2: Ray tracing

Goals

The goal of this practical is to implement the ray-tracing algorithm to obtain reflections and shadows in your renderings. You will start from your previous scene that contains the ray-casting intersections between the camera rays and the scene objects.

Direct illumination using shadow rays the Phong model

The first step will be to implement the direct illumination algorithm to obtain a simple rendering. As seen during the lecture, you will also rely on a shadow ray to test whether the current tested point is in the shadow or not. The following algorithm can be used to obtain the color at a particular point (for a given camera ray):

```
color trace(ray) {
	hit = intersectScene(ray)
	if(hit)
		color = directIllumination(hit)
	else
		color = background_color
	return color
}
```

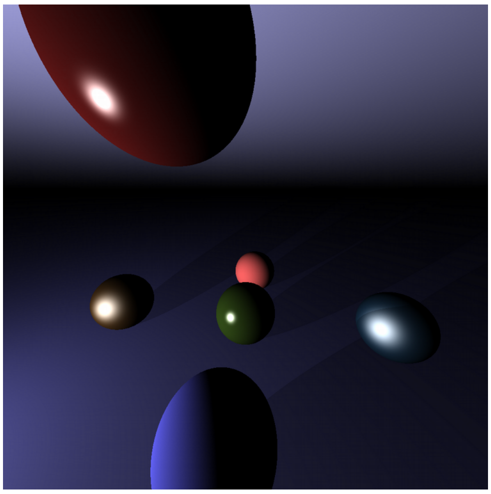
In your implementation, “hit” should contain the information about the intersected object: its type (sphere/plane/etc), its Id (if you use an array of sphere for instance), and its material (diffuse/specular colors and shininess – as we have seen in the Phong model). That way, you will be able to compute a different color/material for each object in your scene.

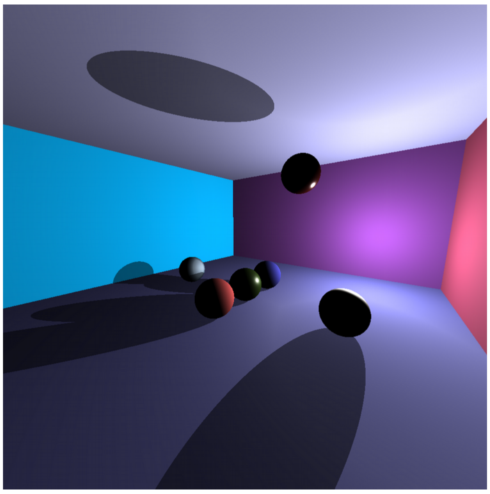
The pseudo code for the “directIllumination” function may look like this:

```
color directIllumination(hit) {
	color = (0,0,0)
	for each light L {
		T = cast shadow ray to L (test if hit is in the shadow)
		if not T (hit is not in the shadow) {
			color += phongModel(hit,L)
		}
	}
	return color
}
```

The Phong equation is the one seen during the lecture: Ka + Kd (**n**.**l**) + Ks $(\mathbf{r}.\mathbf{v})^q$, where Ka, Kd, Ks and q are the material parameters (ambiant, diffuse, specular colors and shininess) and should be defined per object. **n**, **l**, **r** and **v** are respectively the normal, light, reflection and view directions (all 3D vectors). Note that if you have a single light, you do not need the loop in the algorithm.

After this step, you should be able to obtain something like this:





Ray tracing

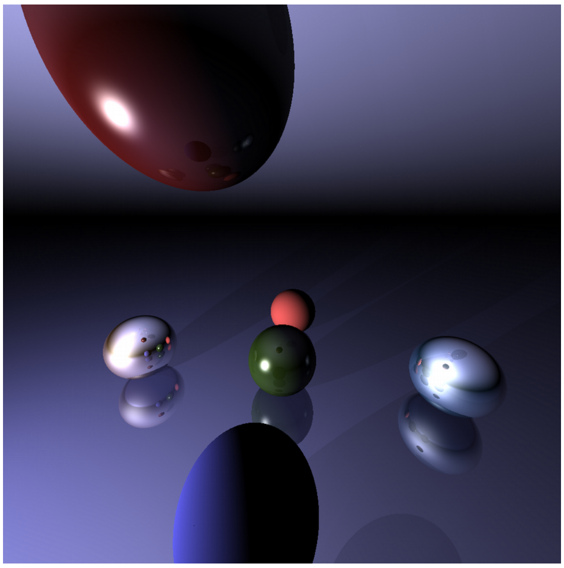
We now want to apply the full ray-tracing algorithm to obtain reflexions on the surfaces. To simplify, we will consider only opaque surfaces (and avoid transparency):

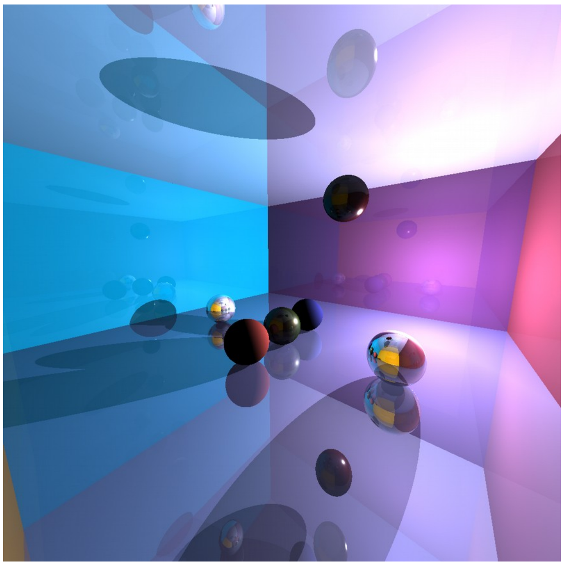
```
color trace(ray) {
        hit = intersectScene(ray)
        if(hit) {
                color = directIllumination(hit)
                if hit is reflective
                        color += c_refl * trace(reflected ray)
        } else
                color = background_color
        return color
}
```

Note that GLSL does not allow to use recursive functions. To implement this algorithm, we will thus rely on a fixed size loop and the use of a mask variable to store the reflection intensities/colors. You will thus rather implement the following algorithm (which is equivalent to the former one):

```
color trace(ray) {
	accum = (0,0,0)
	mask = (1,1,1)
	for i=0 to NB_STEP do {
		hit = intersectScene(ray)
		color = directIllumination(hit)
		accum = accum + mask*color
		mask = mask*c_refl
		ray = reflected ray
	}
	return accum
}
```

Once your algorithm implemented, you can play with material values for each object to obtain mirrored effects or realistic glossy effects. You should obtain something like this:





Bonus

- Animate your objects, your camera or lights with simple functions or with mouse positions.
- Add multiple lights, more objects, etc.