C++ & CUDA For Rendering Sierpinski Fractals

C++ for Scientific Computing

Candidate Number: 1060612

1 Introduction

In this paper, we discuss how to use C++ and heterogeneous computing techniques involving CUDA to render Sierpinski fractals in 2D and 3D. We implement a ray marching algorithm with basic lighting models to visualise 3D environments on a 2D screen and derive signed distance functions for the Sierpinski fractals. Also, to do the mathematical calculations we develop a maths library similar to the GLSL specification.

1.1 Image Format

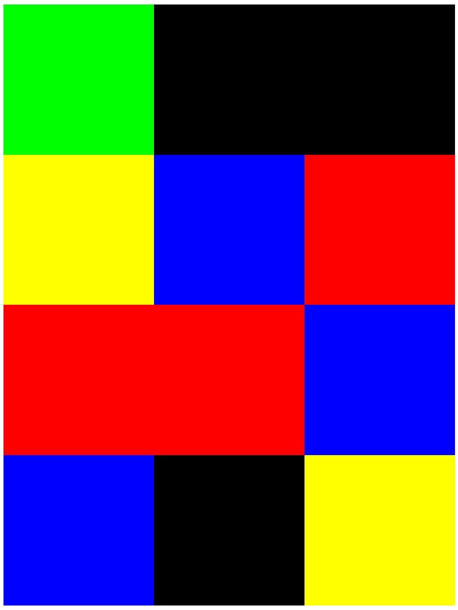
In this project, we use the .PPM file format to save the render of our fractals. This extension stands for Portable Pix (or pixel) Map.

The .PPM file format is as follows: line 1 in this project is "P3"; followed by line 2 which is the resolution of the picture ("Width_Resolution Height_Resolution"). Then line 3 represents the maximum colour value. The following lines each represent the colour of a pixel in the format RGB starting at the top-left pixel and travelling horizontally.

Figure 1a shows an example .PPM file, with the picture it represents shown in Figure 1b.

```
1 P3
2 3 4
3 255
4 0 255 0
5 0 0 0
6 0 0 0
7 255 255 0
8 0 0 255
9 255 0 0
10 255 0 0
11 255 0 0
12 0 0 255
13 0 0 255
14 0 0 0
15 255 255 0
```

(a) .PPM file for Figure 1b



(b) Visual output of .PPM file

Figure 1: .PPM file text with visual output

As can be seen, this file format is a lot simpler than .JPEG and .PNG. However, .PPM takes up a lot more storage space than .JPEG and .PNG. Also, it can not be opened as an image without special software. To open a .PPM a conversion tool is used. Some C++ based ones include ImageMagick [1] and Netpbm [2]. The simplest method is to use an online conversion tool, one can be found here [3]. In our code, we save the render to "Pic.PPM".

2 Math Library

In this project, we created a maths library in C++ which uses the same notation and uses some functions from GLSL (Graphics Library Shading Language). There already exists a library called glm [4], but this was not used in this project because we wanted to test our C++ skills.

Therefore, we created an abstract base class for vectors. The vector class contains the overloading operator functions for $+, -, \times, \div$ and []. And "vec2" and "vec3" represent vectors of length 2 and 3, they are classes which deal with reading and modifying the elements in the vector. We also used the standard "cmath" library for maths operations on single elements. Figure 2 is the inheritance class diagram for vectors.

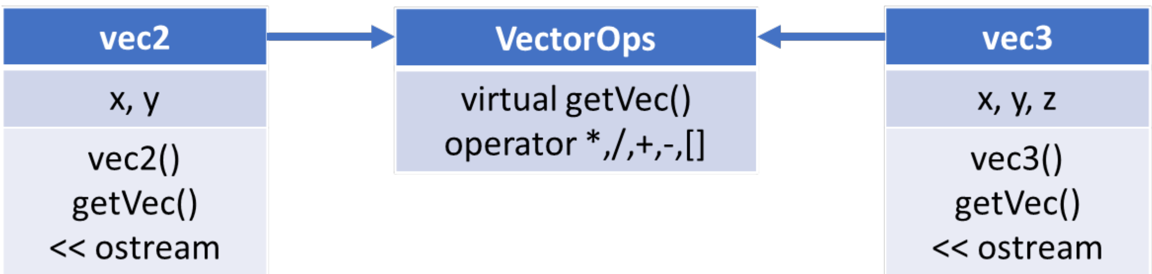


Figure 2: Inheritance class diagram for vectors. Arrows indicate inherited from

Additionally, a union and struct are used to enable different variable names for the same memory location. This is achieved by using the following code

```
//vec3 a = vec3(1.,1.,1.)
//Therefore, a[0], a.x, a.r are linked
//a[1], a.y, a.g are linked
//a[2], a.z, a.b are linked
union{
struct{ double data[3];};
struct{ double x,y,z;};
```

```
struct{ double r,g,b;};};
```

Templates were used to prevent duplication of code. For example, we have the template code below for calculating the absolute value of the elements of a vec2 and vec3.

```
template<typename T, int SIZE>
__CUDA_CALL__ //For GPU implementation
T abs(T a) {
    for(int i=0; i<SIZE; i++) {
        a[i] = abs(a[i]); }
    return a; }
__CUDA_CALL__
vec2 abs(vec2 a) {return abs<vec2,2>(a);}
__CUDA_CALL__
vec3 abs(vec3 a) {return abs<vec3,3>(a);}
```

This means when the specification for a function changes only one section of code needs to be modified. The maths functions mimic the OpenGL shading language specification, which can be found here [5]. However, we used the website [6] to find function requirements as it was quicker to find the function we wanted to code.

2.1 TDD, Test Driven Development

We used test-driven development to develop the maths library which uses a similar notation to GLSL. This involved writing the tests first and then writing the code to pass the tests. We used "cassert" and we developed a simple test framework to compare two vectors.

```
#define VectorCompL(a,b,c,d) VectorComp<a,b>(c,d,__LINE__)
template<typename T, int SIZE>
void VectorComp(T a, T b, int Line = 0) {
    for (int i = 0; i<SIZE; i++) {
        if (abs(a.data[i]-b.data[i]) > 0.000001) {
            //Failed
            std::cout << "\n a=" << a << "\n b=" << b << "\n i="
                            << i << "\n Line= "<< Line <<std::endl;
            assert(1==0); } //To Raise error
    }
}
//Example Tests
VectorComp<vec2,2>(vec2(1.,2.)+3.2,vec2(4.2,5.2));
VectorCompL(vec2,2,vec2(1.,2.)+3.2,vec2(4.2,5.2));
```

The difference between “VectorComp” and “VectorCompL” is “VectorCompL” outputs the line the error happened on. To run the tests we use the following commands.

```
#!/bin/bash
gcc Tests.cpp MathFunctions.cpp -lm -lstdc++ -o tests
./tests
```

All the tests can be found in the file “Tests.cpp” which is shown in appendix B.

3 SDF of Primitive Shapes

The signed distance function (SDF) of a shape is the smallest distance of a given point p from the boundary of the shape. If the point p is inside the boundary, a negative distance is returned. If the point p is outside the boundary, a positive distance is returned.

The derivations for SDF of primitive shapes were inspired by [7].

3.1 Circle and Sphere

These have the simplest signed distance function, the distance between the current position and the circle or sphere’s centre minus the radius. Thus we get code

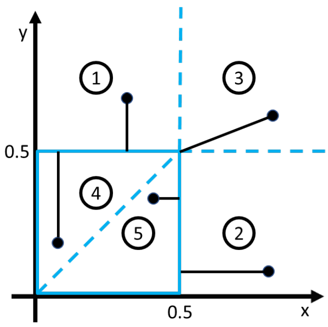
```
double sdCircle(vec2 p) { //Circle with radius 1. at origin
    return length(p) - 1.; }
double sdSphere(vec3 p) {
    return length(p) - 1.; }
```

To change the position and radius of the circle when we call the signed distance function we will transform the space. This is done by modifying the input parameter p. For example for a circle we have

```
//Circle of radius 1. with center at (0.5,0.6)
sdCircle(p-vec2(0.5,0.6));
//Circle of radius 0.5 with center at (0., 0.)
sdCircle(p*2.)/2.;
//Circle of radius r with center at (x, y)
sdCircle((p-vec2(x,y))*(1/r))*r;
```

3.2 Square and Cube

This derivation of the signed distance function of a box is inspired by [8].



(a) Regions for SDF proof.

```
IF in (1): |p.y|-0.5
IF in (2): |p.x|-0.5
IF in (3):
    ((|p.x|-0.5)^2+
    (|p.y|-0.5)^2)^0.5
IF in (4 or 5):
    0
```

(b) Naive outside SDF

```
IF in (1,2 or 3):
    0

IF in (4):
    p.y-0.5
IF in (5):
    p.x-0.5
```

(c) Naive inside SDF

Figure 3: Visuals aids for the SDF derivation for a square

We will derive the SDF for a square with points $(0.5, 0.5), (0.5, -0.5), (-0.5, -0.5)$ and $(-0.5, 0.5)$ by considering the positive quadrant. We take the absolute value of the spatial position p because this square is symmetric. We now use the regions shown in Figure 3a. Next we find two functions, the first function gives the closest distance to the square in regions $1, 2$ and 3, and is 0 in regions 4 and 5. We get the case statements algorithm shown in Figure 3b and this can be simplified to

$$\sqrt{\max(|p.x| - 0.5, 0)^2 + \max(|p.y| - 0.5, 0)^2} = \text{length}(\max(|p| - 0.5, 0)). \tag{1}$$

We have $|\cdot|$ denotes the absolute function and $\max(vec2(), 0)$ takes the element wise maximum.

The second function gives the negative distance to the closest side in regions 4 and 5 and 0 otherwise. This leads to the case statements algorithm shown in Figure 3c and in regions 4 and 5 this can be simplified to

$$\max(|p.x| - 0.5, |p.y| - 0.5). \tag{2}$$

When p is not in regions 4 and 5, equation (2) is positive, thus taking the minimum with 0 gives zero in regions $1, 2$ and 3. Thus for all regions we have

$$\min(\max(|p.x| - 0.5, |p.y| - 0.5), 0). \tag{3}$$

It is trivial to see the addition of these two functions leads to the signed distance function of a square. Thus for a square and unit cube centred at the origin of side length one we get signed distance functions.

```
double sdSquare(vec2 p) {
    //Square centered at origin with side length 1.
    vec2 d = abs(p) - vec2(0.5, 0.5);
```

```
    return length(max(d, vec2(0.0))) + min(max(d.x, d.y), 0.0); }
double sdBox(vec3 p) {
    //Cude centered at origin with side length 1.
    vec3 q = abs(p) - vec3(0.5, 0.5, 0.5);
    return length(max(q, 0.0)) + min(max(q.x, max(q.y, q.z)), 0.0);}
```

We notice the square and circle SDFs do not depend on their dimension. This means increasing the dimension and creating distance functions for hyperspheres and hypercubes is simple.

3.3 Triangle

The derivation for the SDF for the square-based pyramid and the triangular prism involves the same idea. In this paper, for the square-based pyramid and the triangular prism we use the SDFs stated at [7]. These SDFs are used in the rendering of the square-based Sierpinski pyramid.

4 Basic Operations

We now discuss how to create more complicated signed distance functions. When we want to render two different objects we take the minimum of the SDFs. This is known as taking the union of the objects.

When we want to render the intersection of two objects we take the maximum of the two SDFs. Additionally, when we want to cut out a shape from an object, we take the complement (Multiple SDF by -1) of the cutting object and then take the intersection with the main object.

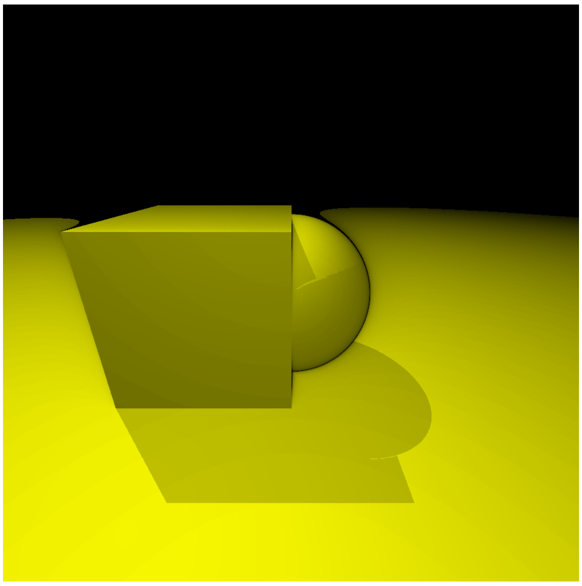
In this paper, we will use $\bigcap$ and $\bigcup$ to denote intersection and union respectively.

We can also display an infinite number of objects for a small increase in computational power. This is done by applying the mod function to the input p of the SDF.

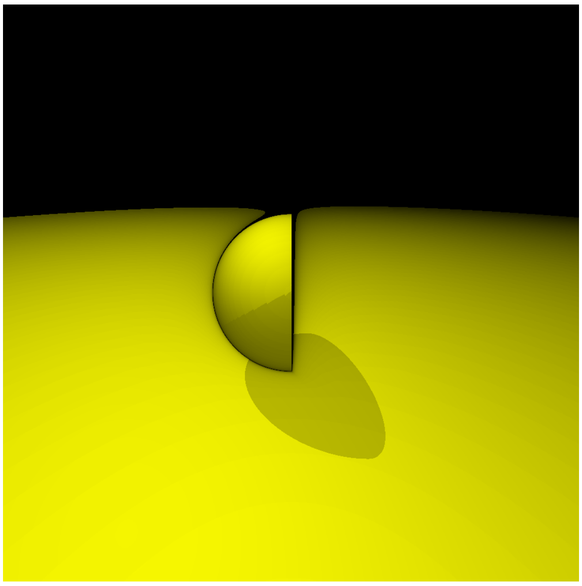
The following code demonstrates these operations with an SDF for a unit sphere at the origin and SDF with a cube with side length 2 centred at $(0, 1, 0)$. Also, the visual output of the SDF created by the code is shown in Figure 4.

```
double distFuncMod(vec3 pos) { //Use Mod to get infinite
    pos.x = mod(pos.x, 4.);     //        number of shapes
    pos.y = mod(pos.y, 4.);
    return min(sdSphere(pos-vec3(2.,2.,0.)), pos.z+1.); }
double distFuncUnion(vec3 p) { //min is Union
```

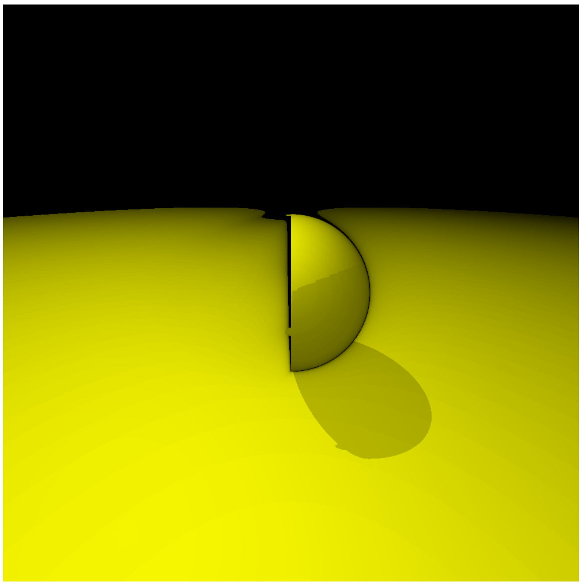
```
    return min(sdSphere(p), sdBox((p-vec3(0.,1.,0.))*0.5)*2.); }
double distFuncIntersection(vec3 p) { //max is Intersection
    return max(sdSphere(p), sdBox((p-vec3(0.,1.,0.))*0.5)*2.); }
double distFuncComplement(vec3 p) { //max negative is Complement
    return max(sdSphere(p), -sdBox((p-vec3(0.,1.,0.))*0.5)*2.); }
```



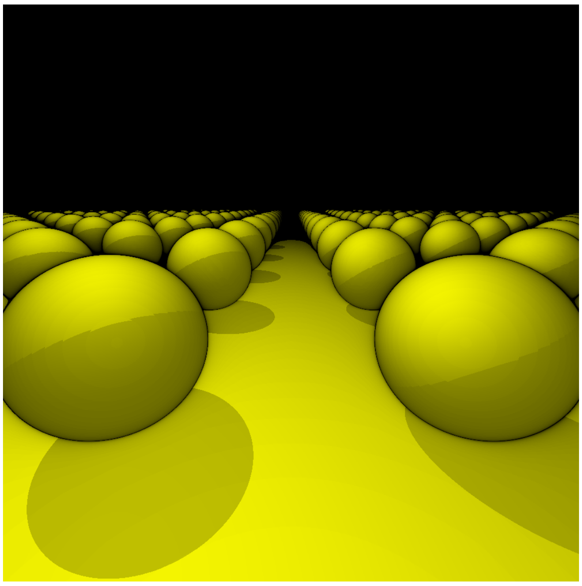
(a) Union of objects



(b) Intersection of objects



(c) Union of (sphere, complement of cube)



(d) Infinite objects by using mod

Figure 4: Output of render at resolution 1000×1000, includes union of surface $z = -0.5$

In Figure 4d the camera is at position $(-5, 0, 1)$ and is pointing towards the origin.

In Figure 4a, 4b and 4c the camera is at position $(-3.5, 0.0, 1.5)$ and is pointing towards the origin.

Additionally, it is worth noting the formula for calculating the modulus of x in the range (x_{low}, x_{high}) is

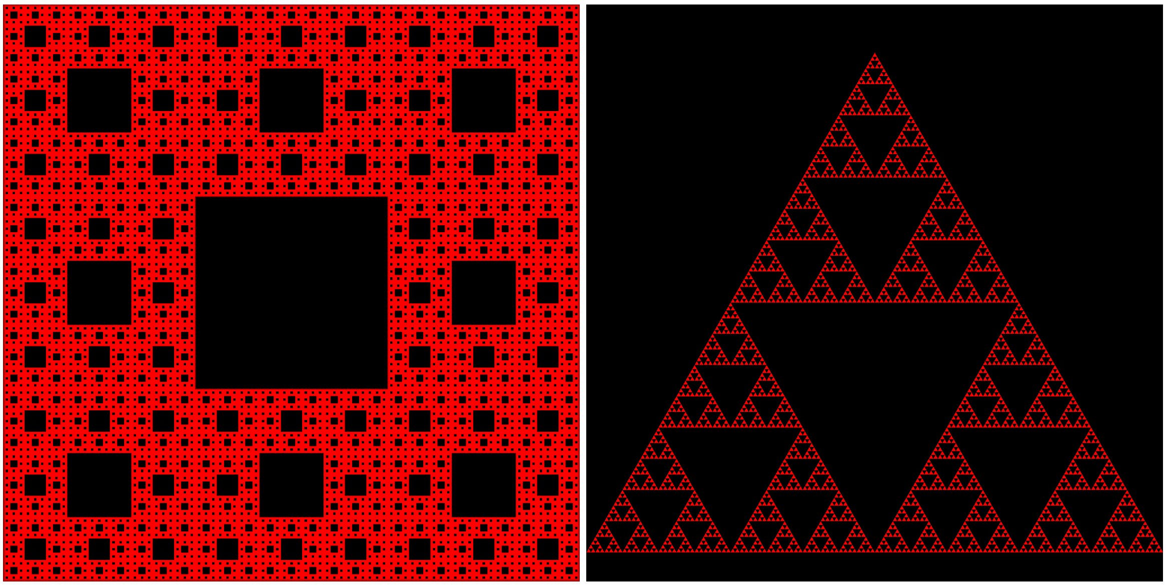
$$\text{mod}\ (x - x_{low}, x_{high} - x_{low}) + x_{low}. \tag{4}$$

This in code is

```
__CUDA_CALL__ //Allow to work on GPU
double mod(double x, double x_low, double x_high)
{return mod(x-x_low,x_high-x_low)+x_low;}
```

5 Sierpinski Polygons

To display these 2D fractals we will use the mod function to display infinite objects, this is shown in Figure 4d. Also, we will use the cutting method shown in Figure 4c. The distance function will be shown for these fractals but it is easier to explain with pictures.



(a) MengerSponge in 2D, Iterations: 6

(b) Sierpinski Triangle in 2D, Iterations: 8

Figure 5: Renders of Sierpinski Polygons

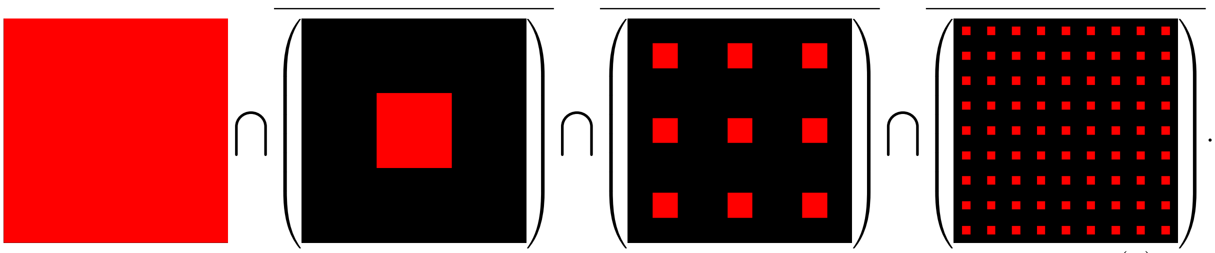
5.1 2D Menger Sponge

The distance function for the 2D Menger Sponge is as follows

```
1 __CUDA_CALL__
2 float my_map(vec2 p) { //MengerSponge
3     float sd = sdSquare(p); float its = 6.0; //Iterations
4     for (float it = 1.; it < its; it++) {
5         vec2 mod_d = mod(p*pow(3.,it)+1.5,3.)-1.5;
6         float d = sdSquare(mod_d)/pow(3.,it);
7         sd = max(sd, -d); }
8     return sd; }
```

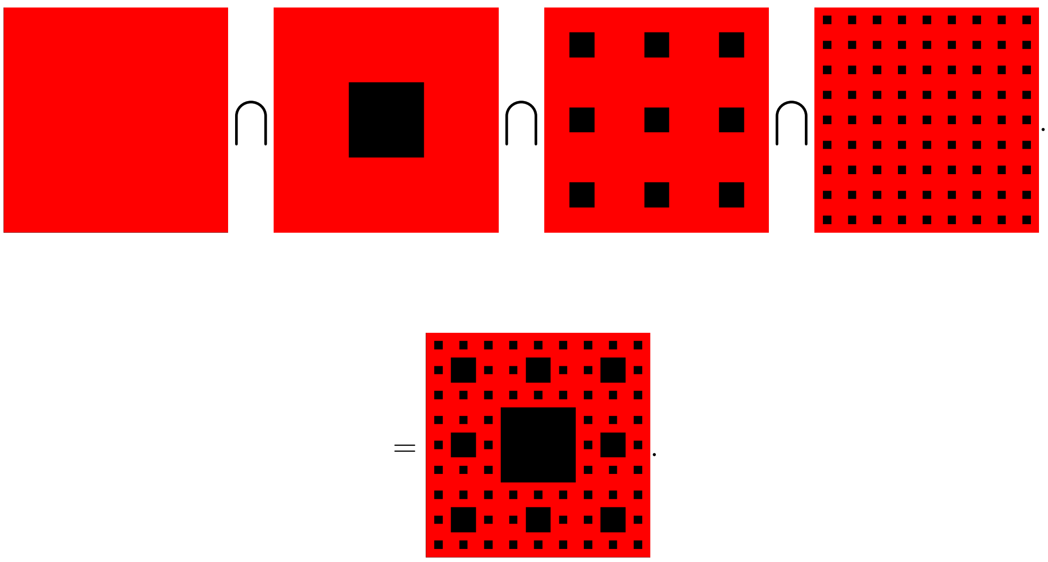
The output of this distance function is shown in Figure 5a.

Now we explain how the code constructs a Menger Sponge when there are 4 iterations. Line 3 creates a red square that fills the screen then each image shown corresponds to one iteration of the loop. The max function is the intersection and the over-bar represents taking the complement (negative) of the distance function.



(5)

This is equivalent to



(6)

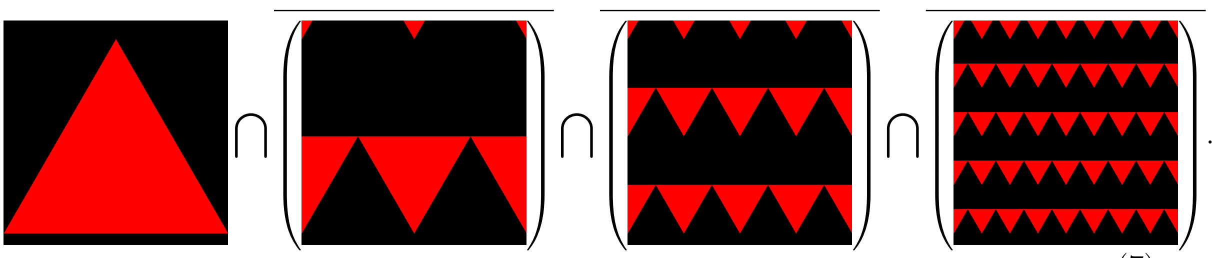
From this, it is trivial to see how it can be extended to higher iterations.

