Lab Assignments
Computer Graphics DT3025, HT2016

Martin Magnusson and Daniel Ricão Canelhas
December 5, 2016

4 Ray tracing

In this fourth lab we will move away from rasterisation, and instead work with ray-based computer graphics. However, we will still implement it in GLSL, to make use of the massive parallelisation that the GPU allows. Although there are many frameworks for performing general purpose computations on GPUs (GPGPU) including Compute Shaders in OpenGL itself, we will base these tasks on Vertex and Fragment shaders due to your familiarity with these.

You are given a ray-casting renderer, and your task will be to extend this to a ray tracer.

Ray casting

This section just describes the ray caster that you will start from. Your task (4.1) is described on page 3.

Ray casting is the simplest form of ray-based rendering and was used in some of the first real-time 3D games ever released [1]. It amounts to the following steps, for each pixel:

1. Define a ray going through the pixel, starting at the camera origin.
2. Find nearest object intersected by the ray.
3. Compute colour at the intersection and set pixel color.

The essential things we need to make this happen are thus a camera, a scene, and rays (lots of them)!

The camera

To fully determine the camera the geometry seen in Figure 1 needs to be completely specified.

Image size Width and height, in pixels, of the image plane (shown in gray in Fig 1). This is modified when the window is resized.

Focal distance Distance between the origin (the black dot at the base of the arrows) and the image plane (in pixels, too!). This is modified by zooming.

Camera origin The translation offset $\mathbf{o} \in \mathbb{R}^3$ from the world's origin to the camera's current position. This is modified by moving; e.g., walking, lifting or lowering the camera.

Viewing direction The vector shown in blue in Figure 1. This is modified by pitch and yaw angles (looking up or to the side).

[1] Wolfenstein 3D, by id Software, source code made public at https://github.com/id-Software

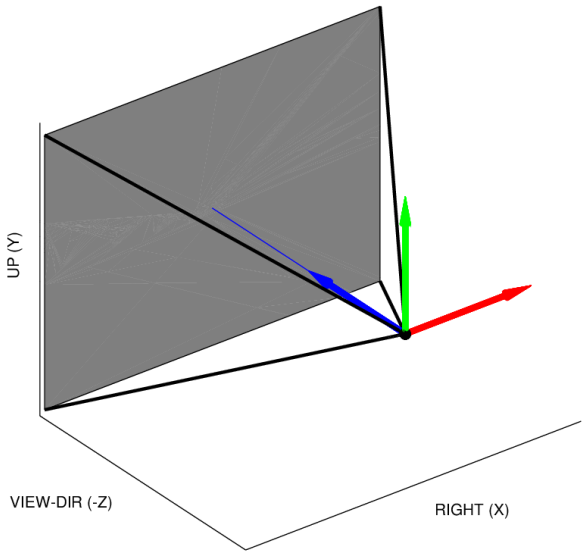

Figure 1: Schematic layout of the camera. The blue axis is the view direction, the other two axes form an orthogonal basis aligned with the pixel coordinates. The image plane is shown in gray.

Right direction The vector shown in red in Figure 1. This is modified by yaw and roll angles (turning sideways).

Up direction The vector shown in green in Figure 1. This is modified by roll and pitch angles (but can be found as the cross product of the view and right directions, if computed in that order)

You may have noticed that none of the listed parameters are constant and will have to be updated for every frame, to account for user input. Some of the relationships are constant though. The last three items form an orthogonal basis, and the view direction always points into the center of the image plane, regardless of how the viewport may be resized. These parameters are passed as `uniform` variables to the shader program.

A ray

A ray is simply a point of origin, a directional unit vector and a scalar term $R = \mathbf{p}_0 + t\boldsymbol{\omega},\ \ t \geq 0$. To obtain rays we simply set $\mathbf{p}_0$ to the camera's orgin $\mathbf{o}$, and if we have updated our basis correctly, we can move t units, along the view direction, half the image height minus the current row along the up axis, the current row minus the half the image width along the right axis. This gives us a directional vector that points from the origin to a specific pixel in the image plane. If we normalize this vector, we get $\boldsymbol{\omega}$. For every value of t we then obtain a 3-dimensional point along that ray. The trick is then to find the value for t that intersects the nearest surface in that direction.

