Computer Graphics - Lab 4
Ray Tracing

Dec 21, 2016

Introduction

In this lab we will work with ray-based graphics rendering instead of rasterization. To start with, a simple ray casting renderer, implemented in the fragment shader, was given as skeleton code. The task of this lab is to extend it to a ray tracer.

A scene that consists of a plane with 5 spheres of different sizes and a sky together with a light source will be rendered.

To account for sharp edges of the spheres anti-aliasing was performed by casting four additional rays after the original ray, in different places of the same pixel. This makes the edges and curves more smooth on the spheres which in turn gives the a smoother surface, when observed from a closer distance.

To make it easier and faster to see changes, the shader program from lab 2 was implemented instead, to enable a reload for the shaders.

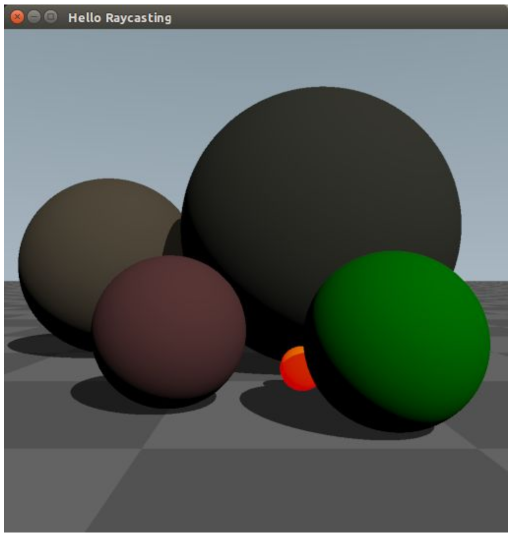
4.1 Ray Tracing

4.1.1 Shadow rays

In this task a "shadow feeler ray" had to be implemented to be able to produce shadows from the spheres. To do this, another instance of the Ray struct, *shadow_ray,* was added in the ray-tracing function. This ray's origin was set to the point of the first intersection, but offset by 1 / 10000 of the distance to the light source with the direction towards the light source. The direction of the ray was set straight towards the light source. The ray was then used to check for an intersection using the *intersect* function. An additional boolean property, *in_shadow*, was added to the *Intersection* struct to signal if shadows were to be cast. This property is true by default, but set to false if the ray don't intersect with an object (if it hits the sky). The irradiance is then set to 0.01 instead of 0.1 (if *in_shadow* is true) times the light source brightness and added to the total colour of the pixel to simulate a darker, more shadowy look.

```
Ray shadow_ray = Ray(isec.point + light_dir * 1e-4, light_dir, 1.0);
Intersection shadow_isec = intersect(shadow_ray);
float irradiance = 0.1 * scene.sun_brightness;
if(shadow_isec.in_shadow){ irradiance = 0.01 * scene.sun_brightness; }
```

The produced result shows that shadows are produced on the plane as well as on the other spheres because of an intersection of the ray going to the light source.