5.2 Sierpinski Triangle

This uses a similar technique to the one used to display the 2D Menger sponge. But this time the space that is modulated is different for each axis and the shape needs to be upside down. This code below gives the distance function for a Sierpinski triangle.

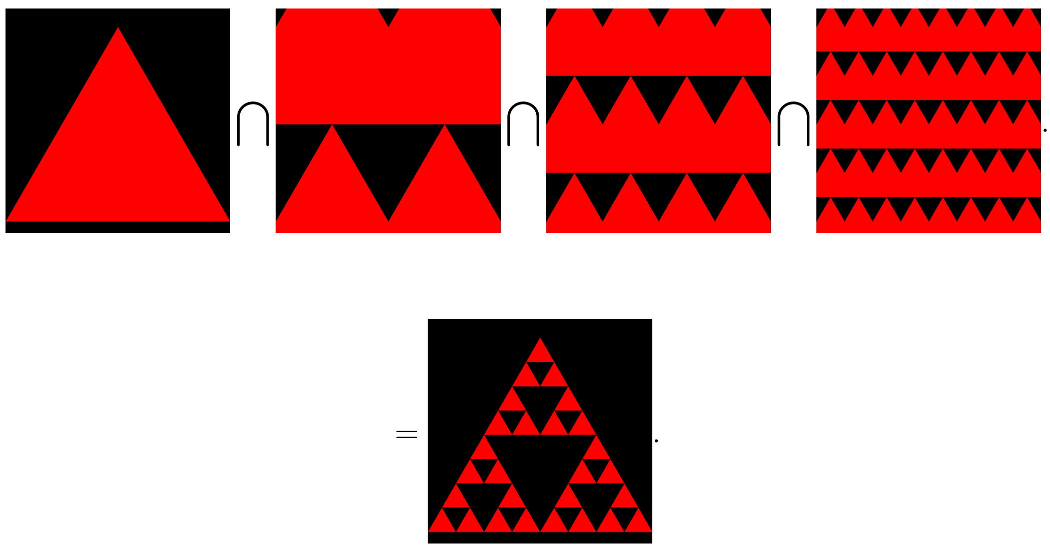
```
__CUDA_CALL__
float my_map(vec2 p) {
    float its = 8.0; const float k = sqrt(3.0);
    float modx = mod(p.x,-1.,1.); float mody = mod(p.y, 2.*k);
    float sd = sdTri(vec2(modx, mody));
    for (float i = 1.; i < its; ++i) {
        modx = mod(p.x*pow(2.,i),-1.,1.);
        mody = mod(-(p.y)*pow(2.,i)+k,2.*k);
        sd = max(sd, -sdTri(vec2(modx, mody))/pow(2.,i)); }
    return sd; }
```

This creates the render shown in Figure 5b. Now similar to the 2D Menger Sponge the best way to describe this algorithm is by visual aids.



(7)

This is equivalent to



(8)

6 RayMarching

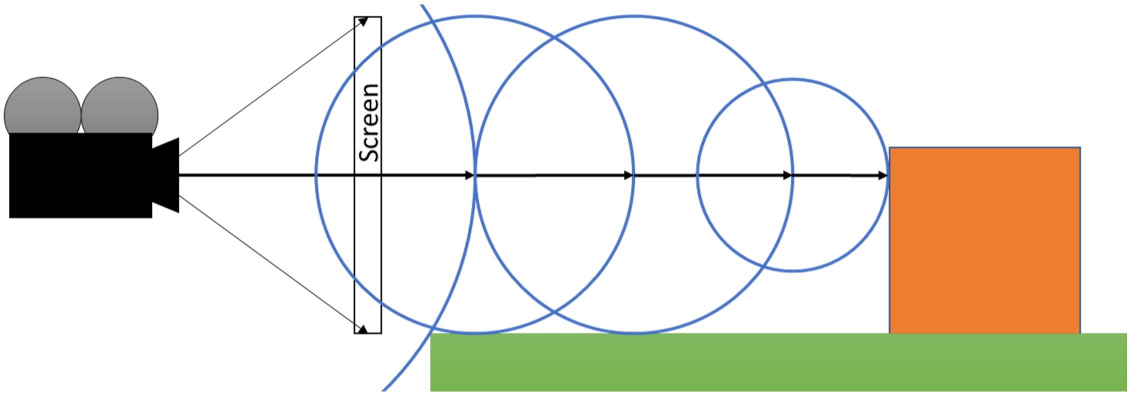


Figure 6: Visual demonstration of a ray being marched.

In Raymarching we calculate a ray direction for every pixel and march a ray in that direction. When marching we take a step in the ray direction. The step size is chosen to guarantee the new position is not inside the object in the scene. Thus the step size is chosen by calculating the SDF. When the SDF returns a small value it implies the ray has hit the object, then we can apply additional effects like shadows and return the colour of the pixel. A visual demonstration of a ray marching can be seen in Figure 6.

6.1 Maths Operations

Here we discuss the mathematical operations used when displaying a 3D scene using ray marching.

6.1.1 Normal

We need the unit normal to the surface of the objects in the scene. This is needed to create believable lighting effects. This is achieved by taking the gradient of the signed distance function for the scene. In this paper, we use finite differences to calculate the gradient. By denoting F as the scene distance function and $\epsilon \ll 1$ we get the equation

$$\frac{\nabla F(\vec{p})}{||\nabla F(\vec{p})||} = \frac{(F(\vec{p}+\epsilon\vec{e}_1), F(\vec{p}+\epsilon\vec{e}_2), F(\vec{p}+\epsilon\vec{e}_3)) - F(\vec{p})}{||(F(\vec{p}+\epsilon\vec{e}_1), F(\vec{p}+\epsilon\vec{e}_2), F(\vec{p}+\epsilon\vec{e}_3)) - F(\vec{p})||}, \tag{9}$$

for the unit normal at $\vec{p} = (x, y, z)$ and $\vec{e}_i$ representing a single entry vector which is 1 at i. This in code is

```
__CUDA_CALL__
vec3 GetNormal(vec3 p) {//Gets normal of the surface
    vec3 GradF = vec3(0.0,0.0,0.0); float F = distFunc(p);
    GradF.x = distFunc(p+epsilon*vec3(1.,0.,0.));
    GradF.y = distFunc(p+epsilon*vec3(0.,1.,0.));
    GradF.z = distFunc(p+epsilon*vec3(0.,0.,1.));
    GradF = GradF-F;
    return normalize(GradF); }
```

6.1.2 Calculating Width and Height Direction

We denote the unit width direction and height direction as $\vec{W}, \vec{H}$ respectively. These denote the direction right relative to the camera ($\vec{W}$) and the direction up to the camera ($\vec{H}$). We use $\vec{D}$ to denote the direction the camera is pointing.

We now calculate $\vec{W}$ and $\vec{H}$ from $\vec{D}$. We restrict the rotation of the camera so the camera can not roll which implies $\vec{W}$ has zero for its z-component. Now we project the camera direction ($\vec{D}$) onto the z-plane, then take the 2D cross product and then normalize.

```
__CUDA_CALL__
vec3 GetW(vec3 d) {
    d = normalize(d); vec3 W = vec3(d.y, -d.x, 0.);
    return normalize(W); }
```

Since $\vec{W}, \vec{H}$ and $\vec{D}$ are perpendicular it is trivial to calculate $\vec{H}$ using

$$\vec{H} = \vec{W} \times \vec{D}, \tag{10}$$

where $\times$ denotes the cross product.

It is important to note using this method the camera direction must have an x and y component. In other words, the camera direction ($\vec{D}$) can not point along the negative or positive z direction.

6.1.3 Calculating Ray Direction

We denote FOV as field of view from the left to right of the screen. Also, we will take the screen being a unit $\vec{D}$ away from the camera position and the screen has the same orientation as the camera. To calculate the ray direction (denoted by $\vec{RD}$) we can add the deviations from the distance traveled in the $\vec{H}$ and $\vec{W}$ directions. This gives us

$$\vec{RD} = \frac{\vec{D} + h_d\vec{H} + w_d\vec{W}}{||\vec{D} + h_d\vec{H} + w_d\vec{W}||}, \tag{11}$$

where from some trigonometry we get

$$w_d = 2\tan(\tfrac{FOV}{2})w/R_x, \tag{12}$$

$$h_d = 2\tan(\tfrac{FOV}{2})h/R_x, \tag{13}$$

with R_x denoting the resolution of the screen in the $\vec{W}$ direction. Also, w and h represent the pixel coordinate on the screen (the middle of the screen has pixel coordinate $(w,h) = (0,0)$). We divide (13) by R_x because if we divided by R_y we would get a stretched image. Implementing (12) and (13) into (11) we get

$$R\vec{D} = \frac{\vec{D} + 2\tan(\frac{FOV}{2})(w\vec{W} + h\vec{H})/R_x}{||\vec{D} + 2\tan(\frac{FOV}{2})(w\vec{W} + h\vec{H})/R_x||}, \tag{14}$$

which in code is

```
__CUDA_CALL__
vec3 Direction(vec3 D, vec3 H, vec3 W, vec2 wh, vec2 iRes) {
    float FOV = PI/2.;
    return normalize(D + 2.*tan(FOV/2.)*(wh.x*W+wh.y*H)/iRes.x); }
```

It is important to note that using this method FOV is restricted to the range $0 \leq FOV < \pi$.

6.2 Lighting Features

In Raymarching hard shadows (sharp shadows) are easy to implement. When the ray hits an object another ray is marched towards the light source. If the ray hits another object before the light source this means there must be a shadow.

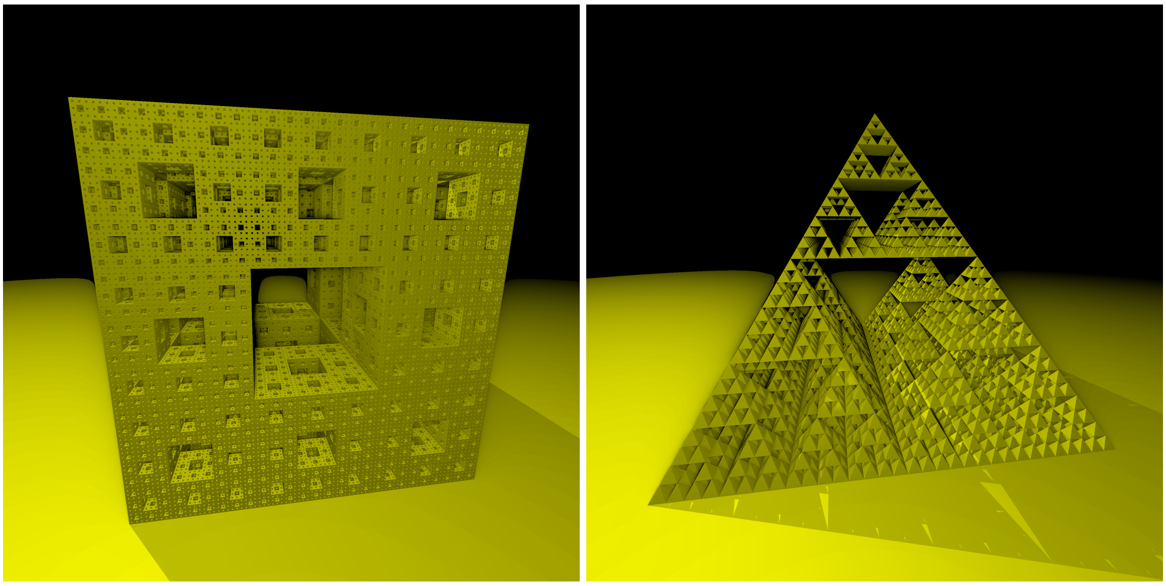
Additionally, reference [9] states a cheap effective way to implement ambient occlusion. It suggests darkening the image based on the number of steps taken to get there. Also, we have implemented specular highlighting via the Blinn-Phong shading model [10]. This states

$$k_e = \max(0, \vec{N} \cdot (\vec{L} + \vec{C}))^2, \tag{15}$$

where $\vec{N}$ is the normal to the surface, $\vec{L}$ is the direction to the light source and $\vec{C}$ is the direction to the camera. The k_e is multiplied by the colour to get a specular highlight effect. All these effects can be seen in the code file “RayMarch.hpp” listed in appendix G.

7 3D Sierpinski

We now discuss how to display Sierpinski fractals in 3D. Similar to the Sierpinski polygons we will use the mod function demonstrated in Figure 4d and the cutting method shown in Figure 4c.



(a) MengerSponge in 3D, Iterations: 5 (b) Sierpinski Pyramid in 3D, Iterations: 5

Figure 7: Renders of Sierpinski Fractals in 3D, resolution 2500x2500.

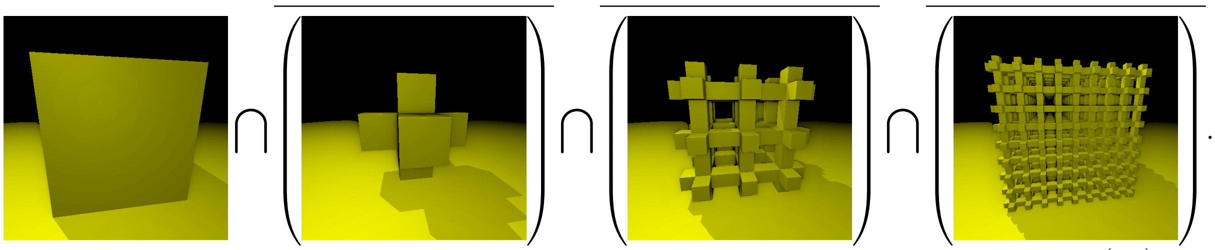
7.1 3D Menger Sponge

The distance function for the 3D Menger Sponge resting on a plane is as follows

```
__CUDA_CALL__
float distFunc(vec3 p) {
    float sd = sdBox(p,vec3(3.,3.,3.));
    for (float it = 0.; it < 5.; it++) {
        vec3 mod_d = mod(p*pow(3.,it)+3.0,6.)-3.0;
        float d = sdCross(mod_d)/pow(3.,it);
        sd = max(sd, -d);
    } return min(sd, p.z+3.); }
```

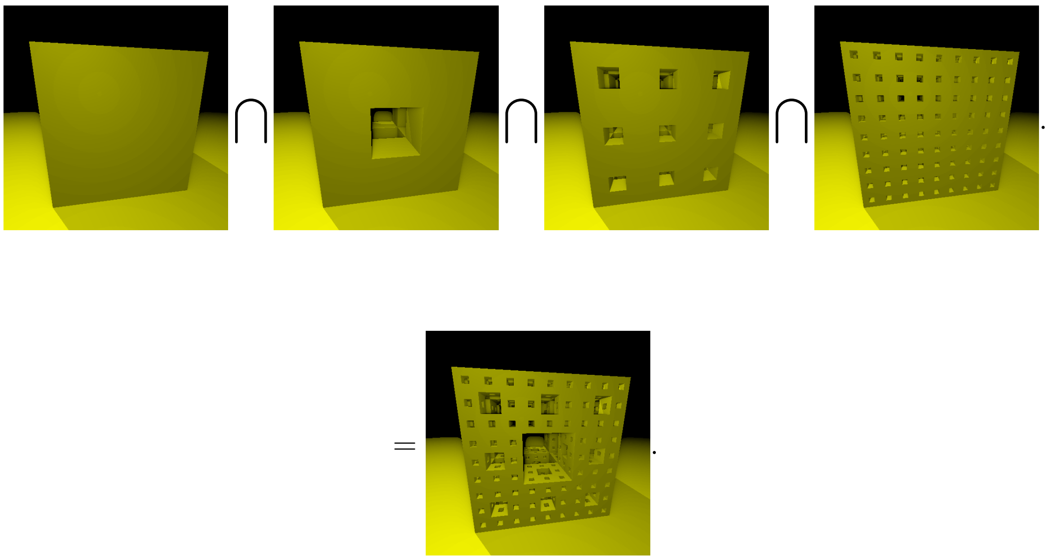
The output of this distance function is shown in Figure 7a.

Now we explain how the code constructs a 3D Menger Sponge when there are 3 iterations (when the break condition is $it < 3.$).



(16)

For the images in equation (16) and (17) we have taken the intersection with the cube, if this is not done the pattern will repeat infinitely and is difficult to visualise. Equation (16) is equivalent to



(17)

From this, it is trivial to see how it can be extended to higher iterations.

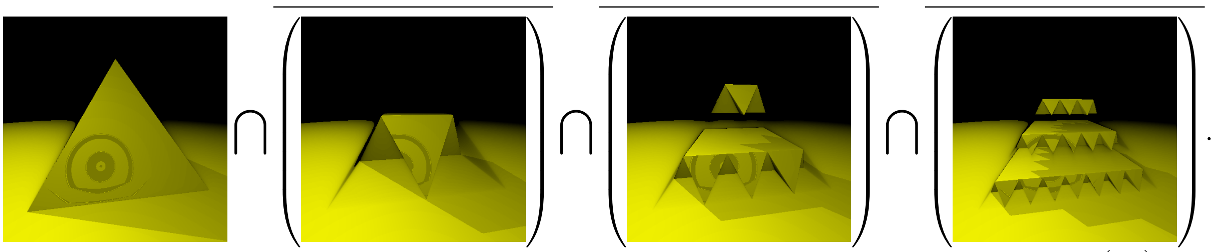
7.2 Sierpinski Square Based Pyramid

This uses a similar technique to the one used to display the 3D Menger Sponge. Just like the Sierpinski triangle, the space is modulated differently for each axis and the shape needs to be upside down. This code below gives the distance function for a Sierpinski square-based pyramid.

```
__CUDA_CALL__
float distFunc(vec3 p) {
    p = (p+vec3(0.,0.,2.))/7.;//So in correct position
    const float k = sqrt(3.0); float sd = sdPyramid(p);
    float modx = 0., mody = 0., modz = 0.;
```

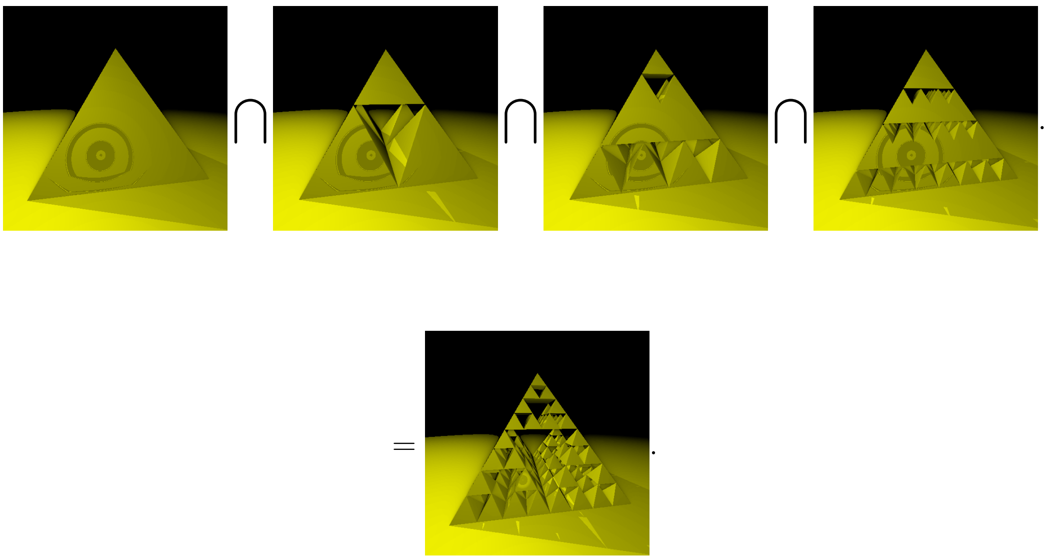
```
    for (float i = 0.; i < 5.; ++i) {
        modx = mod(p.x*pow(2.,i)+0.25, 0.5) - 0.25;
        mody = mod(p.y*pow(2.,i)+0.25, 0.5) - 0.25;
        modz = mod(-p.z*pow(2.,i)+k/4., k/2.);
        sd = max(sd,
            -sdTriCross(vec3(modx, mody, modz)*2.)/pow(2.,i+1.));
    } return min(sd,p.z)*7.; }
```

This creates the render shown in Figure 7b. Now similar to the 3D Menger Sponge the best way to describe this algorithm is by using pictures. Here each picture in equation (18) represents a different iteration of the loop.



(18)

Once again, for the images in Equations (18) and (19) we have taken the intersection with the square-based pyramid to make it easy to visualise. Equation (18) is equivalent to



(19)

8 Accelerated Computing

In this paper, we have developed code to render a fractal using 3 different executions. Technique 1 involved running the program sequentially on the CPU. Technique 2 involved running the program in parallel on the CPU using the C++ library "pthread". Technique 3 involved using a heterogeneous computing approach which used CUDA (Compute Unified Device Architecture) to run the rendering on the GPU.

To implement this two different main files were created: "main.cpp" shown in appendix C for techniques 1 and 2, and "main.cu" shown in appendix D for technique 3. When using CUDA the compiler needs to know if the function will be used on the CPU, GPU or both. This involves adding the phrase "__host__ __device__" before every function. However, this will cause an error when compiling the "main.cpp" file so to reduce duplication of files we implement a "__CUDA_CALL__" which is defined as nothing for compiling with normal c++ and "__host__ __device__" when compiling with CUDA. This is achieved by the following code

```
#ifdef __CUDACC__
#define __CUDA_CALL__ __host__ __device__
#else
#define __CUDA_CALL__
#endif
```

Below is the code to compile the programs and a few examples of how the executable should be used

```
# !/bin/bash
gcc main.cpp MathFunctions.cpp -lm -lstdc++ -lpthread -o main
nvcc main.cu -o cuda_main #Need nvcc installed
#Technique 1 Sequential CPU
./main 1280 720 0
#Technique 2 Parallel CPU
./main 1280 720 1
#Technique 3 GPU
./cuda_main 1280 720
#1st Option = x Resolution
#2nd Option = y Resolution
```

Additionally, to change the fractal being rendered we change the "#include" in "main.cpp" and "main.cu" to the file which contains the distance function we want.

It is important to note that CUDA only works on Nvidia hardware. However, since the syntax is very similar to c++ implementing the code on the GPU is not time-consuming.

8.1 Speed Tests

Now we compare the speeds of the different techniques. The times are in seconds and recorded using our "StopWatch" class. Also, the times include the time taken to save the file as ".PPM".

2D	Sierpinski Triangle			Sierpinski Square		
Resolution	1 Core	8 Cores	GPU	1 Core	8 Cores	GPU
100×100	0.0611202	0.560923	2.53015	0.0710787	0.507549	2.01654
200×200	0.0797239	1.94296	0.807451	0.163357	1.6764	0.67062
400×400	0.298176	8.53181	0.916207	0.58642	6.45865	1.08919
800×800	0.993268	25.6734	1.09634	2.33119	24.8596	1.14159
1600×1600	-	-	2.51188	-	-	2.52175
3200×3200	-	-	9.32252	-	-	7.181

Table 1: Table to test different speeds for rendering 2D Sierpinski Fractals

3D	Sierpinski Pyramid			Sierpinski Cube		
Resolution	1 Core	8 Cores	GPU	1 Core	8 Cores	GPU
100×100	0.228884	0.321559	1.61958	0.678229	0.378749	1.6354
200×200	0.882771	1.20964	0.372528	2.60159	1.37683	0.358463
400×400	3.84839	4.57557	0.486268	10.4276	5.03283	0.467922
800×800	-	18.3025	0.892214	-	20.2503	0.78364
1600×1600	-	788	2.43082	-	-	1.89158
3200×3200	-	788	7.84955	-	-	6.1844

Table 2: Table to test different speeds for rendering 3D Sierpinski Fractals

From Table 1 and Table 2 we infer that using the GPU for rendering our scenes is the superior method. Only when rendering the 2D fractals for small resolutions (200×200 and 400×400) was the 1 Core method faster, this is because the calculations in 2D require fewer computations and since the resolution is small there is not a sufficient amount of calculations to warrant the overhead of using the GPU.

Also, we note that the GPU for resolution 100×100 takes approximately double the amount of time than for the resolution 400×400. This is because for every different fractal the code was recompiled and there is an additional overhead for the first time a CUDA application is run.

Finally, we see the 8 Core method doesn't provide much benefit over the 1 Core method. This strongly suggests the "pthreads" was not implemented correctly. However, it did show promise when rendering the 3D Menger Sponge compared to the 1 Core method.

9 Conclusion

In this paper, we used signed distance functions to render Sierpinski fractals in 2D and 3D with ray marching. Firstly, we used test-driven development to implement our maths functions. Then we combined basic manipulation of signed distance functions to derive the SDFs for Sierpinski fractals. We derived the maths needed to use ray-marching, this involved calculating ray directions and believable lighting effects.

From these results, we have found an efficient way of rendering fractals. For example, to display a Sierpinski Pyramid with 8 iterations by rasterisation it requires over 1.5 million triangles to be drawn but for ray-marching increasing iterations by one only adds an extra iteration to the loop in the SDF. Also, it is clear to see using the GPU is superior to using the CPU. This makes sense as calculating the colour for multiple pixels is an embarrassingly parallel problem. Additionally, all the C++ code shown in this paper was written by myself for this project. Since "MathObjects.hpp" and "MathFunctions.cpp" makes calculations in 2D and 3D trivial. It is also used in my special topic for the simulation of multiple water molecules.

