MTAT.03.015 Computer Graphics (Fall 2013)
Exercise session XI: Raytracing.

Konstantin Tretyakov, Ilya Kuzovkin

November 18, 2013

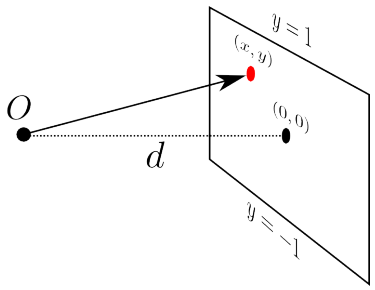
In this exercise session we shall study the basics of raycasting, raytracing and raymarching. For simplicity and convenience we shall be using the ShaderToy website[1] and implement our raytracing code as a fragment shader[2].

The solutions will have to be submitted as a set of text files, zipped in a single archive. Solution to exercise N should be saved in a file named `exercise`N`.glsl`.

1 Basic Raycasting

Let us start with a basic raycasting example. The idea of raycasting is to send rays from the eye through each pixel on screen and check for each ray, whether it intersects any geometry in the scene. The two main steps of a raycasting algorithm are therefore the following:

1. Convert pixel coordinates to a ray direction. It is best understood using the following diagram:



[1] `https://www.shadertoy.com/`

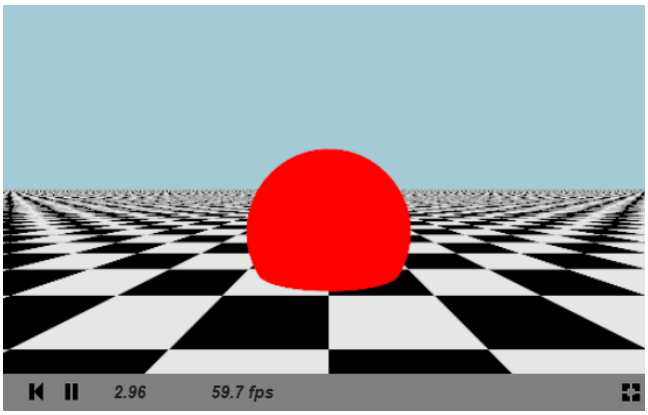
[2] Note, that although GLSL's fragment shaders are conceptually very well suited for implementing raytracing-based techniques, in practice you would not be able to use them for large scenes. Firstly, there is a limit to the amount of data you can keep within the shader code. Secondly, GLSL lacks dynamic memory structures and recursion, which makes it impossible or hard to implement several algorithms. Finally, efficient raytracing may require multiple passes over the whole image – something not possible within a single fragment shader.

Assume our screen is positioned at distance d from the origin. Let the center of the screen have pixel coordinates $(0, 0)$, the top and bottom correspond to $y = 1$ and $y = -1$ respectively. If we assume that the camera is looking along the negative z direction, the direction of the ray passing through pixel (x, y) is $(x, y, -d)$. The parameter d is related to the field of view angle α_y:

$$\frac{1}{d} = \tan\left(\frac{\alpha_y}{2}\right).$$

2. Cast the ray and look for intersections with objects in the scene. Given the ray origin point O (which corresponds to the camera position, i.e. $(0, 0, 0)$ in our case) and the ray direction that we computed in the previous step, we can check for each primitive in the scene whether it is intersected by the ray, and return the color of the pixel at the closest intersection point.

Exercise 1 (0.5pt). The code base for this exercise is provided on the ShaderToy website at `https://www.shadertoy.com/view/ldSGzW`. Examine the code. We would like to raycast the following scene:



The scene consists of a red sphere and a plane with a checkerboard pattern. The code base provides the specifications for both the plane and the sphere in the variables `plane` and `sphere`. The colors of the sky and the sphere are provided in the `sky_color` and `sphere_color` variables. The color of the plane is given as a function: `plane_color` takes a point on the plane and returns its color.

The methods for finding ray-object intersection are also given to you: functions `raycast_plane` and `raycast_sphere` return the distance from ray origin to the intersection with the corresponding object, or `INFINITY` if the ray does not hit the object. Your task is to introduce three changes to the code:

1. First implement the function `raycast_scene`. The function should return the color of the object that is hit by the given ray first.

2. Next, change the main function so that the resulting image would have a y field of view of 60 degrees.

3. At this point you will see the scene rendered like on the picture above. However, it will have a lot of aliasing artifacts: the sphere has jagged edges and the checkerboard pattern turns into nonsense at a distance. Let us fix this using a simple 3x3 discrete box filter. That is, rather than using a single ray for pixel (x, y), cast 9 rays, as if you would be rendering pixels $\{x - 0.2, x, x + 0.2\} \times \{y - 0.2, y, y + 0.2\}$, and average the results[3].

