Procedural Rendering with Ray Marching

Christian A. Robles
University of Southern California
Los Angeles, USA
roblesch@usc.edu

Hoseung Lee
University of Southern California
Los Angeles, USA
hoseungl@usc.edu

Samuel Yin
University of Southern California
Los Angeles, USA
slyin@usc.edu

Vansh Dhar
University of Southern California
Los Angeles, USA
vdhar@usc.edu

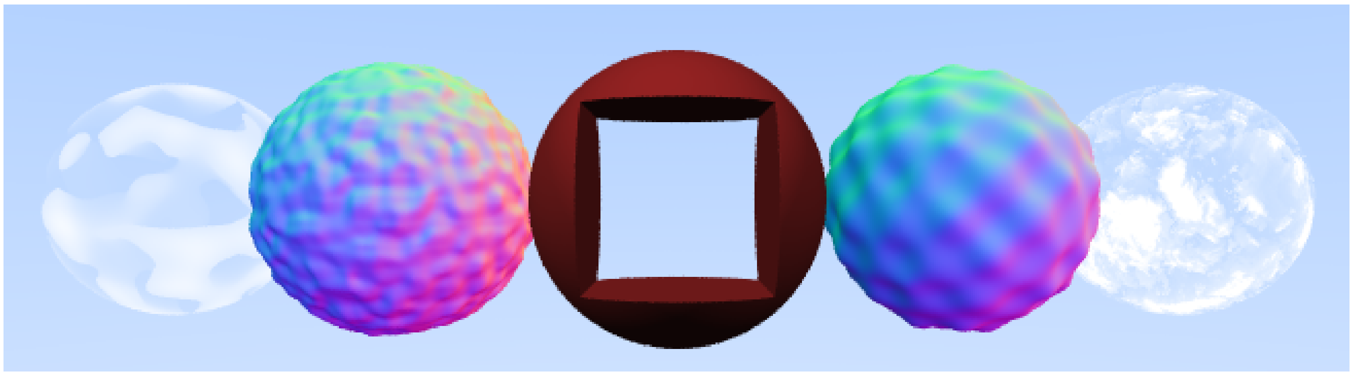

Figure 1: Procedural Perturbations, Constructive Solid Geometry, Marched Depth Clouds

ABSTRACT

Ray tracing requires evaluation of explicit intersections between rays and surface primitives to identify objects visible in a scene to the camera. Complex shapes can be difficult to model as they must be broken down into an aggregate of many primitives, and ray tracing scales poorly with the number of objects in the scene. Ray marching allows us to model complex shapes more efficiently by implicitly identifying intersections with surfaces through the use of a signed distance function. With ray marching, we can efficiently model and render complex shapes, create new shapes with constructive solid geometry, and render procedural 3d depth textures.

1 INTRODUCTION

Let a scene consist of a set of surfaces in a 3d world coordinate space. Place a camera into the scene at some position. We will generate an image by sending rays through each pixel coordinate in the camera's viewport, determining if an intersection occurs, and evaluating a material at that intersection.

First consider a ray tracing renderer. We will determine the color of each pixel by evaluating the scene in the direction of a camera ray. We will compare this ray to each object in the scene and determine if an intersection occurs, then evaluate its material properties. Consider intersection with a relatively simple surface primitive, a sphere. Ray-sphere intersection requires solving $|(P(t) - C)|^2 = r^2$ for some ray $P(t) = A + tb$ and some sphere with center C and radius r. We will then need to decide which intersection point is nearest and in front of the camera, if we are viewing from the inside or outside of the sphere, and calculate a surface normal to calculate the appropriate material color at that intersection. We can see that even for a relatively simple primitive such as a sphere, determining a ray-surface intersection for every surface in a scene is a computationally expensive task.

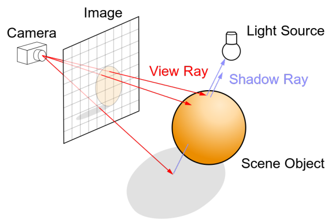

Figure 2: Evaluating a scene with ray tracing, via Wikimedia Commons. (https://w.wiki/55Pu).

In ray tracing, we are limited to only those surface primitives where we can explicitly calculate a ray-surface intersection. Planar surfaces can be solved with a ray-plane intersection and inside outside test. Various shapes and orientations of surfaces can be derived by model transformations of translations rotations and scales. Higher complexity surfaces that would be otherwise too difficult to intersect directly can be instead modeled by composition of many planar primitives, but as we saw performance scales poorly with the number of surfaces in the scene. Complex materials may additionally require even more rays to be cast to determine lighting or transmission. Effects like soft shadows and anti aliasing further degrade performance, requiring a polynomial total number of rays to be cast to determine the color of an individual pixel.

Consider now a ray marching renderer. We again determine the color of each pixel by evaluating the scene in the direction of a camera ray. However, we do not evaluate intersection between this ray and any surface in the scene. Instead, let the scene provide a *Signed Distance Function* which evaluates some point $p(x, y, z)$ and returns the approximate distance d between p and the nearest surface in the scene. If the value is positive, we are d units away from the nearest surface. If it is negative, we are d units inside the surface. If d is close to zero, then we can say we are intersecting the nearest surface. Starting from $t = 0$, we will for each ray evaluate the distance to the closest surface in the scene d and march d units in the direction of the ray. We will evaluate the scene again and repeat this process until d is near-zero, or until the ray terminates due to a march limit or a maximum depth along the ray t_{max}.

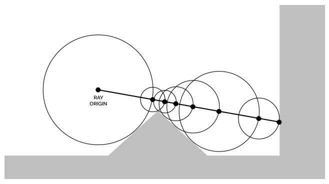

Figure 3: Marching along a ray using the SDF of the scene, via Wikimedia Commons. (https://w.wiki/55Ps).

In the ray marching model, ray-surface intersections no longer require explicit evaluation of intersection formulae. Instead, we implicitly model ray-surface intersections by marching down the ray until the distance to the scene is near-zero. With these implicit intersections, we are no longer constrained to simple surface primitives. We can model any surface that has a defined distance function. Additionally, we can model scenes of complex functions like fractals or noise functions by mapping their values to distance values. We can combine distance functions for multiple surface primitives by union or intersection, and interpolate between them with smoothing functions. We can model 3D materials by stepping through them and evaluating functions along each step. Surface normals no longer need to be explicitly provided by each surface primitive and can instead be calculated by the gradient of the distance function.

2 RENDERER IMPLEMENTATION

We implement our renderer from referenec code provided by Peter Shirley's *Ray Tracing in One Weekend*. We use the template code Peter provides up to Chapter 4: *Rays, a Simple Camera, and Background*. We also use the Anti Aliasing techniques from Chapter 7, and the Positionable Camera from Chapter 11. We follow Peter's interface patterns for surfaces and materials, with some modifications. We rename *hittable* to *surface*, and *hittable_list* to *scene*. We discard all ray tracing functionality and replace it with our implementations of ray marching.

2.1 Interfaces

2.1.1 Surface. The *surface* class is an interface for intersectable objects in the scene. Any class that implements *surface* must accept in its constructor a valid pointer to a *material*. It must also provide an implemention for the following function:

```
double distance(const vec3& p)
```

The *distance* function returns the signed distance from some point $p(x, y, z)$ and the surface. If the result is negative, p is considered inside the surface.

2.1.2 Material. A class that implements the *material* interface must implement the following function:

```
vec3 color(ray &r, vec3 p, vec3 N,
           vector<light> lights)
```

The *color* function accepts an incoming ray, an intersection point, a surface normal, and a list of directional lights. It returns the color of the material evaluated by those parameters.