Other fractals that would be easy to implement would be Newton fractals and the Mandelbrot set. A complex number class would need to be created, this would lead to a class similar in structure to "vec2".

Further research and development would be focused on speeding up the maths calculations in "MathObjects.hpp" and "MathFunctions.cpp". A performance improvement is possible in our C++ code, it takes 0.892214 seconds to render a Sierpinski Pyramid at resolution 800×800 but on shadertoy it can render it at 30 frames per second. This is because for some definitions of the operator functions it is passing the vector through by value, this involves copying the data, instead, it should pass by reference to the pointer. And on shadertoy it uses floats while in our code we use doubles. Also, an implementation of vector swizzling would be useful. Finally, developing a header file to translate a ".PPM" image file to a ".PNG" would be interesting.

References

[1] *ImageMagick*. Last accessed 06.07.2022. URL: https://imagemagick.org/.

[2] *PPM Viewer*. Last accessed 06.07.2022. URL: http://netpbm.sourceforge.net/.

[3] Catherine E. Welsh. *PPM Viewer*. Last accessed 06.07.2022. URL: https://www.cs.rhodes.edu/welshc/COMP141_F16/ppmReader.html.

[4] Christophe Groovounet. *OpenGL Mathematics (GLM)*. Last accessed 06.07.2022. URL: https://github.com/g-truc/glm.

[5] Randi Rost John Kessenich Dave Baldwin. "The OpenGL Shading Language 4.5". In: 7 (2017). URL: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf.

[6] *docs.GL*. Last accessed 06.07.2022. URL: https://docs.gl/sl4/mod.

[7] Inigo Quilez. *3D SDF*. Last accessed 06.07.2022. URL: https://iquilezles.org/articles/distfunctions/.

[8] Inigo Quilez. *Youtube: The SDF of a Box*. Last accessed 06.07.2022. URL: https://www.youtube.com/watch?v=62-pRVZuS5c.

[9] Tom Beddard. *4D Quaternion Julia Set Ray Tracer*. Last accessed 06.07.2022. 2009. URL: http://2008.sub.blue/blog/2009/9/20/quaternion_julia.html.

[10] Steven M.Lavalle. *Blinn-Phong shading*. Last accessed 06.07.2022. URL: http://vr.cs.uiuc.edu/node198.html.

A Code File: "README.txt"

```
To Compile the c++ version run
gcc MathFunctions.cpp main.cpp -lm -lpthread -lstdc++ -o main

To Compile the CUDA version run
nvcc main.cu -o cuda_main

```

```
To Compile the tests we run
gcc MathFunctions.cpp Tests.cpp -lm -lpthread -lstdc++ -o tests

Note the CUDA version requires a NVIDIA GPU and it needs
to be installed.

To change the fractal being rendered in "main.cpp" (C++)
Change the include on line 13.

To change the fractal being rendered in "main.cu" (CUDA)
Change the include on line 9

Possible fractal includes are
#include "MengerSponge2D.hpp"
#include "SierpinskiTriangle.hpp"
#include "MengerSponge3D.hpp"
#include "SierpinskiPyramid.hpp"


When we define a float we mean double.
This is changed on line 10 of "MathObjects.hpp" by
#define float double
```

B Code File: “Tests.cpp”

```cpp
#include<iostream>
#include<cassert>
#include"MathObjects.hpp"

#define VectorCompL(a,b,c,d) VectorComp<a,b>(c,d,__LINE__)

using namespace std;






template<typename T, int SIZE>
void VectorComp(T a, T b, int Line = 0)
{
    for (int i = 0; i<SIZE; i++)
    {
        if (abs(a.data[i]-b.data[i]) > 0.000001)
```

```
        {
            //Failed
            std::cout << "\n a=" << a
                      << "\n b=" << b
                      << "\n i=" << i
                      << "\n Line= "<< Line <<std::endl;
            assert(1==0); //To Raise error
        }

    }
}

// ***********************************
// ***********************************
// **                               **
// ** Vec2 Standard Operations Tests **
// **                               **
// ***********************************
// ***********************************
void Test_vec2_adds_with_other_vec2()
{
  vec2 a = vec2(1.,2.);
  vec2 b = vec2(1.,4.);
  vec2 c = vec2(2.,6.);
    VectorComp<vec2,2>(a+b,c);
}

void Test_vec2_adds_with_other_scalars()
{
  vec2 a = vec2(1.,3.);
  double b = 3.2;
  vec2 c = vec2(4.2,6.2);
    VectorComp<vec2,2>(a+b,c);
    VectorComp<vec2,2>(b+a,c);
}

void Test_vec2_subtracts_with_other_scalars_and_vec2()
{
    //Scalar
    vec2 a = vec2(1.,2.);
  double b = 3.2;
    VectorCompL(vec2,2,a-b,vec2(-2.2,-1.2));
    VectorCompL(vec2,2,b-a,vec2(2.2,1.2));
```

```

    //vec2
    vec2 c = vec2(0.,-3.);
    VectorCompL(vec2,2,-a,vec2(-1.,-2.));
    VectorCompL(vec2,2,a-c,vec2(1.,5.));

    //From Function
    VectorCompL(vec2, 2,
                abs(vec2(0.5,0.5))-vec2(0.5,0.5),vec2(0.0,0.0));
    VectorCompL(vec2, 2,
                vec2(0.5,0.5)-abs(vec2(0.5,0.5)),vec2(0.0,0.0));
}

void Test_vec2_divides_with_other_scalar()
{
    vec2 a = vec2(1.,2.);
  double b = 2.0;
  vec2 c = vec2(0.5,1.0);
    VectorComp<vec2,2>(a/b,c);
    VectorComp<vec2,2>(a/2.0,c);
}

void Run_vec2_tests()
{
    Test_vec2_adds_with_other_vec2();
    Test_vec2_adds_with_other_scalars();
    Test_vec2_subtracts_with_other_scalars_and_vec2();
    Test_vec2_divides_with_other_scalar();

    std::cout << "vec2 tests passed\n";
}

// ***********************************
// ***********************************
// **                               **
// ** Vec3 Standard Operations Tests **
// **                               **
// ***********************************
// ***********************************
void Test_vec3_adds_with_other_vec3()
{
  vec3 a = vec3(1.,2.,3.);
  vec3 b = vec3(1.,4.,-2.);
```

```
  vec3 c = vec3(2.,6.,1.);
    VectorComp<vec3,3>(a+b,c);
}

void Test_vec3_adds_with_other_scalars()
{
  vec3 a = vec3(1.,2.,3.);
  double b = 3.2;
  vec3 c = vec3(4.2,5.2,6.2);
    VectorComp<vec3,3>(a+b,c);
    VectorComp<vec3,3>(b+a,c);
    VectorComp<vec3,3>(vec3(1.,2.,3.)+3.2,c);
    VectorComp<vec3,3>(3.2+vec3(1.,2.,3.),c);
}

void Test_vec3_subtracts_with_other_scalars_and_vec3()
{
    //Scalar
    vec3 a = vec3(1.,2.,3.);
  double b = 3.2;
    VectorCompL(vec3,3,a-b,vec3(-2.2,-1.2,-0.2));
    VectorCompL(vec3,3,b-a,vec3(2.2,1.2,0.2));

    //vec3
    vec3 c = vec3(0.,-3.,2.);
    VectorCompL(vec3,3,-a,vec3(-1.,-2.,-3.));
    VectorCompL(vec3,3,a-c,vec3(1.,5.,1.));
}

void Test_vec3_mults_elementwise_with_other_vec3()
{
  vec3 a = vec3(1.,2.,3.);
  vec3 b = vec3(1.,4.,-2.5);
  vec3 c = vec3(1.,8.,-7.5);
    VectorComp<vec3,3>(a*b,c);
}

void Test_vec3_mults_with_other_scalars()
{
  vec3 a = vec3(1.,2.,3.);
  double b = 4.2;
  vec3 c = vec3(4.2,8.4,12.6);
    VectorComp<vec3,3>(a*b,c);
```

```
    VectorComp<vec3,3>(b*a,c);
    VectorComp<vec3,3>(vec3(1.,2.,3.)*4.2,c);
    VectorComp<vec3,3>(4.2*vec3(1.,2.,3.),c);
    VectorComp<vec3,3>(b*vec3(1.,2.,3.),c);
    VectorComp<vec3,3>(4.2*a,c);
}

void Test_vec3_divides_elementwise_with_other_vec3()
{
    vec3 a = vec3(1.,2.,3.);
  vec3 b = vec3(1.,4.,-2.5);
  vec3 c = vec3(1.,0.5,-1.2);
    VectorComp<vec3,3>(a/b,c);
}

void Test_vec3_divides_with_other_scalar()
{
    vec3 a = vec3(1.,2.,-3.);
  double b = 2.0;
  vec3 c = vec3(0.5,1.0,-1.5);
    VectorComp<vec3,3>(a/b,c);
}

void Test_vec3_defines_correctly_with_different_inputs()
{

    VectorComp<vec3,3>(vec3(vec2(1.,2.),3.),vec3(1.,2.,3.));
    VectorComp<vec3,3>(vec3(1.,vec2(2.,3.)),vec3(1.,2.,3.));
    VectorComp<vec3,3>(vec3(),vec3(0.0,0.0,0.0));
    VectorComp<vec3,3>(vec3(5.0), vec3(5.0,5.0,5.0));

}

void Test_vec3_reads_data_correctly_with_subscript()
{
    vec3 a = vec3(1., 2., -3.);
    assert(a[0]==1.);
    assert(a[1]==2.);
    assert(a[2]==a.data[2]);
}

void Test_vec3_stores_data_correctly_with_subscript()
{
```

```
    vec3 a = vec3(1.0,2.0,3.);
    a[0] = 2.0;
    assert(a[0]==2.0);
    a[2] = a[2]+2.;
    assert(a[2]=5.);
}



void Run_vec3_tests()
  {
    Test_vec3_adds_with_other_vec3();
    Test_vec3_adds_with_other_scalars();
        Test_vec3_subtracts_with_other_scalars_and_vec3();
        Test_vec3_mults_elementwise_with_other_vec3();
        Test_vec3_mults_with_other_scalars();
        Test_vec3_divides_elementwise_with_other_vec3();
        Test_vec3_divides_with_other_scalar();
        Test_vec3_defines_correctly_with_different_inputs();
        Test_vec3_reads_data_correctly_with_subscript();
        Test_vec3_stores_data_correctly_with_subscript();
    std::cout << "vec3 tests passed" << "\n";
  }


// ******************************
// ******************************
// **                          **
// ** Swizzling Operations Tests **
// **                          **
// ******************************
// ******************************
void swizzling_outputs_correct_val_for_one_swizzle()
{
    vec3 a1 = vec3(1.0,2.0,3.0);
    vec2 a2 = vec2(1.0,2.0);
    assert(a1.x == 1.0);
    assert(a1.y == 2.0);
    assert(a1.z == 3.0);
    assert(a2.x == 1.0);
    assert(a2.y == 2.0);

}
```

```
void swizzling_allows_setting_value()
{
    vec3 a1 = vec3(1.0,2.0,3.0);
    vec2 a2 = vec2(1.0,2.0);
    a1.x = 2.0;
    VectorCompL(vec3,3,a1,vec3(2.0,2.0,3.0));
    a2.y = 0.0;
    VectorCompL(vec2, 2, a2, vec2(1.0,0.0));

}

void swizzling_works_for_input_of_function_for_one_swizzle()
{

    assert((min(vec2(1.2,3.2).x, vec2(1.2,3.2).y)-1.2)<0.00001);

}

void Run_swizzling_tests()
{
    swizzling_outputs_correct_val_for_one_swizzle();
    swizzling_allows_setting_value();
    swizzling_works_for_input_of_function_for_one_swizzle();
    std::cout << "swizzling tests passed" << "\n";
}

// ********************************
// ********************************
// **                            **
// ** Mathematics Function Tests **
// **                            **
// ********************************
// ********************************

void Test_abs_outputs_correct_value()
{
    vec3 a = vec3(-1.0,2.0,-3.0);
    VectorComp<vec3,3>(abs(a), vec3(1.,2.,3.));
    vec2 b = vec2(-1.,-2.0);
    VectorComp<vec2,2>(abs(b),vec2(1.,2.0));
}
```

```
void Test_min_and_max_outputs_correct_value_for_vecs()
{
    vec3 a = vec3(5.0,-1.0,2.0);
    vec3 b = vec3(4.0,1.0,-3.0);
    vec3 c1 = max(a,b);
    vec3 c2 = min(a,b);
    VectorComp<vec3,3>(c1, vec3(5.,1.,2.));
    VectorComp<vec3,3>(c2, vec3(4.,-1.,-3));

}

void Test_clamp_outputs_correct_value()
{
    //double case
    assert(clamp(0.5,-1.,1.) == 0.5);
    assert(clamp(-1.5,-1.,1.) == -1.);
    assert(clamp(1.5,-1.,1.) == 1.);

    //vec2 case
    VectorCompL(vec2,2,
                clamp(vec2(0.5,-1.5), -1.,1.), vec2(0.5,-1.));

    //vec3 case
    VectorCompL(vec3,3,
               clamp(vec3(0.5,-1.5,1.5), -1.,1.), vec3(0.5,-1.,1.));
}

void Test_mod_outputs_correct_value()
{
    assert(abs(mod(342.4,7.2)-4.) < 0.00001);
    VectorCompL(vec2,2, mod(vec2(342.4,86.), 7.2), vec2(4.,6.8));
    assert(abs(mod(4.5,1.,3.)-2.5)<0.00001);
}


void Test_smoothstep_outputs_correct_value()
{
    double a1 = -1.;
    vec2 a2 = vec2(-1.,1.5);
    vec3 a3 = vec3(-1.,1.5,2.5);
    assert(smoothstep(1.,2.,a1)==0.);
    VectorCompL(vec2,2,smoothstep(1.,2.,a2),vec2(0.,0.5));
    VectorCompL(vec3,3,smoothstep(1.,2.,a3),vec3(0.,0.5,1.));
```

```

}

void Run_Math_Function_Tests()
{
    Test_abs_outputs_correct_value();
    Test_min_and_max_outputs_correct_value_for_vecs();
    Test_clamp_outputs_correct_value();
    Test_mod_outputs_correct_value();
    Test_smoothstep_outputs_correct_value();
    std::cout << "Math Function Tests Passed" << "\n";
}


// ***************************
// ***************************
// **                       **
// ** Vector Function Tests **
// **                       **
// ***************************
// ***************************

//Cross
void Test_cross_calculates_cross_product_correctly()
{
    vec3 a = vec3(1.0,2.0,3.0);
    vec3 b = vec3(5.0,3.0,6.0);
    vec3 c = vec3(3.,9.0,-7.0);
    VectorCompL(vec3,3,cross(a,b),c);
}

//Length
void Test_length_calculates_correct_value_on_vec2_vec3()
{
    vec3 a1 = vec3(1.0,2.0,3.0);
    vec2 a2 = vec2(2.0,4.0);
    assert(abs(length(a1)-3.74166)<0.0001);
    assert(abs(length(a2)-4.47214)<0.0001);
    //Tests Dot and Sum as well
}

//Dot