2 Raytracing

Let us now extend the basic raycasting algorithm we just implemented to support shadows and reflections. This results in what is typically referred to as *recursive raytracing*. Open the code base given at `https://www.shadertoy.com/view/Ms2GzW`. This is a slightly extended version of the previous exercise. Note the following changes:

1. **Lighting.** We shall now be using lighting computations. Consequently, rather than specifying object colors, we specify their materials in `sphere_mat` and `plane_mat()`. Ambient light is given in the `ambient` variable, and a single light source is described in `light`. Finally, the function `lighting` implements the familiar Blinn-Phong light model. Note that this function takes a `viewer_pos` argument rather than assuming the viewer to be fixed at $(0, 0, 0)$. Think why it is done this way.

2. **More detailed raycasting.** The `raycast_scene` routine is updated to not only return the distance to the closest object, but also the actual point location, the normal at that point and the material of the object.

3. The actual scene rendering routine, `render_scene` is separate from the raycasting function (which is now just a useful primitive).

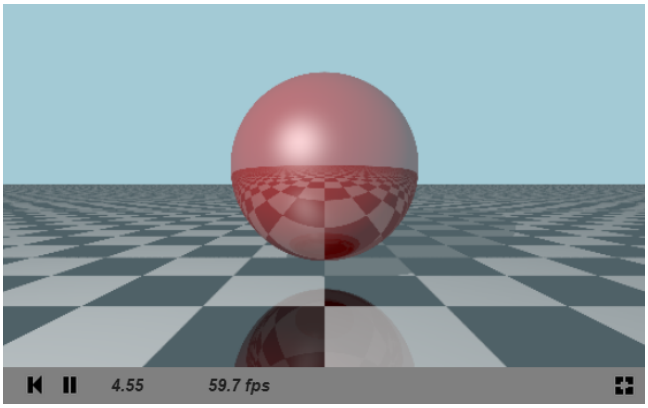
Exercise 2 (1pt). Update the function `render_scene` to implement recursive raytracing with shadows and reflections.

1. To implement shadows, cast a ray from the point towards the light source. If this *shadow ray* hits anything on the way, you can consider the point to be in shadow (and thus only affected by ambient light).

2. To implement reflections, note that the `Material` structure has a field `reflectivity`. This is the weight with which the reflected ray contributes to the visible color of a point with this material. The full color of a point is then its "usual" color + reflectivity weight times the color "seen" by the reflected ray. Implement at least two reflection steps. (Hint: this is naturally done using a loop).

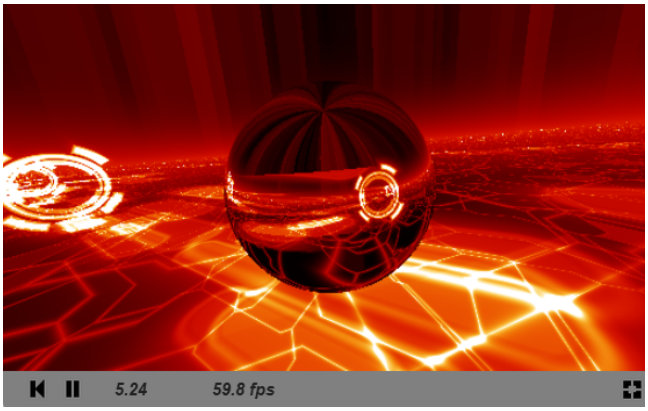
[3] You are free to try larger grids, however note that having long nested loops in shader code may lead to weird results on some graphics cards. Also note that the use of `break` statement in `for`-loops can sometimes result in strange behaviour.

The resulting image should look as follows:



Hint: Remember that ray direction vectors must always be normalized.
Hint: You can use the `reflect` GLSL function to compute reflection vectors.

Exercise 3* (0.5pt). An example of a great purely-raytracing based scene is the shader "Relentless": `https://www.shadertoy.com/view/lss3WS`. Study the code. Your task is to add a perfectly reflective sphere right in front of the camera, getting a result like the following:



Let the sphere radius be 1. Have its center positioned straight in front of the camera, at distance 2. For simplicity you may assume that the sphere is always the topmost object (i.e. resolve ray intersection with the sphere first of all, ignoring the possibility of other objects being in front).

3 Raymarching

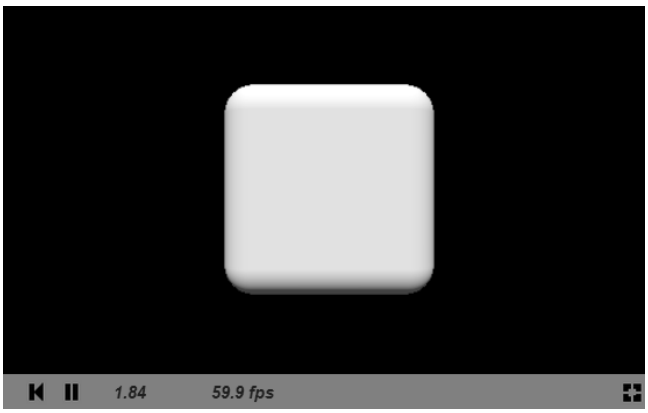
When objects have complex descriptions, it may be easier to search for ray-object intersections using iterative numerical methods, resulting in the family of *raymarching* algorithms. *Sphere tracing* is a particularly fast version of raymarching, which can be applied if the scene is modeled as a *distance field*. A distance field is a function that for any point (x, y, z) returns the distance to the nearest object in the scene. For example, if the scene consists of a single sphere of radius R centered at the origin, the corresponding distance function is as simple as

