reason two is to avoid shadow-acne when sending shadow rays (shadow-acne happen when shadow ray intersects with the object itself near the intersection point).

finding the Intersection point P :

What we know :
1/ P = Origin + Dir * t;
2/ Sphere equation is $\|p-c\|^2 = r^2$
where p is a point in the sphere's surface and c is the center.
3/ If we assume P belongs to the surface, that means : **dot_product(P-C, P-C) — r^2= 0;**
We have all information we need to solve for t.

(P-C) · (P-C) — r^2 = 0;
(O — C + D*t) · (O — C + D*t) — r^2 = 0;
(O — C) · (O — C) + (O — C) · D*t + (O — C) · D*t + D*t · D*t — r^2;
(D · D) * t^2 + (2t * ((O — C) · D)) + (O — C) · (O — C) — r^2;

which is just a quadratic equation where
a = (D · D);
b = 2 * ((O — C) · D);
c = (O — C) · (O — C) — r^2;
delta = $b^2 - 4ac$
and our solutions are t = $(-b \pm \sqrt{\text{delta}}) / 2a$

```
int sphere_intersection(t_ray *ray, t_sphere *sphere)
{
 double t0, t1;
 double delta;
 double a, b, c;
 t_vec4 oc;

 oc = vector_sub(ray->origin, sphere->center);
 a = dot_product(ray->dir, ray->dir);
 b = 2 * dot_product(ray->dir, oc);
 c = dot_product(oc, oc) - sphere->radius * sphere->radius;
```

```
    delta = b * b - 4 * a * c;
    if (delta < 0)
        return (0);
    t0 = (-b + sqrt(delta)) / (2 * a);
    t1 = (-b - sqrt(delta)) / (2 * a);
    t0 = t0 < t1 && t0 > NEAR ? t0 : t1;
    if (t0 > NEAR && t0 < ray->t)
    {
        ray->t = t0;
        return (1);
    }
    return (0);
}
```

Intersecting with every type of object is beyond the scope if this article so I'll redirect you to this excellent page by ChrisDragan.

. . .

Illumination model : how do we compute the shading ?

I : light intensity;

Ia : ambient light intensity, defined as AMBIANT(0–1); (difference being that this doesn't depends on light sources)

Kd : surface diffuse reflectivity (value between 0–1);

Ks : surface specular reflectivity (value between 0–1);

N => surface normal;

L => direction vector from intersection point towards light position;

R => reflection direction;

V => camera direction;

SPECULAR_POW => the bigger the value the more concentrated the highlight will be;

```
La = Ia * object color;
Ld = I * Kd * dot_product(N.L) * object color;
Ls = I * Ks * pow( max(0, dot_product(R.V)), SPECULAR_POW ) * light
color;

Pixel color = L ambient + L diffuse + L specular;
```

. . .

Light loop and shading :

compute_color() is the function that will return the final color value using the previously mentioned Illumination model. it requires the color of the object, color of light and information about the shading that we will temporarily store in t_shader_x structure :

```
typedef struct s_shader_x
{
   t_vec3 diff; // foreach value R-G-B, between 0 and 1
   t_vec3 spec; // R-G-B, between 0 and 1
}              t_shader_x;
```

while the final R-G-B result is :

R_final = diff.x * R + spec.x * Light_color_R + AMBIANT * R;

G_final = diff.y * G + spec.y * Light_color_G + AMBIANT * G;

B_final = diff.z * B + spec.z * Light_color_B + AMBIANT * B;

Now, to calculate t_shader_x values we have to get the **surface normal** first, then **loop through lights** in the scene and **add t_shader_x of each light**.

```
// normal of a sphere is the vector (P - C) normalized.
t_vec4 get_sphere_normal(t_ray *ray, t_sphere *sphere)
{
   t_vec4 P;
   P = vector_add(ray->origin, vector_scalar(ray->dir, ray->t));
   return (normalize(vector_sub(P, sphere->center)));
}
```

.

```
int sphere_shader(t_data *data, t_ray *ray, t_sphere *sphere)
{
 t_shader_x   sh_x;
 t_vec4       sphere_normal;
 sphere_normal = get_sphere_normal(ray, sphere);
 sh_x = ray_inter_lights(data, sphere_normal, ray,
                        sphere->diffuse, sphere->specular);
 return (compute_color(sphere->color, &sh_x));
}
```

NB : t_data *data is a pointer to a struct containing scene, camera, lights, window width etc…

ray_inter_lights() is used for all types of objects, it will do the actual crunching of numbers and give us the final color that we'll use to light the pixel, after this part is done we will have at our hands the foundation from which we start developing our ray tracer.

get_shadow_ray() is used for each intersection point for each light. we pass as parameters the ray from which we calculate the intersection point which is the origin of the shadow ray, and the light to calculate ray direction.

```
t_ray get_shadow_ray(t_ray *ray, t_light *light)
{
 t_vec4 inter_point;
 t_ray shadow_ray;

 inter_point = vactor_add(ray->origin, vector_scalar(ray->dir,
 ray->t));
 shadow_ray.origin = inter_point;
 shadow_ray.dir = normalize(vector_sub(light->origin,
                                        shadow_ray.origin));
 return (shadow_ray);
}
```

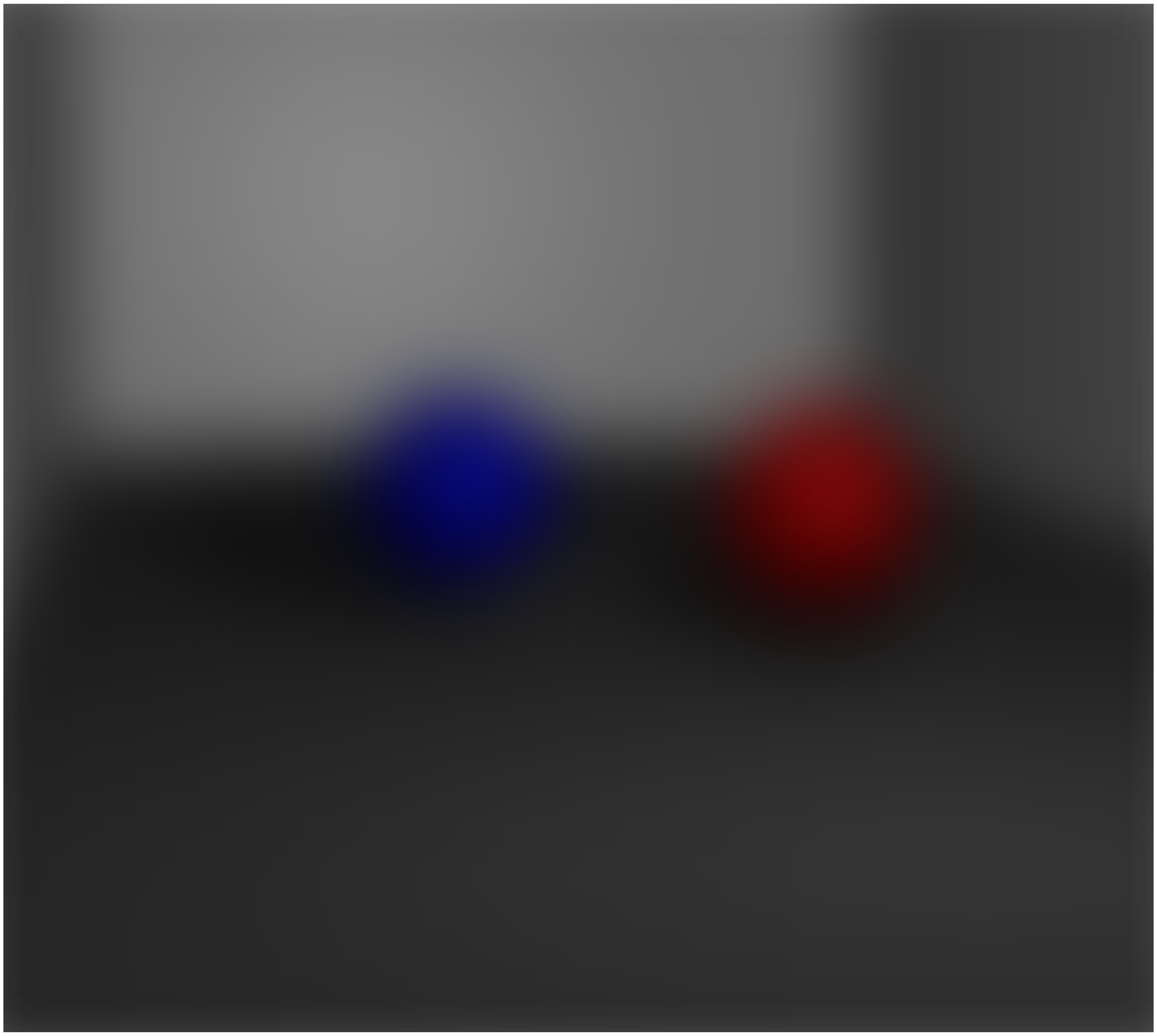
The following function was expanded from multiple function to make explaining, it easier, let's go through it step by step :

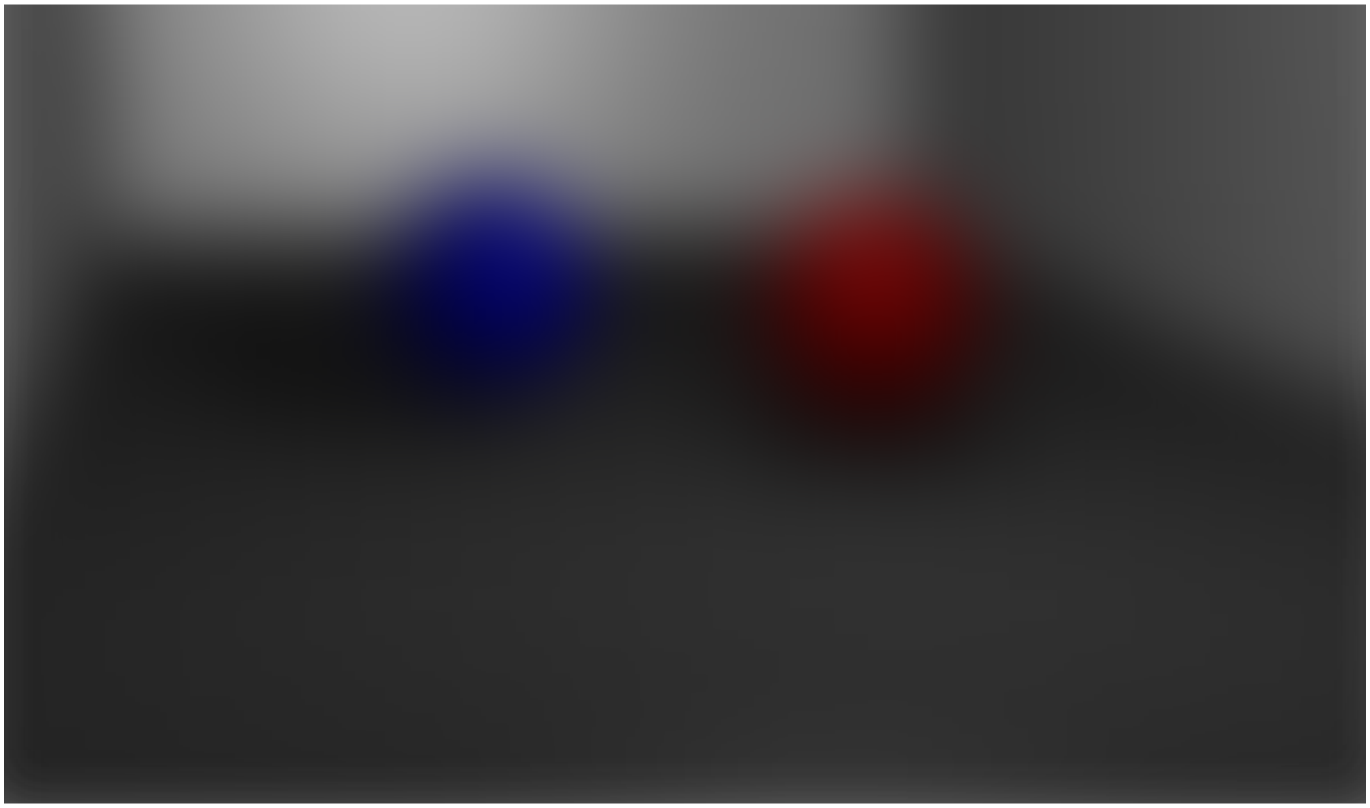
```
t_shader_x ray_inter_lights(t_data *data, t_vec4 normal,
                t_ray *ray, t_vec3 diffuse_kd, double specular_ks)
{
 t_ray      shadow_ray;
 t_shader_x shx;
 t_list     *lights;
 double     N_dot_L;
 t_light    *light;
```

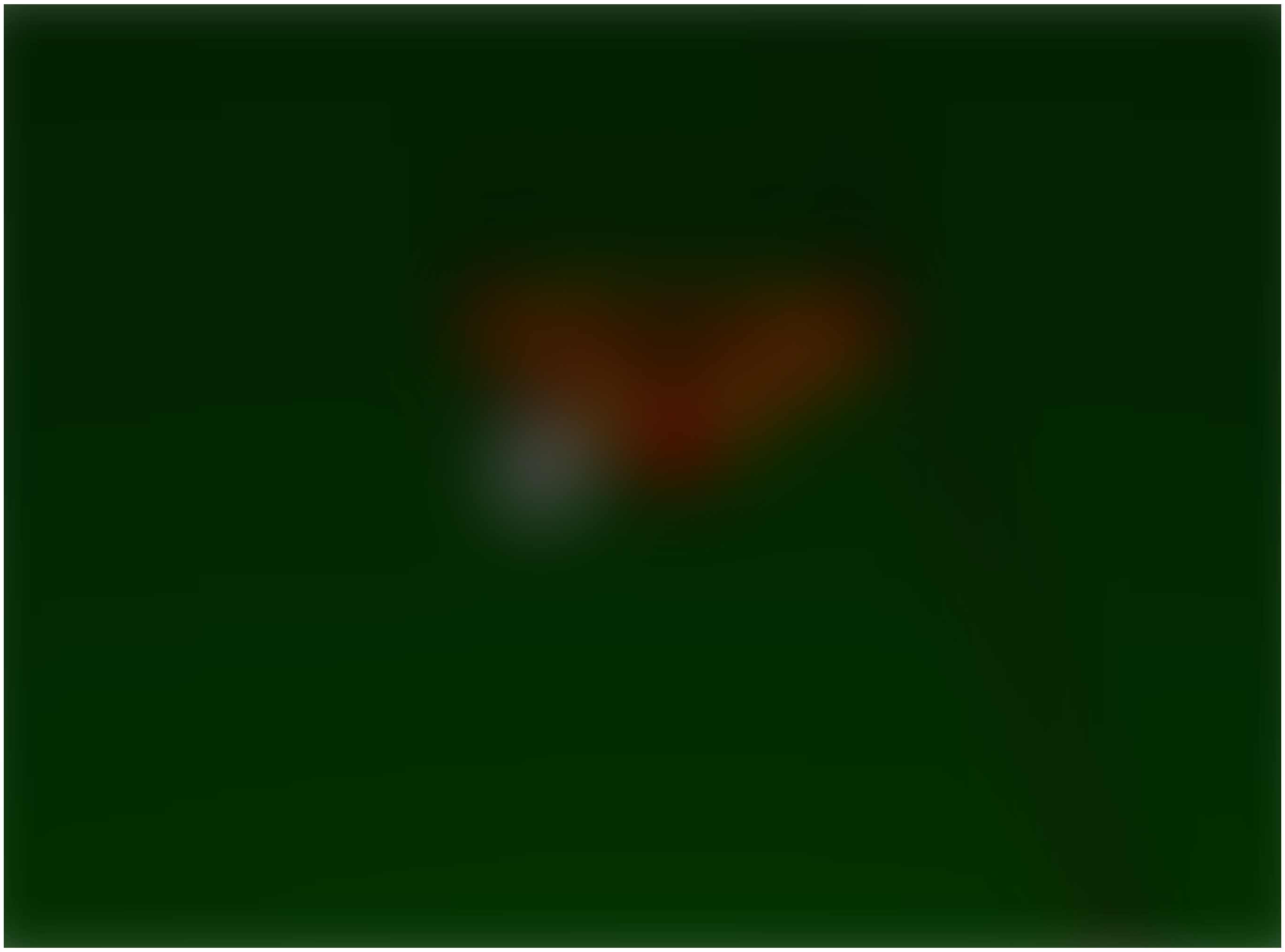
```
  lights = data->light_list;
  shx.diff = create_vector(0.0, 0.0, 0.0);
  shx.spec = create_vector(0.0, 0.0, 0.0);
  while (lights)
  {
    light = lights->content;
    shadow_ray = get_shadow_ray(ray, light);
/*
if N.L < 0 it means that light is under intersection
point,therefore it shouldn't contribute to the color value.
*/
    N_dot_L = max(dot_product(shadow_ray.dir, normal), 0);

/*
- ray_inter_objects() loops again through all objects and checks if
shadow_ray is blocked by any of them, if it's not blocked AND
distance_to_light > distance_to_new_intersection_point (the one
shadow_ray intersected with) then the light reached intersection
point P and we should start  calculating the shading.
- distance_to_light == magnitude(vector_sub(light->origin,
shadow_ray.origin);
*/

if(!ray_inter_objects(data->scene,&shadow_ray,distance_to_light)))
  {
    // DIFFUSE
    shx.diff.x += N_dot_L * diffuse_kd.x * light->intensity_r;
    shx.diff.y += N_dot_L * diffuse_kd.y * light->intensity_g;
    shx.diff.z += N_dot_L * diffuse_kd.z * light->intensity_b;

// SPECULAR
    t_vec4 V;
    t_vec4 R;
    if (N_dot_L == 0)
       tmp = 0;
    else
    {
       // view direction
       V = scalar(ray->dir, -1);
       // reflection direction
      R = vector_sub(scalar(normal, 2 * N_dot_L), shadow_ray.dir);
      tmp = pow(max(dot_product(r, v), 0), SPECULAR_POW);
    }
    shx.spec.x += tmp * specular_ks * light->intensity_r;
    shx.spec.y += tmp * specular_ks * light->intensity_g;
    shx.spec.z += tmp * specular_ks * light->intensity_b;
   }
   lights = lights->next;
  }
  return (shx);
}
```

We have completed the routine that will be repeated for each pixel and light each one with the appropriate value and we have enough information to implement many things (reflections, refractions, textures etc…).
it's up to you to use your favorite library, framwork or API and start coding!

Here's the github repo for my project if you're curious or just want to see some examples. here are few :







Programming Computer Science Ray Tracing Rendering C