In the fragment shader, `struct Ray { vec3 origin, dir; };` defines a simple ray structure. The scalar t that gives the ray its length is kept elsewhere as

each ray will have several possible t values, for each object that it intersects in the scene.

The scene

The scene defines the geometry and also provides descriptions of the material properties and lights in the environment. Given a scene and a ray, we will need to compute their intersection. At the intersection, we need all the information required to compute the color at that point, just as before; that is, material properties and surface normal.

In a naive implementation, each ray will be tested against every object, potentially producing several intersections per ray, the closest of which has to be found. Most of the work in optimising rendering systems takes place here, by reducing the number of objects that have to be tested against each ray.

Our rendering engine only considers spheres and planes, but with a few additional steps you could also implement ray–triangle intersections, for example (see the lecture instructions).

The skeleton code defines the scene in the `init()` function, where a fre spheres and a plane are placed in the world.

Intersections

The most interesting things in a ray caster/tracer happens when rays intersect objects. There is a `struct Intersection` defined in the fragment shader, which contains the information necessary from a successful ray–object intersection: the point of intersection and the normal at that point, as well as the material properties of the object.

In a C++ implementation, the material should be an object, to allow for more flexibility. In this case, the material properties declared as part of `Intersection` are diffuse and glossy colour and a glossy roughness term (which can be used in a Blinn-Phong model, for example). The provided code uses a "fixed-function" Lambertian function.[2]

The `intersect` function tests for intersection with the objects in the scene, and assigns material properties to the `Intersection` based on which object was hit. For the plane, it also creates a procedural chess-board texture.

The provided code only implements one light source.

4.1 Ray tracing

Raytracing takes a similar approach to raycasting, but adds additional rays to deal with transparency, reflection and cast shadows.

Your task is to upgrade the raycaster to include additional rays that are produced in 3 different cases:

1. Any surface: a secondary ray is cast from the surface in the direction of the light source (a "shadow feeler"). If this ray intersects any object on its way towards the light, this surface is in shadow and should not receive light from this light source.

2. Reflective surface: an additional ray is cast from the surface, taking the incident ray and reflecting it around the normal. The final color will be a combination of diffuse shading and the color at the endpoint of the

[2] If you want, you could also add a member variable of `Intersection` that defines what type of BSDF to execute, and some more material properties.