2.1.3 Scene. The *scene* class manages objects and lights in a scene, and maintains internal lists of pointers to instances of these classes. It provides the following key functions:

```
bool near_zero(double d)
```

Returns *true* if d is smaller than machine-epsilon: the difference between 1.0 and the next value representable by the floating-point type.

```
double distance_estimator(vec3 p)
```

Iterates over the list of scene surfaces and returns the distance from some point $p(x, y, z)$ and the surface nearest that point.

```
vec3 normal(vec3 &p)
```

Returns the "surface normal" at some point $p(x, y, z)$. We take a small step along each of the x, y, z axes and record the gradient of the distance estimator in each direction. Returns the unit length vector in the direction of the gradient of the distance estimator.

```
bool march(ray& r, hit_record& rec)
```

Beginning at $t = 0$, *march* evaluates the distance from the point along the ray r at t to the nearest surface in the scene. If the distance is near zero, *march* will terminate and return true, and store the information about the intersection and the material of the surface intersected on the hit record. If the distance is not near zero, it will increase t by d with a minimum step size of 0.001 and evaluate the distance estimator again. Will terminate if a surface is hit or if $t \geq t_{max}$.

```
vec3 ray_color(ray& r)
```

Evaluates a ray by performing marching. If a surface is hit, queries the material properties for the color evaluated at the point of intersection. Otherwise, returns the background color of the scene.

2.2 Surface Primitives

The following classes provide implementations for *surface*.

2.2.1 Sphere. The *sphere* class implements *surface*. An instance of *sphere* is specified by a center C and a radius r. It provides the following implementation of *distance* -

$$distance = |(p - C)| - r$$

2.2.2 Box. The *box* class also implements *surface*. An instance of *box* is specified by a vector P representing the distance from the point to the box and R the vector representing the distance from the center to the edges of the box -

$$distance = length(max(|P| - R, 0)$$

2.2.3 Equilateral Triangular Prism. An instance of a triangular prism is specified by a center C, length L, and height H. The "center" determines the coordinate location of the object. The "length" determines the length of the prism, and the height determines the size of the equilateral triangle on the triangular prism.

$$q = |p - center|;$$
$$distance =$$
$$max(q.z - length, max(q.x * 0.866025 + p.y * 0.5, -p.y) - height * 0.5);$$

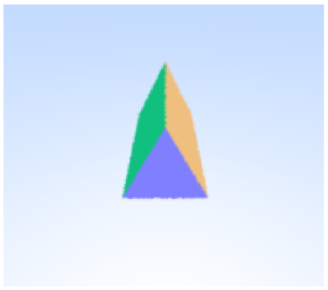

Figure 4: A triangular prism with a height of 0.4 and length of 0.8

2.2.4 Cylinder. An instance of a cylinder is specified by a center C, height H, and radius R. The "center" determines the coordinate location of the object. The "height" determines the length of the cylinder, and the "radius" determines the radius of the cylinder.

$$q = p - center;$$
$$d.x = length(q.xz) - radius;$$
$$d.y = length(q.y) - height;$$
$$distance :$$
$$min(max(d), 0) + length(max(d))$$

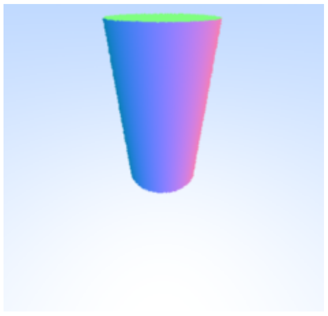

Figure 5: A cylinder with a height of 0.9 and radius of 0.5

2.2.5 Pyramid. An instance of a pyramid is specified by a center C and height H. The "center" determines the coordinate location of the object. The "height" determines the height of the pyramid (where the square base of the pyramid is height = 0). The base of the pyramid is constant (1 by 1).

$$a = height * height + 0.25;$$
$$p = p - center$$
$$p.xz = |p.xz|$$
$$if p.z > p.x : p.xz = p.zx$$
$$p.xz = p.xz - 0.5$$
$$b = (p.z, height * p.y - 0.5 * p.x, height * p.x + 0.5 * p.y);$$
$$c = max(-b.x, 0.0);$$
$$d = clamp((b.y - 0.5 * p.z)/(a + 0.25), 0.0, 1.0);$$
$$e = a * (b.x + s) * (b.x + s) + b.y * b.y;$$
$$f = a * (b.x + 0.5 * d) * (b.x + 0.5 * d) + (b.y - a * t) * (b.y - a * d);$$
$$if min(q.y, -q.x * a - q.y * 0.5) > 0.0 : g = 0.0$$
$$else : g = min(a, b);$$
$$distance =$$
$$\sqrt{(g + b.z * b.z)/a} * sign(max(b.z, -p.y));$$

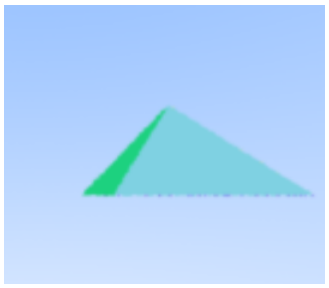

Figure 6: A pyramid with a height of 0.6, the base is 1x1

2.2.6 Infinite Cylinder. An instance of an infinite cylinder is specified by a center C and radius R. The "center" determines the coordinate location of the object and the "radius" determines the radius of the cylinder. As you can see as long as the equation for a primitive exists, surfaces can be rendered, even mathematically infinite ones.

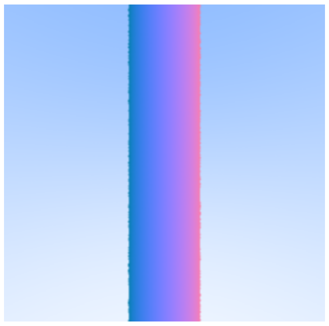

Figure 7: An infinite cylinder with a radius of 0.3

2.3 Materials

The following classes provide implementations for *material*.

2.3.1 Diffuse. The *diffuse* class implements the *color* function to return the color C as

$$C = (Ks \sum_L (le(R \cdot E)^s]) + (Kd \sum_L (le(N \cdot L)) + (Ka * la)$$

2.3.2 Clouds. We implement the materials perlin_cloud_2d, perlin_cloud_3d, gardner_cloud_2d and gardner_cloud_3d by mapping surface coordinates to a noise function and attenuating output color by march depth. Described in section 4.

2.3.3 Normals. Maps a normal N to RGB as

$$C = 0.5 * (N + 1)$$

2.3.4 Flat. Returns a constant color.

3 CONSTRUCTIVE SOLID GEOMETRY

Constructive Solid Geometry the technique of combining primitives to form a more complex object using Boolean operators such as union, difference, and intersection. Ray marching makes CSG easy. To understand how CSG works in ray marching, its important to understand Signed Distances Function mentioned earlier. When a signed distance function returns a positive value, it means it is outside of the object. A negative SDF means we are inside of the object. When can use this property to implement the Boolean operations.

3.0.1 Union. The unions of two objects is computed simply by taking the closest of the two distances between the objects or the minimum of the distances of the two objects. Since we are already returning the surface nearest to the point in our distance estimator function, this operation is already handled by the ray marching algorithm.

3.0.2 Intersection. The intersection of two objects is the area in which both objects converge. To get the intersection of two objects, distance to both objects should be less than or equal to zero. This creates an object that only exists at the intersection of the two objects. This is because the only way we can have distance that is zero or less is if both SDF values are zero or less.

3.0.3 Difference. Finally, the difference of two objects involves taking the maximum of the negative distance of one object and the regular distance of a another. Flipping the sign inverts the object so everything that was considered inside an object is now outside and vice versa. The object now only exists if it is inside the non inverted objected and outside of the inverted object. The result is one area "cutting" into the other when brought together.

3.0.4 Smoothing. To blend the area where two objects intersect, we need to modify the min function. We only want to change the are where the distance of the objects are close to each other. We use the following function to apply the smoothing for the union operator.

```
h = max(k - abs(d1 - d2), 0.0);
return min(d1, d2) - h * h * 0.25/k;
```

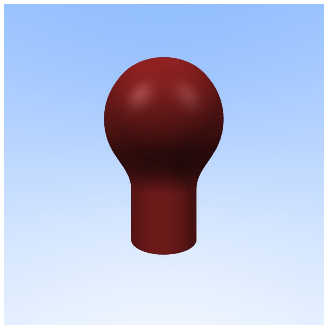

Figure 8: The intersection between a sphere and cylinder blended together

k is some value between 0 and 1 and controls the amount of smoothing. For intersection and difference, we apply the same logic as before: using max instead of min and flipping the sign for difference.

4 PROCEDURAL TEXTURES

Procedural generation of 3D clouds is a complex challenge in rendering. Clouds in real-time applications are often drawn by mapping pre-rendered 2D textures at distances or on volumes not interactable by the camera. However, these approaches fall apart when viewed at shallow angles or when passed through by the camera. In this section, we will examine how ray marching can be leveraged to generate 3D textures on marchable depth surfaces that approximate the appearance of clouds.

We will examine two noise functions for generating cloud textures. The first is proposed by Geoffrey Y. Gardner in his paper *Visual Simulation of Clouds*. The second is the famous Perlin Noise, originally proposed by Ken Perlin in his paper *An image synthesizer*. Both were originally published in the 1985 ACM SIGGRAPH Computer Graphics Volume 19, Issue 3.

4.1 Gardner Noise

In his 1985 paper *Visual Simulation of Clouds*, Geoffrey Gardner proposes a mathematical texturing function for generating textures that approximate the appearance of clouds. We refer to this texture function as *Gardner Noise*. Gardner Noise models the texture as a product of sums of sine waves

$$T(X,Y,Z) = k \sum_{i+1}^{n} [C_i \sin(FX_i x + PX_i) + T_0] \quad (1)$$

$$\times \sum_{i=1}^{n} [C_i \sin(FY_i Y + PY_i) + T_0] \quad (2)$$

The textures are produced as a sum of four to seven sines where frequencies and coefficients are chosen by

$$FX_{i+1} = 2FX_i \quad (3)$$

$$FY_{i+1} = 2FY_i \quad (4)$$

$$C_{i+1} = .707C_i \quad (5)$$

We use the 3D formulation of Gardner Noise where PX_i and PY_i are phase shifts determined by

$$PX_i = \pi/2\sin(.5FY_{i-1}Y) + \pi\sin(FX_iZ/2) \qquad for\ i > 1 \qquad (6)$$

$$PY_i = \pi/2\sin(.5FX_{i-1}iX) + \pi\sin(FX_iZ/2) \qquad for\ i > 1 \qquad (7)$$

T_0 is a parameter controlling contrast, and k is computed such that the maximum of $T(X, Y, Z)$ is approximately 1.

4.2 Perlin Noise

First described by Ken Perlin in his 1985 paper *An image synthesizer*, *Perlin Noise* is a noise function that accepts a n-vector and returns a value with the qualities of statistical invariance under rotation and translation and a narrow bandpass limit in frequency. We use 3D noise, which is constructed by

1. Construct a 3 dimensional grid of size x by y by z.
2. Assign to each grid intersection a pseudo-random unit length gradient vector $v_{x_i,y_i}(x, y, z)$
3. For some input $p(x, y, z)$, determine which grid cell contains p. Identify the corners of that cell and calculate an offset vector from each corner to p.
4. For each corner, calculate the dot product of its offset vector and its gradient vector.
5. Interpolate the grid corner dot products at p and return this value as the result.

4.3 Procedural Clouds

Once the noise functions are formulated, using them to generate textures is simple. We evaluate a ray-surface intersection and pass the resulting point $p(x, y, z)$ to one of these functions and use the $[0, 1.0]$ result as a texture value T. Gardner phase and coefficient parameters can be tuned to modify the size and variance of the cloud-like textures, and Perlin noise can be tuned to map input coordinates to larger or smaller gradient grids, resulting in fuzzier or more granular textures.

Because each of these noise functions are continuous on three dimensions of input, we can implement a fragment shader that marches through the surface and attenuates the color output by the number of marches. We can accept as a material parameter the approximate depth of a surface, and use this value to determine a step bound and a step size within the surface. We use a cutoff threshold to march past low texture values and attenuate the final result by a combination of step size and step count.

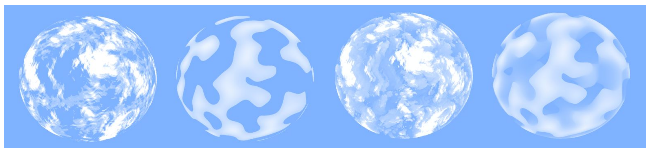

Figure 9: Surface Mapped and Depth Marched Gardner & Perlin Clouds

5 PROCEDURAL SURFACES

In section 3 we introduced Constructive Solid Geometry, and saw how we can alter surface evaluations by changing the calculation of a surface's SDF. By combining these techniques with the noise functions introduced in section 4, we can create a new class of surfaces we call procedural surfaces.

Procedural surfaces are analogous to the bump mapping texturing technique. Bump maps use lighting effects to provide the appearance of surface variation, but may not always behave as intended because the underlying surface is not affected. With procedural surfaces, we can create more convincing effects by altering the surface directly. Additionally, we can render fractals and other complex functions by mapping values to an SDF over some defined set of inputs.

5.1 Displacement Surfaces

Sphere perturbations are simple - for some displacement function, we add its value to the SDF of a sphere

$$SDF = (p - C) - r + D$$

For some point $p(x, y, z)$, a sphere described by a center C and radius r, and a displacement value $D(x, y, z)$.

We implement two displacement surfaces. *Perlin Sphere* uses Perlin Noise (4.1.2) to evaluate D. *Perturbed Sphere* implements D as a sum of sine waves -

$$D(x, y, z) = c * \sin(\varphi + x) * \sin(\varphi + y) * \sin(\varphi + z)$$

Where c is some coefficient that modulates the intensity of displacement and φ is some constant phase shift.

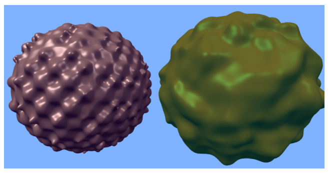

Figure 10: Diffuse Perturbed and Perlin Spheres

5.2 Fractals

Fractals are infinitely complex mathematical shapes, patterns that repeat forever and ever the more you zoom into an area of a fractal. This infinitely complex nature of fractals make it ideal for being rendered with ray marching instead of ray tracing. Not only is the ray marching of fractals easier to parallelize since each rendered pixel can be calculated separately unlike the more complex storage structures used in ray tracing. In addition, because the step lengths for ray marching isn't constant, we can use distance functions to render complex details such as the Mandelbulb fractal. The general equation for generating a Mandelbulb is shown below. Fractals, more specifically this mandelbulb structure, are very hard to render with traditional ray tracing techniques because of their lack of analytic intersections.

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\phi = \arctan \frac{y}{x} = \arg(x + yi)$$

$$\theta = \arctan \frac{\sqrt{x^2 + y^2}}{z} = \arccos \frac{z}{r}$$

$$\mathbf{v}^n := r^n \big\langle \sin\big(f(\theta,\phi)\big)\cos\big(g(\theta,\phi)\big), \sin\big(f(\theta,\phi)\big)\sin\big(g(\theta,\phi)\big), \cos\big(f(\theta,\phi)\big)\big\rangle$$

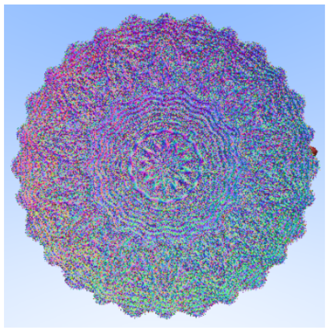

Figure 11: Mandelbulb rendered with 1 rays per pixel

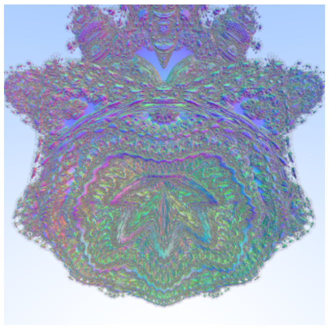

Figure 12: Mandelbulb rendered with 200 ray per pixel

It is worth nothing that for the more complex mandelbulb shapes, or more detailed geometric structures in general, a higher ray per pixel value is required otherwise the details of the fractal are obscured as pixelated noise. This increase in rays per pixel will impact the render time, Figure 10. took about 5 seconds to rende, while Figure 11. took a little over 2 hours.

