1: Ray casting

Initialization

You will implement a ray-casting algorithm directly in a fragment shader. The given code simply displays a quad that covers the entire viewport. The fragment shader will then process each pixel of the screen in parallel. The only file that you need to edit is “fragment_shader.fs”. Consider that you are on a single pixel and that you want to find its final color. The following input variables will be useful to control and animate your rendering:

- fragCoord: the coordinates of the current pixel (in [-1,1]),
- mousePos: the position of the mouse (in [-1,1])
- time: number of seconds (since the beginning of the application)
- aspectRatio: width/height

Goals

The goals of this practical consist in reproducing a simple scene using the ray-casting technique using the tools seen during the lecture:

- Define a perspective camera to generate rays,
- Define at least one planar surface in the scene
- Define at least one sphere in the scene
- Intersect the ray with the scene (plane(s) + sphere(s))
- Display normals and depths

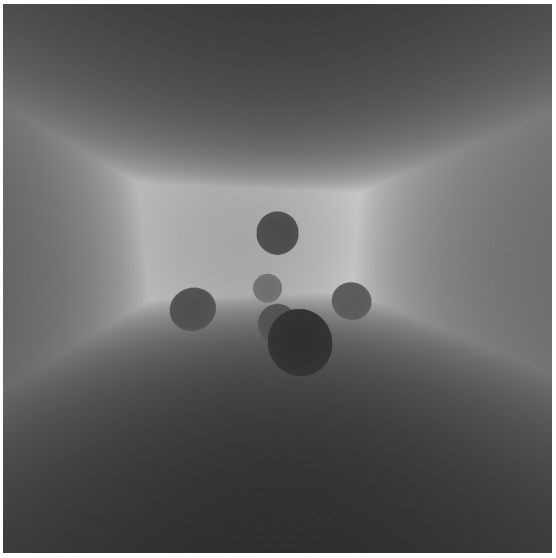


Illustration 1: Depths

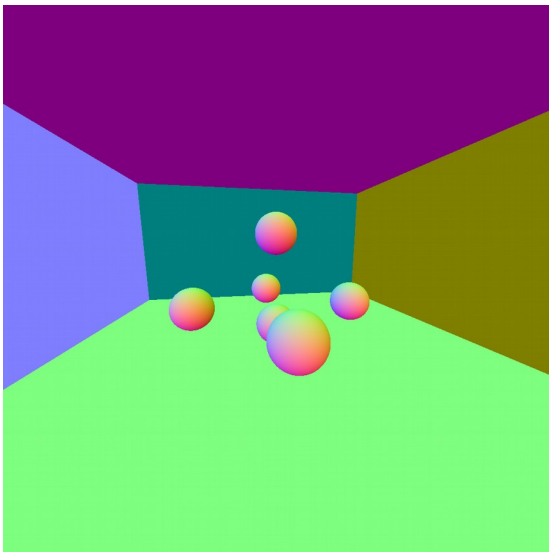


Illustration 2: normals

Reminder: ray-casting

- For each pixel
  - generate a camera ray
  - for each object in the scene (planes+spheres)
    - test the intersection between the object and the ray
    - remember this object if its distance is less than the other tests
  - compute the normal of the intersected object
  - compute the color of the pixel (display normal or depth values here)

GLSL tips

You can define GLSL structures (as in C). For instance, a ray and a plane can be defined using the following structure:

```
// ray structure

struct Ray {
vec3 ro; // origin
vec3 rd; // direction
};


// plane structure
struct Plane {
vec3 n; // normal
float d; // offset
};
```

The creation of an object can be done using the following:

```
Plane pl = Plane(vec3(0,1,0),0); // plane y=0
```

You can create arrays in GLSL. But be careful: their lengths must be defined with constant values (dynamic arrays and pointeurs do not exist in GLSL):

```
Plane pls[2] = Plane[](Plane(vec3(0,1,0),0), Plane(vec3(0,-1,0),5)); // 2 planes in an array
```

It is finally advised to write functions to create a clear code, and do loops as C:

```
for(int i=0;i<2;++i)
        computeIntersectionWithPlane(pls[i]);
```

Camera

How to generate a camera ray (reminder):

x and y : pixel coordinates (between -1 and 1)
u, v and w : (orthogonal) frame of the camera
ro = e : location/origin of the camera ray (= eye)
rd = unit vector of the camera towards the scene
D = 1/tan(alpha/2), alpha being the field of view

$$r = (x.u, y.v, D.w)$$
$$r_d = r/||r||$$
$$r_o = e$$
$$P(t) = r_o + r_d * t$$

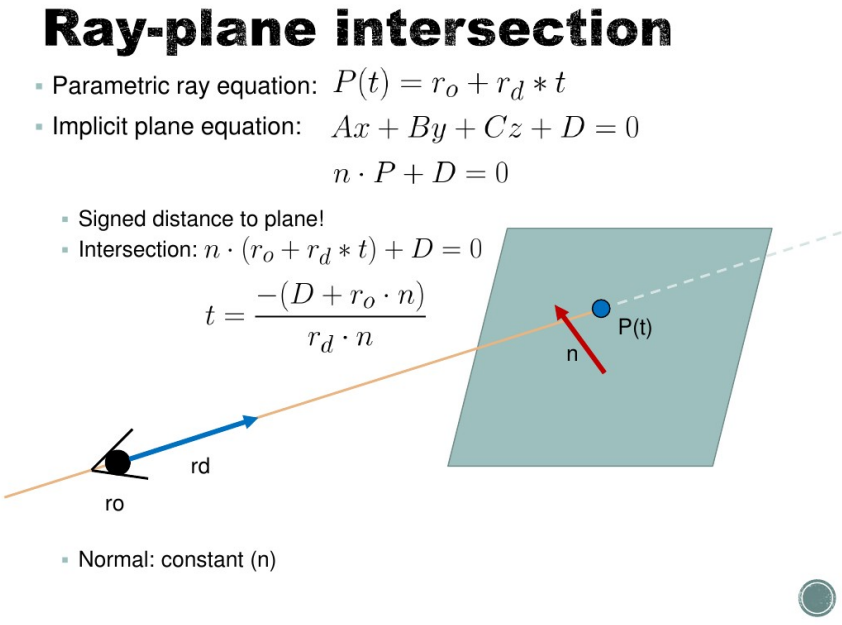
The simplest is to first compute the view vector w using the position of the camera and the location in the scene to look at. Reminder that u,v and w are unit vectors.

The v vector v can be initialized at (0,1,0) to specify that the camera is oriented toward the top of the scene.

The u vector can then be computed using the vectorial product of v and w. This operation should also be repeated to compute the "true" v vector (otherwise the frame might not be orthogonal).

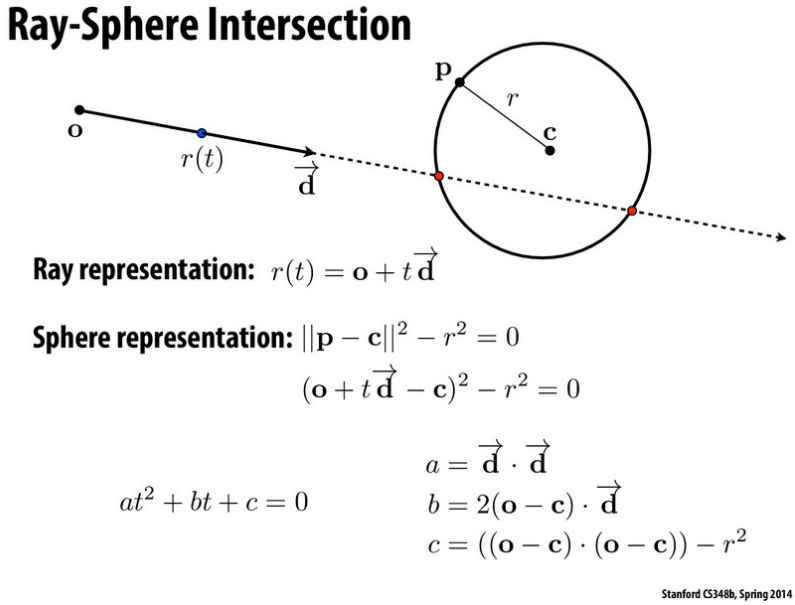
Once the frame computed, the other parameters can directly be computed.

Ray-plane intersection



The intersection test between a ray and a plane should return a distance value (the "t" value, obtained with the plane equation and the ray parameters, as described in the slide above). If your scene contains multiple objects, your function should return not only the distance to the object, but also an ID that will allow to access it afterwards (to compute its normal/depth for instance).

Ray-sphere intersection



$$d = \sqrt{(b^2 - 4ac)}$$

$$t = \frac{-b \pm d}{2a}$$

Note that this version is slightly different from the one seen during the course, but compute the same thing, more efficiently.

Remind that the distance to store is the minimum positive one. If the determinant is negative, there is no solution. If the determinant is equal to 0, there is only one solution (silouhette).

Last advices

Use function and sub-functions to easily test your shaders. If you have to debug, note that you do not have any way to print something in GLSL → you will thus need to improvise and to debug with colors (display red/green to test a condition for instance).

Bonus

- Animate your objects, your camera with simple functions or with mouse positions.
- Add some elements in your scene:
  - other spheres/planes
  - try novel objects (cylinder/etc)
- Try to obtain a simple shading (how to color your scene?)