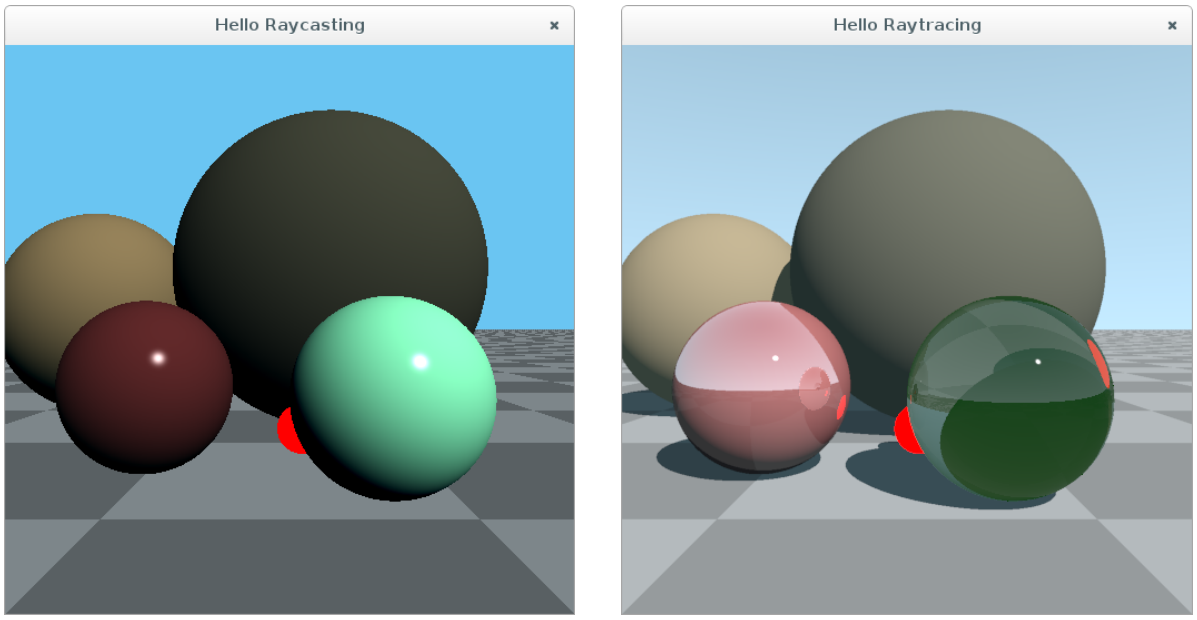

Figure 2: These two images show the result of the basic ray caster given in the skeleton code, and what the result might look like after you have extended it to a ray tracer. Note that the surfaces here use Blinn-Phong reflectance, and not the plain Lambertian that is included in the skeleton code.

secondary ray (which might, in turn hit a reflective object and bounce further).

3. Transparent surface: an additional ray is refracted into the material. The final color is a combination of the diffuse shading, the color accumulated by the refractive and reflective rays.

You may be wondering, what happens if the reflected/refracted rays hit another reflective/refractive surface, and another, and another? Naturally, a maximum depth is required, otherwise there is no guarantee for how long the program will take to finish, e.g. in the case of a ray bouncing between two parallel mirrors.

The traditional strategy for implementing a raytracer would involve pointers, recursion and dynamically allocated memory, none of which are available in GLSL. Instead, the skeleton code implements a ray stack (in lieu of a proper function call stack) and a `push` operation.

Tasks

1. For all intersections, cast a ray towards the light source and check if it intersects an object on the way there. If so, set the irradiance to a small "ambient" value, and otherwise, use the irradiance from the light souce as in the skeleton code. This can be done in a one-shot manner, without updating the ray stack.

   For objects with *specular* properties (reflection or transmission) you need to push a new ray to the ray stack, for further processing.

2. For objects with reflective properties, shoot an additional ray, reflected about the normal (hint: use the `vec3 reflect(vec3 I, vec3 N )` function).

Attribute a weight to the reflected ray (based on the amount of reflection of the material). The reflected ray will be handled in the next iteration of the loop in the `raytrace` function.

3. For transmissive objects, it is slightly more complicated.

   The direction in which a transparency ray should be cast can be determined using the material's index of refraction as well as the *current* index of refraction (IOR). To do this properly, the index of refraction needs to be propagated through all calls to the raytracing function. However, to simplify things, you can assume that you don't have nested refractive objects, and that the IOR is always 1.0 (vacuum) on one side of the surface.

   Using this assumption, you still need to check if the current ray is hitting a surface from the outside or the inside of the object. Compute the direction of the refracted rate using the IOR on the inside of the surface (`isec.material.ior`) and the IOR on the outside (1.0).

   To be physically correct, you should compute the amounts of light being refracted into the material and reflected off the surface using the Fresnel Equation (or with Schlick's approximation of it). De Greve[3] describes how to compute the refraction angle in a pedagogical way. Combine the color collected by the refracted and reflected ray. However, it is OK to use fixed ratios, as specified by `isec.material.reflection` and `isec.material.transmission` too. (Just make sure that the amounts of transmission and reflection don't add up to more than one).

   You may also include absorption by accounting for the length of the path the ray traversed inside the object and add some of its diffuse color (or a simple constant term, for simplicity). This should reduce the weight of the ray that comes out on the other side.

Questions

1. Is the raytracing 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?

2. 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?

Deadline

The deadline is December 16. If you hand in your report after this, it will still be graded, but only when there is time.

[3] Bram De Greve. *Reflections and refractions in ray tracing*. 2004. URL: http://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf.