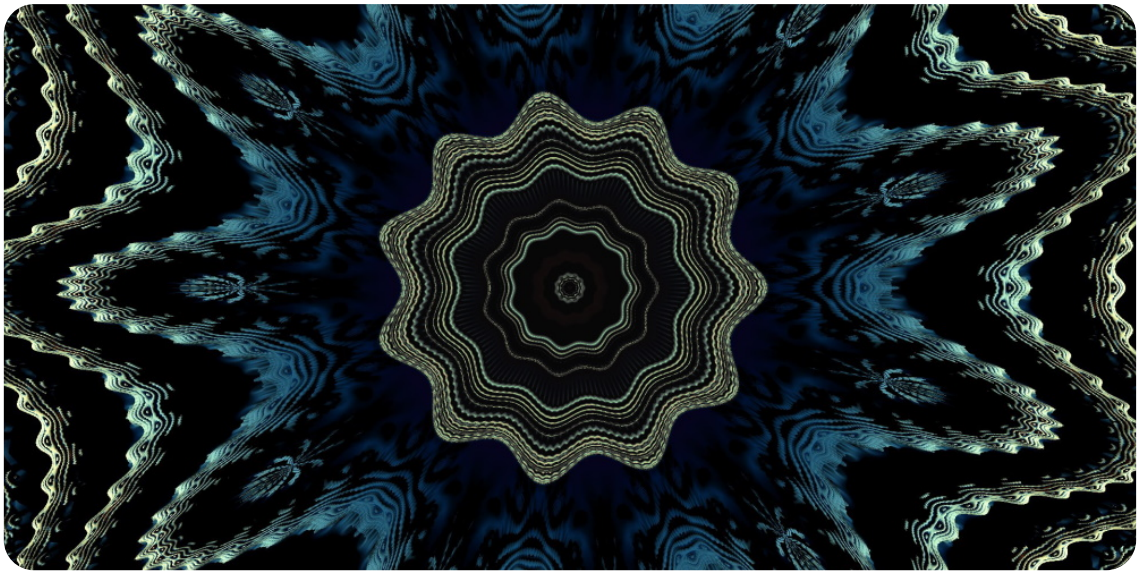
MANDELBULB



24. Juli 2022

Im Rahmen des Moduls Computergrafik habe ich mich an der Uni mit OpenGL und diversen Renderingtechniken bzw. Shaderprogrammierung beschäftigt. Um zu zeigen, dass wir die erlernten Kenntnisse anwenden können, mussten wir eine Belegarbeit zu dem Thema anfertigen. Ich habe mich dabei für das Rendern von 3D Fraktalen entscheiden, da dies mathematische, grafische als auch programmiertechnische Aspekte aufweist.

Vorbereitung und Theorie

Komplexe Zahlen
Es gibt viele Fragen, die man sich zum Anfang stellen kann: Was sind komplexe Zahlen? Wofür brauche ich komplexe Zahlen und warum brauche ich sie für das Darstellen von einem Mandelbulb? All diese Fragen habe ich mir am Anfang meiner Recherche gestellt und ich werde sie in den nachfolgenden

Absätzen Stück für Stück beantworten.

Die Frage warum komplexe Zahlen für den Mandelbulb benötigt werden ist eigentlich ganz einfach zu beantworten: Der Mandelbulb ist über eine komplexe Hyperebene definiert, d.h. um ein Mandelbulb darstellen zu können muss mit komplexen Zahlen gerechnet werden.

Der Grund warum komplexe Zahlen für die Berechnung des Mandelbulbs notwendig sind, ist mit folgender Beispielaufgabe schnell klar: *"Berechnen Sie die Wurzel aus -4"*. Intiutiv würde man wahrscheinlich -2 beantworten, jedoch ist $(-2)^2 = 4$, da minus mal minus gleich plus ergibt. Man kann also keine Wurzel aus negativen Zahlen ziehen. Oder doch?
Angenommen man legt fest, dass i^2 gleich -1 ergibt:

$$i^2 = -1$$

Dann würde die Wurzel aus -4 gleich $2i$ ergeben:

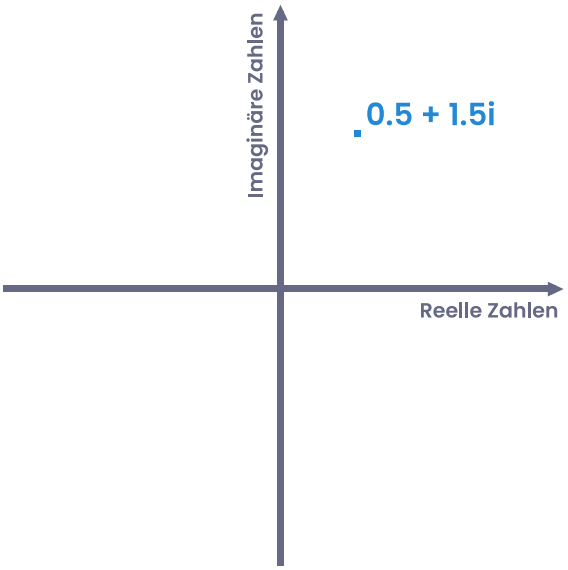
$$\sqrt{-4} = \sqrt{4} \cdot \sqrt{-1}$$

$$\sqrt{-4} = \sqrt{4} \cdot \sqrt{i^2}$$

$$\sqrt{-4} = 2i$$

Definition komplexe Zahl
Komplexe Zahl = Realteil + Imaginäre Einheit · Imaginärteil

$$z = 0.5 + 1.5i$$



Das Mandelbrot Set
Der Mandelbulb ist eine Repräsentation des Mandelbrot Sets in 3D, das bedeutet, um zu verstehen, wie der Mandelbulb funktioniert muss erst geklärt werden, wie das Mandelbrot Set funktioniert. Das Mandelbrot Set ist eine

Menge von komplexen Zahlen, welches nicht gegen unendlich divergiert. Divergenz bedeutet, dass eine Funktion immer weiter ins Unendliche geht. Konvergenz wiederrum bedeutet, dass eine Funktion sich einem bestimmten Wert annähert und diesen im Unendlichen erreicht.

Beispiel:

$f(x) = e^x \Rightarrow$ divergiert: $lim_{x \to \infty}\ f(x) = \infty$

$g(x) = \frac{1}{x} \Rightarrow$ konvergiert: $lim_{x \to \infty}\ f(x) = 0$

Mandelbrot Algorithmus

Die Formel für die Berechnung des Mandelbrot Sets ist:

$z_{k+1} = {z_k}^2 + c$

Der Algorithmus basiert auf dem Konzept von Rekursion, d.h. die Funktion ruft sich solange selbst auf, bis eine Abbruchbedingung erreicht ist. In diesem Fall wäre das eine vordefinierte Anzahl an Durchläufen (Iterationen).

Ablauf:
1. Belibiegen Punkt auf komplexer Ebene wählen (z.B. 0.5 + 1.5i)
2. Punkt für c in Formel einsetzen
3. Ergebnis als neues z definieren und 2. wiederholen

Beispiel: Punkt $(0.5 + 1.5i)$

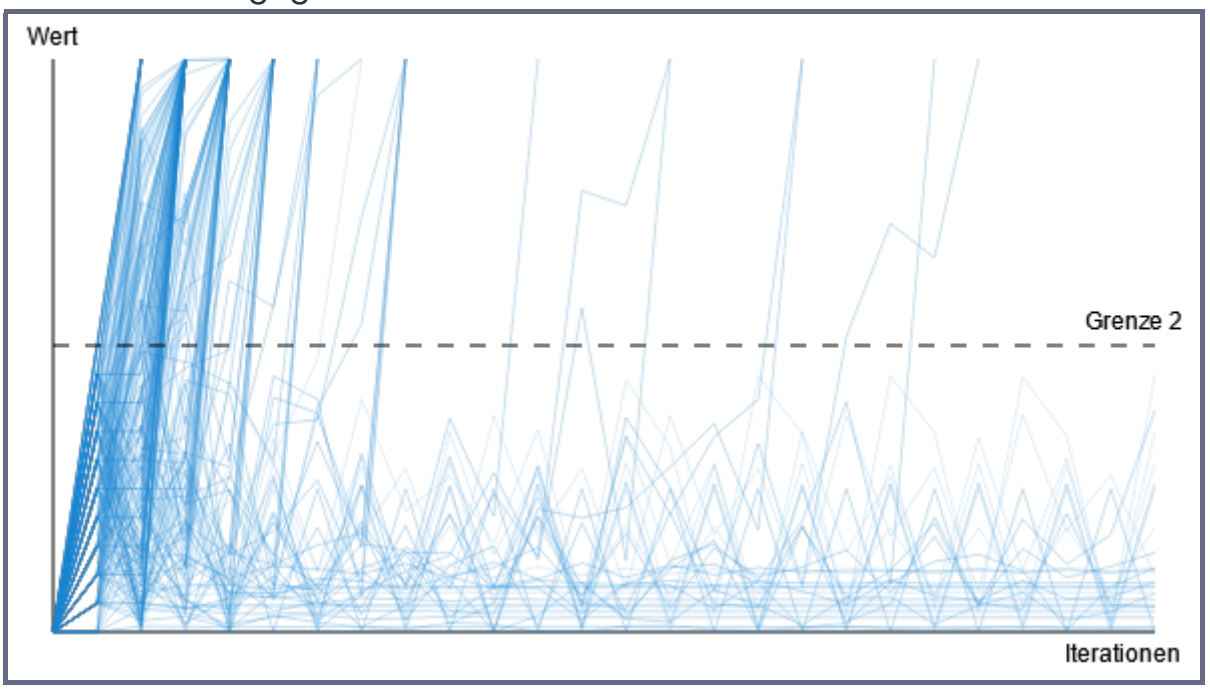
$z_0 = 0 + 0i$

$$
\begin{aligned}
z_1 = {z_0}^2 + c &= (0 + 0i)^2 + 0.5 + 1.5i && = 0.5 + 1.5i \\
z_2 = {z_1}^2 + c &= (0.5 + 1.5i)^2 + 0.5 + 1.5i \\
&= (0.5^2 + 2 \cdot 0.5 \cdot 1.5i + [1.5i]^2) + 0.5 + 1.5i \\
&= 0.25 + 1.5i - 2.25 + 0.5 + 1.5i && = -1.5 + 3i \\
z_3 = {z_2}^2 + c &= (-1.5 + 3i)^2 + 0.5 + 1.5i \\
&= ([-1.5]^2 - 2 \cdot 1.5 \cdot 3i + [3i]^2) + 0.5 + 1.5i \\
&= 2.25 - 9i - 9 + 0.5 + 1.5i && = -6.25 - 7.5i
\end{aligned}
$$

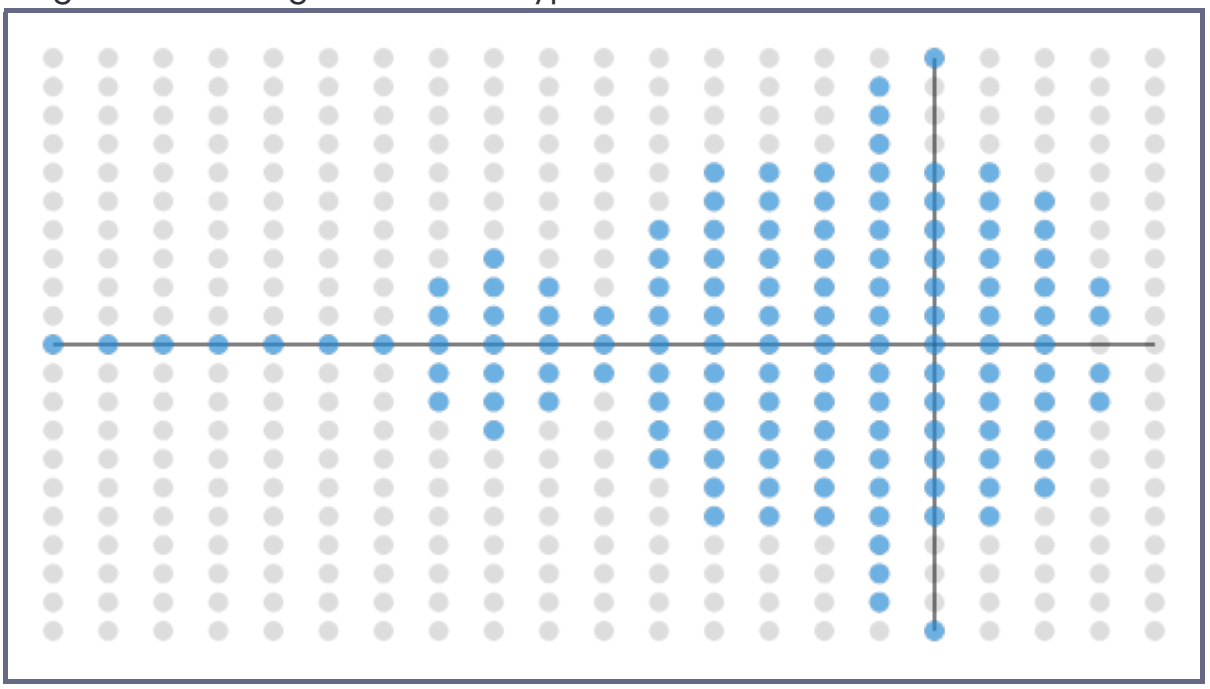
$z_4 = \dots$

Das Beispiel oben zeigt, dass der gewählte Punkt schnell aus dem Werteberech des Mandelbrot Sets verschwindet. Er divergiert gegen unendlich ist also kein Punkt des Sets. Als Orientierung wurde für Mandelbrot die Grenze 2 festgelegt, d.h. wenn der Punkt nicht in dieser Grenze bleibt, dann ist er kein Teil vom Mandelbrot Set. Wenn man sich jetzt 400 Beispielpunkte im Bereich von -2 bis +2 anschaut erhält man folgende Grafiken:

Hier sieht man den Verlauf der Werte von den einzelnen Punkten. Man kann sehr schön erkennen, dass manche Punkte über die Grenze von 2 kommen und andere hingegen unterhalb der Grenze bleiben:



Eine andere Darstellungsmöglichkeit ist die Punke auf der komplexen Ebene einzutragen. Hier habe ich die Punkte, welche unterhalb der Grenze liegen blau eingefärbt. Das ergibt dann das typische Muster des Mandelbrot Sets:

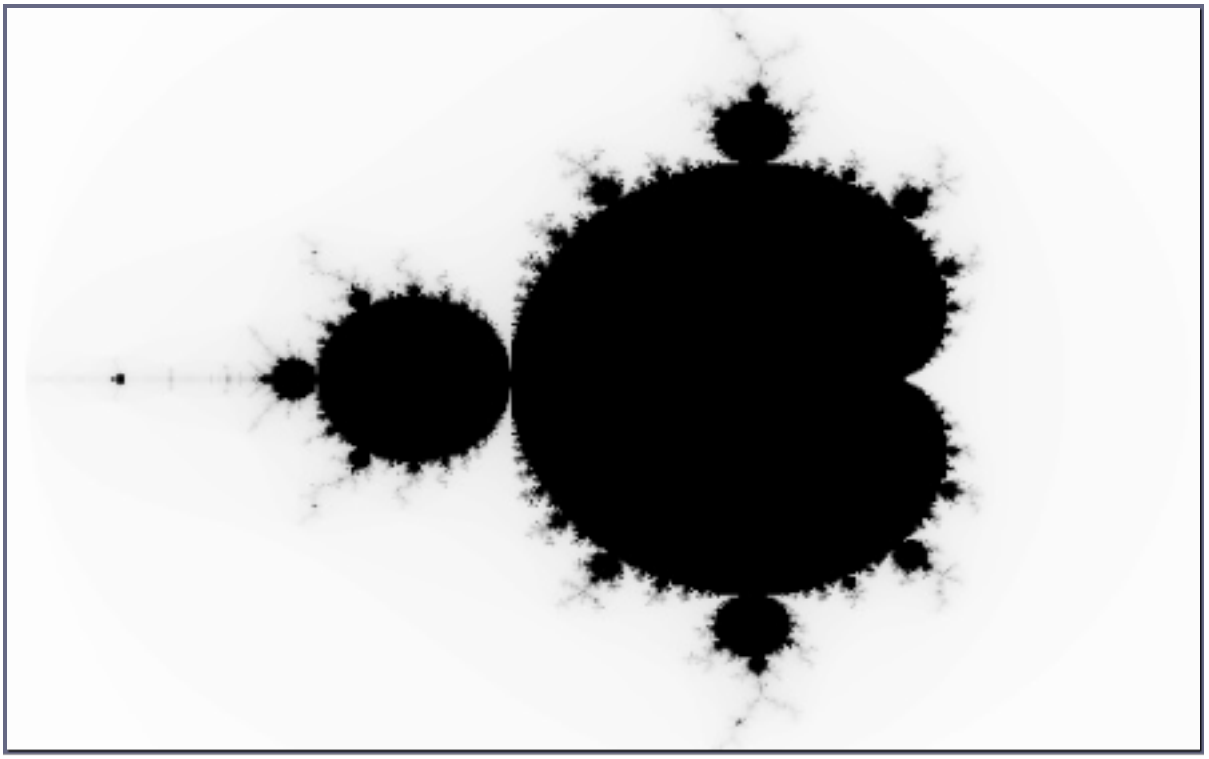


Rendern des Mandelbrot Sets

Implementierung
Da jetzt alle theoretischen Grundlagen gegeben sind, kann man das Mandelbrot Set ganz einfach rendern. Dafür habe ich einen Fragment Shader gebaut:

```glsl
void main() {
    vec2 uv = gl_FragCoord.xy / RESOLUTION;

    float x = uv.x * 3.65 - 2.45;      // Wertebereich festlegen
    float y = uv.y * 2.24 - 1.12;      // Wertebereich festlegen

    float z = 0.0;      // Realer Teil
    float c = 0.0;      // Imaginärer Teil
    int i = 0;
    int iterations = 150;

    for (i = 0; i < iterations; i++) {
        float tmp = x + z*z - c*c;
        c = y + 2.0*z*c;
        z = tmp;

        if (abs(z) >= 2.0) break;
    }

    vec3 color = vec3(float(i) / float(iterations));      // Auf colo
    gl_FragColor = vec4(color, 1.0);
}
```

Das Ergebnis nach dem Rendern von 150 Iterationen sieht dann ungefähr so aus:

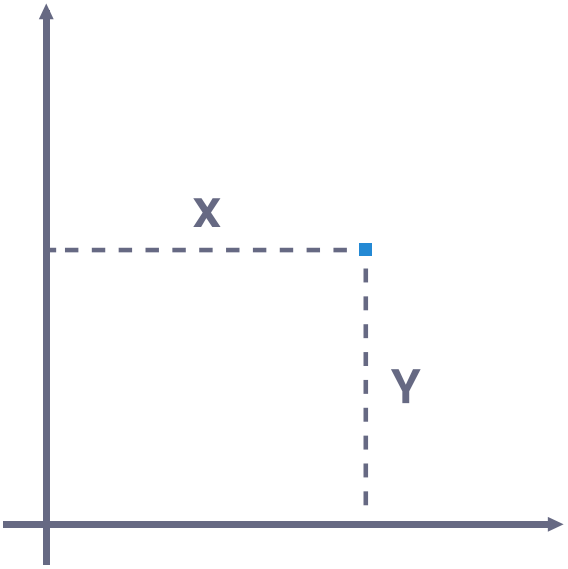


Der Mandelbulb

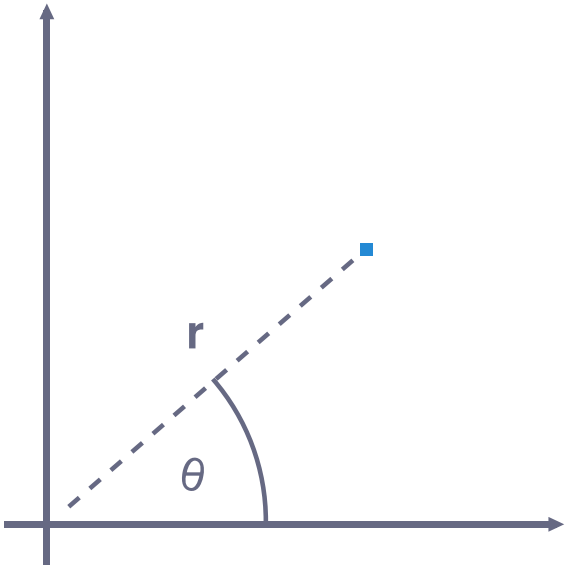
Komplexe Zahlen in 3D

Der Mandelbulb ist ein dreidimensionales Fraktal, das bedeutet die Darstellung basiert auf komplexen Zahlen in 3D. Hier wird mit sogenannten triplexen Zahlen oder allgemeiner hyperkomplexen Zahlen gerechnet. Um zu verstehen, wie die Berechnung letztendlich erfolgt, muss erst geklärt werden, was sphärische Koordinaten sind. Mit diesen werden die Punkte des Mandelbulbs definiert.

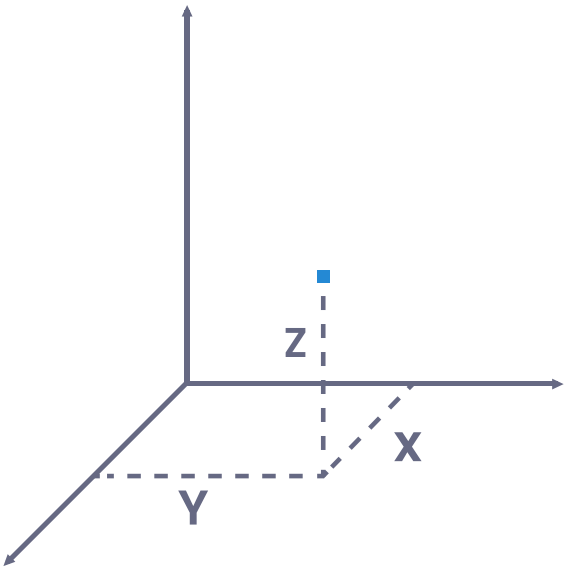
Kartesische Koordinaten (2D)



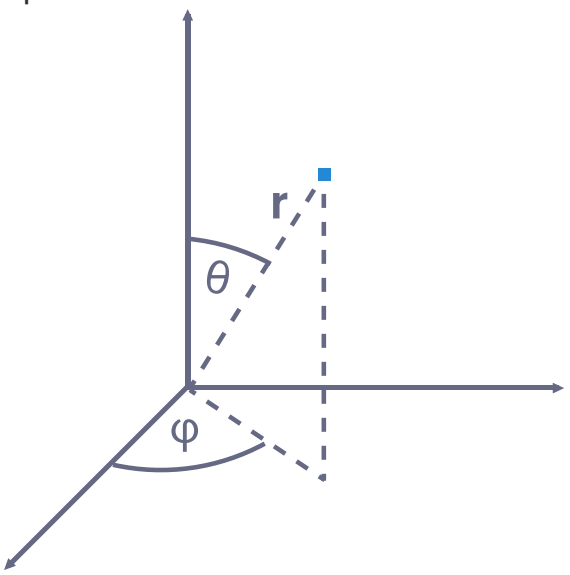
Polare Koordinaten



Kartesische Koordinaten (3D)



Sphärische Koordinaten



Der primäre Unterschied von kartesischen zu sphärischen Koordinaten ist also die Verwendung von Winkeln bei sphärischen Koordinaten.

Die Formel für den Mandelbulb

Bereits bekannt von dem Mandelbrot Set ist die Formel $z_{k+1} = {z_k}^2 + c$. Die Formel für den Mandelbulb unterscheidet sich nicht viel:

$$z_{k+1} = {z_k}^n + c$$

Die Frage, die sich daraus ergibt ist: Wie nehme ich eine triplexe Zahl zu der n-ten Potenz? Daniel White und Paul Nylander haben für dieses Problem eine Formel entwickelt, mit der ein Vektor v mit beliebigen Exponenten n potenziert werden kann:

$$v^n := r^n \begin{pmatrix} sin(n\theta)cos(n\phi) \\ sin(n\theta)sin(n\phi) \\ cos(n\theta) \end{pmatrix}$$

mit

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

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

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

Rendern von unendlich detallierten Objekten

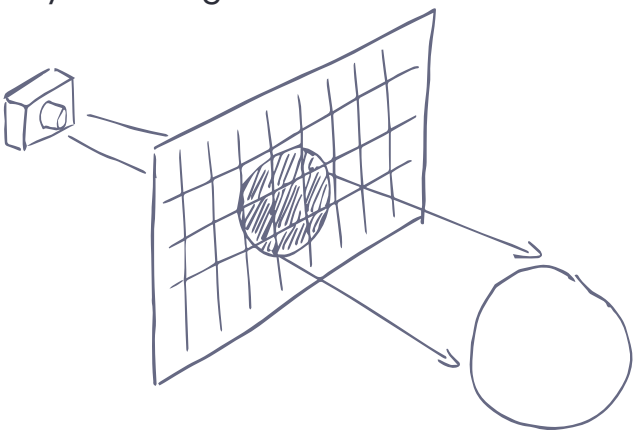
Genau wie bei dem Mandelbrot Set ist der Mandelbulb unendlich detailliert. Jetzt ist die Frage, wie rendert man ein solches Objekt? In der Computergrafik werden Objekte normalerweise in Dreiecke (Polygone) aufgeteielt und gerendert. Diese Methode funktioniert in diesem Fall jedoch eher weniger, da die Dreiecke ermittelt und verbunden werden müssten. Fraktale einfacher zu

rendern ist mit einer Technik namens *Ray Marching* möglich.

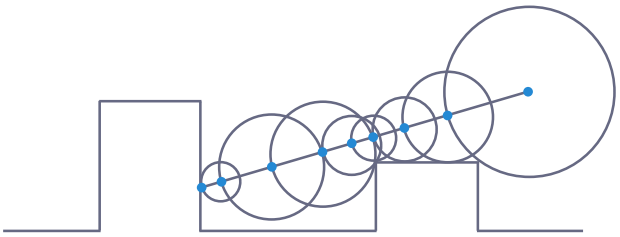
Wenn man sich den Bildschirm eines Betrachters als Ebene vorstellt, dann würde jeder Pixel auf dem Bildschirm einen Punkt auf der Ebene entsprechen. Wenn nun von jedem Punkt aus ein Strahl in die Scene geschossen wird, dann kann man feststellen, ob dieser Strahl ein Objekt trifft oder nicht. Dieses Objekt ist über eine Formel definiert, es existieren also nicht wirklich Objekte in der Scene. Das ist auch garnicht die Anforderung, da klassische Objekte wie bereits erwähnt mit Polygonen dargstellt werden können. Diese Formel wird auch *Signed Distance Function* oder auch SDF genannt.

Wenn ein Strahl in Richtung des Objekts geschossen wird, welches mit einer SDF beschrieben wird, dann wird der kürzeste Abstand zum Objekt mit der SDF berechnet und auf eine Gesamtdistanz addiert. Wenn die Distanz zum Objekt einen gewissen Schwellenwert nach X Iterationen erreicht, wissen wir es gab einen Hit mit dem Objekt. Wenn die Gesamtdistanz einen vorher definierten Maximalwert überschreitet, wissen wir es gab keinen Hit. Jeder Pixel der ein Hit ist wird eingefärbt, so wird das Objekt auf dem Bildschirm sichtbar.

Ray Marching Scene



Ray Marching Algorithmus



Implementierung des Ray Marching Algorithmuses

Um simpel anzufangen, möchte ich zunächst nur eine Kugel in der Scene erzeugen. Eine SDF für eine Kugel lautet wie folgt:

$$f(x,y,z) = \sqrt{x^2+y^2+z^2} - 1$$
$$\Rightarrow f(\vec{p}) = ||\vec{p}|| - 1$$

Im Code:

```glsl
float sphereSDF(vec3 p) {
    return length(p) - 1.0;
}
```

Später für mehrere Objekte in der Scene:

```glsl
float sceneSDF(vec3 p) {
    return sphereSDF(p);
}
```

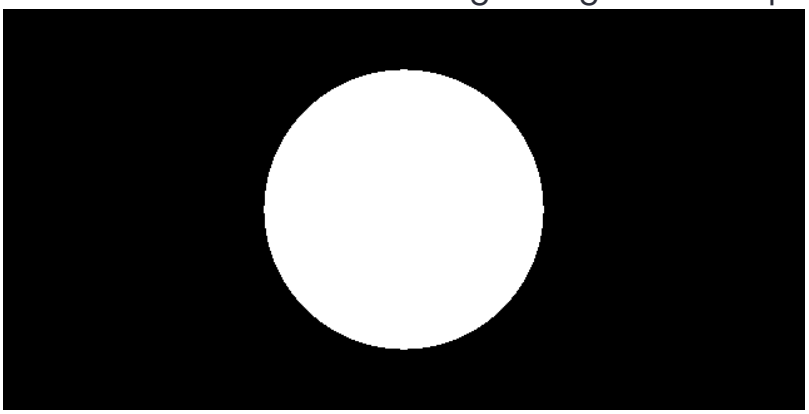
Ray Marching im Code:

```glsl
float ray_march(vec3 ray_origin, vec3 ray_direction) {
    float total_dist = 0.0;
    const int NUMBER_OF_STEPS = 128;
    const float MAXIMUM_TRACE_DISTANCE = 10.0;

    for (int i = 0; i < NUMBER_OF_STEPS; i++) {
        vec3 current_position = ray_origin + total_dist *
                                ray_direction;

        float dist = sceneSDF(current_position);

        if (total_dist > MAXIMUM_TRACE_DISTANCE) {
            total_dist = -1.0;
            break;
        }
        total_dist += dist;
    }
    return total_dist;
}
```

Main Methode für den Shader:

```glsl
void main() {
    vec2 uv = gl_FragCoord.xy / RESOLUTION.xy * 2.0 - 1.0;
    uv.x *= RESOLUTION.x / RESOLUTION.y;

    vec3 dir = vec3(uv, 1.0);
    vec3 cam = vec3(0.0, 0.0, -1.75);
    vec3 color = vec3(0.0);
    float dist = ray_march(cam, dir);

    // Objekt getroffen
    if (dist > 0.0) {
        color = vec3(1.0);
    }

    gl_FragColor = vec4(color, 1.0);
}
```

Die oberen Zeilen Code erzeugen folgenden Output:



Ray Marching Beleuchtung

Das Objekt, welches bis jetzt gerendert wurde ist nicht besonders schön. Der Grund dafür ist, dass noch kein Licht in der Scene existiert. Um zu ermitteln welche Pixel heller und welche dunkler sein müssen, kann man den Normalenvektor von der Oberfläche des Objekts mit dem Richtungsvektor der Lichtquelle Multiplizieren (Skalarprodukt).

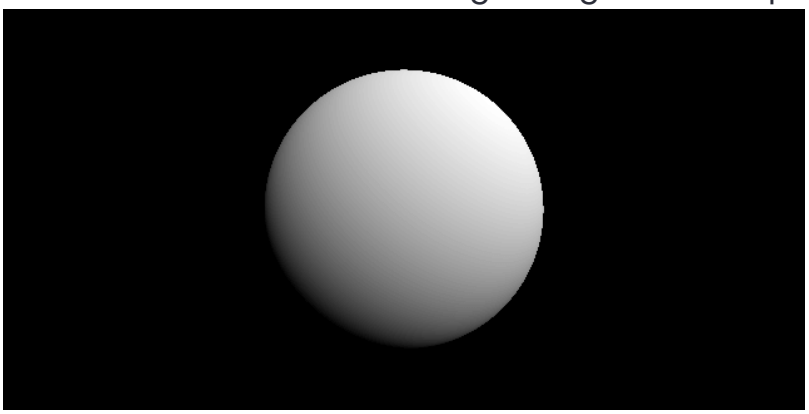
Dafür habe ich folgende Methode verwnedet:

```glsl
vec3 calculate_normal(vec3 p) {
    const vec3 eps = vec3(0.001, 0.0, 0.0);

    float gradient_x = sceneSDF(p + eps.xyy) - sceneSDF(p - eps.xyy)
    float gradient_y = sceneSDF(p + eps.yxy) - sceneSDF(p - eps.yxy)
    float gradient_z = sceneSDF(p + eps.yyx) - sceneSDF(p - eps.yyx)

    return normalize(vec3(gradient_x, gradient_y, gradient_z));
}
```

Die Main Methode wird dadurch auch etwas angepasst:

```glsl
void main() {
    vec2 uv = gl_FragCoord.xy / RESOLUTION.xy * 2.0 - 1.0;
    uv.x *= RESOLUTION.x / RESOLUTION.y;

    vec3 dir = vec3(uv, 1.0);
    vec3 cam = vec3(0.0, 0.0, -1.75);
    vec3 light_dir = normalize(vec3(0.8, 1.0, -1.2));

    vec3 color = vec3(0.0);
    float dist = ray_march(cam, dir);

    if (dist > 0.0) {
        vec3 p = cam + dist * dir;
        vec3 n = calculate_normal(p);
        float diffusion = max(0.0, dot(n, light_dir));
        color = vec3(1.0) * diffusion;
    }

    gl_FragColor = vec4(color, 1.0);
}
```

Die oberen Zeilen Code erzeugen folgenden Output:



Implementierung des Mandelbulbs im Fragmentshader

Der Grund, warum ich das Mandelbrot Set vorher als Shader definiert habe, ist weil dies sehr viel schneller in der Berechnung ist, da die Berechnung auf der GPU nicht auf der CPU durchgeführt wird. Die Berechnung des Mandelbulbs ist sehr viel komplexer und ein Shader ist dringend notwendig, um ein vernünftiges Ergebnis zu erhalten.

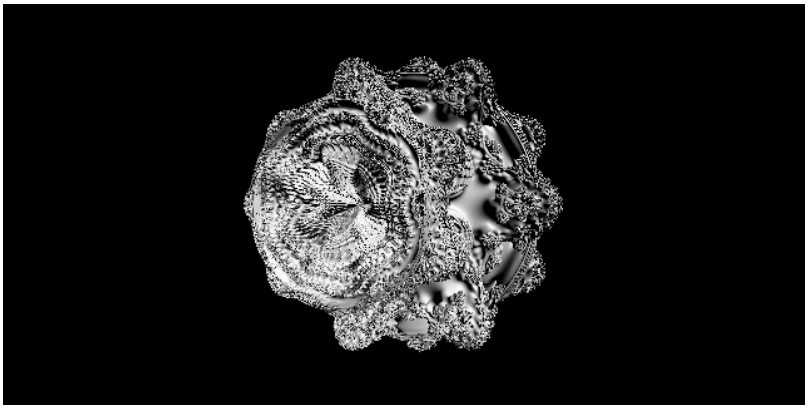
Folgenden Code habe ich als SDF verwnedet, um den Mandelbulb zu erhalten:

```glsl
float mandelbulb(vec3 p) {
    vec3 z = p;
    float power = 8.0;
    float r;
    float dz = 1.0;
    float w = 1.0;

    for (int i = 0; i < 4; i++) {
        r = length(z);
        float theta = acos(z.z / r) * power;
        float phi = atan(z.y, z.x) * power;

        dz = pow(r, power - 1.0) * dz * power + 1.0;
        r = pow(r, power);
        z = r * vec3(sin(theta) * cos(phi), sin(phi) *
                     sin(theta), cos(theta)) + p;
        w = min(w, r);

        if (r > 256.0) break;
    }
    return 0.25 * log(r) * r / dz;
}
```

Jetzt kann die Mandelbulb Funktion einfach in die Scene eingetragen werden:

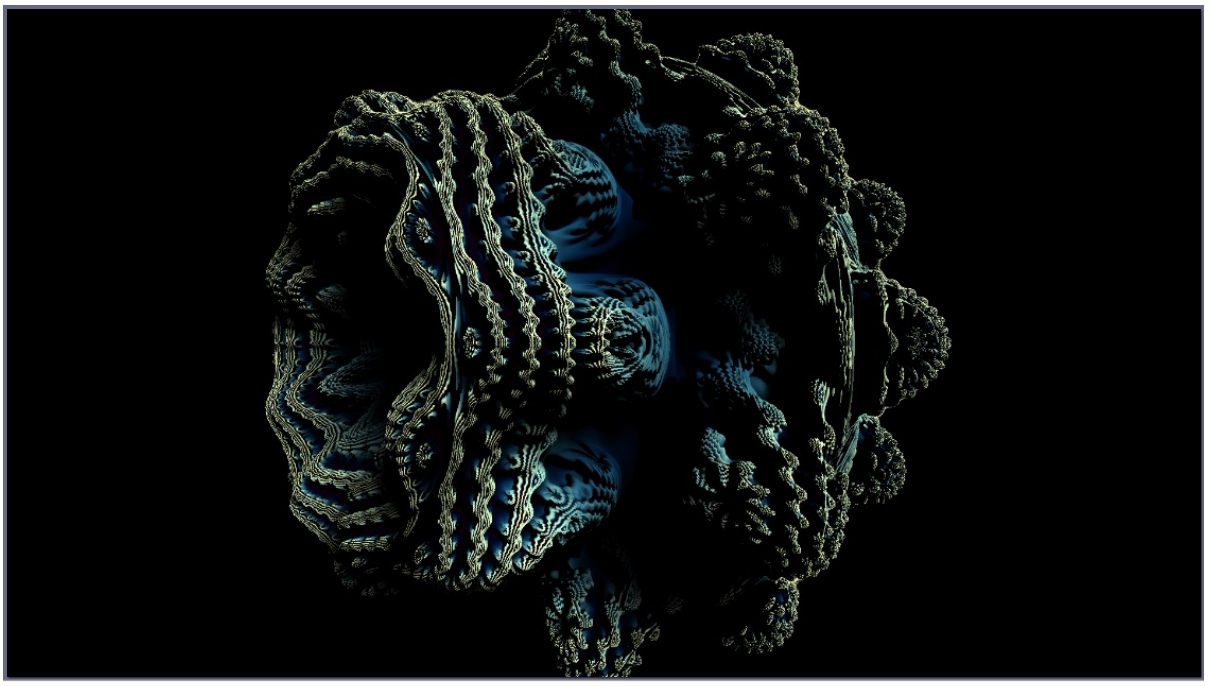
```glsl
float sceneSDF(vec3 p) {
    return mandelbulb(p);
}
```

Hier sieht man den Mandelbulb aus dem Fragmentshader:



Mandelbulb in Aktion

Nach vielen weiteren Stunden der Recherche habe ich den Mandelbulb in Farbe gerendert. Das Rendering zu erklären würde aber deutlich den Rahmen sprengen, desshalb hier einfach nur das finale Ergebnis:



Referenzen

Andrzej Katunin; Kamil Fedio (01.06.2015). ON A VISUALIZATION OF THE CONVERGENCE OF THE BOUNDARY OF GENERALIZED MANDELBROT SET TO (n-1)-SPHERE
https://reader.digitarium.pcss.pl/Content/295117/JAMCM_2015_1_6-Katunin_Fedio.pdf
Barrallo, Javier (2010). Expanding the Mandelbrot Set into Higher Dimensions
https://archive.bridgesmathart.org/2010/bridges2010-247.pdf
Weisstein, Eric W. (25.05.2000). Mandelbrot Set
https://mathworld.wolfram.com/MandelbrotSet.html
Dickerson, Richard E. (03.12.2003). Higher Order Mandelbrot Fractals: Experiments in Nanogeometry
http://www.fractal.org/Bewustzijns-Besturings-Model/Higher-order-Mandelbrot-Fractals.pdf
Fredriksson, Bastian (15.01.2015). An introduction to the Mandelbrot set
https://www.kth.se/social/files/5504b42ff276543e4aa5f5a1/An_introduction_to_the_Mandelbrot_Set.pdf
Walczyk, Michael (11.12.2020). Ray Marching
https://michaelwalczyk.com/blog-ray-marching.html
Quilez, Inigo (25.03.2013). 3D Signed Distance Functions
https://iquilezles.org/articles/distfunctions/