```

```
//Normalize
void Test_normalize_outputs_correct_normalised_vector()
{
    assert(normalize(-5.5)==-1.);
    VectorCompL(vec2,2, normalize(vec2(3.,4.)), vec2(0.6,0.8));
    VectorCompL(vec3,3,
                normalize(vec3(0.,4.,3.)), vec3(0.,0.8,0.6));
}

//Distance

void Run_Vector_Function_Tests()
{
    Test_cross_calculates_cross_product_correctly();
    Test_length_calculates_correct_value_on_vec2_vec3();
    Test_normalize_outputs_correct_normalised_vector();
    std::cout << "Vector Function Tests Passed" << "\n";
}

int main()
{
    Run_vec2_tests();
  Run_vec3_tests();
    Run_swizzling_tests();
    Run_Math_Function_Tests();
    Run_Vector_Function_Tests();

  std::cout << "All tests passed\n";
}
```

C Code File: “main.cpp”

```
#include <iostream>
#include <fstream>
#include <vector>
#include <future>

#include "MathObjects.hpp"
using namespace std;



//Fractal to Display
```

```
//#include "SierpinskiPyramid.hpp"
#include "MengerSponge3D.hpp"

vec3 mainImage(vec2 fragCoord, vec2 iRes)
{
  /*
  Calculates the color for each pixel

  Inputs
  ------
  fragCoord:
    Gives the pixel position in the image, with (0.,0.) being
  bottom left, and (iResolution.x, iResolution.y),being top right.

  Outputs
  -------
  fragColor:
    returns the rgb of the current pixel. (range 0. to 1.)
  */
  vec3 fragColor = GetColor(fragCoord, iRes);

  return fragColor;
}


//Sequential
void main_func_sequential(vec2 iResolution) {

    StopWatch My_Watch;
    My_Watch.Start();
    // Image

    const int image_width = (int)(iResolution.x);
    const int image_height = (int)(iResolution.y);

    // Render
  ofstream MyFile("Pic.ppm");
  std::cerr << "Res: " << image_width << "x"
        << image_height << "\n";
    MyFile << "P3\n" << image_width << ' '
        << image_height << "\n255\n";

  //Each loop is independent (loop over every pixel)
```

```
    for (int y_pos = 0; y_pos < image_height; ++y_pos) {
        std::cerr << "\rScanlines remaining: "
            << image_height - y_pos << ' ' << std::flush;
        for (int x_pos = 0; x_pos < image_width; ++x_pos) {

      vec2 fragCoord = vec2((double)(x_pos),
                             (double)(image_height - y_pos - 1));
      vec3 fragColor256 = clamp(
                   mainImage(fragCoord, iResolution),0.,1.)*255.999;


      //Save to file
            MyFile << (int)(fragColor256.x)
            << ' ' << (int)(fragColor256.y)
      << ' ' << (int)(fragColor256.z) << '\n';
        }
    }
    std::cerr << "\nDone.\n";
  MyFile.close();
    My_Watch.Stop();
    std::cout << "Time: " << My_Watch.Time() << "\n";
}


//Asynchronous
vec3 GetfragColor256(vec2 fragCoord, vec2 iRes)
{
    return clamp(mainImage(fragCoord, iRes),0.,1.)*255.999;
}

void main_func_asynchronous(vec2 iResolution) {

    StopWatch My_Watch;
    My_Watch.Start();
    // Image

    const int image_width = (int)(iResolution.x);
    const int image_height = (int)(iResolution.y);

    // Render
  ofstream MyFile("Pic.ppm");
  std::cerr << "Res: " << image_width << "x"
        << image_height << "\n";
```

```
    MyFile << "P3\n" << image_width << ' '
        << image_height << "\n255\n";

  //Each loop is independent (loop over every pixel)

    for (int y_pos = 0; y_pos < image_height; ++y_pos) {
        std::cerr << "\rScanlines remaining: "
            << image_height - y_pos << ' ' << std::flush;
        std::vector<std::future<vec3>> futures;
        for (int x_pos = 0; x_pos < image_width; ++x_pos) {

      vec2 fragCoord = vec2((double)(x_pos),
                            (double)(image_height - y_pos - 1));
      futures.push_back (std::async(GetfragColor256,
                                    fragCoord, iResolution));

        }

        for(auto &e : futures)
        {
      vec3 fragColor256 = e.get();
      //Save to file
            MyFile << (int)(fragColor256.x)
            << ' ' << (int)(fragColor256.y)
      << ' ' << (int)(fragColor256.z) << '\n';
        }
    }
    std::cerr << "\nDone Threads.\n";
    int count = 0;

  MyFile.close();
    My_Watch.Stop();
    std::cout << "Time: " << My_Watch.Time() << "\n";
}

int main(int argc, char *argv[])
{
    //argv[1] iResolution.x
    //argv[2] iResolution.y
    //argv[3] Run in Asynchronous mode

    vec2 iResolution;
    bool Async;
```

```

    if (argc == 3) {
        iResolution = vec2((double)(atoi(argv[1])),
                           (double)(atoi(argv[2])));
        Async = true;
    } else if (argc == 4) {
        iResolution = vec2((double)(atoi(argv[1])),
                           (double)(atoi(argv[2])));
        Async = (bool)(atoi(argv[3]));
    } else {
        iResolution = vec2(600., 300.);
        Async = true;
    }

    if (Async){
        main_func_asynchronous(iResolution);
    } else {
        main_func_sequential(iResolution);
    }
    return 0;
}
```

D Code File: "main.cu"

```
#include <iostream>
#include <fstream>

#include "MathFunctions.cpp" //This includes MathsObjects.hpp
using namespace std;


//Fractal to Display
#include "MengerSponge3D.hpp"

__device__
vec3 mainImage(vec2 fragCoord, vec2 iRes)
{
  /*
  Calculates the color for each pixel

  Inputs
  ------
  fragCoord:
```

```
    Gives the pixel position in the image, with (0.,0.) being
  bottom left, and (iResolution.x, iResolution.y),being top right.

  Outputs
  -------
  fragColor:
    returns the rgb of the current pixel. (range 0. to 1.)
  */
  vec3 fragColor = GetColor(fragCoord, iRes);

  return fragColor;
}

__global__
void sequence_kernal(double *r, double *g,
                     double *b, int N, vec2 iResolution)
{
  // Unique id of thread
  int index = blockIdx.x*blockDim.x+threadIdx.x;
  int stride = blockDim.x*gridDim.x;//Get number of threads in block
  for(int i = index;i<N;i += stride)
  {
      int x_pos = i%((int)(iResolution.x));
      int y_pos = (int)(iResolution.y)-i/((int)(iResolution.x))-1;
      vec2 fragCoord = vec2((double)(x_pos), (double)(y_pos));
      vec3 fragColor = mainImage(fragCoord, iResolution);

      //Update Memory
      r[i] = fragColor.x;
      g[i] = fragColor.y;
      b[i] = fragColor.z;

  }
}


int main(int argc, char *argv[])
{

    //argv[1] iResolution.x
    //argv[2] iResolution.y

    vec2 iResolution;
```

```

    if (argc == 3) {
        iResolution = vec2((double)(atoi(argv[1])),
                           (double)(atoi(argv[2])));
    } else {
        iResolution = vec2(2400., 1200.);
    }

    StopWatch My_Watch;
    My_Watch.Start();
    // Image
    const int image_width = (int)(iResolution.x);
    const int image_height = (int)(iResolution.y);
    const int N = image_width*image_height;

    // Render
  ofstream MyFile("Pic.ppm");
  std::cerr << "Res: " << image_width
        << "x" << image_height << "\n";
    MyFile << "P3\n" << image_width << ' '
        << image_height << "\n255\n";

    //Create Vars
    double *r, *g, *b;

    //for (int i = 0; i < N; ++i) {calc[i] = false;}



    // Allocate Unified Memory      accessible from CPU or GPU
    cudaMallocManaged(&r, N*sizeof(double));
    cudaMallocManaged(&g, N*sizeof(double));
    cudaMallocManaged(&b, N*sizeof(double));


    //Run Code On GPU
    sequence_kernal<<<256,256>>>(r, g, b, N, iResolution);

    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();

    std::cerr << "\nDone.\n" << "Saving to file \n";
    for (int i = 0; i<N; ++i)
```

```
    {
        MyFile << (int)(r[i]*255.999) << ' '
               << (int)(g[i]*255.999) << ' '
               << (int)(b[i]*255.999) << '\n';
    }


  MyFile.close();

    // Free memory
    cudaFree(r);
    cudaFree(g);
    cudaFree(b);

    My_Watch.Stop();
    std::cout << "Time: " << My_Watch.Time() << "\n";
}
```

