Real-Time Graphics Programming - Raymarching

Enes Sezgin

July 2022

1 Introduction

In this project, raymarching technique and its capabilities are examined and tried on two different scene. While examining raymarching, the other classic methods to create better scenes such as collision detection, camera movement, illumination techniques are applied and combined with raymarching.

First scene contains adjustable number of spheres that collide and merge together. Purposes of this scene are seeing how object count affects performance, examining combination techniques of raymarching and apply collusions to create visually plausable renders. Second scene contains a fractal object. Aims of the scene are seeing how scene complexity affects performance, observing the capability of creating different objects mathematically defined, applying repetition and substraction on objects, and observing the basic lighting applied on the scene.

Captured images and performance analyses are created using OpenGL (v4.10) and GLSL. The rendering computer has Intel® Core™ i5-8300H Processor, 16 GB RAM, and GEFORCE GTX 1050(4GB) graphics card. Additionally, all images captured and fps analyses are generated at 900x900 pixel dimensions. Before seeing how things work on the scenes, how raymarching and SDF's work should be explained.

2 What is Raymarching

Ray marching is an image-based rendering technique and depends on *ray casting* methodology. In *ray casting*, we send rays from the camera through each pixel on the image plane and find the closest object blocking the path of the ray. Once we hit an object, we can calculate color and shading using the information of the intersection point and assign it to our pixel. In techniques like *ray tracing* using *ray casting*, the intersection point is calculated analytically, generally non-performant for real-time graphics. A *ray marcher*, however, looks for an approximate solution by marching along the ray using *signed distance functions*. The marching operation ends when our ray is close enough to an object. Since we use approxi-

mations and *signed distance functions*, *ray marching* is efficient enough to achieve a certain degree of realism like in an acceptable time.

Figure 1: Beautiful scene created via Ray Marching [1]

2.1 Raymarching Algorithm

To create visuals, we use *RayMarch* function that gives the distance between the camera and the object. We use this distance to calculate the point in space, normal, and much more information about the point we hit. Then, we use these parameters to calculate the color of our pixel. Our objects are defined as mathematical functions in space, and the *SceneSDF* function gives the distance to the closest object. As we march through the ray, we get closer to the object that our ray should hit. Marching using *SceneSDF* is important because instead of marching a pre-determined amount, we can calculate the next position with *SceneSDF*. After all, it guarantees that there is no other object between our point and the closest object, so we can safely continue marching.

```
float RayMarch(vec3 origin, vec3 direction){
    float dist = 0.5; //current distance
    for(int i=0; i<MAX_MARCHING_STEPS ;i++){
        vec3 point = origin + direction * dist; //Find current point in
   space
        float distanceToClosest = sceneSDF(point);
        dist += distanceToClosest;
        if( distanceToClosest < SURFACE_DIST || dist > MAX_DIST)
            break;
    }
    return dist;
}
```

Listing 1: Raymarching algorithm

Function seen in Listing 1 is the ray marching function used in the project. Ray marching function gives us the distance between camera and point that our ray hit. So, we have our *origin* and direction of ray as parameters. For each marching (iteration), we find the current point in the space using our current distance. Then *sceneSDF* gives us closest distance to any object from that point. When we close enough or we are out of our boundary, we return the distance.

To demonstrate the function visually, I created the scene can be seen at Figure 2 containing some simple objects.

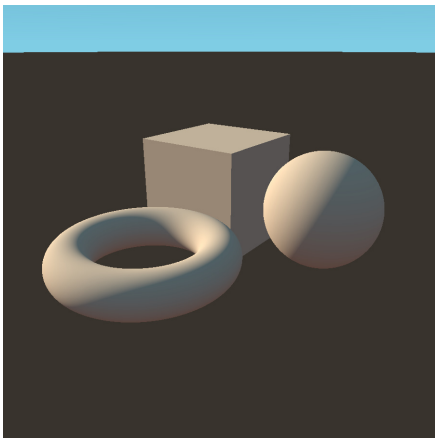


Figure 2: Test scene with a cube, a sphere and a torus.

With Figure 3, Figure 4 and Figure 5, you can see step by step demonstration of function *RayMarch*. Each distance is found by *sceneSDF* function and demonstrated with a circle. Red dots demonstrates step positions.

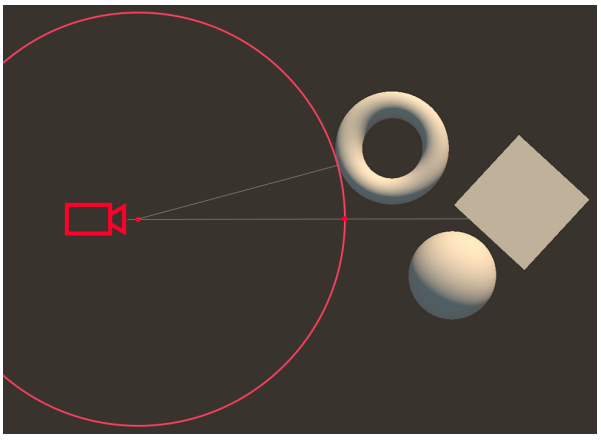


Figure 3: First march

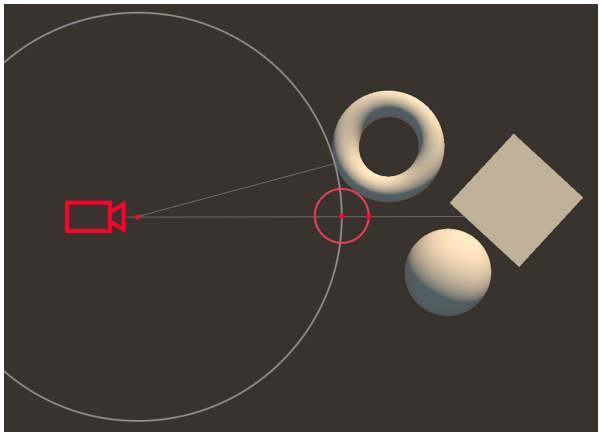


Figure 4: Second March

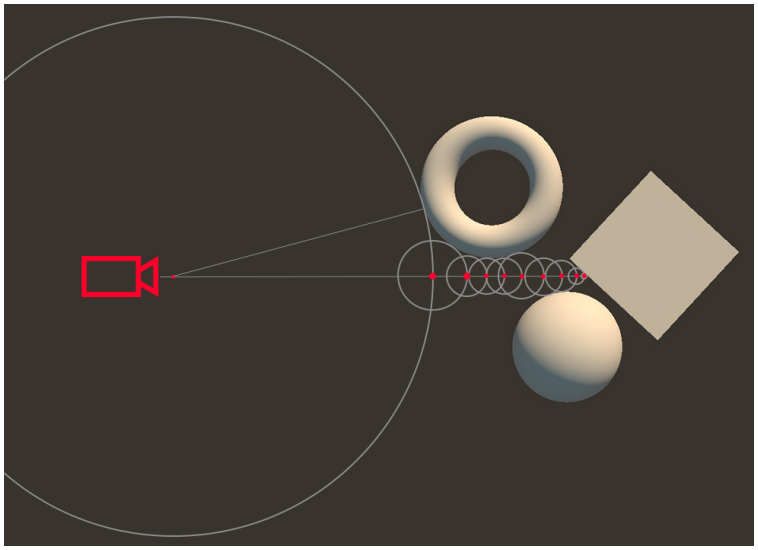


Figure 5: Final March

The *sceneSDF* function mainly determines the performance of finding distance along the ray. Since approximated distance is adequate for us, the ray marching function ends when it is close enough to a surface or out of boundaries. The critical case is when the ray passes near the object and does not intersect. Since it is close, but there is no intersection, there are many iterations, so there are many *sceneSDF* calls. Consequently, the project's performance is determined by how heavy the sceneSDF function is and how complicated the scene is. Iteration count and demonstration of how many times *sceneSDF* is called can be seen in Figure 6. Pure red means reached the max iteration, and pure black means zero iteration. Before performance analysis, we should check how signed distance functions work.

