Introduction to Parallel Computing - Homework 2

1. Goal

In this assignment, you are asked to implement a simplest Ray Marching algorithm to render the "Classic" Mandelbulb and parallelize your program with MPI and OpenMP libraries. We hope this assignment helps you to learn the following concepts:

- The differences between **Static Scheduling** and **Dynamic Scheduling**.
- The importance of **Load Balance**.
- It is very slow and difficult to use a single core to render a high-quality 3D image, not to mention rendering an entire 3D movie, so designing an efficient parallel algorithm is important.

For more information about Mandelbrot Set and Mandelbulb, please refer to Appendix A.

2. Problem Description

We provided you a sequential Mandelbrot rendering program, and you are asked to parallelize it.

Input / Output Format

The program accepts 10 parameters:

```
./executable $num_threads $x1 $y1 $z1 $x2 $y2 $z2 $width $height $filename
```

- $num_threads — number of threads per process [integer]
- $x1 $y1 $z1 — camera position [three doubles, -5.0 ~ 5.0]
- $x2 $y2 $z2 — camera target position [three doubles, -5.0 ~ 5.0]
- $width $height — size of the image [positive integer]
- $filename — the file name of the output PNG image [string, xxx.png]

The output should be a 32bit PNG image with RGBA channels.

Parameters

These are the parameter settings for the sequential implementation. If you have any better settings that produces the same results as the sequential version for any input parameters, please write in your report and give some explanations.

- power — 8.0
  - The power of the equation
- md_iter — 24
  - The mandelbulb's maximum iteration count
- ray_step — 10000
  - The maximum step count of ray marching
- shadow_step — 1500
  - The maximum step count of shadow casting
- step_limiter — 0.2
  - The limit length of each step when casting shadow
- ray_multiplier — 0.1
  - A multiplier for ray marching to prevent over-shooting
- bailout — 2.0
  - The escape radius
- eps — 0.0005
  - The precision of the rendering calculation
- FOV — 1.5
  - Field of view
- far_plane — 100
  - The maximum depth of the scene

Libraries

These are the libraries used for the sequential version. All these libraries are already installed on apollo. Please refers to TA's sample code and `build.ninja`.

1. lodepng — loading/saving PNG images. [C/C++]
   GitHub – lvandeve/lodepng: PNG encoder and decoder in C and C++
2. GLM — vector/matrix arithmetic. [C++]
   GitHub – g-truc/glm: OpenGL Mathematics (GLM)

Verify Your Results

We provide a simple script for you to verify your rendering results. Type this command to compare two PNG pictures, and see how many pixels are matched or mismatched:

```
$ /home/ipc20/ta/hw2/hw2-diff a.png b.png
```

```
$ /home/ipc20/ta/hw2/runner.py xx.txt ./bin
```

3. Report

Report must contain the following contents, and you can add more as you like. You can write in either **English** or **Traditional Chinese**.

1. Name, Student ID
   - ✧ Name — your name
   - ✧ Student ID — your student ID
2. Implementation

   Explain your implementation, especially in the following aspects:
   - ✧ How do you implement your program, what scheduling algorithm did you use — static, dynamic, guided, etc.?
   - ✧ How do you partition the task?
   - ✧ What techniques do you use to reduce execution time?
   - ✧ Other efforts you make in your program
3. Analysis
   - ✧ Design your own plots to show the load balance of your algorithm between threads/processes.
   - ✧ If you have modified the default parameter settings, please also compare the results of the default settings and your settings.
   - ✧ Other things worth mentioning.
4. Conclusion

   Your conclusion of this assignment.
   - ✧ What have you learned from this assignment?
   - ✧ What difficulty did you encounter in this assignment?
   - ✧ Any feedback or suggestions to this assignment or spec.

4. Grading

1. Correctness (50%)
   - ✧ Your implementation must finish each case in 10 minutes with 8 cores or more.
   - ✧ TA will use the hidden testcases and the scripts mentioned above to test your implementation.
   - ✧ You will get 0 score if your implementation cannot get 95% pixels correct for each testcase. If you have more than 95% pixels correct, the score for each testcase is given as:

$$\left[\min\left(1, \frac{number\ of\ correct\ pixels}{number\ of\ pixels} \times \frac{1000}{996}\right)\right]^6 \times score\ for\ the\ testcase$$

   You are expected to have at least 99.6% pixels correct.

For example, if you have at least 99.6% pixels correct, you get full points.
If you have 98% pixels correct, you get about 90.7% score for the testcase.

2. Performance (20%)
   - ✧ TA will use multiple cases to test your program, each case will be run many times and the median execution time will be used to grading. Your program must produce the right answer to get performance points.
3. Report (30%)
   - ✧ Grading based on your evaluation, discussion and writing.

5. Submission

Please upload your code and report to iLMS without compression.
You should name your file as:

- `hw2.cc`
- `build.ninja`
- `report.pdf`

Note

Your executable file produced by the ninja command must be named as `hw2`

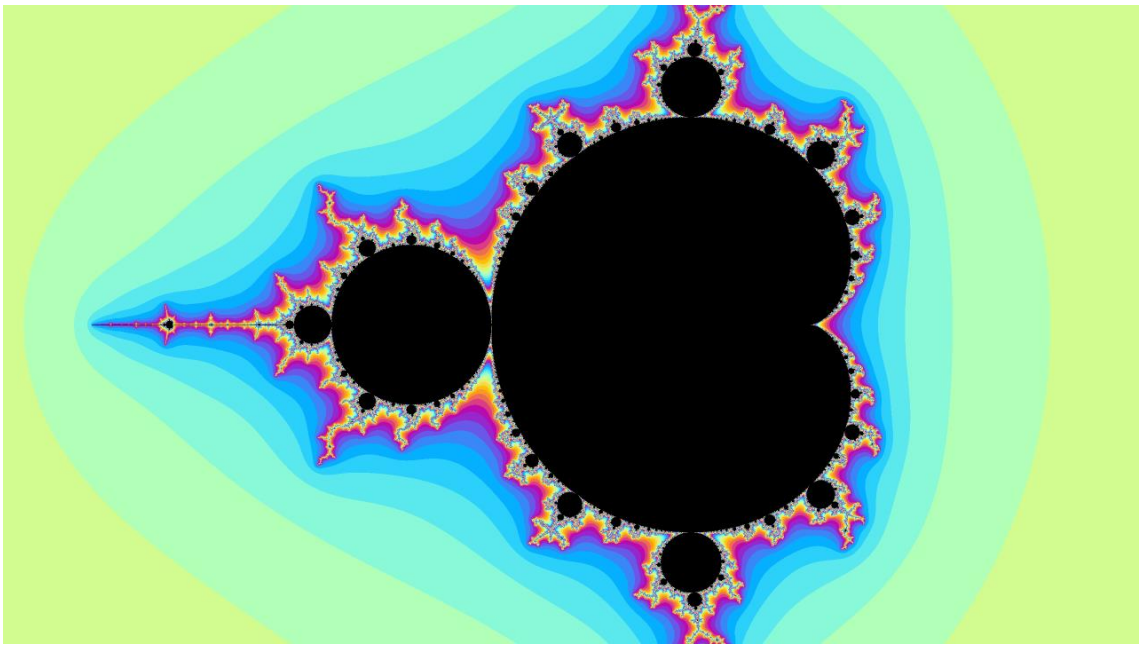
6. Reminder

- Please submit your homework before 2020/04/13 23:59
- You can check your performance on the scoreboard (iLMS)
- Any questions about this assignment? Please ask on iLMS

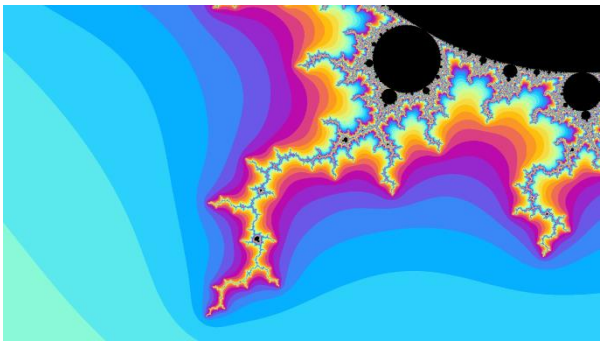
Before we talk about what is Mandelbulb, we must first understand the Mandelbrot set.

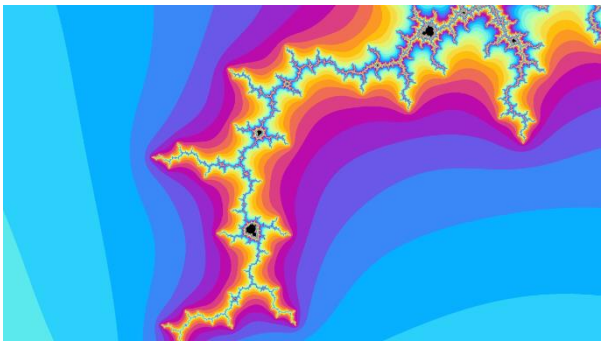
What is Mandelbrot set?

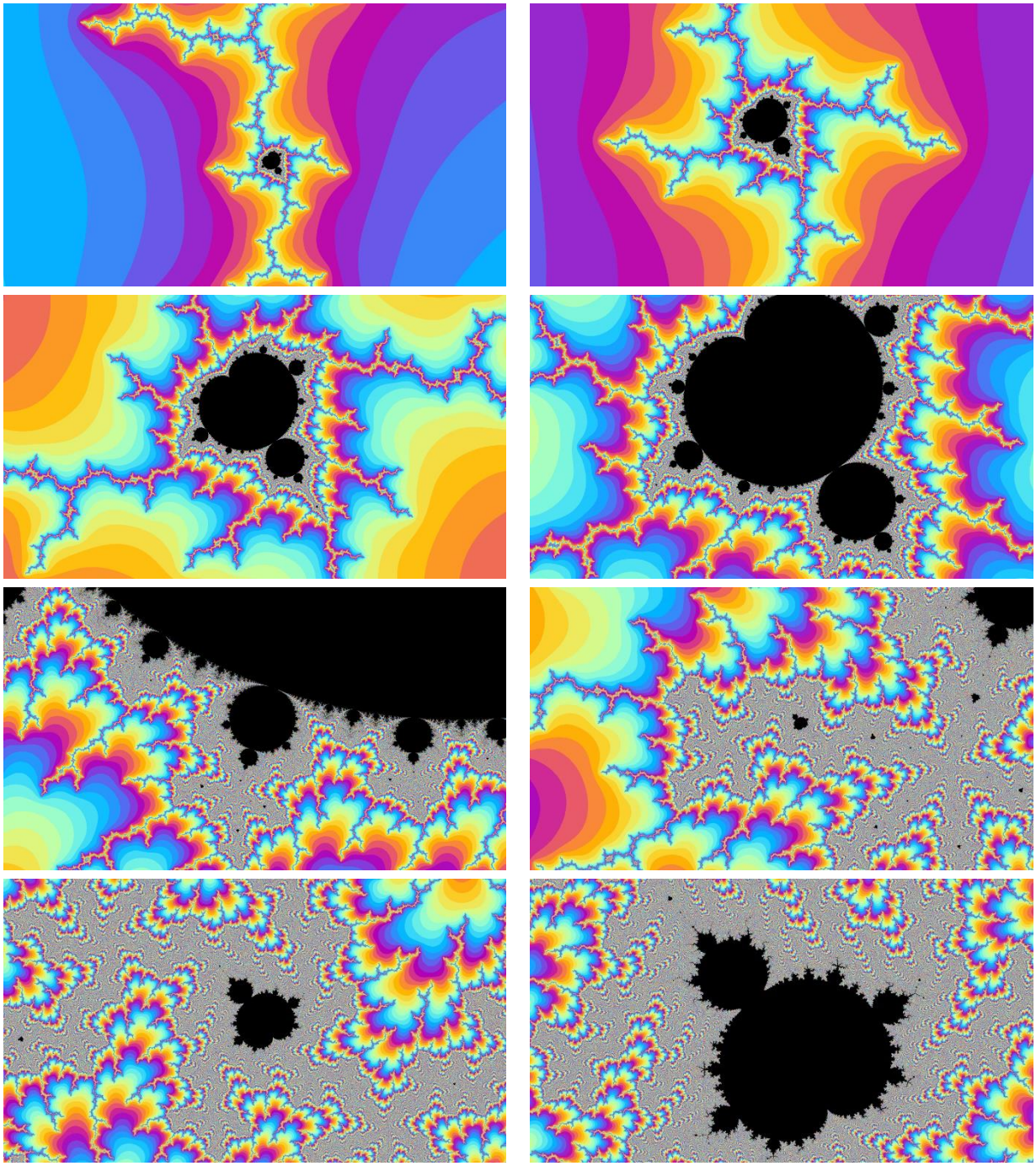
Mandelbrot set is the set of complex numbers c for which the quadratic recurrence equation $z_{k+1} = z_k^2 + c$ does not diverge when iterated from $z_0 = 0$. When plotting the Mandelbrot set, we need to convert each pixel to the corresponding coordinates on the complex plane, then plug them into the iterative function until either (a) the absolute value of the result exceeds the escape radius called "Bailout"—that means the inequality $|z_k| \leq 2$ must hold, otherwise it escaped —or (b) the iteration count achieves the given limitations. If a point does not "escape" after this procedure, then it belongs to the Mandelbrot set. As such, we can color the pixels according to the iteration count as shown in the image below.



If we scale up the image, we can see the Mandelbrot set has self-similar patterns in a smaller scale.



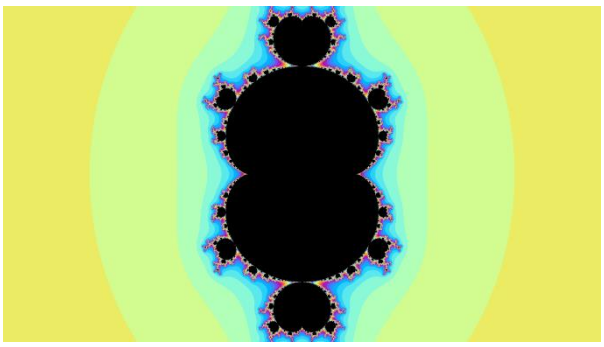




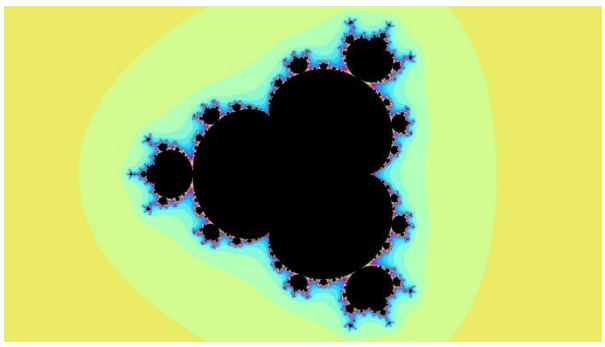
Objects that have a similar phenomenon to this are called “fractals” —therefore we can say the Mandelbrot set is a 2D fractal.

Another interesting thing is that if we change the power of the equation, we can get something like this:

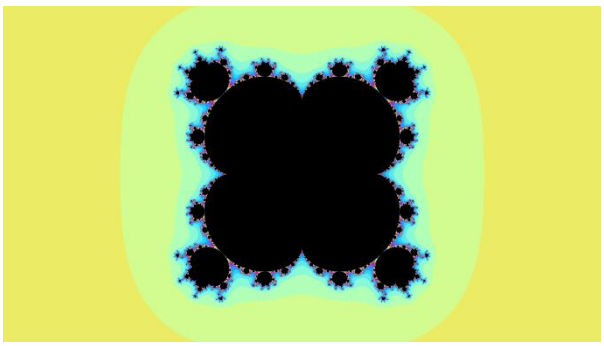
Power = 3



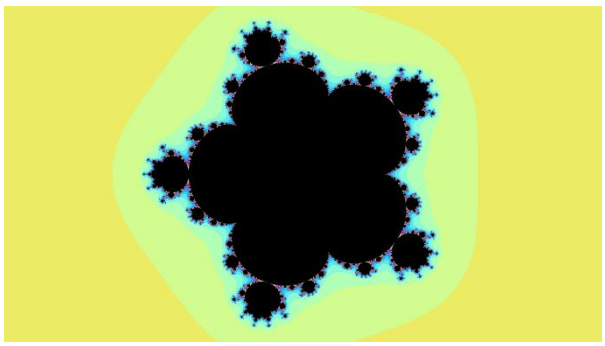
Power = 4



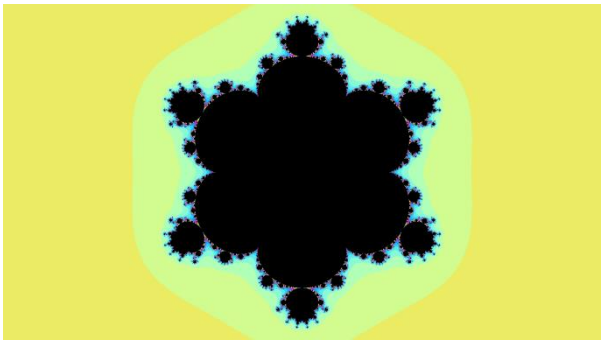
Power = 5



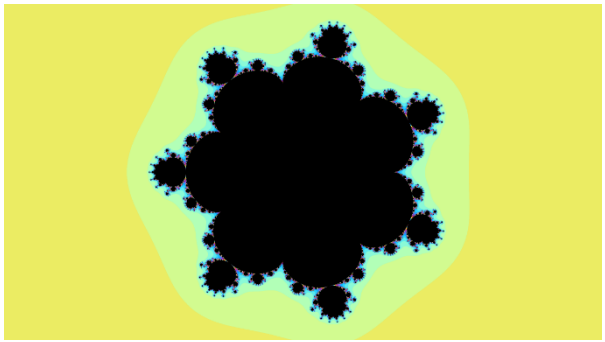
Power = 6



Power = 7



Power = 8



What is Mandelbulb ?

The Mandelbulb is a 3D fractal. Instead of using the 2D space of complex numbers, the mandelbulb uses spherical coordinates to represent its 3D space. Thus, we can rewrite the formula as such:

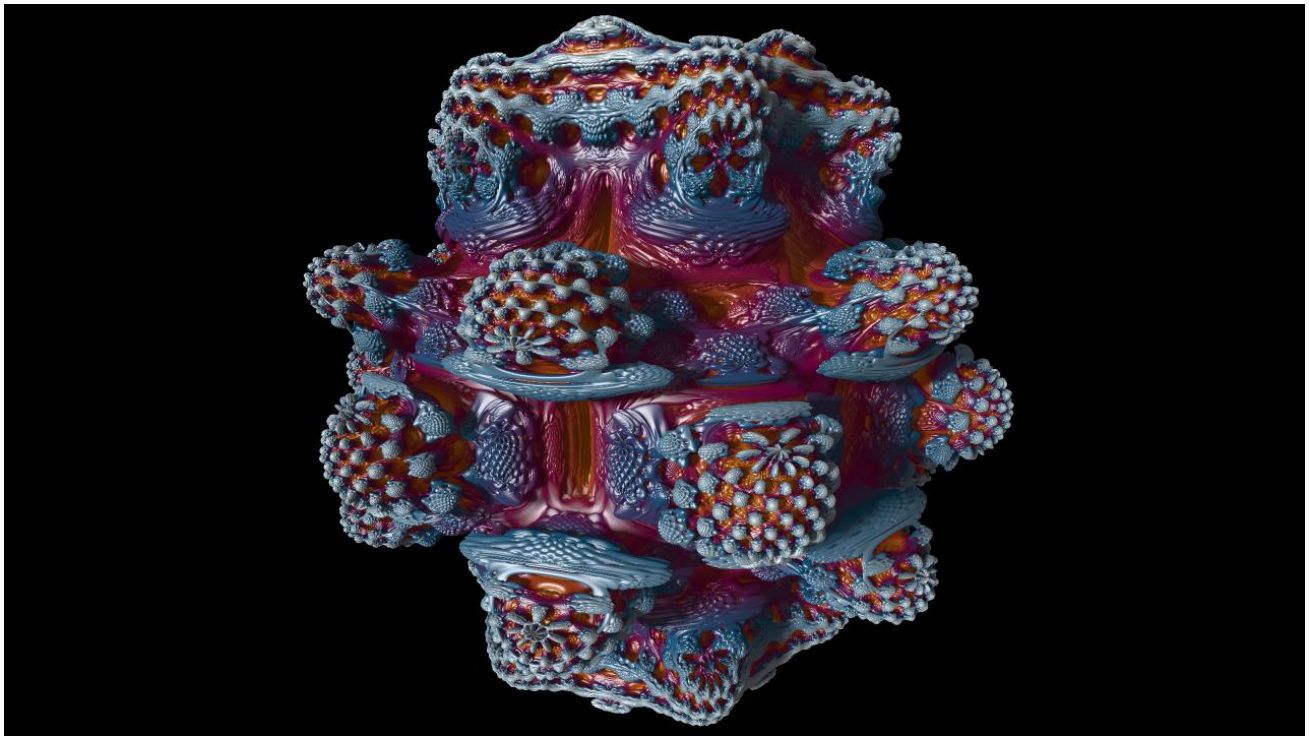
$$v_{k+1} = v_k^n + c$$

Where the nth power of vector $v = \langle x, y, z \rangle$ in $\mathbb{R}^3$ is:

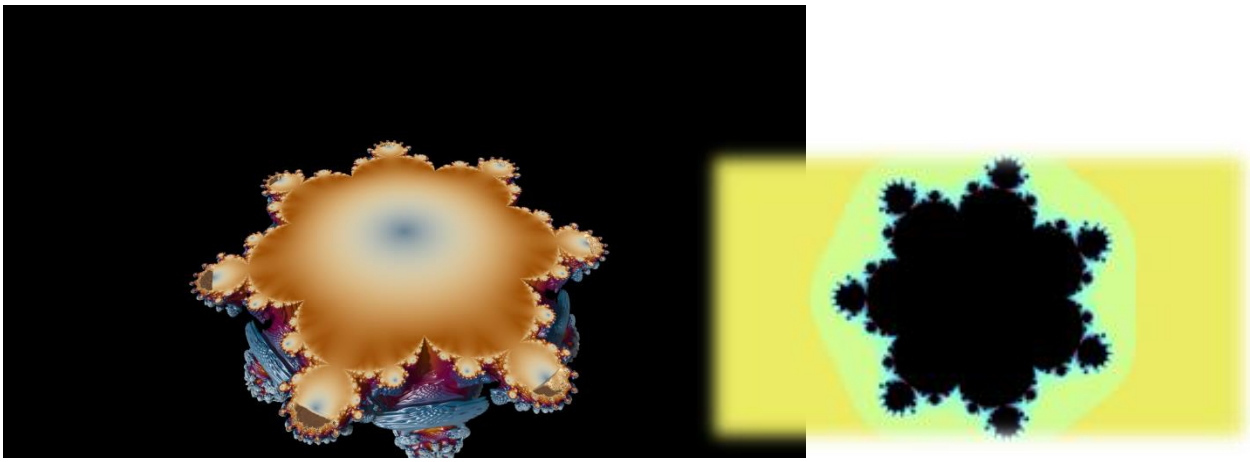
$$v^n := r^n \langle \cos(n\theta)\cos(n\phi), \cos(n\phi)\sin(n\theta), -\sin(n\phi) \rangle$$

$$\begin{cases} r = \sqrt{x^2 + y^2 + z^2} \\ \theta = \arctan(\frac{y}{x}) \\ \phi = \arcsin(\frac{z}{r}) \end{cases}$$

Modifying the power n of the Mandelbulb doesn't change much. Typically, when we mention the Mandelbulb, we refer to the Mandelbulb with a power of 8.



