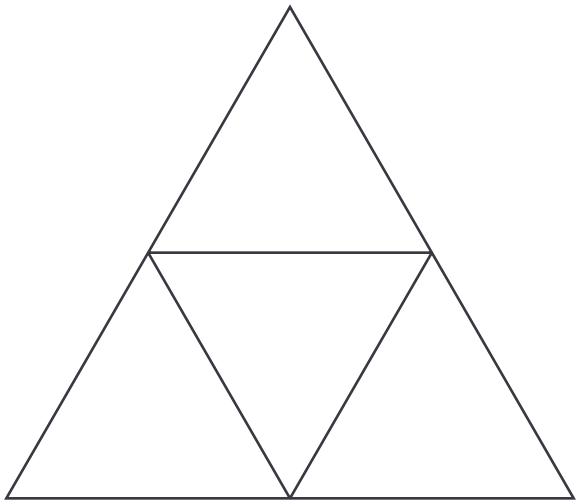
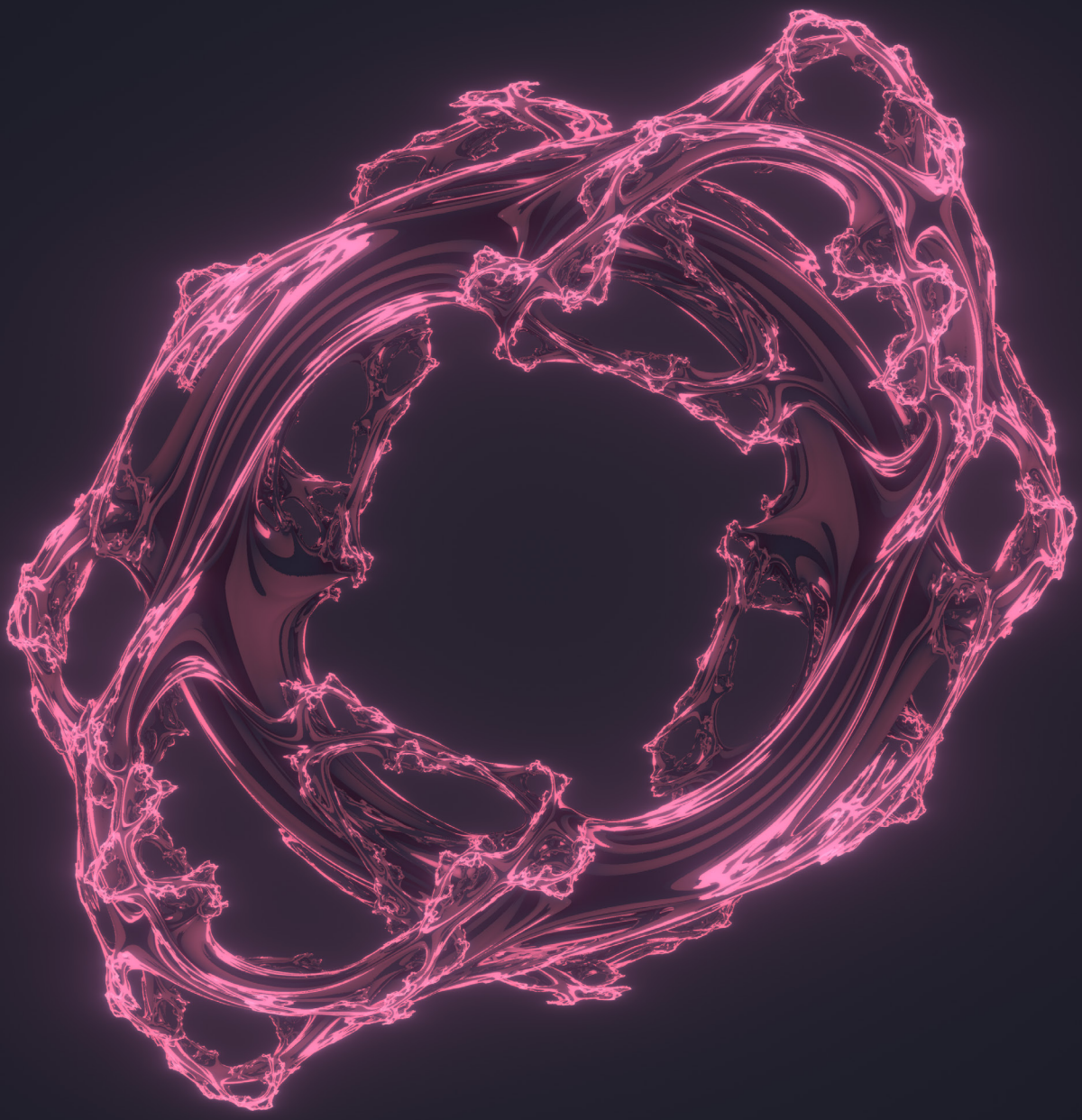
FRAKTALE