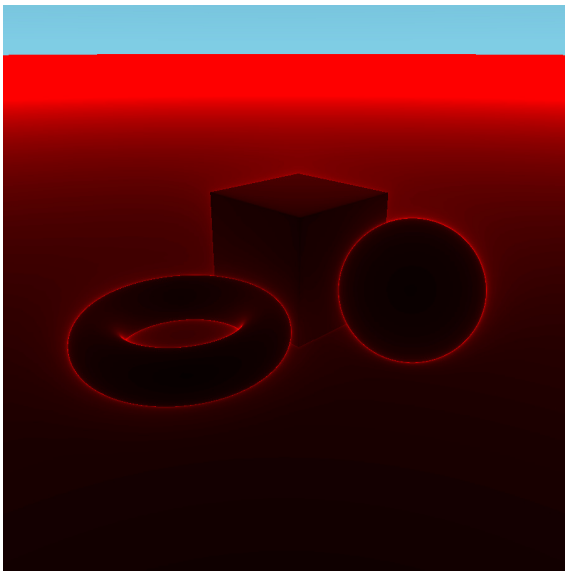


Figure 6: Iteration ount on the scene

3 Signed Distance Functions

Signed distance function (SDF) is a way to mathematically define objects in space. They take a point as a parameter and return the distance of determined object. While SDF functions can represent basic objects like sphere or cube, also they can be easily combined in different ways and represent various things. Let's see some examples of how they works.

3.1 Sphere

```
float sphereSDF(vec3 p, vec3 position, float size)
{
    return length(p - position) - size;
}
```

Listing 2: Sphere Signed Distance Function

SphereSDF simply gives distance between our point p and position of sphere minus size.

3.2 Box

```
float sdBox( vec3 p, vec3 b ,vec3 position)
{
    p -= position;
    vec3 q = abs(p) - b;
    return length(max(q,0.0));
}
```

Listing 3: Box Signed Distance Function

Before calculating in 3D, we can demonstrate how we achieve box in 2D because it simply applies one more dimension. So as you can see in Figure 7, we can divide our box into three areas. Other areas can be calculated using an absolute function; basically, only calculating positive areas will give other areas too. Let's say that distance is d and the current point is p. For area number 1, $d = p_x - b_x$. For area number 2, $d = p_y - b_y$. For area number 3, d is the distance of our point to the corner, which means $d = \sqrt{(p_x - b_x)^2 + (p_y - b_y)^2}$ We can combine these three equations to one with this line: $d = length(max(abs(p) - b, 0))$. We are only working on the positive area. When any of the parameters of p is negative, with abs, we calculate it as a positive number, so we mirror it in axes.

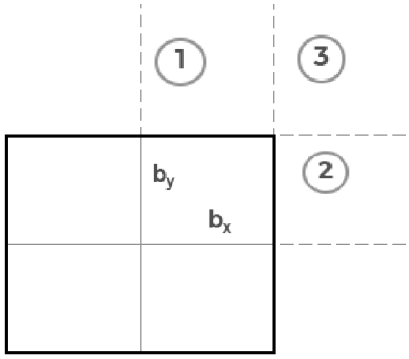


Figure 7: Final March

3.3 Combination

By combining SDFs, we can create complex structures made of simple objects. These operations are *union*, *subtraction*, and *intersection*. For example, when our ray intersects with three different objects, the camera only will see the closest one. So, using the minimum function, we achieve the union of objects, and we are able to see them together in the scene. Visual examples and code that I use in my scenes can be seen.

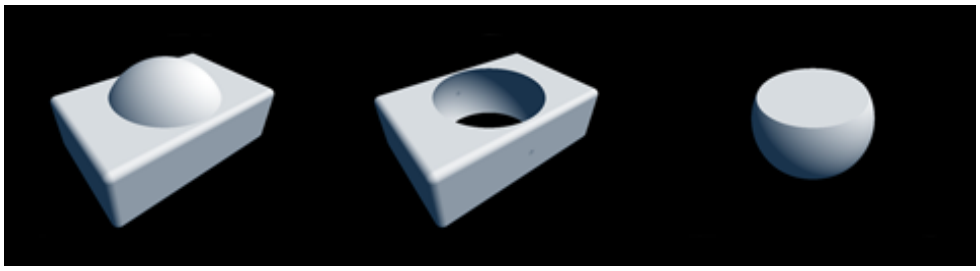


Figure 8: Primitive Combinations [2]

```
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); }
```

Listing 4: Box Signed Distance Function

When we use these direct operations, the resulting shape has discontinuities in its derivatives. Or in other words, the resulting surface of unifying two smooth objects is not a smooth surface anymore. The way to smoothly blend the shapes is to get rid of the discontinuity of the min() function, and we can achieve this by adding some smoothness factor when two object are really close to each other. You can see polynomial smoothness function in listing below.

```
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);
}
```

Listing 5: Polynomial Smoothness Function

4 Bouncy Balls Scene

The bouncy balls scene contains an adjustable number of spheres that collide and merge together.