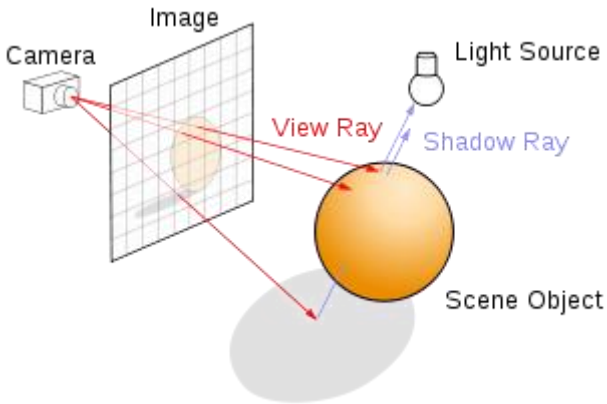
If we cut the Mandelbulb along the $y = 0$ plane, we can see the section has the same shape as a power 8 mandelbrot set.



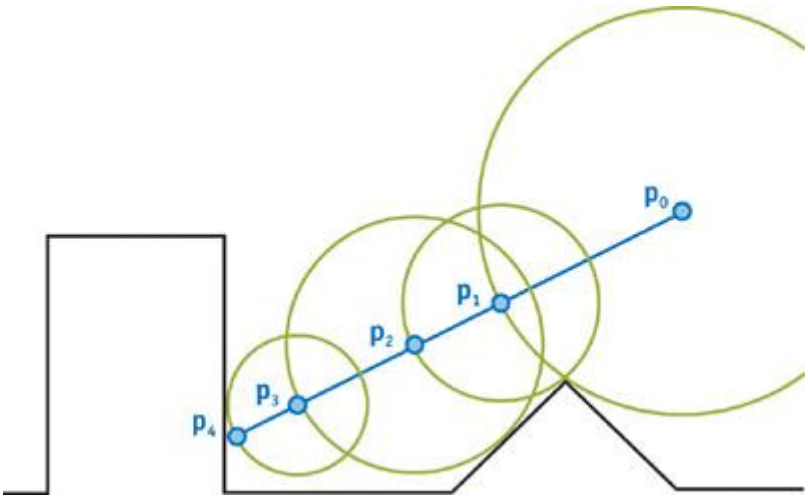
The rendering procedure is similar to the 2D Mandelbrot set: we plug each 3D point into the equation until the same condition is satisfied. However, to render 3D space onto the 2D screen, we need another rendering method called "ray marching".

What is Ray Marching?

Ray marching is a kind of ray tracing algorithm. As opposed to the common 3D-rendering method that uses a geometry definition (eg. Vertices, triangles, surfaces) to describe 3D objects in 3D space, ray marching renders objects by casting a ray for each screen pixel, then uses a mathematical function called **Distance Function** (or **Distance Estimator**) to verify if the ray intersects with any objects. If so, then we draw the color onto this pixel.



In this diagram, we cast the ray (View Ray) for each image pixel from the camera position, and verify if it intersects with the sphere object in this scene.



The Distance Function tells us the minimum distance from the current ray position to the object's surface. Let's define it as:

$$r = f(x, y, z)$$

Where r represents the minimum distance from point (x, y, z) to the surface of map function f.

First, we cast the ray from the camera position p_0 and calculate minimum distance r_0 — this means that the distance that the ray travels (until it hits the object's surface) also has at

least r_0 distance. The next step is to move the ray to the next point $p_1 = p_0 + r_0 \cdot \widehat{rd}$, where $\widehat{rd}$ is a unit vector in the direction of the ray. We then repeat this procedure on points $p_2, p_3, p_4, \ldots$ until either the minimum distance is smaller than some given threshold, or the iteration count achieves the given limitations.
Here is the pseudo code:

```
ro = camera position
rd = ray direction  // (unit vector)
td = 0              // total distance
mx = 100            // max iteration
st = 0.75           // step multiplier to prevent over-shooting
for i=0 to mx:
    r = map(ro + rd*td) // calculate minimum distance of current point
    if absolute of r < threshold then   // if hit the surface
        return hit surface
    else
        td += r * st  // move to the next point
return miss
```

What does the distance function look like?