und ihre Anwendung im Game-Design

Maturaarbeit von Alec Franco
Betreut von Yves Gärtner
Kantonsschule Reussbühl Luzern
2020



Inhalt



1 Einleitung

1.1 Vorwort

Das Ziel dieser Arbeit ist es, mich mit Fraktalen auseinanderzusetzen und sie zu verstehen lernen. Es ist mir wichtig, dass ich dabei möglichst viel Neues lerne.

Das Resultat dieser Arbeit werde ich in Form einer Applikation namens «Fractals» zugänglich machen. Diese Applikation ist eine Zusammenstellung der verschiedenen Fraktale und anderer Projekte, welche ich im Rahmen dieser Arbeit programmiert habe. Sie ist einfach zugänglich und benutzerfreundlich gestaltet. Die Applikation besitzt einen hohen Lernfaktor und ermöglicht so ein spielerisches Erlernen der Fraktale. Die Applikation bietet auch die Möglichkeit, Bildschirmfotos zu erstellen, damit jeder eindrückliche Bilder von Fraktalen erstellen kann.

Der Code und das Programm sind open-source auf GitHub verfügbar: github.com/srpnt3/Fractals

Am Ende dieser Arbeit befindet sich ein Glossar, in dem fachspezifische Wörter genauer erklärt sind und nachgeschlagen werden können.

1.2 Abstract

In dieser Arbeit werden verschiedene 2-, 3- und n-dimensionale Fraktale mathematisch beschrieben und programmiert. Es werden verschiedene Technologien, um Fraktale zu rendern, analysiert. Die Umsetzung der Fraktale in Code wird genau dokumentiert. Zuletzt werden verschiedene Anwendungen der Fraktale in verschiedenen Bereichen, vor allem dem Game-Design, besprochen.

1.3 Zu meiner Person

Ich hatte bereits ein frühes Interesse für Technisches und Mathematisches. Angefangen zu programmieren habe ich in der Primarschulzeit und ich konnte mir seitdem sehr viel Neues beibringen und lernen.