6 PARALLELIZATION

Rendering an image is a computationally-intensive task and a ray marching rendering system requires more computational resources when compared with a ray tracing rendering system due to the simple fact that in ray tracing computation is done only at the point of object intersection whereas in ray marching computation is done at every iteration of the march. Thus, to improve the execution time of our renderer: we utilize the power of parallel computing with the help of CUDA programming and NVIDIA GPU.

6.1 Platform Used

Hardware specification: The CUDA program was tested on a 64-bit operating system with 16 GB of Ram and a AMD Ryzen 9 5900HX processor, along with a Nvidia GeForce RTX 3070 GPU (Laptop version).
Software Specification: The hardware ran on Windows 11 OS and we used Microsoft Visual Studio as our programming IDE to develop the CUDA program. We also used Nvidia Nsight Compute to profile our CUDA program to better understand the resource utilisation of the GPU while the program was running.

6.2 CUDA Programming Overview

In GPU programming, there are several parameters on both the software side and the hardware side that affect performance. Whenever a part of the program is outsourced to a GPU to be computed, On the software side: the total number of threads/ processors used are represented by a "Grid", which in our case will represent the image being rendered, therefore, total number of threads will be equal to the number of pixels in our image. These threads are further divided into blocks which are independent units of execution with no communication possible between different blocks.

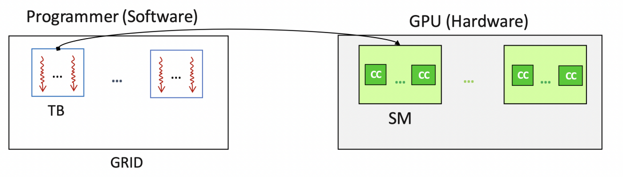

Figure 13: Diagram depicting software and hardware abstraction for GPU programming

Now on the hardware side: The total number of cores/processors in the GPU are divided between Streaming Multiprocessors (SMs) such that each block can be executed only on a single SM with the possibility of several blocks being executed concurrently on the same SM if the hardware resource allows it. Thus, fine-tuning parameters like block size for a image of specific resolution being renderer based on the resource utilisation of the hardware can lead to significant increase in SpeedUp.

$$SpeedUp = S/P \tag{8}$$

here, S: is the Program Serial Execution Time and P: is the Program Parallel Execution time

6.3 Parallel Algorithm

This section describes the algorithm/psuedo code run by each thread of the program.

Resolution	Block Size	Number of Samples	Parallel Time	Serial Time	Speed Up	Branch Efficiency	A/T Occupancy
256x256	8x8	8	0.027s	29.483s	1091.963	99.27	83.42/87.5
256x256	16x16	8	0.031s	29.483s	951.065	99.27	80.14/87.5
256x256	32x8	8	0.042s	29.483s	701.976	97.54	76.59/83.33
256x256	128x1	8	0.038s	29.483s	775.868	99.27	77.37/83.33
256x256	256x1	8	0.033s	29.483s	893.424	98.65	79.28/87.5
512x512	8x8	8	0.073s	117.817s	1613.932	99.38	82.74/87.5
512x512	16x16	8	0.081s	117.817s	1454.531	99.27	78.94/87.5
512x512	32x8	8	0.096s	117.817s	1227.260	99.27	72.41/83.33
512x512	128x1	8	0.092s	117.817s	1280.620	98.65	74.53/87.5
512x512	256x1	8	0.093s	117.817s	1266.849	98.65	74.68/87.5
1024x1024	8x8	8	0.191s	476.936s	2497.047	99.38	83.65/87.5
1024x1024	16x16	8	0.209s	476.936s	2281.990	98.65	81.12/87.5
1024x1024	32x8	8	0.243s	476.936s	1962.700	99.38	74.64/83.33
1024x1024	128x1	8	0.241s	476.936s	1978.988	98.65	75.23/83.33
1024x1024	256x1	8	0.244s	476.936s	1954.656	98.65	75.87/83.33