There exist two kinds of distance functions:

- Signed Distance Function
- Unsigned Distance Function

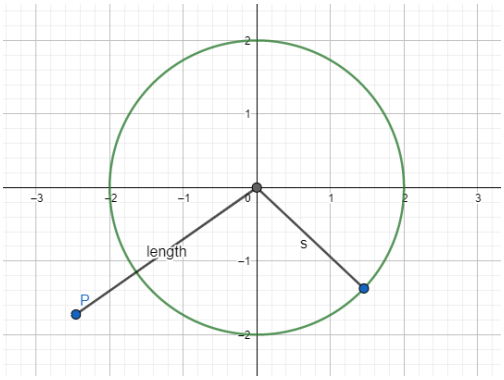
The first will return a signed distance from the function. A positive value means that the point is outside the object, while a negative value means the point is inside the object. The latter function simply returns a positive value wherever the point is.

Here I take a sphere for example — we can simply define its signed distance as:

$$r = f(x, y, z) = \sqrt{x^2 + y^2 + z^2} - \mathrm{s}$$

where s is the radius of the sphere.

Then we can get the distance from point p to the surface.



What does the Mandelbulb's distance function look like?

Here, I've chosen to only write down the result, as the proof is too tedious. You may read the following article written by Syntopia if you would like to learn more:

Distance Estimated 3D Fractals (V): The Mandelbulb & Different DE Approximations

The approximate distance function of the mandelbulb is:

$$DE = \frac{0.5r\ln(r)}{dr}$$

Where $r = |v_k|$ and $dr = |v_k'|$.

We can get dr by scalar derivative $dr_{k+1} = n|v_k|^{n-1}dr_k + 1$ and $dr_0 = 1$

Finally

Let's sum up everything we've learned:

- Mandelbulb equation: $v_{k+1} = v_k^8 + c$
  - c is the 3D point
  - $v_0 = 0$
  - $\mathrm{v}^{\mathrm{n}} := \mathrm{r}^{\mathrm{n}}\langle \cos(n\theta)\cos(n\phi), \cos(n\phi)\sin(n\theta), -\sin(n\phi)\rangle$

$$\begin{cases} r = \sqrt{x^2 + y^2 + z^2} \\ \theta = \arctan(\frac{y}{x}) \\ \phi = \arcsin(\frac{z}{r}) \end{cases}$$

- Mandelbulb Distance Function: $DE = 0.5r\ln(r) \,/\, dr$
  - $r = |v_k|$ and $dr = |v_k'|$
  - $dr_{k+1} = n|v_k|^{n-1}dr_k + 1$
    - $dr_0 = 1$

Pseudo code:

```
map(x, y,z):
    c = vec3(x, y, z)  // current position (a 3D vector)
    v = c
    dr = 1
    r = length(v)
    mx = 100  //max iteration

    for i=0 to mx:
        theta = atan(v.y/v.x) * power
        phi = asin(v.z/r) * power
        dr = power * r^(power-1)*dr + 1 //dr_k+1
        v = c + r^power *
            vec3(cos(theta)*cos(phi),
                 cos(phi)*sin(theta),
                 -sin(phi))  //v_k+1
        r = length(v)  //r_k+1
        if r > bailout then
            break
    DE = 0.5 * log(r) * r / dr
    return DE
```

If you are still confused how to implement this, don't worry about that! The very nice (and exceedingly handsome) TAs will provide a sequential code for this.

Other Useful References

- Modeling with Distance Functions – Inigo Quilez
  - More distance functions of primitives and operations.
- Ray Marching and Signed Distance Functions – Jamie Wong
  - More details on ray marching.
- Mandelbulb: The Unravelling of the Real 3D Mandelbrot Fractal
  - The history of the mandelbulb
- Mandelbulb Gallery by Daniel White
  - Mandelbulb Gallery
- Summary of 3D Mandelbrot Set Formulas – Fractal Forums
  - Do you remember I said "Classic" mandelbulb? This site summarized all the mandelbulbs' formulas. You can use them in your design as well, but please specify in your report.