Projektbericht «Quaternion Julia Fraktale»

Studenten: Thöni Stefan, Sidler Matthias

Betreuer: Marx Stampfli

1.1 Inhaltsverzeichnis

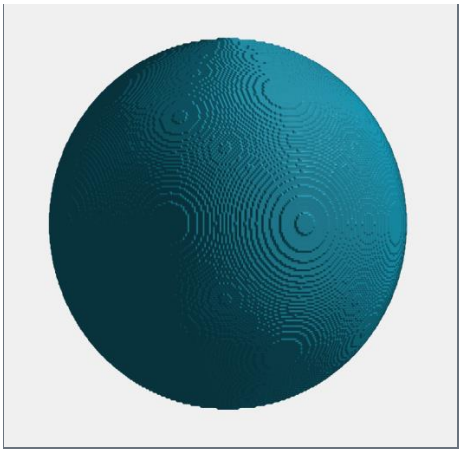

Abbildung 1: Quaternion Julia Fraktal mit k=0

1.2 Abstract

Im Rahmen der Module «Programmieren in Matlab / Octave» und «Objektorientiere Geometrie» haben wir die fraktalen Eigenschaften der Mandelbrot- sowie Julia-Mengen in der komplexen Zahlenebene sowie im Hyperkomplexen Raum untersucht. Es wurde ein Matlab-Programm erstellt, welches diese Fraktale generiert und auf grafische Weise darstellt.

1.3 Mathematische Grundlangen

Im Folgenden sollen die mathematischen Grundlagen kurz erläutert werden, um die Thematik des Projektes besser verstehen zu können.

1.3.1 Komplexe Zahlen $\mathbb{C}$

Die komplexen Zahlen ermöglichen die Lösung der Gleichung $x^2 + 1 = 0$, indem die imaginäre Zahl i mit der Eigenschaft $\mathrm{i}^2 = -1$ eingeführt wird. Dargestellt werden komplexe Zahlen in der Form

$$c = a + b \cdot \mathrm{i} \qquad a, b \in \mathbb{R}$$

Dabei ist der Real-Teil $\mathrm{Re}(c) = a$ und der Imaginär-Teil $\mathrm{Im}(c) = b$, welche als x/y Koordinaten aufgefasst und so die komplexe Zahlenebene bilden.

1.3.1.1 Rechnen mit komplexen Zahlen

Wir beschränken uns nur auf die Operationen, welche für die Berechnung nachfolgenden Folgen benötigt werden.

Gegeben zwei komplexe Zahlen: $z_1 = a + b\mathrm{i}$ und $z_2 = c + d\mathrm{i}$

Addition $z_1 + z_2 = (a + c) + (b + d)\,\mathrm{i}$

Multiplikation $z_1 \cdot z_2 = (ac - bd) + (ad + bc)\,\mathrm{i}$

1.3.2 Mandelbrot-Menge

Um die Mandelbrot-Menge zu berechnen, wird für jeden Punkt $c \in \mathbb{C}$:$\mathrm{Re}(c) \in [x_1, x_2], \mathrm{Im(c)} \in [y_1, y_2]$ die rekursive Folge

$$z_0 = 0$$

$$z_{n+1} = z_n^2 + c$$

durchgeführt. Ist die Folge für den Punkt c nach n Iterationen beschränkt, so gehört dieser Punkt zur Mandelbrot-Menge, welche also eine Teilmenge der komplexen Zahlenebene ist.

Bildet man nun die komplexen Zahlen auf einen Pixelbereich ab und zeichnet die Mandelbrot-Menge ein, ergibt dies ein 2D Fraktal. Oft werden die Punkte je nach Anzahl der benötigten Iterationen bis zu einem Schwellenwert unterschiedlich eingefärbt.

1.3.3 Julia-Menge

Die Julia-Mengen wird ähnlich die der Mandelbrot-Menge erstellt.

Für jeden Punkt $c \in \mathbb{C}$:$\mathrm{Re(c)} \in [x_1, x_2], \mathrm{Im(c)} \in [y_1, y_2]$ und der Konstante $k \in \mathbb{C}$ wird die rekursive Folge

$$z_0 = c^2 + k$$

$$z_{n+1} = z_n^2 + k$$

durchgeführt. Ist auch hier die Folge für den Punkt c nach n Iterationen beschränkt, so gehört dieser Punkt zur Julia-Menge. Die Gestalt der Menge hängt von der gewählten Konstante k ab. Für jedes k in der komplexen Ebene existiert eine korrespondierende Julia-Menge. Gewisse Eigenschaften der Julia-Menge lassen sich durch die relative Lage der Konstanten k zur Mandelbrot-Menge bestimmen.