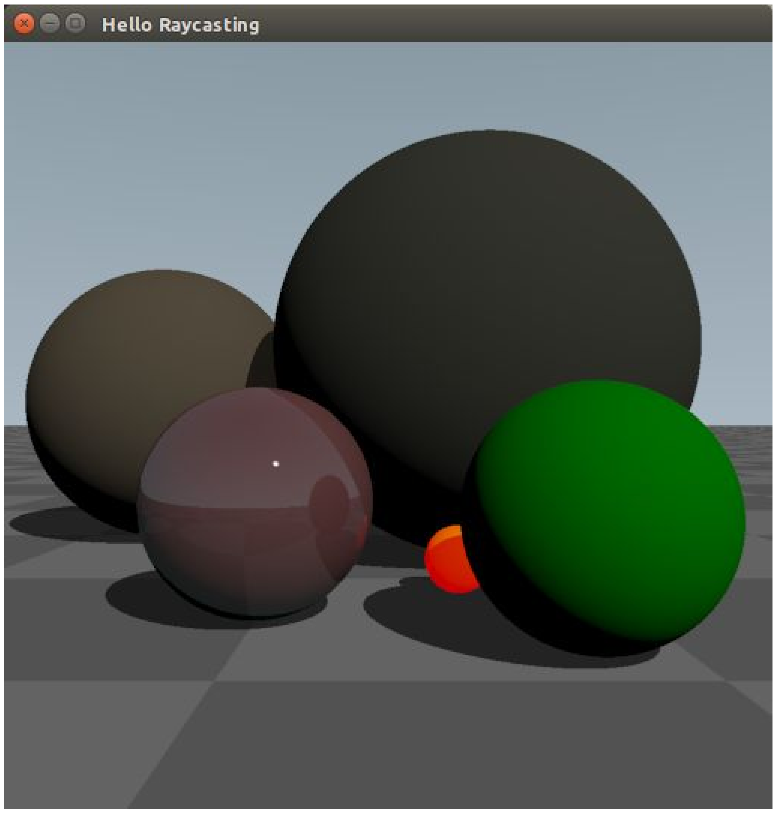


4.1.2 Reflective Surface

To create a reflective object, a ray needs to be cast from the surface to introduce a reflective specular lightning. This ray was sort of recursively called through the stack calls in a for-loop of the ray-tracing function, to present a final color for the pixel reflection of the intersection of the first ray. The direction was calculated by using the GLSL function *reflect* to get the direction of the ray, using the parameters of the direction of the ray and the normal of the intersection. So for the sphere, the normal was orthogonal to its surface and the reflection vector was a mirror-reflection of the incident vector. To create the ray, the direction and the intersection origin was applied together with a reflection weight, which decides how much reflection there's going to be. Below is the entire code to introduce reflective highlighting, before the total sum up of the color:

```
vec3 direction = reflect(ray.direction, intersection.normal);
vec3 origin = intersection.point + direction * 1e-4;
Ray reflection_ray = Ray(origin, direction, intersection.material.reflection);
push(reflection_ray);
```

The produced result can be seen on the purple sphere on the front left:



4.1.3 Transparent surface

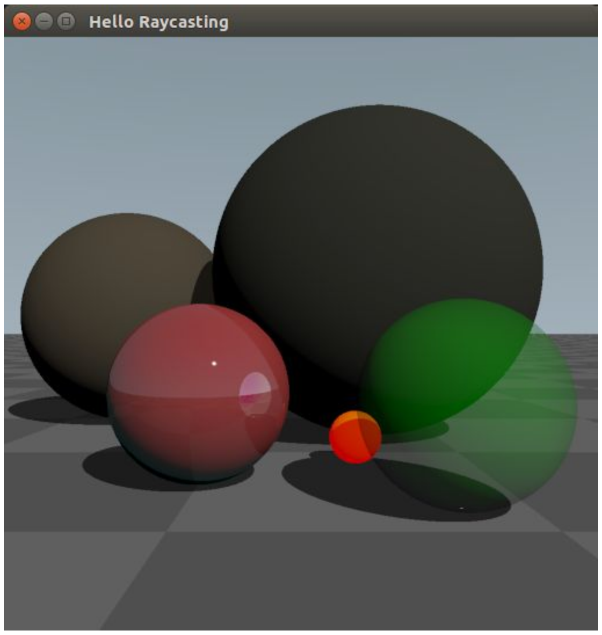
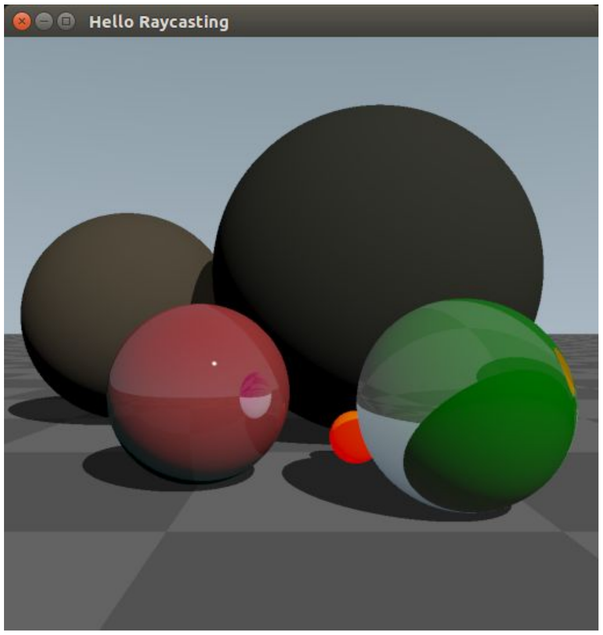
To give the spheres a transparent look there was more work to be done than the previous tasks. Transparency largely depends the material of the object, which tells us what kind of index of refraction (*IOR*) it has and decides which direction the light travels when entering the material. For the simplicity of this task, the only cases of refraction that will occur will be between air (*IOR* = 1.0) and a fixed *IOR* for a sphere, i.e. 1.33 for water and 1.5 for glass. To avoid total internal reflection, it is important to make sure that the sum of the reflection and the transparency values do not exceed the value of 1.

