MSIAM TP4: Ray Marching

M. Garcia

April 9, 2019

1 Introduction

In the last practicals we saw how to simulate light behaviour using Ray tracing. Moreover, this method lie on the principle that we can compute analytically ray-surface intersections, making computation quite fast. However, this is also a limitation as we are bound to display only surfaces for which we are able to compute ray-surface intersection. In this practical, we will implement a generalization of Ray Tracing named Ray Marching which will allow us to display any implicit surface defined by an expression $f(x, y, z) = 0$. In our context the f function will be named a Signed Distance Function (SDF) as when $f(p) > 0$ means that p lie outside the surface and when $f(p) < 0$ it is inside.

The fundamental principle remains the same, for each pixel of the screen we will cast a ray going from the camera center to the pixel but instead of computing intersection by solving an intersection equation, we will go through the generated ray step by step and check if we intersect an object. More precisely, for a given ray $r(t) = at + c_{camera}$ we will increase the value of t until we have $f(r(t)) \leq 0$ meaning that the current point on the considered ray is inside a surface of our scene.

There are two main ways of increasing the value of t while going through a light ray: one is to increase t by a constant small value until we intersect one surface while the other is to compute the minimal distance we can travel without intersecting anything by taking the minimum of all $f(r(t))$. This second approach is called Sphere (or Spherical) Marching and will be the approach we will use in this practical.

Below is the code of the ray marching main loop which is very similar to our previous trace function. Notice that IntersectObjects has been replaced by rayMarch which will be the function returning the minimal $f(r(t))$ among all the scene surfaces.

```
vec3 march(in Ray r)
{
        vec3 accum = vec3(0.0f);
        vec3 mask = vec3(1.0f);
        int nb_refl = 0;
        float c_refl = 0.3;
        Ray curr_ray = r;
        for(int i = 0 ; i <= nb_refl ; i++)
        {
                ISObj io = rayMarch(curr_ray);
                if(io.t >= 0)
                {
                        HitSurface hs = HitSurface(curr_ray.ro + io.d*curr_ray.rd
                                        ,computeSDFNormal(io,curr_ray.ro + io.d * curr_ray.rd)
                                        ,color,roughness,ao,metal) ;
                        vec3 color = directIllumination(hs,true,c_refl);
                        accum = accum + mask * color;
                        mask = mask*c_refl;
                        curr_ray  = Ray(hs.hit_point + 0.001*hs.normal
                        ,reflect(curr_ray.rd,hs.normal));
                }
        }
        return accum;
}

ISObj rayMarch(in Ray r)
{
        int nb_step = 255;
        float depth = 0.0f;
        float eps = 0.001;
        for (int i = 0; i < nb_step; i++)
        {
                ISObj io = sceneSDF(r.ro + depth * r.rd);
                if (io.d <= eps)
                {
                        return ISObj(depth,io.t,io.i);
                }
                // Move along the view ray
                depth += io.d - DIST_MIN;

                 if (depth >= DIST_MAX)
                 {
                        // Gone too far; give up
                        return ISObj(DIST_MAX,-1,-1);
                 }
        }

        return ISObj(DIST_MAX,-1,-1);
}
```

Additionally, in this practical surfaces will be represented as surface operation units SdOpU which represent a surface operation between two SDF. As seen during the lecture, it is quite easy to blend, unite, intersect or invert surfaces using the SDF representation.

```
//SDF operation Unit structure
struct SdOpU
{
        int s_t; /start type
        int s_id; //start id
        int e_t; // end type
        int e_id; //end id
        int op_id; // operation id
};
```

Using this structure, we will need two additional functions to retrieve a SdOpU SDF and compute the distance with respect to an input point.

```
float getSDF(int type, int id,in vec3 p)
{
        if(type == 0)
        {
                return SDFPlane(planes[id],p);
        }
        else if(type == 1)
        {
                return SDFSphere(spheres[id],p);
        }
        else if(type == 2)
        {
                return SDFBox(boxes[id],p);
        }
        return 0.0;
}

float SDFSdOpU(in SdOpU unit, in vec3 p)
{
        float s_sdf = getSDF(unit.s_t,unit.s_id,p);
        float e_sdf = getSDF(unit.e_t,unit.e_id,p);
        if(unit.op_id == 0)
        {
                return opUnion(s_sdf,e_sdf);
        }
        else if(unit.op_id == 1)
        {
                return opSubtraction(s_sdf,e_sdf);
        }
        else if(unit.op_id == 2)
        {
                return opIntersection(s_sdf,e_sdf);
        }
        else if(unit.op_id == 3)
        {
                return opSmoothUnion(s_sdf,e_sdf,1.0f);
        }

        return 0.0;

}
```

The getSDF function returns the corresponding SDF related to the given surface type and its id while SDFSdOpU compute the SdOPU operation between the two SDF of the unit. Below are the code for the different operation we will use.