E Code File: "MathFunctions.cpp"

```
#include"MathObjects.hpp"
#include<iostream>
using namespace std;



// **************************
// **************************
// **                      **
// ** Standard Math Objects **
// ** (Print functions)     **
// **************************
// **************************

std::ostream& operator<<(std::ostream& output, const vec2& rVec)
{
    output << "(" << rVec.x << ", " << rVec.y << ")";
    return output;
}

std::ostream& operator<<(std::ostream& output, const vec3& rVec)
{
    output << "(" << rVec.x << ", " << rVec.y
```

```
                                << ", " << rVec.z << ")";
    return output;
}

// *************************
// *************************
// **                     **
// ** Mathematics Functions **
// **                     **
// *************************
// *************************

//Absolute
template<typename T, int SIZE>
__CUDA_CALL__
T abs(T a)
{
    for(int i=0; i<SIZE; i++)
    {
        a[i] = abs(a[i]);
    }
    return a;
}

__CUDA_CALL__
vec2 abs(vec2 a) {return abs<vec2,2>(a);}
__CUDA_CALL__
vec3 abs(vec3 a) {return abs<vec3,3>(a);}


//Max and Min
template<typename T, int SIZE>
__CUDA_CALL__
T Max(T a, T b)
{
    for(int i=0; i<SIZE; i++)
    {
        a[i] = max(a[i], b[i]);
    }
    return a;
}

__CUDA_CALL__
```

```
vec2 max(vec2 a, vec2 b) {return Max<vec2,2>(a,b);}
__CUDA_CALL__
vec3 max(vec3 a, vec3 b) {return Max<vec3,3>(a,b);}

template<typename T, int SIZE>
__CUDA_CALL__
T Min(T a, T b)
{
    for(int i=0; i<SIZE; i++)
    {
        a[i] = min(a[i], b[i]);
    }
    return a;
}
__CUDA_CALL__
vec2 min(vec2 a, vec2 b) {return Min<vec2,2>(a,b);}
__CUDA_CALL__
vec3 min(vec3 a, vec3 b) {return Min<vec3,3>(a,b);}


//Clamp
template<typename T>
__CUDA_CALL__
T Clamp(T x, double minVal, double maxVal)
{return min(max(x, minVal), maxVal);}
__CUDA_CALL__
double clamp(double x, double minVal, double maxVal)
 {return Clamp<double>(x, minVal, maxVal);}
__CUDA_CALL__
vec2 clamp(vec2 x, double minVal, double maxVal)
 {return Clamp<vec2>(x, minVal, maxVal);}
__CUDA_CALL__
vec3 clamp(vec3 x, double minVal, double maxVal)
 {return Clamp<vec3>(x, minVal, maxVal);}


//Floor (Tested as Needed for Mod)
template<typename T, int SIZE>
__CUDA_CALL__
T Floor(T x)
{
   for(int i=0; i<SIZE; i++)
    {
```

```
        x[i] = floor(x[i]);
    }
    return x;
}
__CUDA_CALL__
vec2 floor(vec2 x) {return Floor<vec2,2>(x);}
__CUDA_CALL__
vec3 floor(vec3 x) {return Floor<vec3,3>(x);}

//SmoothStep
template<typename T>
__CUDA_CALL__
T SmoothStep(double edge0, double edge1, T x)
{
    T t;
    t = clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0);
    return t * t * (3.0 - 2.0 * t);
}
__CUDA_CALL__
double smoothstep(double edge0, double edge1, double x)
{return SmoothStep<double>(edge0, edge1, x);}
__CUDA_CALL__
vec2 smoothstep(double edge0, double edge1, vec2 x)
{return SmoothStep<vec2>(edge0, edge1, x);}
__CUDA_CALL__
vec3 smoothstep(double edge0, double edge1, vec3 x)
{return SmoothStep<vec3>(edge0, edge1, x);}

//Sign
__CUDA_CALL__
double sign(double x)
{
    if (x<0.00001 & x>-0.00001)
        return 0.;
    return normalize(x);
}



//Mod
template<typename T>
__CUDA_CALL__
T Mod(T x, double y) {return x+(-1.)*y*floor(x/y);}
```

```
__CUDA_CALL__
double mod(double x, double y) {return Mod<double>(x,y);}
__CUDA_CALL__
vec2 mod(vec2 x, double y) {return Mod<vec2>(x,y);}
__CUDA_CALL__
vec3 mod(vec3 x, double y) {return Mod<vec3>(x,y);}

//ModRange (Not in GLSL)
__CUDA_CALL__
double mod(double x, double x_low, double x_high)
{return mod(x-x_low,x_high-x_low)+x_low;}


// **********************
// **********************
// **                  **
// ** Vector Functions **
// **                  **
// **********************
// **********************

//Cross
__CUDA_CALL__
vec3 cross(vec3 x, vec3 y)
{
    return vec3(x[1]*y[2]-x[2]*y[1],
                x[2]*y[0]-x[0]*y[2],
                x[0]*y[1]-x[1]*y[0]);
}

//Sum
template<typename T, int SIZE>
__CUDA_CALL__
double Sum(T x)
{
    double total = 0.;
    for(int i = 0; i<SIZE; i++)
    {
        total += x[i];
    }
    return total;
}
__CUDA_CALL__
```

```
double sum(double x) {return x;}
__CUDA_CALL__
double sum(vec2 x) {return Sum<vec2,2>(x);}
__CUDA_CALL__
double sum(vec3 x) {return Sum<vec3,3>(x);}


//Dot
template<typename T>
__CUDA_CALL__
double Dot(T x, T y) {return sum(x*y);}
__CUDA_CALL__
double dot(double x, double y) {return Dot<double>(x,y);}
__CUDA_CALL__
double dot(vec2 x, vec2 y) {return Dot<vec2>(x,y);}
__CUDA_CALL__
double dot(vec3 x, vec3 y) {return Dot<vec3>(x,y);}

//Length
template<typename T>
__CUDA_CALL__
double Length(T x)
{
    return sqrt(dot(x,x));
}
__CUDA_CALL__
double length(double x) {return Length<double>(x);}
__CUDA_CALL__
double length(vec2 x) {return Length<vec2>(x);}
__CUDA_CALL__
double length(vec3 x) {return Length<vec3>(x);}


//Normalize
template<typename T>
__CUDA_CALL__
T Normalize(T x) {return x/length(x);}
__CUDA_CALL__
double normalize(double x) {return Normalize<double>(x);}
__CUDA_CALL__
vec2 normalize(vec2 x) {return Normalize<vec2>(x);}
__CUDA_CALL__
vec3 normalize(vec3 x) {return Normalize<vec3>(x);}
```

```

//Distance



// **************
// **************
// **          **
// ** StopWatch **
// **          **
// **************
// **************
void StopWatch::Start()
{
    t1 = std::chrono::high_resolution_clock::now();
}

void StopWatch::Stop()
{
    t2 = std::chrono::high_resolution_clock::now();
}

float StopWatch::Time()
{
    std::chrono::duration<double> time_span =
 std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);
    return time_span.count();
}
```

F Code File: "MathObjects.hpp"

```
#include <iostream>
#include <math.h>


//StopWatch
#include <ctime>
#include <ratio>
#include <chrono>

#define float double
//Set __CUDA_CALL__ to correct vaule depends if c++ or cuda
//__CUDA_CALL__ means it can be called by GPU
```

```
#ifdef __CUDACC__
#define __CUDA_CALL__ __host__ __device__
#else
#define __CUDA_CALL__
#endif

using namespace std;


// **********************
// **********************
// **                  **
// ** RayMarch Constants **
// **                  **
// **********************
// **********************

#define PI 3.1415926538
#define MAX_STEPS 70
#define MAX_DISTANCE 100.
#define MIN_STEP_SIZE 0.001
#define epsilon 0.00001
#define inf 100000000.


// *************************
// *************************
// **                     **
// ** Standard Math Objects **
// **                     **
// *************************
// *************************

template<typename T, int SIZE>
struct VectorOps
{

    virtual __CUDA_CALL__ T* getVec() = 0;

    __CUDA_CALL__
    T operator+(const T& b)
    {
        T* a = this->getVec();
```

```
        T c;
        for (int i=0; i<SIZE; i++)
        {
            c.data[i] = a->data[i] + b.data[i];
        }
        return c;
    }

    __CUDA_CALL__
    T operator+(const double& b)
    {
        T a = *(this->getVec());
        return a+T(b);
    }

    __CUDA_CALL__
    T operator-(T b)
    {
        T a = *(this->getVec());
        return a+b*(-1.);
    }

    __CUDA_CALL__
    T operator-(const double& b)
    {
        T a = *(this->getVec());
        return a+T(-b);
    }

    __CUDA_CALL__
    T operator-()
    {
        T a = *(this->getVec());
        return a*(-1.);
    }

    __CUDA_CALL__
    T operator*(const T& b)
    {
        T* a = this->getVec();
        T c;
        for (int i=0; i<SIZE; i++)
        {
```

```
            c.data[i] = a->data[i] * b.data[i];
        }
        return c;
    }

    __CUDA_CALL__
    T operator*(const double& b)
    {
        T a = *(this->getVec());
        return a*T(b);
    }

    __CUDA_CALL__
    T operator/(const T& b)
    {
        T* a = this->getVec();
        T c;
        for (int i=0; i<SIZE; i++)
        {
            c.data[i] = a->data[i] / b.data[i];
        }
        return c;
    }

    __CUDA_CALL__
    double& operator[](int i) // works for assignment as well :)
    {
        T* a = this->getVec();
        return a->data[i];
    }
};

template<typename T>
__CUDA_CALL__
T operator* (double const& lhs, T rhs) {
  return rhs*lhs;
}

template<typename T>
__CUDA_CALL__
T operator+ (double const& lhs, T rhs) {
  return rhs+lhs;
}
```

```
template<typename T>
__CUDA_CALL__
T operator- (double const& lhs, T rhs) {
  return -(rhs-lhs);
}


//Not used
template<int POS>
struct scalar_swizzle
{
    double v[POS+1];
    double& operator=(const double x)
    {
        v[POS] = x;
        return v[POS];
    }
    operator double() const
    {
        return v[POS];
    }
};

//Overload subscript operator. []

struct vec2 : VectorOps<vec2, 2>
{
    /*
    union{
        double data[2];
        scalar_swizzle<0> x;
        scalar_swizzle<1> y;
    };
    */

    union{
        struct{ double data[2];};
        struct{double x,y;};
        struct{double r,g;};
    };

```

```
    __CUDA_CALL__
    vec2()
    {
        this->x=0.0;
        this->y=0.0;
    }

    __CUDA_CALL__
    vec2(double x, double y)
    {
        this->x=x;
        this->y=y;
    }

    __CUDA_CALL__
    vec2(double x)
    {
        this->x=x;
        this->y=x;
    }

    //VectorOps Inheritence
    __CUDA_CALL__
    vec2* getVec()
    {
        return this;
    }
    using VectorOps<vec2, 2>::operator+;
    using VectorOps<vec2, 2>::operator*;

    friend std::ostream& operator<<(std::ostream& output,
                                    const vec2& rVec);
};



struct vec3 : VectorOps<vec3, 3>
{
    /*
    union{
        double data[3];
        scalar_swizzle<0> x;
        scalar_swizzle<1> y;
```

```
        scalar_swizzle<2> z;
    };
    */
        union{
        struct{ double data[3];};
        struct{double x,y,z;};
        struct{double r,g,b;};
    };

    __CUDA_CALL__
    vec3()
    {
        this->x=0.0;
        this->y=0.0;
        this->z=0.0;
    }

    __CUDA_CALL__
    vec3(double x, double y, double z)
    {
        this->x=x;
        this->y=y;
        this->z=z;
    }

    __CUDA_CALL__
    vec3(vec2 a, double z)
    {
        this->x=a.x;
        this->y=a.y;
        this->z=z;
    }

    __CUDA_CALL__
    vec3(double x, vec2 b)
    {
        this->x=x;
        this->y=b.x;
        this->z=b.y;
    }

    __CUDA_CALL__
    vec3(double x)
```

```
    {
        this->x=x;
        this->y=x;
        this->z=x;
    }

    //VectorOps Inheritence
    __CUDA_CALL__
    vec3* getVec()
    {
        return this;
    }
    using VectorOps<vec3, 3>::operator+;
    using VectorOps<vec3, 3>::operator*;

    friend std::ostream& operator<<(std::ostream& output,
                                        const vec3& rVec);
};


// **************************
// **************************
// **                      **
// ** Mathematic Operations **
// **                      **
// **************************
// **************************

//Absolute Value
__CUDA_CALL__
vec2 abs(vec2 a);
__CUDA_CALL__
vec3 abs(vec3 a);


//Max
__CUDA_CALL__
vec2 max(vec2 a, vec2 b);
__CUDA_CALL__
vec3 max(vec3 a, vec3 b);

//Min
__CUDA_CALL__
```

```
vec2 min(vec2 a, vec2 b);
__CUDA_CALL__
vec3 min(vec3 a, vec3 b);

//Clamp
__CUDA_CALL__
double clamp(double x, double minVal, double maxVal);
__CUDA_CALL__
vec2 clamp(vec2 x, double minVal, double maxVal);
__CUDA_CALL__
vec3 clamp(vec3 x, double minVal, double maxVal);

//Floor
__CUDA_CALL__
vec2 floor(vec2 x);
__CUDA_CALL__
vec3 floor(vec3 x);

//SmoothStep


//Sign
__CUDA_CALL__
double sign(double x);

//Mod
__CUDA_CALL__
double mod(double x, double y);
__CUDA_CALL__
vec2 mod(vec2 x, double y);
__CUDA_CALL__
vec3 mod(vec3 x, double y);


//ModRange (Not in GLSL)
__CUDA_CALL__
double mod(double x, double x_low, double x_high);

//SmoothStep
__CUDA_CALL__
double smoothstep(double edge0, double edge1, double x);
__CUDA_CALL__
vec2 smoothstep(double edge0, double edge1, vec2 x);
```

```
__CUDA_CALL__
vec3 smoothstep(double edge0, double edge1, vec3 x);


// ********************
// ********************
// **                **
// ** Vector Functions **
// **                **
// ********************
// ********************

//Cross
__CUDA_CALL__
vec3 cross(vec3 x, vec3 y);


//Sum (Not in GLSL, here to make code neater)
//(Tested by being used by other functions)
__CUDA_CALL__
double sum(double x);
__CUDA_CALL__
double sum(vec2 x);
__CUDA_CALL__
double sum(vec3 x);

//Length
__CUDA_CALL__
double length(double x);
__CUDA_CALL__
double length(vec2 x);
__CUDA_CALL__
double length(vec3 x);

//Dot
__CUDA_CALL__
double dot(double x, double y);
__CUDA_CALL__
double dot(vec2 x, vec2 y);
__CUDA_CALL__
double dot(vec3 x, vec3 y);

//Normalize
```

```
__CUDA_CALL__
double normalize(double x);
__CUDA_CALL__
vec2 normalize(vec2 x);
__CUDA_CALL__
vec3 normalize(vec3 x);

//Distance


// **************
// **************
// **          **
// ** StopWatch **
// **          **
// **************
// **************
class StopWatch {
private:
 std::chrono::high_resolution_clock::time_point t1;
 std::chrono::high_resolution_clock::time_point t2;

public:
 void Start();
 void Stop();
 float Time();
};
```