Table 1: In the above table, we list the different parameter configurations of our CUDA program and the SpeedUp achieved for those configurations. We also list the Branch Efficiency and Achieved/Theoretical Occupancy for each configuration.

Algorithm 1 : Algorithm executed by each Thread/Pixel

```
    $i \leftarrow ThreadIdx.x + BlockDim.x * BlockIdx.x$
    $j \leftarrow ThreadIdx.y + BlockDim.y * BlockIdx.y$
Ensure: $i < Image\_Width$ and $j < Image\_Height$
    $N \leftarrow No\_of\_Samples$
    $Color_{ij} \leftarrow 0$
    while $N \neq 0$ do
        $i+ = Random(-1, 1)$
        $j+ = Random(-1, 1)$
        $ray = Compute\_Ray(Camera\_Info, i, j)$
        $Color_{ij}+ = Compute\_Color(3D\_Scene, ray)$
        $N \leftarrow N - 1$
    end while
    $FrameBuffer[j, i] = Color_{ij}/No\_of\_Samples$
```

6.4 Experiments and Results

As discussed in section 6.2, several parameters can affect the execution time of our rendering system. The focus of our experiments will be to understand the range of the parameters for which we can achieve maximum speedup. The main parameters of focus will be: Block size (Number of Threads in each block), Grid size (Image Resolution). To better understand Resource utilization of our GPU based on these parameters we will also look at Branch Efficiency and Achieved Occupancy.

Branch Efficiency is the ratio of similar work done by threads in a block to the total work done by threads in a block. (Note: Threads in a block run concurrently only if they do similar work, thus, this ratio should be high to achieve maximum performance). Achieved Occupancy is the ratio of number of active processors to the total number of processors in a Streaming Multiprocessor. (Note: This ratio should be as close to theoretical occupancy as possible to achieve maximum performance).

Thus, for a range of parameter values the Program Parallel Execution Time, Program Serial Execution Time and SpeedUp can be seen in Table 1. By analysing the results in Table 1, we can see that SpeedUp scales with Image Resolution (with the highest SpeedUp achieved being ≈ 2500x), for any Image Resolution Block Size of '8x8' yield highest SpeedUp. Branch Efficiency remains high for any parameter configuration, therefore, divergent behaviour in our program is bare minimum. Although the Theoretical Occupancy is not close to 100% due to the register utilization of our CUDA program, the achieved occupancy for various parameter configuration remains close to theoretical occupancy, thus extracting high performance benefit from the GPU.

7 CONCLUSION

Thus, In this project we developed a Ray Marching based rendering system with several features which are easily executed using the concepts of Ray Marching and Signed Distance Function as compared to other similar rendering techniques. These features include: Constructive Solid Geometry, Procedural Generation of 3D clouds, Displacement Surfaces, Fractal patterns. We also developed several material types and surfaces that our system is capable of rendering. At last, we utilized the power of parallel computing via Nvidia GPU and CUDA programming to achieve a 2500x faster execution time of our rendering system when compared with its serial execution time.

REFERENCES

[1] https://raytracing.github.io/books/RayTracingInOneWeekend.html
[2] Geoffrey Y. Gardner. Visual Simulation of Clouds. SIGGRAPH ‘85
[3] Ken Perlin. 1985. An Image Synthesizer. SIGGRAPH ‘85
[4] https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda/
[5] https://www.skytopia.com/project/fractal/mandelbulb.html
[6] https://iquilezles.org/articles/distfunctions/
[7] http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/
[8] https://michaelwalczyk.com/blog-ray-marching.html