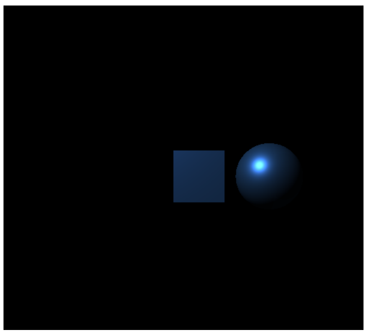
```
float opUnion( float d1, float d2 ) {  return min(d1,d2); }

float opSubtraction( float d1, float d2 ) { return max(-d1,d2); }

float opIntersection( float d1, float d2 ) { return max(d1,d2); }

float opSmoothUnion( float d1, float d2, float k )
{
    float h = clamp( 0.5 + 0.5*(d2-d1)/k, 0.0, 1.0 );
    return mix( d2, d1, h ) - k*h*(1.0-h);
}
```

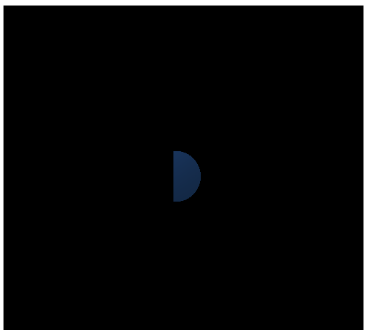
Finally, in the same manner we implemented intersectScene in our ray tracing, we will implement the sceneSDF function, returning the distance to the closest surface in the scene. Additionally, we will compute the surface normal using its discrete derivative $f(p+\epsilon) - f(p-\epsilon)$

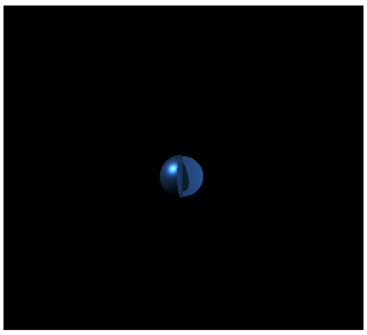
```
ISObj sceneSDF(in vec3 point)
{
    ISObj nearest = ISObj(DIST_MAX,-1,-1);
    for(int j = 0 ; j < suo_nbr ;j++)
    {
        float dist = SDFSdOpU(suos[j],point);
        if(dist >= 0 && dist < nearest.d)
                nearest = ISObj(dist,10,j);
    }
    return nearest;
}



vec3 computeSDFNormal(in ISObj is,in vec3 p)
{
        vec3 p_x_p = p + vec3(0.001, 0, 0);
        vec3 p_x_m = p - vec3(0.001, 0, 0);
        vec3 p_y_p = p + vec3(0, 0.001, 0);
        vec3 p_y_m = p - vec3(0, 0.001, 0);
        vec3 p_z_p = p + vec3(0, 0, 0.001);
        vec3 p_z_m = p - vec3(0, 0, 0.001);

        else if(is.t == s_op_id)
        {
                return normalize(vec3(SDFSdOpU(suos[is.i],p_x_p) - SDFSdOpU(suos[is.i],p_x_m)
                                ,SDFSdOpU(suos[is.i],p_y_p) - SDFSdOpU(suos[is.i],p_y_m)
                                ,SDFSdOpU(suos[is.i],p_z_p) - SDFSdOpU(suos[is.i],p_z_m)
                                ));
        }

    return vec3(0,0,0);
}
```

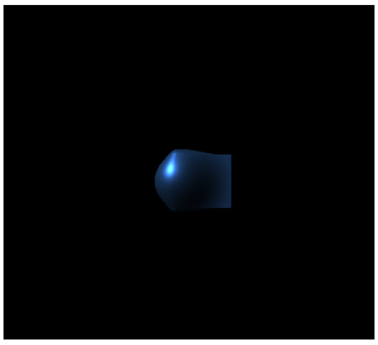
Below are some outputs of you should obtained with one box and one sphere combined with different operators.

```
// signed distance function of Box
float SDFBox(in Box b, in vec3 p)
{
        vec3 d = abs(p - b.c) - (b.b) ;
        return length(max(d,0)) + min(max(max(d.x,d.y),d.z),0);
}
```


(a) Sphere Box Union


(b) Sphere Box Intersect


(c) Sphere Box Subtract


(d) Sphere Box Blend