G Code File: "RayMarch.hpp"

```
__CUDA_CALL__
vec3 GetNormal(vec3 p) //Gets normal of the surface
{
    vec3 GradF = vec3(0.0,0.0,0.0);
    float F = distFunc(p);
    GradF.x = distFunc(p+epsilon*vec3(1.,0.,0.));
    GradF.y = distFunc(p+epsilon*vec3(0.,1.,0.));
    GradF.z = distFunc(p+epsilon*vec3(0.,0.,1.));
    GradF = GradF-F;
    return normalize(GradF);
}

__CUDA_CALL__
```

```
vec3 march(vec3 dir, vec3 start_pos)
{   //return a vec3
    //vec3.x is distance
    //vec3.y number of steps taken
    //vec3.z if we hit an object
    int steps = 0;
    vec3 pos = start_pos;
    float step_size = 0.;
    float dist = 0.;
    while (steps < MAX_STEPS && dist < MAX_DISTANCE)
    {
        //Checks for maximum safe distance to step
        step_size = distFunc(pos);
        if (step_size < MIN_STEP_SIZE)
        {
            return vec3(dist, float(steps),1.);
        }
        pos = pos + step_size*dir;
        steps = steps + 1;
        dist += step_size;
    }
    return vec3(dist, float(steps),0.);
}


__CUDA_CALL__
vec3 Direction(vec3 D, vec3 H, vec3 W, vec2 wh, vec2 iRes)
{
    float FOV = PI/2.;
    return normalize(D + 2.*tan(FOV/2.)*(wh.x*W+wh.y*H)/iRes.x);
}

__CUDA_CALL__
vec3 GetW(vec3 d)
{
    d = normalize(d);
    vec3 W = vec3(d.y, -d.x, 0.);
    return normalize(W);
}

__CUDA_CALL__
vec3 GetColorRayMarch(vec2 fragCoord, vec3 LightPos, vec3 camPos,
                      vec3 camDir, vec2 iRes)
```

```
{
    vec3 W = GetW(camDir);
    vec3 H = cross(W,camDir);

    // Normalized pixel coordinates (from -Res to Res)
    vec2 wh = fragCoord - iRes/2.;

    vec3 RayDir = Direction(camDir, H, W, wh, iRes);
    float out_color = 0.0;
    vec3 pos_hit = march(RayDir, camPos);
    if (abs(pos_hit[2]-1.)<0.001)
    {
        //Hit
        out_color = 0.5;

        //Get Normal and other needed directions
        vec3 currentPos = pos_hit.x*RayDir+camPos;
        vec3 Norm = GetNormal(currentPos);
        vec3 ToLight = normalize(LightPos-currentPos);
        vec3 ToCamera = normalize(camPos-currentPos);

        //Specular highlight
        //Via Blinn Phong shading model
        out_color *=
            pow(max(0.,dot(Norm, normalize(ToLight+ToCamera))),2.0);

        //Now calculate shodow
        vec3 shad_hit = march(normalize(LightPos-currentPos),
                              currentPos+2.*Norm*MIN_STEP_SIZE);
        if (abs(shad_hit[2]-1.)<0.001)
        {
        //If hit something there should be shadow
            out_color *= 0.6;
        }
        //Ambient Oclusion
        //out_color += 0.5;
        out_color += 0.5*(1.-1.2*pos_hit.y/float(MAX_STEPS));
    }

    // Output to screen
    vec3 fragColor = vec3(out_color,out_color,0.);
    return fragColor;

```

```
}
```

H Code File: "MengerSponge2D.hpp"

```
__CUDA_CALL__
float sdSquare(vec2 p) {
    //Square centered at the origin with side length 1.
    vec2 d = abs(p)-vec2(0.5,0.5);
    return length(max(d,vec2(0.0))) + min(max(d.x,d.y),0.0);
}

__CUDA_CALL__
float my_map(vec2 p) //MengerSponge
{
    float sd = sdSquare(p);
    float its = 6.0; //Iterations
    for (float it = 1.; it < its; it++)
    {
        vec2 mod_d = mod(p*pow(3.,it)+1.5,3.)-1.5;
        float d = sdSquare(mod_d)/pow(3.,it);
        sd = max(sd, -d);
    }
    return sd;
}

__CUDA_CALL__
vec3 GetColor(vec2 fragCoord, vec2 iRes)
{
    // Normalized pixel coordinates (from -0.5 to 0.5)y
    vec2 uv = (fragCoord-iRes/2.)/iRes.y;

    // Output to screen
    return vec3(smoothstep(-0.00001,0.00001,-my_map(uv)),0.0,0.0);
}
```

I Code File: "MengerSponge3D.hpp"

```
__CUDA_CALL__
float sdBox( vec3 p, vec3 b )
{
  vec3 q = abs(p) - b;
  return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0);
}
```

```

__CUDA_CALL__
float sdCross(vec3 p )
{

  float da = sdBox(p, vec3(inf,1.0,1.0));
  float db = sdBox(vec3(p[1],p[2],p[0]),vec3(1.0,inf,1.0));
  float dc = sdBox(vec3(p[2],p[0],p[1]),vec3(1.0,1.0,inf));
  return min(da,min(db,dc));
}

__CUDA_CALL__
float distFunc(vec3 p)
{
    float sd = sdBox(p,vec3(3.,3.,3.));
    for (float it = 0.; it < 4.; it++)
    {
        vec3 mod_d = mod(p*pow(3.,it)+3.0,6.)-3.0;
        float d = sdCross(mod_d)/pow(3.,it);
        sd = max(sd, -d);
    }
    return min(sd, p.z+3.);
}

#include "RayMarch.hpp"

__CUDA_CALL__
vec3 GetColor(vec2 fragCoord, vec2 iRes)
{
    vec3 LightPos = vec3(100.,100.,200.);
    //vec3 camPos = vec3(-10.,0.,1.5);
    //vec3 camDir = normalize(vec3(1.,0.,-0.2));
    //vec3 LightPos = vec3(0.);
    vec3 camPos = vec3(-7.,1.,1.5);
    //float th = 0.2;//iTime/3.0;
    //camPos.xy =mat2(cos(th),sin(th), -sin(th), cos(th))*camPos.xy;
    vec3 camDir = -normalize(camPos);
    // Output to screen
    vec3 fragColor = GetColorRayMarch(fragCoord, LightPos,
                                      camPos, camDir, iRes);
    return fragColor;
}
```

J Code File: “SierpinskiTriangle.hpp”

```
__CUDA_CALL__
float sdEquilateralTriangle(vec2 p)
{
    //Triangle with points
    //(-1,-sqrt(3)/3),(1,-sqrt(3)/3),(0,2*sqrt(3)/3)
    const float k = sqrt(3.0);
    p.x = abs(p.x) - 1.0;
    p.y = p.y + 1.0/k;
    if( p.x+k*p.y>0.0 ) p = vec2(p.x-k*p.y,-k*p.x-p.y)/2.0;
    p.x -= clamp( p.x, -2.0, 0.0 );
    return -length(p)*sign(p.y);
}

__CUDA_CALL__
float sdTri(vec2 p)
{
    //Triangle with points
    //(-1,-0),(1,0),(0,sqrt(3))
    const float k = sqrt(3.0);
    return sdEquilateralTriangle(vec2(p.x,p.y-k/3.));

}


__CUDA_CALL__
float my_map(vec2 p) {
    float its = 8.0; const float k = sqrt(3.0);
    float modx = mod(p.x,-1.,1.); float mody = mod(p.y, 2.*k);
    float sd = sdTri(vec2(modx, mody));
    for (float i = 1.; i < its; ++i) {
        modx = mod(p.x*pow(2.,i),-1.,1.);
        mody = mod(-(p.y)*pow(2.,i)+k,2.*k);
        sd = max(sd, -sdTri(vec2(modx, mody))/pow(2.,i)); }
    return sd; }

__CUDA_CALL__
vec3 GetColor(vec2 fragCoord, vec2 iRes)
{
    // Normalized pixel coordinates (from -0.5 to 0.5)y
    vec2 uv = (fragCoord-iRes/2.)/iRes.y;

```

```
    // Output to screen
    return vec3(smoothstep(-0.00001,0.00001,
                                -my_map((uv*2.)+vec2(0.,0.9))),0.0,0.0);
}
```

K Code File: "SierpinskiPyramid.hpp"

```

__CUDA_CALL__
float sdTriPrism( vec3 p)
{
  //Long Triangle
  //Made bigger in y direction
  //(-0.5,0.)(0.5,0.)(0, sqrt(3.)/2.)
  double temp = p.x;
  p.x = p.y;
  p.y = temp;
  p.z = p.z - sqrt(3.)/6.;
  vec3 q = abs(p);
  vec2 h = vec2(sqrt(3.)/3.,inf);
  const float k = sqrt(3.)/2.;
  return max(q.y-h.y,max(q.x*k+p.z*0.5,-p.z)-h.x*0.5);
}

__CUDA_CALL__
float sdTriCross(vec3 p)
{
    return min(sdTriPrism(p),sdTriPrism(vec3(p.y,p.x,p.z)));
}

__CUDA_CALL__
float sdPyramid( vec3 p)
{
  //Square Based Pyrimid with
  //(-0.5,0.,0.5) (0.,sqrt(3.)/2.,0.)
  //h means height of triangle
  //p.xyz = p.xzy;
  const float h = sqrt(3.)/2.;
  float m2 = h*h + 0.25;

  p.x = abs(p.x);
  p.y = abs(p.y);
  p.x = (p.y>p.x) ? p.y : p.x;
```

```
  p.x -= 0.5;
  p.y = (p.y>p.x) ? p.x : p.y;
  p.y -= 0.5;

  vec3 q = vec3( p.y, h*p.z - 0.5*p.x, h*p.x + 0.5*p.z);

  float s = max(-q.x,0.0);
  float t = clamp( (q.y-0.5*p.y)/(m2+0.25), 0.0, 1.0 );

  float a = m2*(q.x+s)*(q.x+s) + q.y*q.y;
  float b = m2*(q.x+0.5*t)*(q.x+0.5*t) + (q.y-m2*t)*(q.y-m2*t);

  float d2 = min(q.y,-q.x*m2-q.y*0.5) > 0.0 ? 0.0 : min(a,b);

  return sqrt( (d2+q.z*q.z)/m2 ) * sign(max(q.z,-p.z));
}


__CUDA_CALL__
float distFunc(vec3 p)
{
    p = (p+vec3(0.,0.,2.))/7.;
    const float k = sqrt(3.0);
    float modx = 0., mody = 0., modz = 0.;
    float sd = sdPyramid(p);
    for (float i = 0.; i < 5.; ++i)
    {
        modx = mod(p.x*pow(2.,i)+0.25, 0.5) - 0.25;
        mody = mod(p.y*pow(2.,i)+0.25, 0.5) - 0.25;
        modz = mod(-p.z*pow(2.,i)+k/4., k/2.);
  sd = max(sd, -sdTriCross(vec3(modx, mody, modz)*2.)/pow(2.,i+1.));
    }
    return min(sd,p.z)*7.;
}

#include "RayMarch.hpp"

__CUDA_CALL__
vec3 GetColor(vec2 fragCoord, vec2 iRes)
{
    vec3 LightPos = vec3(100.,100.,100.);
    //vec3 camPos = vec3(-10.,0.,1.5);
    //vec3 camDir = normalize(vec3(1.,0.,-0.2));
```

```
    //vec3 LightPos = vec3(0.);
    vec3 camPos = vec3(-7.,1.,1.5);
    //float th = 0.2;//iTime/3.0;
    //camPos.xy = mat2(cos(th),sin(th),-sin(th), cos(th))*camPos.xy;
    vec3 camDir = -normalize(camPos);
    // Output to screen
    vec3 fragColor = GetColorRayMarch(fragCoord, LightPos,
                                      camPos, camDir, iRes);
    return fragColor;
}
```