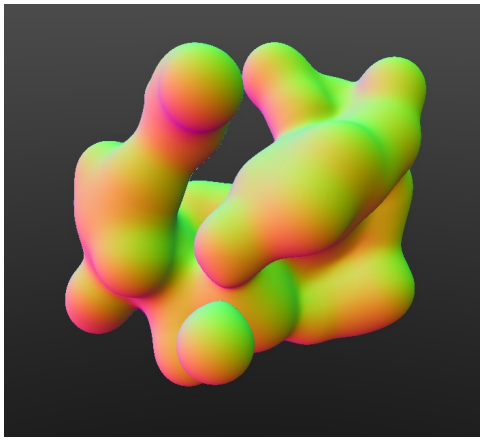


Figure 9: Bouncy Balls Scene Example

4.1 Physics

For the physics part, spheres' collision information is calculated in the program and passed to shaders with a uniform. Since spheres' rotation information is not essential, only position information is passed. Spheres' collide with each other, and with an invisible box around them with 0.9 bounciness. To create an environment where collisions and union operations are fully visible, I applied random force to each object by holding it down to the space key. Ball count and collision size can be adjusted via the GUI tool created using *Dear ImGUI*[3]. The physics part of the code calculated for each frame can be seen in listing below.

```
// we cycle among all the Rigid Bodies
int num_cobjs = bulletSimulation.dynamicsWorld->getNumCollisionObjects();
btTransform transform;
GLfloat matrix[16];
for (int i=6, x=0; i<num_cobjs;i++)
{
    // we take the Collision Object from the list
    btCollisionObject* obj = bulletSimulation.dynamicsWorld->
   getCollisionObjectArray()[i];
    // we upcast it in order to use the methods of the main class
   RigidBody
    btRigidBody* body = btRigidBody::upcast(obj);
    // we take the transformation matrix of the rigid boby, as calculated
   by the physics engine
    body->getMotionState()->getWorldTransform(transform);
    // we update collision size
    body->getCollisionShape()->setLocalScaling(btVector3(collisionSize,
   collisionSize, collisionSize));
```

```
    bulletSimulation.dynamicsWorld->updateSingleAabb(obj);
    // apply impulse when pressed space
    if(keys[GLFW_KEY_SPACE])
    {
        btVector3 impulse(rand()%5-2, rand()%5-2, rand()%5-2);
        body->activate(true);
        body->applyCentralImpulse(impulse);
    }
    //Set new positions
    bouncyBalls[x++] = glm::vec3(transform.getOrigin().getX(), transform.
    getOrigin().getY(),transform.getOrigin().getZ());
}
// Send information to the shaders
glUniform1f(glGetUniformLocation(shader.Program, "visibleSize"),
    visibleSize);
glUniform3fv(glGetUniformLocation(shader.Program, "ballPos"), ballCount,
    glm::value_ptr(*bouncyBalls));
glUniform1i(glGetUniformLocation(shader.Program, "ballCount"), ballCount);
glUniform1f(glGetUniformLocation(shader.Program, "blending"), blending);
```

4.2 Shader Part and Normals

Objects are created and combined in *sceneSDF* function in the shader part. The blending parameter and visible size of spheres' are adjustable again via GUI. For the visual aspect of the spheres, I used basic coloring using parameters of *normal* calculation. At any point on a surface defined in the distance field, the *gradient* of the distance field is the same as the object's normals at that point. The gradient of a scalar field (such as a signed distance field) is essentially the derivative of the field in the x, y, and z directions. In other words, for each dimension d, we fix the other two dimensions and approximate the derivative of the field along d. Intuitively, the distance field value grows fastest when moving directly away from an object (along its normal). So, by calculating the gradient at some point, we have also calculated the surface normal at that point. The normal calculation part of the code can be seen in listing below. Parameter p is a point in the surface of object.

```
vec3 GetNormal(vec3 p)
{
    float d = sceneSDF(p);
    vec2 e = vec2(0.01, 0);
    vec3 n = vec3(d) - vec3(
        sceneSDF(p-e.xyy),
        sceneSDF(p-e.yxy),
        sceneSDF(p-e.yyx)
    );
    return normalize(n);
}
```

Listing 6: Normal Calculation Function

4.3 Performance

Since performance based on mostly how complicated is sceneSDF, bouncy balls scene fits perfectly for object count test. To test, I increased ball count one by one in every two seconds and take average of FPS in these intervals. Here you can see the graph.

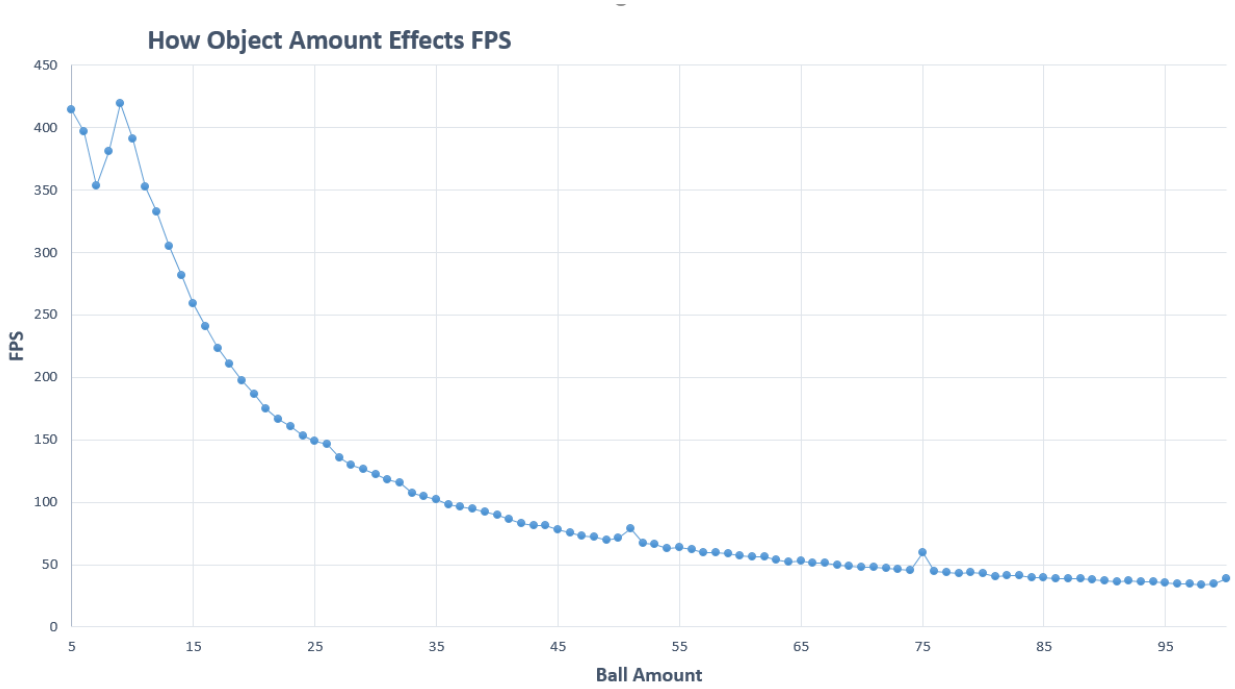
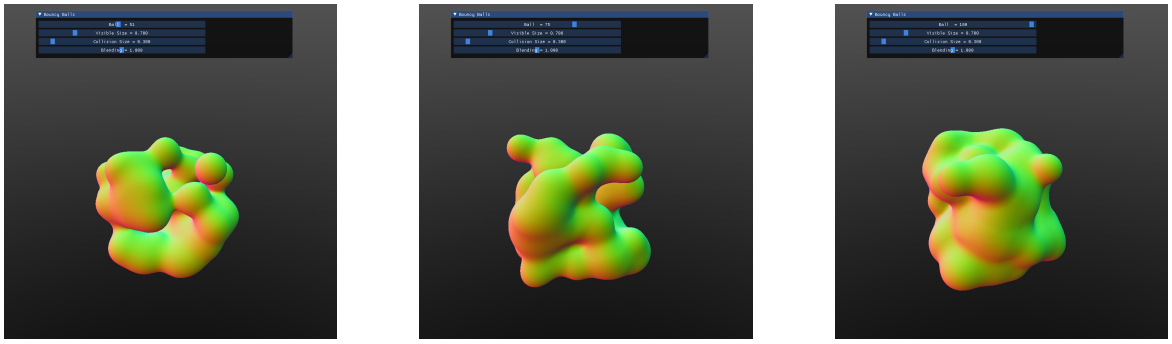


Figure 10: Ball Count vs FPS

It can be seen that object count significantly affects the performance. As object count increases, FPS drops. Since this scene is simple in terms of effects, like there is no lighting or effects like ambient occlusion, the maximum object count should be limited to 40-45 to get a smooth experience. If there are more effects or objects will be more complicated, the count should be even more lowered.



(a) 50 Ball (b) 75 Ball (c) 100 Ball

Figure 11: Scenes while testing with different ball counts

5 Fractal Scene

In fractal scene, there is an fractal created using ray marching's a strong feature: repetition. Addition to calculation of fractal, I added basic lighting, and shadows to the scene.

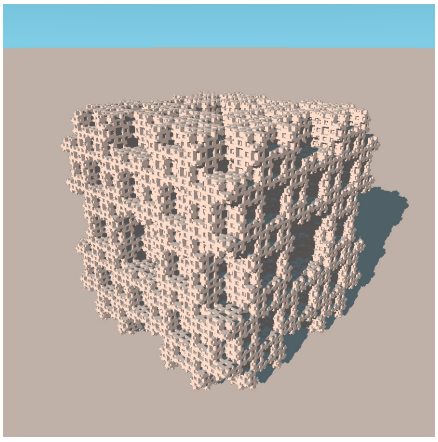


Figure 12: Fractal Scene

5.1 Repetition

In ray marching, as I mentioned, object count highly affects performance. However, there is a feature that ray marching can create unlimited items with very high performance: repetition. You can create an object, and you can repeat it as many as you want. This can be achieved by folding the space in the scene. What I mean by folding the space is mirroring with absolute and using modulus.

As you can see in Listing 7, when you take the absolute value of x, you mirror the positive values of x to negative ones. You behave negative values as positive. The good thing about this is when you use mirroring, your usage of sceneSDF does not change, and you can get double of objects with the similar resource usage.

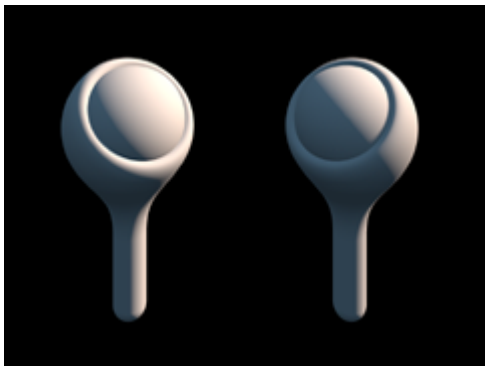


Figure 13: X Symmetry [4]

```
vec3 opSymX(vec3 p)
{
    p.x = abs(p.x);
    return p;
}
```

Listing 7: X Symmetry Function

We can extend our mirroring ability to an infinite amount. As you mirror axes at regular intervals, you will achieve infinite mirroring. And this can be achieved by modulus operation. Since we only take the modulus of our point(mirroring) and then use the same *sceneSDF*, the performance will be good even there is infinite amount of objects. In below listing and figure, you can see *opRep* function applied to a single cube.

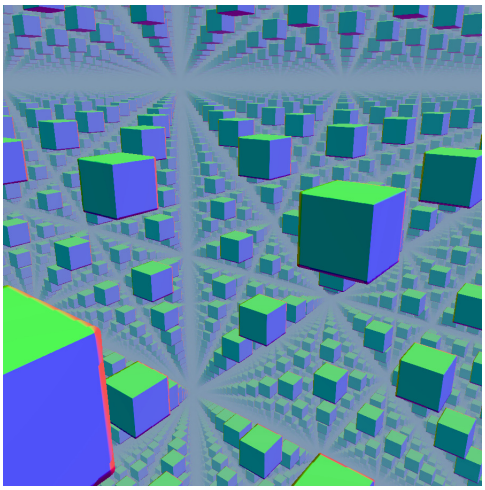
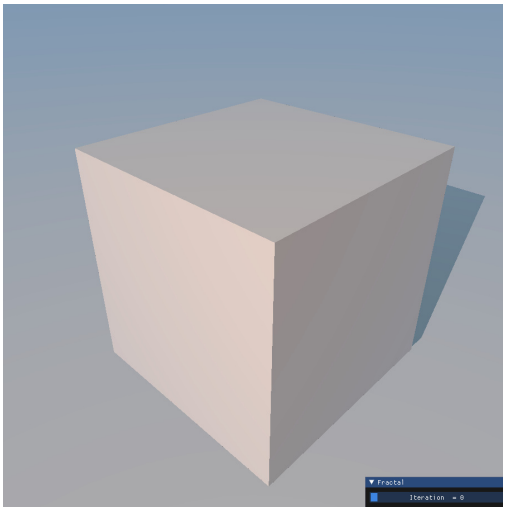


Figure 14: Infinite cubes

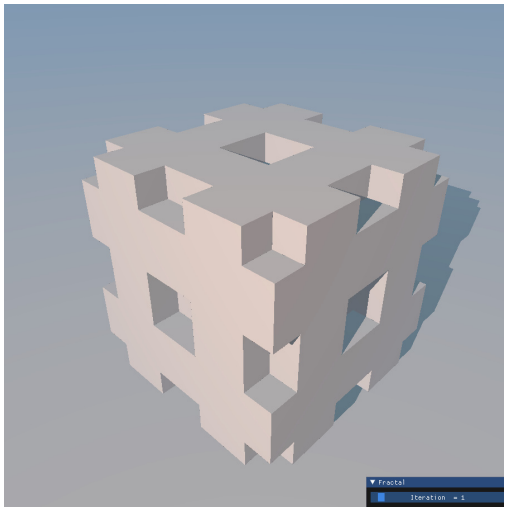
```
vec3 opRep( vec3 p, vec3 c)
{
    //c is period
    vec3 q = mod(p+0.5*c,c)-0.5*c;
    return q;
}
```

5.2 Fractal Generation

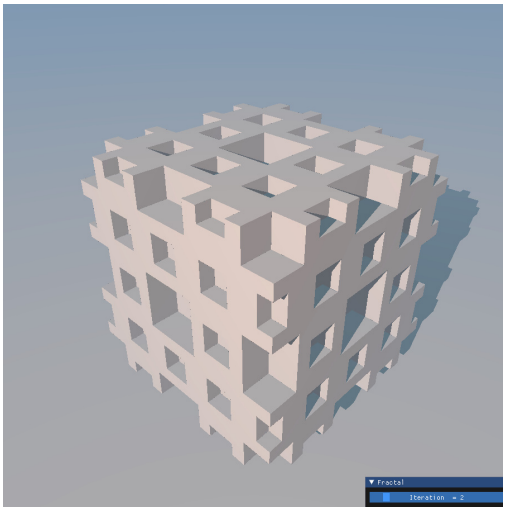
For the fractal generation, I used repetition and subtraction together. Firstly, I created a single cube,then subtracted an infinite cube sequence. Fractal deepness and complexity are determined by fractal iteration. For each iteration, subtraction continued an infinite cube sequence with smaller cubes and different repetition period. The size of the cubes and period are determined by a scale factor multiplied by parameters in every iteration.



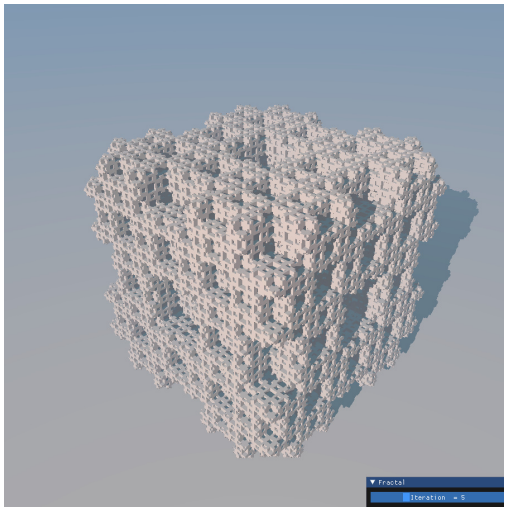
(a) 0 iteration



(b) 1 iteration



(c) 3 iteration



(d) 5 iteration

Figure 15: Scenes while testing iteration count

```
float sceneSDF(vec3 p){
    float d;
    d = sdBox(p, vec3(4), vec3(0, 4, 0));
    float scale = 1;
    float x = 4;
    for(int i=0;i<fractalIteration;i++)
    {
        vec3 q = mod(p+0.5*x*scale,x*scale)-0.5*x*scale;
        float c = sdBox(q, vec3(1*scale), vec3(0));
        d = max(d,-c);
        scale/=2;
    }
    float f = (p + vec3(0,1,0)).y + 0.1;
    return min(f,d);
}
```