$$f(\mathbf{p}) = \|\mathbf{p}\| - R$$

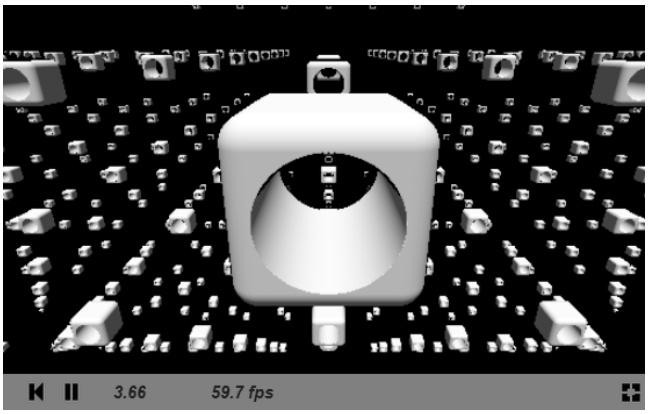
Distance fields are a very flexible means for modeling complex scenes, see `http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm`.

Exercise 4 (0.5pt). Open the code in `https://www.shadertoy.com/view/MsjGRD`. The `main` function presents you with a familiar structure of a raytracer. The only difference is that all the hard work is done by a function named `raymarch` now rather than `raycast`. Implementing this function is your first task.

The function `raymarch(ro, rd)` must "walk" from the point `ro` in the direction `rd`, making steps of size `scene(x)`, looking for a point `x` where the function `scene` becomes nearly zero. If you implement this algorithm correctly, you should see the rendering of the scene, which is a simple rounded box:

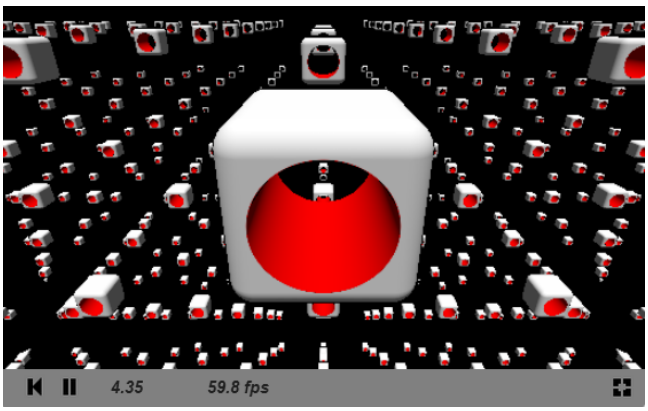


Exercise 5 (0.5pt). Next, let us make the scene a bit more interesting. For that we shall use some of the basic operations with distance field functions, that are already given to you in the code. Make the changes to the `scene` function, so that instead of a static rounded box, the scene would become a rotating grid of rounded boxes, each with a hole drilled in it. That is, turn the scene into this:



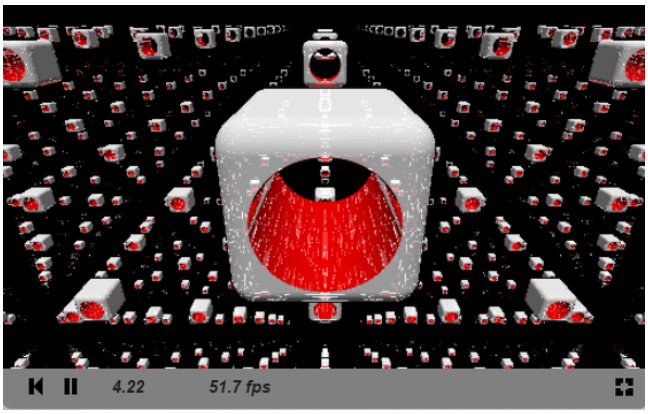
The comments in the code will provide you the appropriate parameter values.

Exercise 6* (0.5pt). Modify the code so that the insides of the holes, drilled in the cubes would be colored red:



Hint: For this you will need to change the `scene` function so that it would not only return the distance, but also the color of the corresponding region of the field. The simplest approach is to have `scene` return a `vec4` (instead of a `float`), so that its first three components correspond to the color and the fourth is the distance.

Exercise 7* (0.5pt). Add reflections to the raymarching algorithm:



Hint: The approach is similar to the one used in Exercise 2. Make a new function `render`, that will invoke `raymarch` inside a loop, accumulating the resulting color as a weighted sum of multiple reflections. Assume the reflectivity coefficient to be 0.2.

Exercise 8* (1pt). Read the article *Distance Estimated 3D Fractals*[4]. Present a shader implementation of any Mandelbulb fractal.

[4] `http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i/`