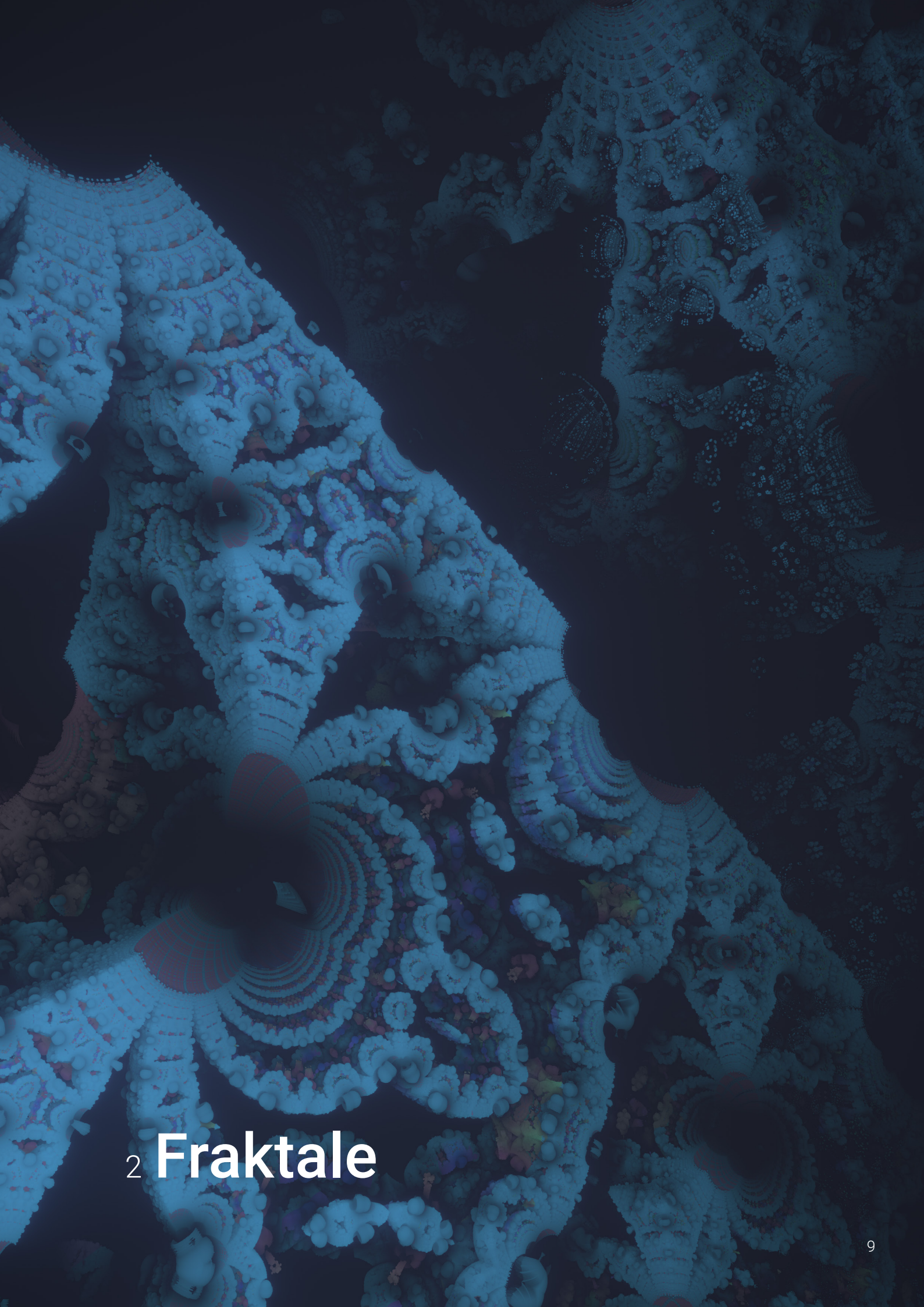
Design ist ein weiterer Bereich, der mich interessiert. Ich habe in meiner Freizeit bereits viele verschiedene Projekte programmiert und gestaltet. Darunter finden sich mehrere Webseiten, Applikationen und Logos. Ausserdem wird seit einigen Jahren an der Kantonsschule Reussbühl Luzern ein Programm für das stichprobenartige Prüfen von Schülern verwendet, welches von mir programmiert und designt wurde.

In Kontrast zu dem praktiziere ich die japanische Kampfkunst Jiu-Jitsu seit etwa 6 Jahren.

1.4 Persönliche Motivation

Fraktale kenne ich schon eine Weile. Mein Interesse wurde jedoch erst richtig geweckt, als sie im Schwerpunktfach Mathematik erstmals behandelt wurden. Der Betreuer dieser Arbeit, Yves Gärtner war damals schon mein Mathematiklehrer und hat definitiv dazu beigetragen, dass ich mich für dieses Thema entschieden habe. Noch am selben Tag habe ich mein erstes Fraktal programmiert.