Listing 8: sceneSDF Function of Fractal Scene

5.3 Performance

As mentioned before, in ray marching's performance, the complexity of the scene is as important as the number of objects. Since, fractal scene is a good choice to test scene complexity and performance relation. To measure, I changed *fractalIteration* variable via GUI and take the average FPS.

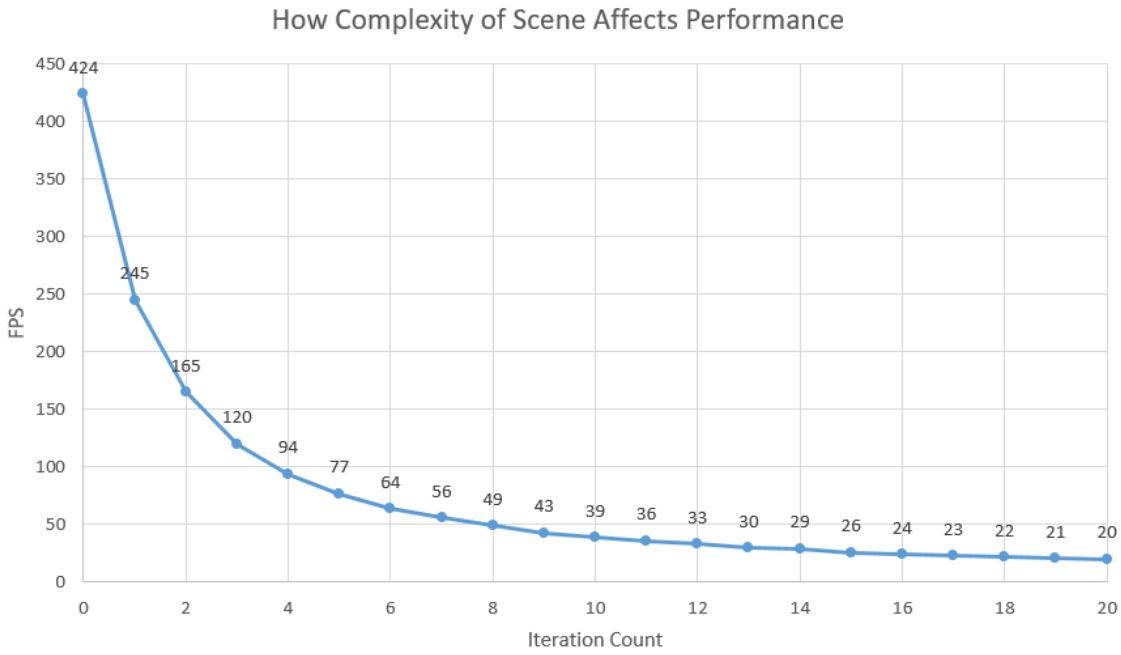


Figure 16: Primitive Combinations by Inigo Quilez

6 Conclusion

Ray marching is a great image-based rendering technique that can be used for creating beautiful yet efficient scenes. This technique uses mathematical definitions via signed distance functions instead of meshes to create objects in the scene. Simple objects defined by signed distance functions can be combined in various ways to create more complex objects. It is essential to consider object count and scene complexity to get a good performance. Ray marching loses its power when there are too many objects in the scene; however, by using the power of mirroring and repetition abilities, you can easily get powerful concepts with good performance. Since you can infinitely repeat objects, you can achieve scenes that would be impossible to create with regular rendering techniques.

References

[1] Inigo Quilez (2020) *Selfy Girl*, https://www.shadertoy.com/view/WsSBzh.

[2] Inigo Quilez, *Primitive combinations*, https://iquilezles.org/articles/distfunctions/.

[3] *Dear ImGUI*, https://github.com/ocornut/imgui/.

[4] Inigo Quilez, *Symmetry*, https://iquilezles.org/articles/distfunctions/