Raymarching inside unity

Robin Opheij

Grafische Lyceum Utrecht

Author Note

In this document I will talk about who I am, what raymarching is and how it started.

Talk a little bit about its future. And in the end actually creating a raymarching shader inside unity.

Abstract

I am Robin Opheij aspiring game developer, graphics programmer and machine learning enthusiast. This is the proper readable paper of the tutorial I posted on my website (link: add link to online tutorial.) If you downloaded this paper of my website you may have seen my other tutorials for instance the “Your first shader” tutorial. I first wanted to write a research paper about the engine I’m building but that is taking way longer than I expected and I won’t make the deadline for it. I will just work on that in my own time. But I will release that sometime. It is a C++ engine with C# for the game code. But for now lets focus on raymarching since that is also a very interesting subject.

Introduction to raymarching

Raymarching is a rendering technique used for for instance 3D fractals and volumetric rendering. I wanted to learn this technique because I enjoy doing graphics programming and this might be the way to render anything in the future especially in animation because of the level of detail you can acquire with easy equations and fast rendering.

Raymarching

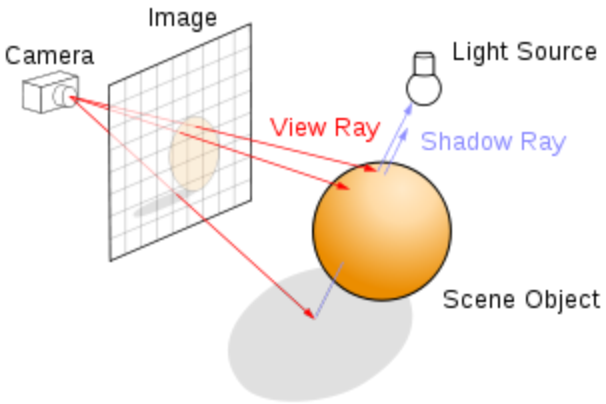
Tools

For this tutorial we are going to use Unity3D and its shader language to create a image effect shader that renders endless amounts of 3D balls in space.

Prerequisites

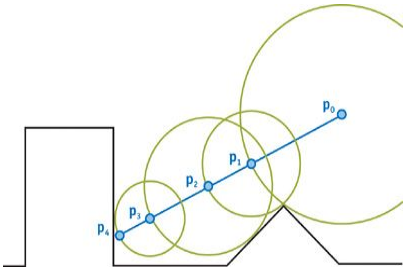
Since this is a pretty advanced tutorial I will assume you know how unity works and at least a little bit about shaders. If you don't, check out my shader tutorial since it will explain quite a bit about shaders. I also hope that you understand math a bit.

Raymarching through the years. Before raymarching there was raytracing, a simple technique that uses rays to send straight out from each pixel in the camera into the scene and once objects gets hit it renders the color of the pixel it hit. This also works with lights by casting another ray from the hit point to the light source. (*img 1)



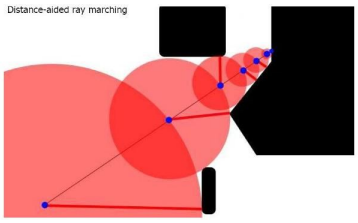
Img 1: ray tracing visual from wikipedia

Raymarching. Raymarching is a evolved form of raytracing, it has less computations and more accuracy which makes it faster and better looking. However it still is a very heavy operation for you graphics card.



(img 2)

Let's say we have point p0 which is the starting point from which we send the ray from this point we draw a circle until it hits something in the scene. After that we draw the next point on the radius of the circle which we then keep repeating until we hit an actual point right in front of the first point.



Where are we now and what does the future look like: When back in the day when people first started doing raymarching they just created simple spheres and cubes but now people are making beautiful visuals with it from fractals to hyper realistic environments. Sadly these barely ever get used outside of some very specific vfx jobs but something that does happen a lot that uses raymarching are anything that is being rendered volumetrically. For instance clouds in a lot of games are made with volumetric rendering which is raymarching is a big thing in doing that. But since it's still very heavy do make extremely detailed big environments using this technique it's very hard to implement it anywhere. Lucky the graphic cards get better day by day which overtime will make these operation much easier on the graphics card and it could become widely used. For instance in 3D animation engines could just all run on raymarching techniques where objects get generated into math which then will be rendered real time.

The how and why: I am going to explain to you how to make a raymarching scene inside of a unity image effect shader. Inside your unity project we are going to start making 2 “Scripts”, one which will be a normal C# script and one Image effect shaders. Project Tab>Right mouse button(RMB)>Create>C# script, Project Tab>RMB>Create>Shaders>Image Effect Shader.

We don’t need much to get the c# script up and running so lets start of with that.

C# programming. Open the script and add this piece of code:

```
[ExecuteInEditMode]
public class Blitter : MonoBehaviour
//I named the script blitter name it whatever you want though.
{
//The material with the image effect shader on it.
    [SerializeField] private Material m_Shader;

    private void OnRenderImage(RenderTexture src, RenderTexture dest)
    {
//Renders the materials to the rendertexture of our camera
        Graphics.Blit(src,dest,m_Shader);
    }
}
```

Add this script to the camera object and create a new material that uses the image effect shader we made earlier. Select the shader your created>RMB>Create>Material now the material uses the shader you had selected. Since the script uses the [ExecuteInEditMode] you should immediately see inverted colors on the screen once you add the material to the script on the camera.