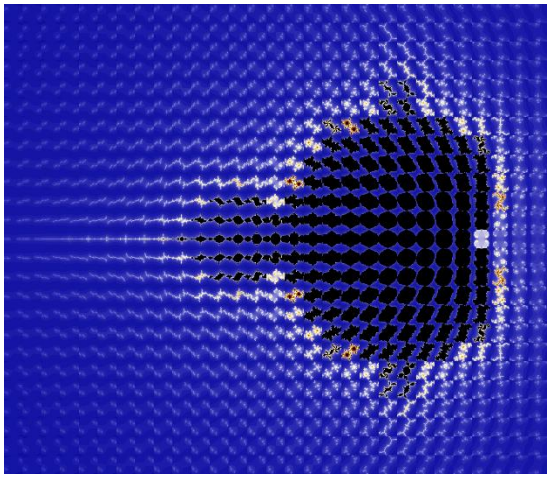

Abbildung 2: Julia Mengen für bestimmte k machen die Mandelbrot-Menge erkennbar Quelle: wikipedia.org/wiki/Julia-Menge

Mit der gleichen Idee wie bei der Mandelbrot-Menge können nun die Julia-Menge als Fraktal gezeichnet werden.

1.3.4 Quaternionen $\mathbb{H}$

Die Quaternionen erweitern - ähnlich der komplexen Zahlen – den Zahlenbereich der reellen Zahlen. Quaternionen erlauben es insbesondere, Drehungen im dreidimensionalen euklidischen Raum auf mathematisch elegante Weise zu beschreiben. Konstruiert werden Quaternionen indem 3 imaginäre Einheiten i, j und k zum Realteil hinzugefügt werden. Dargestellt werden Quaternionen in der Form

$$Q = r + a \cdot i + b \cdot j + c \cdot k \qquad r, a, b, c \in \mathbb{R}$$

Also ist r die reale Komponente und der Vektor (r, a, b, c) kann als Vektor im 4D Quaternion Raum angeschaut werden. Die imaginäre Zahl i behält die Eigenschaft $\mathrm{i}^2 = -1$. Die Beziehung zwischen i, j und k ist wie folgt:

$$i^2 = j^2 = k^2 = -1$$

$$ij = k \qquad \mathrm{jk} = i \qquad \mathrm{ki} = j$$

$$j\imath = -k \qquad \mathrm{kj} = -i \qquad \mathrm{ik} = -j$$

1.3.4.1 Rechnen mit Quaternionen

Wir beschränken uns nur auf die Operationen, welche für die Berechnung nachfolgenden Folgen benötigt werden.

Gegeben zwei Quaternionen: $Q_1 = r_1 + a_1 i + b_1 j + c_1 k$ und $Q_2 = r_2 + a_2 i + b_2 j + c_2 k$

Addition $$Q_1 + Q_2 = (r_1 + r_2) + (a_1 + a_2)i + (b_1 + b_2)j + (c_1 + c_2)k$$

Multiplikation (nicht kommutativ!)
$$\begin{aligned} Q_1 \cdot Q_2 = {} & (r_1 r_2 - a_1 a_2 - b_1 b_2 - c_1 c_2) + \\ & (r_1 a_2 + a_1 r_2 + b_1 c_2 - c_1 b_2)\mathrm{i} + \\ & (r_1 b_2 + b_1 r_2 + c_1 a_2 - a_1 c_2)\mathrm{j} + \\ & (r_1 c_2 + c_1 r_2 + a_1 b_2 - b_1 a_2)\mathrm{k} \end{aligned}$$

Betrag / Länge $$|Q_1| = \sqrt{r_1^2 + a_1^2 + b_1^2 + c_1^2}$$

1.3.5 Mandelbrot-Menge in 3D

Die Mandelbrot-Menge in 3D wird ähnlich der normalen Mandelbrot-Menge erstellt. Man benutzt anstelle der komplexen Zahlen jedoch die Quaternionen mit ihren vier Komponenten.

Für jeden Punkt $c \in \mathbb{H}$ wird die rekursive Folge

$$z_0 = 0$$

$$z_{n+1} = z_n^2 + c$$

durchgeführt. Ist der Betrag des Funktionswertes $|z_n|$ nach n Iterationen beschränkt, so gehört dieser Punkt zur Mandelbrot-Menge.

Bildet man nun diese Menge auf einen Pixelbereich als 3D ab und zeichnet die Mandelbrot-Menge ein, ergibt dies ein 3D Fraktal:

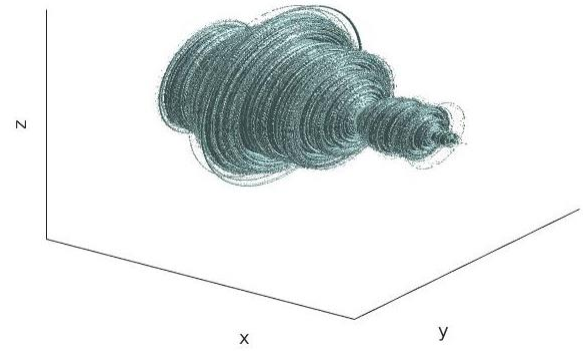

Abbildung 3: 3D Mandelbrot

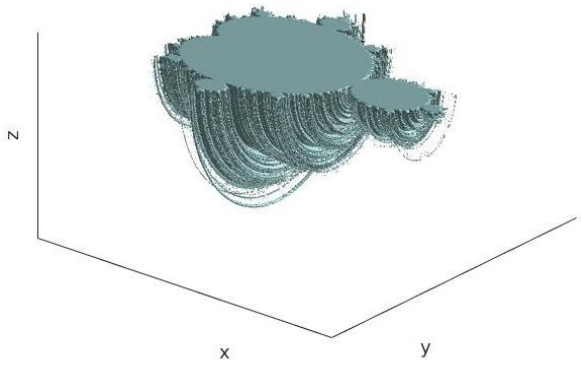

Abbildung 4: 3D Mandelbrot halfcut

1.3.6 Julia-Menge in 3D

Julia-Menge in 3D wird ähnlich der normalen Julia-Menge und der 3D Mandelbrot-Menge erstellt. Weil dabei Quaternionen verwendet werden, nennen wir die dabei entstehende Menge *Quaternion Julia Fraktale*.

Für jeden Punkt $c \in \mathbb{H}$ und der Konstante $k \in \mathbb{H}$ wird die rekursive Folge

$$z_0 = c^2 + k$$

$$z_{n+1} = z_n^2 + k$$

durchgeführt. Ist der Betrag des Funktionswertes $|z_n|$ nach n Iterationen beschränkt, so gehört dieser Punkt zur Julia-Menge. Diese Menge hängt also wie auch schon das zweidimensionale Äquivalent von der gewählten Konstante k ab.

Natürlich spielt auch die gewählte Funktion eine Rolle. Im Rahmen dieses Projektes wurde jedoch nur die oben definierte nicht lineare Funktion untersucht.

Mit der gleichen Idee wie bei der Mandelbrot-Menge können nun die Julia-Menge als 3D Fraktal gezeichnet werden.

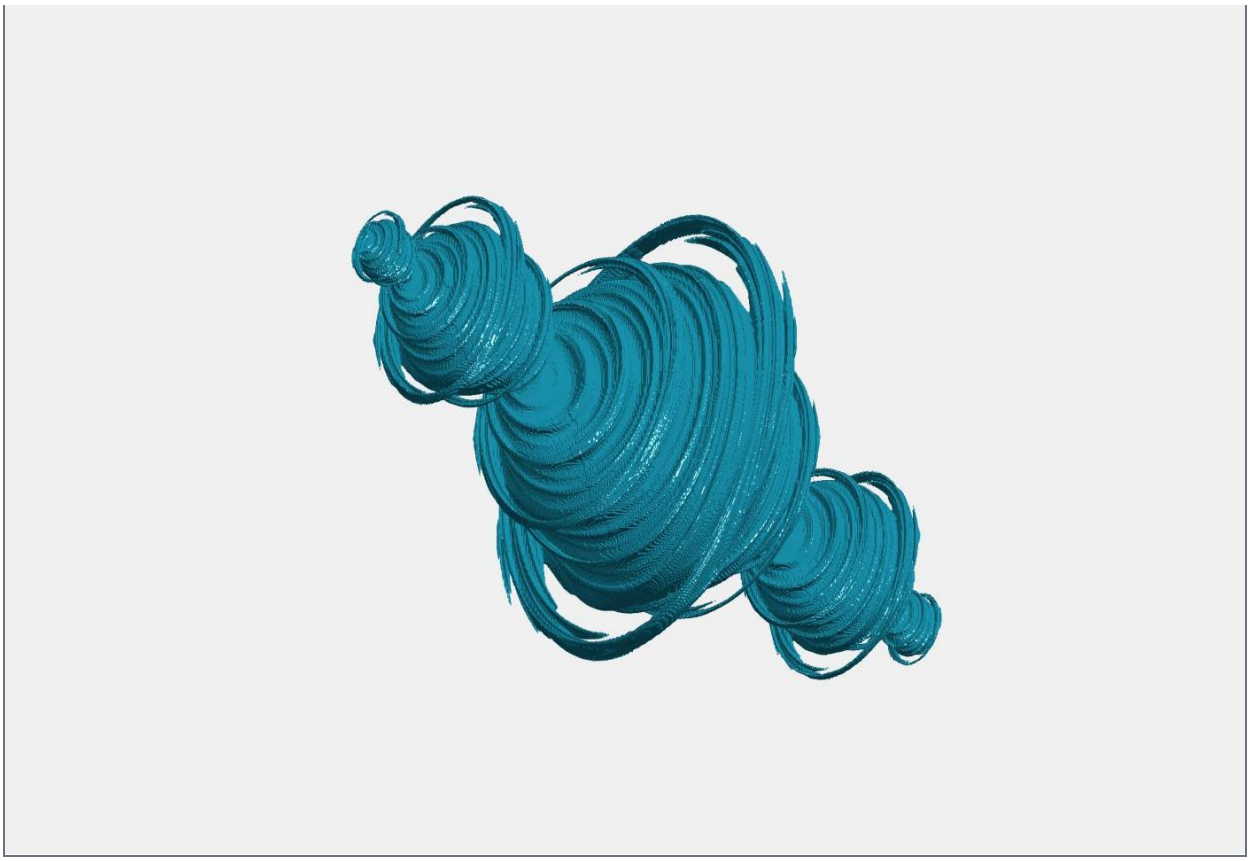

Abbildung 5: Quaternion Julia Fraktal mit (-1, 0.2, 0, 0) als k

1.4 Implementation

1.4.1 Systemanforderung

Die Voraussetzungen für die Ausführung des finalen Programms sind:

- MATLAB, in Version R2017b (getestet, andere Version ev. auch möglich)
- NVIDIA GPU mit CUDA Unterstützung, mind. 2 GB RAM
- CUDA Drivers von NVIDIA
- Mind. 16 GB freie RAM

Das benötigte Memory ist abhängig von der gewählten Samplingrate. Oben genannte Mindestanforderung an RAM gilt bei einer Samplingrate von 400.

1.4.2 Vorgehensweise

Um ein besseres Verständnis von Fraktalen zu erhalten, haben wir zu Beginn des Projektes mit MATLAB Newton Fraktale (Folder 1_newton_fraktale) generiert und dargestellt. Danach setzten wir uns intensiv mit der Theorie zu Mandelbrot- und Julia-Mengen auseinander. Da diese Fraktale schon oft untersucht wurden, gab es dazu fast fertige MATLAB Programme, welche wir getestet haben. Dabei spielten wir mit den Parametern bzw. der Konstante, um deren Wirkung auf das Ergebnis zu verstehen.

In einem nächsten Schritt haben wir die Quaternionen implementiert und uns überlegt, wie wir die 3D Fraktale am effizientesten generieren können. Als Test haben wir eine 3D Mandelbrotmenge generiert und dargestellt (Folder 2_mandelbrot_mengen_3d) und so unser erstes 3D Fraktal sichtbar gemacht.

Danach haben wir das Skript umgebaut und erweitert, damit wir auch Quaternion Julia Fraktale darstellen können. Dies war sehr zeitintensiv, da wir auf verschiedene Herausforderungen trafen, wie die Wahl der optimalen Parametern k, Samplingrate, Anzahl Iterationen und Grenze, effiziente Berechnung der Quaternion Multiplikation und Addition und schöne Darstellung sowie das Hineinzoomen (siehe u.a. 1.5 Erkenntnisse / Probleme).

1.4.3 Finales Programm

Im Folgenden soll die Funktion der Komponenten des finalen Programmes (Folder final) erklärt und deren Highlights hervorgehoben werden.

1.4.3.1 demo.m

Eigentlicher Startpunkt, um Fraktal mit entsprechenden Werte für Konstante k zu generieren.

1.4.3.2 animate_highres.m

Dies dient dazu, um über die Werte von k zu iterieren und so eine Animation von der Entstehung und dem Zerfall eines Fraktal zu generieren (Folder gallery/animate_-1,0.2,0,0).

1.4.3.3 generateFractal.m

Generiert ein Julia Fraktal mit dem übergebenen k. Dabei wird der CUDA Kernel geladen, ein Würfelraum erstellt, Memory initialisiert und der Kernel aufgerufen.

```
% Load the kernel
kernel = parallel.gpu.CUDAKernel( ptxFilename, cudaFilename );
[…]
% create a evenly spaced vector for the space that we want to sample
x = gpuArray.linspace( xlim(1), xlim(2), gridSize );
y = gpuArray.linspace( ylim(1), ylim(2), gridSize );
z = gpuArray.linspace( zlim(1), zlim(2), gridSize );

% Create a coordinate cube from the sample-space
[xGrid, yGrid, zGrid] = meshgrid(x, y, z);
[…]
% Call the kernel and iterate on gpu
count = feval( kernel, count, xGrid, yGrid, zGrid, wGrid, cx, cy, cz, cw, maxIterations, numElements );
% Gather the result from the GPU memory
count = gather( count );
```

1.4.3.4 generateHighResolutionFractal.m

Generiert ein Julia Fraktal mit dem übergebenen k in 8-fach höherer Auflösung als generateFractal.m. Das Konzept ist, den Würfelraum in 8 kleinere Würfel aufzuteilen, für jeden generateFractal.m aufzurufen und das Ergebnis zusammenzustellen.

```
% specify limits (sampling space)
xlim = linspace(-1.5, 1.5, 3);
ylim = linspace(-1.5, 1.5, 3);
zlim = linspace(-1.5, 1.5, 3);
wlim = [-1.5 1.5];

% createFractal is called for 8 sub-cubes and then patched together using cat
% front, bottom, left cube
fbl = createFractal(cx,cy,cz,cw, xlim(1:2), ylim(1:2), zlim(1:2), wlim, sampleSize, linux);

% front, bottom, right cube
fbr = createFractal(cx,cy,cz,cw, xlim(2:3), ylim(1:2), zlim(1:2), wlim, sampleSize, linux);

% cat fbl and fbr together to get front-bottom element
fb = cat(2, fbl, fbr);

% bottom back left cube
bbl = createFractal(cx,cy,cz,cw, xlim(1:2), ylim(2:3), zlim(1:2), wlim, sampleSize, linux);

% bottom back right cube
bbr = createFractal(cx,cy,cz,cw, xlim(2:3), ylim(2:3), zlim(1:2), wlim, sampleSize, linux);

% bbl and bbr concatenated results in bottom back element
bb = cat(2, bbl, bbr);

% fb and bb concatenated results in the bottom plane of the cube
bottom = cat(1,fb,bb);

ftl = createFractal(cx,cy,cz,cw, xlim(1:2), ylim(1:2), zlim(2:3), wlim, sampleSize, linux);
ftr = createFractal(cx,cy,cz,cw, xlim(2:3), ylim(1:2), zlim(2:3), wlim, sampleSize, linux);

ft = cat(2, ftl, ftr);

btl = createFractal(cx,cy,cz,cw, xlim(1:2), ylim(2:3), zlim(2:3), wlim, sampleSize, linux);
btr = createFractal(cx,cy,cz,cw, xlim(2:3), ylim(2:3), zlim(2:3), wlim, sampleSize, linux);
bt = cat(2, btl, btr);
top = cat(1, ft,bt);

% the bottom and top concateneted gives the result
result = cat(3, bottom, top);
```

1.4.3.5 processQuatJulEle[_linux].cu

Dies ist der Kernel für die GPU, worin die eigentliche Berechnung der Folge geschieht.

```
__global__ void main ( […] ) {
[…]
    // Get our X and Y coords
    double const xPart0 = x[globalThreadIdx];
    double const yPart0 = y[globalThreadIdx];
    double const zPart0 = z[globalThreadIdx];
    double const wPart0 = w[globalThreadIdx];

    // Run the itearations on this location
    unsigned int const count = doIterations( xPart0, yPart0, zPart0, wPart0, cx, cy, cz, cw,
maxIters );
    out[globalThreadIdx] =  double( count  );
}

__device__ unsigned int doIterations( […] ) {
    // Initialise: z = z0
    double xPart = xPart0;
    double yPart = yPart0;
    double zPart = zPart0;
    double wPart = wPart0;
    unsigned int count = 0;

 // Loop until escaped. Check Quaternion magnitude smaler than 4
    while ( ( count <= maxIters )
            && ((xPart*xPart + yPart*yPart + zPart*zPart + wPart*wPart) <= 16.0) ) {
        ++count;
        // Update: z = z*z + z0;
        double const oldXPart = xPart;
        double const oldYPart = yPart;
        double const oldZPart = zPart;
        double const oldWPart = wPart;

        // Quat mult and add constant
        xPart = oldXPart*oldXPart-oldYPart*oldYPart-oldZPart*oldZPart-oldWPart*oldWPart + cx;
        yPart = oldXPart*oldYPart+oldYPart*oldXPart-oldZPart*oldWPart+oldWPart*oldZPart + cy;
        zPart = oldXPart*oldZPart+oldYPart*oldWPart+oldZPart*oldXPart-oldWPart*oldYPart + cz;
        wPart = oldXPart*oldWPart-oldYPart*oldZPart+oldZPart*oldYPart+oldWPart*oldXPart + cw;
        //xPart = xPart*xPart - yPart*yPart + xPart0;
        //yPart = 2.0*oldRealPart*yPart + yPart0;
    }
    return count;
}
```

1.4.3.6 processQuatJulEle[_linux].ptx

Das vom NVIDIA Compiler kompilierte File von processQuatJulEle.cu.

1.4.3.7 render.m

Am Schluss zuständig, dass das Fraktal gerendert und als Bild gespeichert wird. Dabei wird die Kamera optimal eingestellt (durch Versuchsreihe möglichst optimiert), Licht gesetzt und die Fraktal-Menge gepatched.

```
% Empirical ('Good looking') values for view and cameraposition.
view(-20,-50);
campos('manual');
campos([-2.5 -2.5 -2.5]);
camva(45);
[…]
% patch the isosurface and create a mesh with fancy colors
iso = patch(fract,...
    'FaceColor',[0.11,.66,.78],...
    'EdgeColor','none');

% light and coloring parameters
iso.AmbientStrength = 0.3;
iso.DiffuseStrength = 0.6;
iso.SpecularColorReflectance = 1;
iso.SpecularExponent = 50;
iso.SpecularStrength = 1;
```

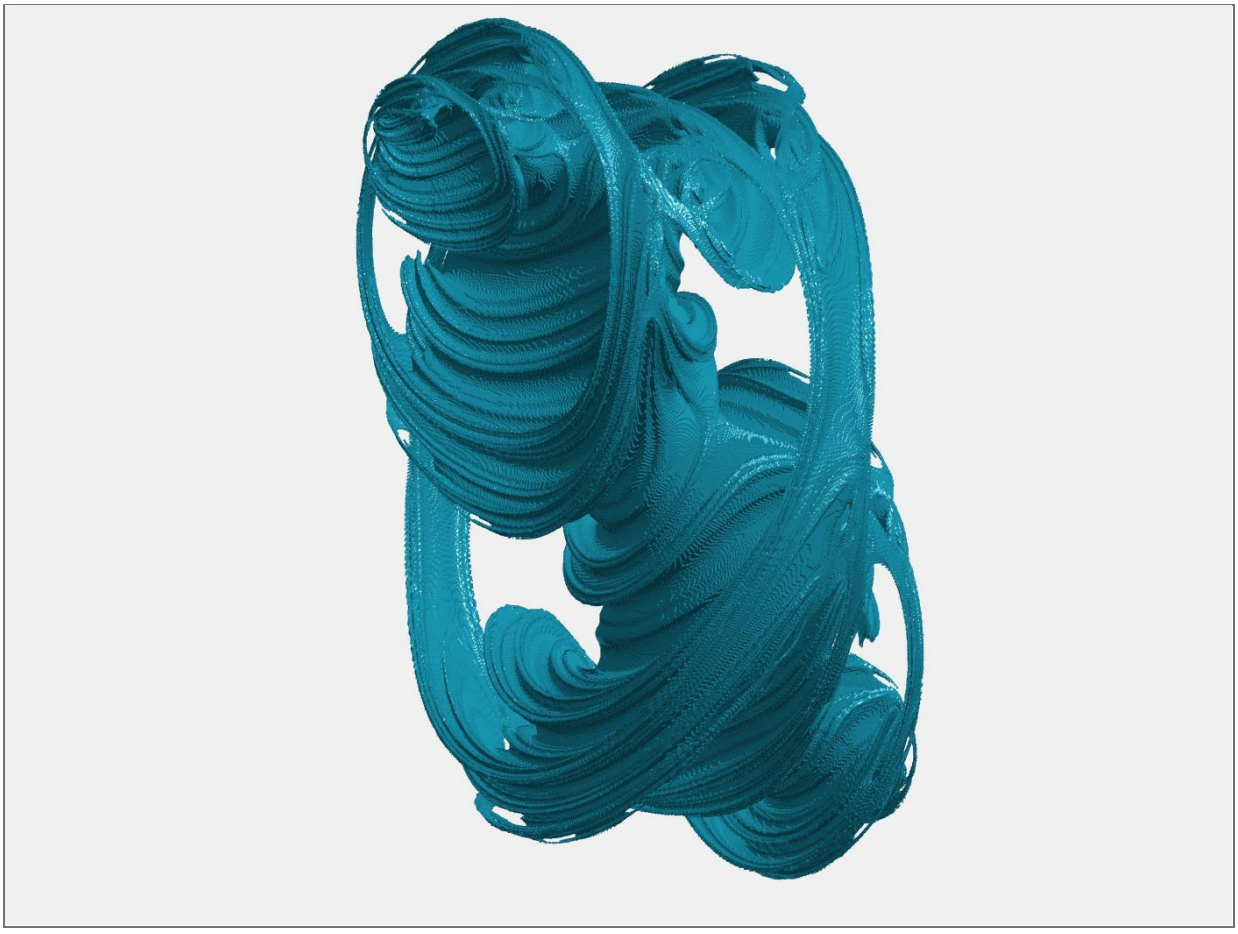

Abbildung 6: Quaternion Julia Fraktal mit (-0.125, -0.256, 0.847, 0.0895) als k

1.5 Erkenntnisse / Probleme

1.5.1 Wahl der optimalen Parameterwerte

Die Wahl der Parameterwerte beeinflusst logischerweise direkt das Resultat. Aufgrund von Referenzen und insbesondere durch x-faches ausprobieren, haben wir versucht, die optimalen Werte herauszufinden.

1.5.1.1 Samplingrate

Die Samplingrate ist insbesondere für die Auflösung und detailtreue verantwortlich. Ist sie zu gering, entfallen Details, ist sie zu gross, wird das Resultat zwar genauer, jedoch steigt der Rechenbedarf unverhältnismässig. Gewählter Wert: 400