When the ray intersects a surface, in this case a sphere, the IOR of the material and the normal needs to be determined. Since the intersection occurs outside the sphere, the normal is positive, the new ray that will be refracted into the sphere, will have the origin at the intersected point, the IOR of the intersected material and the direction will be calculated with the IOR and, by using the *refract* GLSL function. This ray will then be pushed to the stack to be calculated again in the same function. This time the ray will hit the inside of the sphere, causing the normal to be negative. In this case, the intersected material will once again be the material of the sphere, but now we want the new ray to be going out of the sphere, so the same direction it went into the sphere the first time. The new direction will therefore depend on the new IOR which needs to be an inverse of the first IOR, to get it back to its original direction. This will keep going on until the ray stack reaches MAX_DEPTH of 10, causing it to stop the refraction.

The code below describes how it was solved:

```
float ior = 1 / isec.material.ior;
float norm_dir = sign(-dot(isec.normal, ray.dir));
if (norm_dir == 0.0 || norm_dir == -1.0) {
   ior = isec.material.ior;
}
vec3 dir = refract(ray.dir, isec.normal * norm_dir, ior);
vec3 pos = isec.point + dir * 1e-4;
Ray trans_ray = Ray(pos, dir, isec.material.transmission);
push(trans_ray);
```

The produced result of the transparency can be seen in the image below on the green sphere.

	
IOR = 1.0 - Giving full transparency, a sphere made of air	IOR = 1.5 - Resembling the transparency of a glass sphere