Shader Scripting:

Now the shader scripting might be a bit harder to understand so I will go over it bit by bit. We Don't have to do much in the actual shader file before the vert function. The only thing we really care about is the frag function so lets edit that so it works for what we want to create. We first want to create 2 helper functions called *Trace* and *Map.* The map function is kind of the heart of most raymarching shaders. It returns the scalar of a given point in 3D space. And for a sphere which is what we are going to be rendering it's quite easy to get the scalar because it's just the length of the given point then subtracting the radius of the object which in our case will just be 0.25 (but you can change this however you like depending on which size you want your sphere to be). The map function should look something like this:

```
 float map(float3 point)
{
      float3 q = frac(p) * 2.0 - 1.0; //for multiple spheres delete
this if you want 1 sphere
      return length(point) - 0.25;
}
```

In the trace function we go through iterations to find where we converge the intersection point than map that point then to add the point to a variable that we keep outside the loop and multiply it all by 0.5, you can experiment with this number because it depends on what you subtracted in the math function.

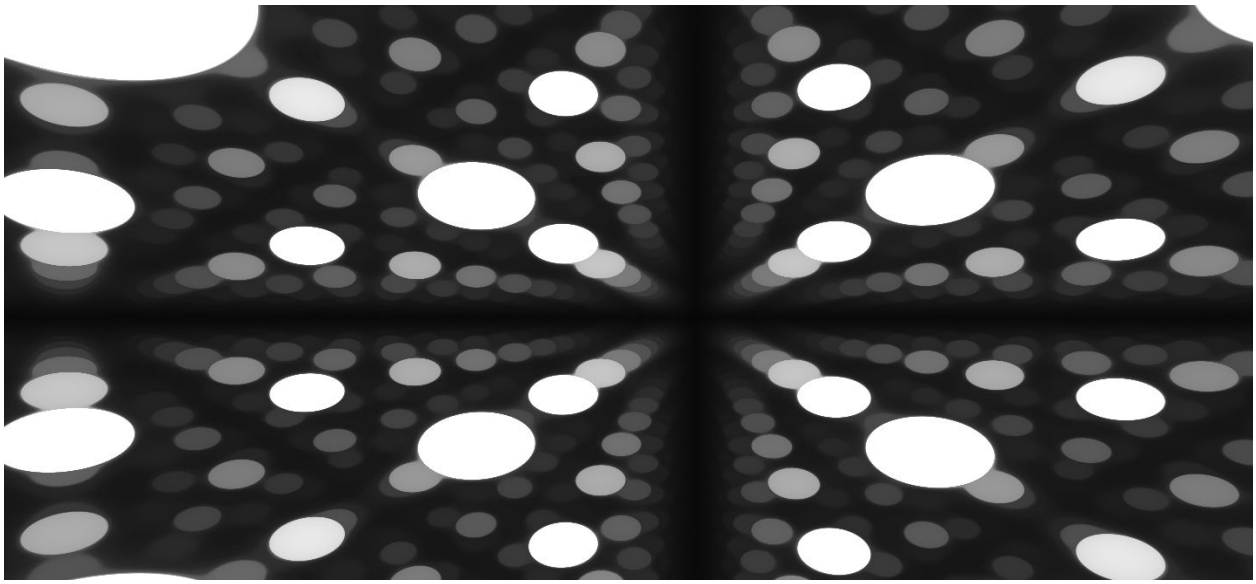
```
float trace(float3 o, float3 r)
{
      float t = 0.0;
      for (int i = 0; i < 32; i++)
      {
            float3 p = o + r * t;
            float d = map(p);
            t += d * 0.5;
      }
            return t;
}
```

And now long and behold we finally get to the actual frag function. This is where we create the ray and actually render our spheres.

```
fixed4 frag(v2f i) : SV_Target
{
   float2 uv = i.uv; //removing the i. from the uv variable
   uv = uv * 2 - 1.0; //transform coords from 0 - 1 to -1 - 1

   uv.x *= uv.x / uv.x; //correct the aspect ratio

   float3 r = normalize(float3(uv, 1.0)); //normalize so the ray doesn't poke trough on close objects
   //the 1.0 is basicly the fov or z value for the shader. You can play around with it.
   float rot = _Time.y * 0.25;
   r.xz = mul(r.xz, float2x2(cos(rot), -sin(rot), sin(rot), cos(rot))); //rotate the scene
   float3 o = float3(0.0, _Time.w, _Time.y); //camera position
   float t = trace(o,r); //trace the origin and find the intersection
   float fog = 1.0 / (0.1 + t * t * 0.1); //rendering the spheres First number is the bloom and visible amount
   //second number is bloom and the last number is amount of visible
   float3 fc = float3(fog,fog,fog);

   return float4(fc,1.0); //return the pixel colors with all our calculation
}
```

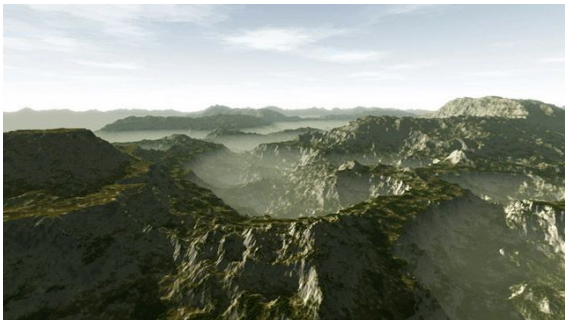
Sorry that this bit will be a screenshot the tools I am using completely butcher the code and don't allow me to fix it properly.

Results



Discussion

As you can see this is a very easy example of raymarching which you can easily create in 10 minutes if you know what you are doing. But if you are a starter you can play around with the numbers, try different algorithms, maybe even generate boxes instead of spheres. But if you keep learning someday you may be able to create images like these with raymarching.



References

Iquilezles. Big resource for raymarching. *https://iquilezles.org/www/index.htm* .