Abbildung 7: Quat Julia k =(-0.2, 0.8, 0, 0), 400 samples

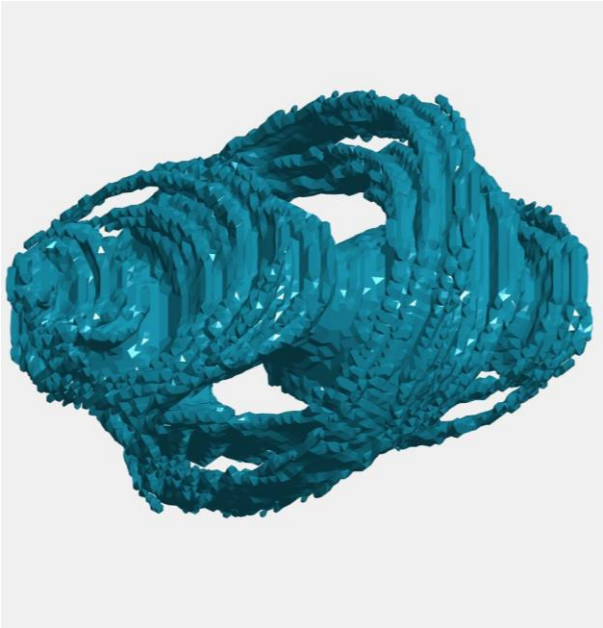

Abbildung 8: Quat Julia k=(-0.2, 0.8, 0, 0), 100 samples

1.5.1.2 Anzahl Iterationen, Wahl der Grenze

Die Anzahl der Iterationen und die Wahl der Grenze, um zu entscheiden, ob die Funktion beschränkt ist, beeinflussen die Genauigkeit des Fraktals.

Ist die Anzahl der Iterationen zu gering, fallen Punkte in die Menge, welche nicht sollten. Ist sie zu gross, erhöht sich der Rechenbedarf, ohne dass man ein genaueres Resultat erhält. Gewählter Wert: 50

Ist die Grenze zu klein, fallen Punkte aus der Menge, die eigentlich dazu gehören. Ist sie zu gross, fallen wieder um Punkte in die Menge, welche nicht sollten. Gewählter Wert: 4

1.5.2 Effiziente Berechnungszeit

Wir mussten die Berechnung der Quaternion Multiplikation und Addition sehr effizient implementieren.

Um ein hochaufgelöstes Bild zu erzeugen bedeutet das 2,46 * 10^10 reine Quaternion Multiplikationen und Additionen. (Anzahl Iterationen * Sampling Size^3 * Anzahl Teilbilder: 50*400^3*8).

Als wir zuerst mit dem CPU testeten, realisierten wir ziemlich schnell, dass die Leistung viel zu schwach war. Bei einer Schätzung für ein einzelnes High Resolution Bild kamen wir auf eine Berechnungszeit für mehrere Monate. Dies wäre natürlich unpraktikabel gewesen für ein Semesterprojekt.

Als zweiten Ansatz haben wir die eingebauten GPU Arrays von Matlab verwendet. Diese brachten aber leider immer noch nicht die gewünschte Verbesserung und die Schätzung blieb bei einigen Tagen.

Erst als wir unser eigener CUDA Kernel implementierten und so nativ die GPU benutzen konnten, hatten wir je nach Hardware rund 1 Minute und konnten so unsere Tests starten.

1.5.3 Anschauliche Darstellung

Die Daten welche Punkte zum Quaternion Julia Fraktal gehören, haben wir nun also effizient berechnen können. Eine weitere Hürde war dann die schöne und anschauliche Darstellung, welche wir mit der Matlab Funktionen Isosurface und Patch schliesslich umsetzen konnten. Insbesondere die Matlab Funktion Isosuface, welche oft verwendet wird zum Rendern von CRT-Scans, erwies sich als sehr nützlich, da wir bereits die korrekte Datenstruktur zu deren Verwendung vorliegen hatten. Das Einstellen der Kamerawinkel, Kameraposition, Beleuchtung, Färbung etc. war ein rumtüfteln und ausprobieren bis ein hübsches Bild vorlag.

Bei 2D Fraktale konnte die Anzahl der benötigten Iterationen unterschiedlich farbig dargestellt werden. Dies ist im 3D leider nicht ganz trivial, insbesondere deswegen, weil nur die Oberfläche des Fraktals sichtbar ist. Die Iterationen könnten eventuell mittels Schattierungen/Transparenzen oder mit einer Animation dargestellt werden. Dies wurde aber vorerst auf ein regnerisches Wochenende verschoben.

1.5.4 Zoom In

In einem Test haben wir auch versucht, in das Fraktal hineinzuzoomen und so neue Details sichtbar zu machen. Dadurch zeigte sich schön die fraktale und faserige Struktur der Quaternion-Julia-Menge. Die Mengen scheinen aus vielen unendlich feinen Fasern zu bestehen.

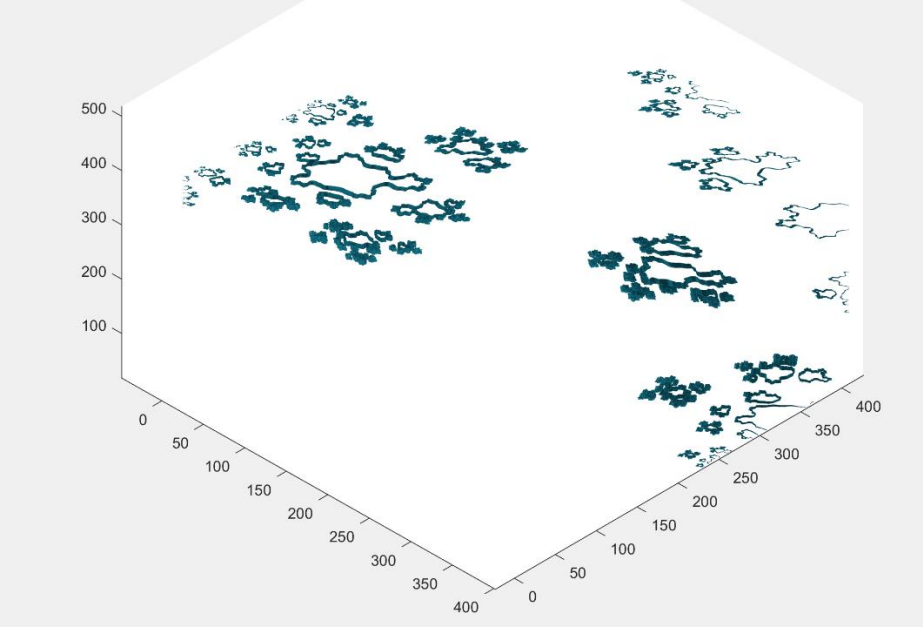

Abbildung 9: Querschnitt der reingezoomten «Fasern»

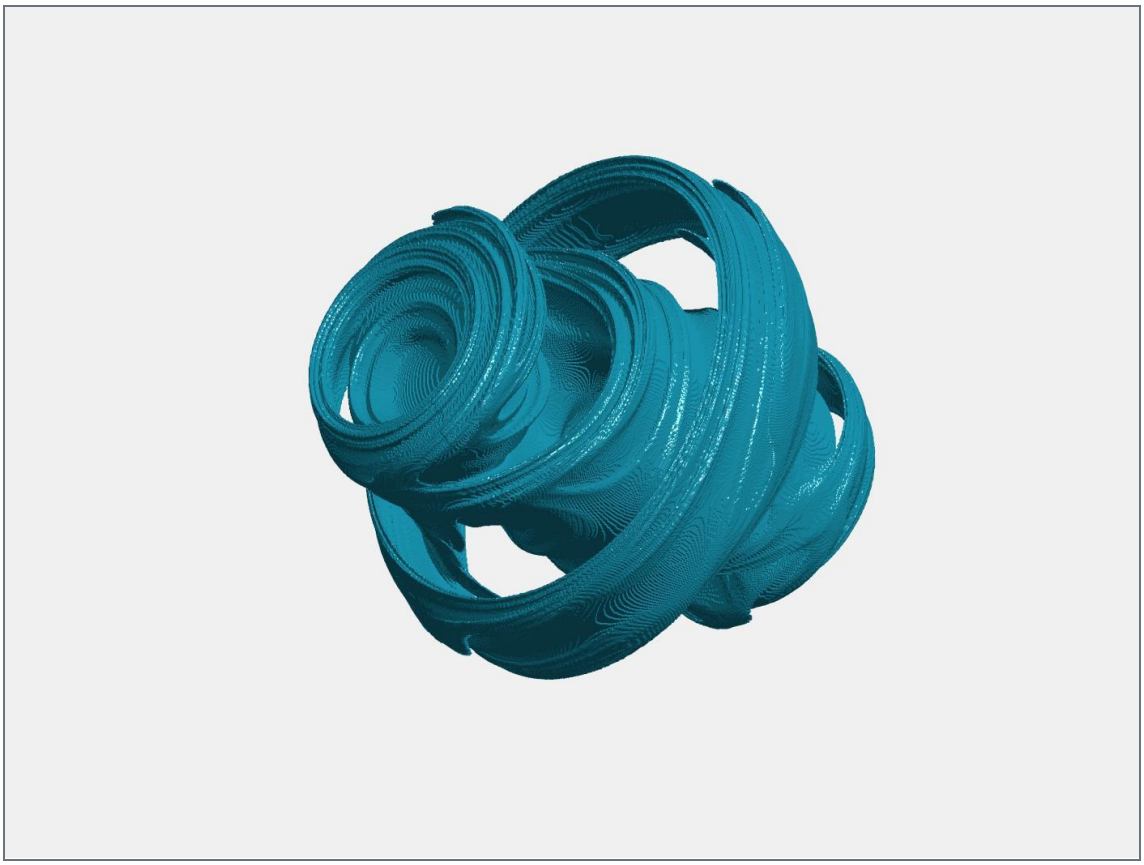

Abbildung 10: Quaternion Julia Fraktal mit (-0.445, 0.339, -0.0889, -0.56) als k

1.6 Quellen

https://en.wikipedia.org/wiki/Mandelbrot_set
https://en.wikipedia.org/wiki/Julia_set
http://bugman123.com/Hypercomplex/
https://en.wikipedia.org/wiki/Quaternion
https://ch.mathworks.com
http://paulbourke.net/fractals/quatjulia/

1.7 Abbildungsverzeichnis