Discussion

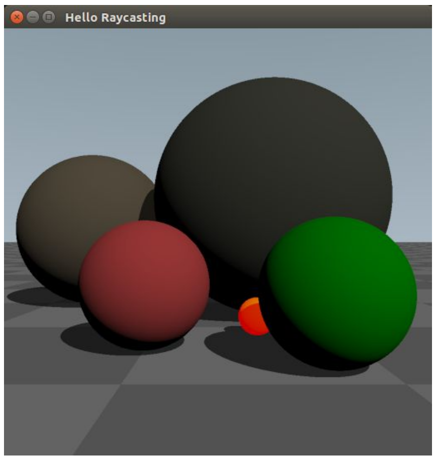
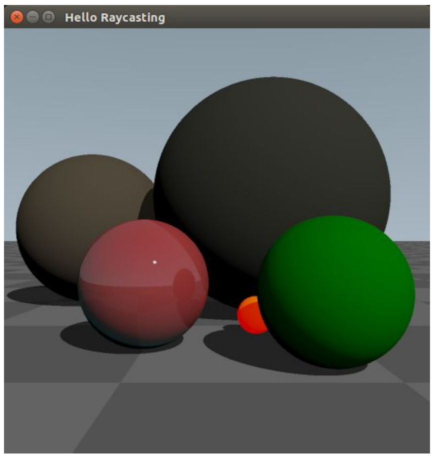
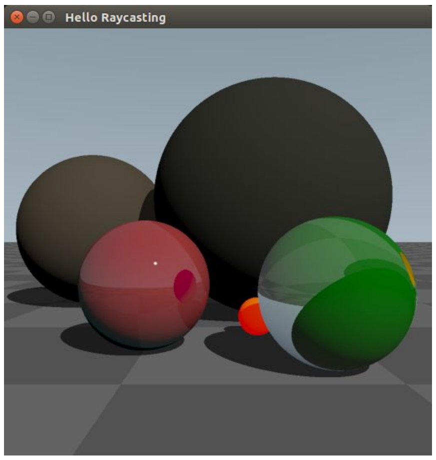
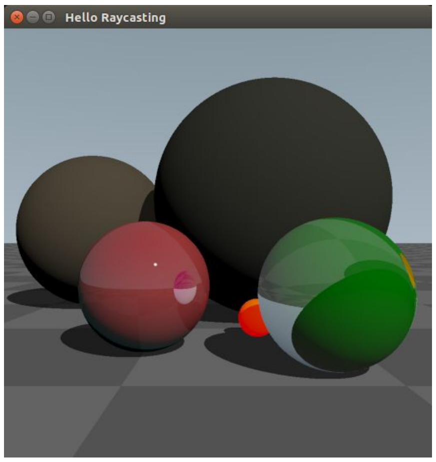
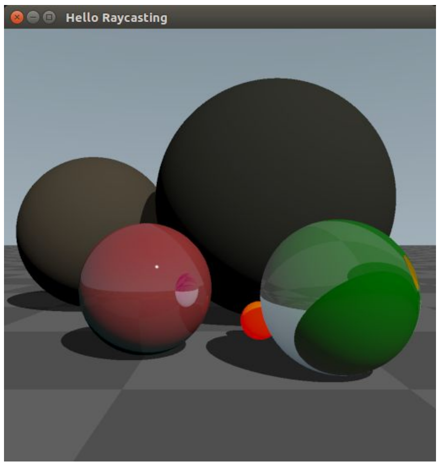
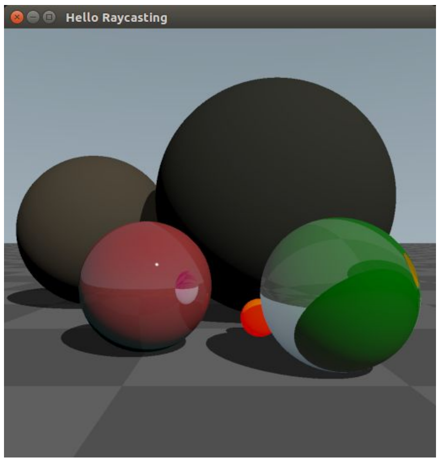
Is the ray tracing result realistic? If you compare it to the terms included in the Rendering Equation, what has been left out and what has been represented only approximately?

Yes, it is really realistic. The specular lighting is not explicitly calculated in an approximate way nor with a physically based formula. Instead it appears more naturally through ray tracing by adding up colours of objects that are reflected in the surface. The *irradiance* factor times the *lambertian_brdf* function sets the diffuse lighting of the object which is an approximation based on the direction of the light.

How does the maximum depth influence the appearance of the scene? How would a refractive object appear at a ray depth of 1, 2 and 3? How about reflective objects? At which depth does your scene no longer change by increasing the ray depth further? Why?

The maximum depth sets the maximum number of new rays to be shot after the original ray was shot. The lower the number, less level of detail of reflection and refraction is applied. If it is set to 1, only the original ray is used. In this case, this will only bring the surface and shadows, but no reflections or transparency. If it is set to 2, reflective objects will appear reflective, but without reflection / transparency details of the objects that can be seen as reflections in the observed surface. If set to 3, same situation as 2, but with refraction applied to transparent objects.
The scene stops to change after the maximum depth is set to 6. The reason for this is that only one of the spheres is reflective and another one is transparent. Even if the maximum depth would be higher, no more new rays would be produced here, since no more reflections will occur. If for example two of the spheres were reflective, there could be situations of “infinite reflections” between them, which would result in an infinite number of new rays, if allowed.

The difference going from depth 1 to depth 6.

 **Depth 1** Only shadows and the scene is rendered	 **Depth 2** Reflection is rendered
 **Depth 3** Transparency and 2nd reflection ray	 **Depth 4** 2nd Transparency and 3rd reflection ray
 **Depth 5**	 **Depth 6**