Das Thema dieser Arbeit ist eine Vereinigung zweier grossen Interessensbereiche von mir. Daher hatte ich viel Motivation ein umfangreiches und sorgfältig gestaltetes Programm zu kreieren. Zusätzlich zu dem Programm verwende ich die Fraktale in verschiedenen Designs. Diese Arbeit ist eines meiner eher grösseren Projekte und hat daher auch einen grossen Lernfaktor für mich.



2 Fraktale

2.1 Was sind Fraktale?

2.1.1 Begriff

Der Begriff Fraktal wurde vom französischen Mathematiker Benoit B. Mandelbrot 1975 eingeführt. Der Begriff kommt aus dem Lateinischen vom Adjektiv «fractus». Das zugehörige Verb ist «frangere» und bedeutet «brechen». Er selbst sagt dazu in seinem Buch «The Fractal Geometry of Nature»:

> *«I coined fractal from the Latin adjective fractus. The corresponding Latin verb frangere means 'to break': to create irregular fragments. It is therefore sensible – and how appropriate for our needs! – that, in addition to 'fragmented' (as in fraction or refraction), fractus should also mean 'irregular,' both meanings being preserved in fragment»*
>
> Mandelbrot 1982: 4

2.1.2 Definition

Die meisten Fraktale sind selbstähnlich, das heisst identische Kopien des ganzen Fraktals sind in einer vergrösserten Version des Fraktals wiederzufinden. Damit könnten Fraktale einfach definiert werden, so wie das auch Lauwerier in «Fraktale verstehen und selbst programmieren» macht:

> *«Ein Fraktal ist eine geometrische Figur, in der sich das gleiche Motiv in stets kleinerem Massstab wiederholt.»*
>
> Lauwerier 1992a: 9

Doch laut Mandelbrot sind Fraktale viel mehr als nur selbstähnliche Formen, sondern ein viel allgemeinerer Begriff. Fraktale sind unendlich raue Objekte, deren Oberfläche unabhängig vom Massstab rau bleibt. Somit brechen sie jede Art von Glattheit in unendlich kleinere Fragmente auf. Um Fraktale besser zu beschreiben, benutzte er die Hausdorff-Besicovitch-Dimension. Die Hausdorff-Dimension ist eine Art Massstab für Rauigkeit. Es gibt verschiedene Methoden sie zu berechnen. Zwei davon sind in den folgenden Abschnitten erläutert.

Die erste der beiden Methoden basiert auf der Selbstähnlichkeit des Objekts, daher wird diese Dimension oft auch Selbstähnlichkeitsdimension genannt. Die Methode funktioniert jedoch nur für selbstähnliche Fraktale. Wird zum Beispiel eine Linie in 2 gleich lange Stücke geteilt, so verkleinert sich die Länge der Stücke um den Faktor 1/2. Wird ein 2-dimensionales Objekt, wie zum Beispiel ein Quadrat, in 4 gleiche Stücke geteilt, d.h. um den Faktor 1/4 verkleinert, so verkleinert sich die Seitenlänge wieder um den Faktor 1/2. Analog verkleinert sich die Kantenlänge eines 3-dimensionalen Würfel wieder um den Faktor 1/2, wenn er in 8 gleiche grosse Stücke geteilt wird, d.h. um den Faktor 1/8 verkleinert wird (vgl. Sanderson 2017).

Zwischen dem Verkleinerungsfaktor 1/s einer Länge und dem entsprechenden Verkleinerungsfaktor 1/a des Objekts gilt die Beziehung $1/a = (1/s)^D$ bzw. $a = s^D$, wobei D die Dimension des Objekts ist. Die zweite Beziehung $a = s^D$ kann nun zur Berechnung der Selbstähnlichkeitsdimension verwendet werden:

$$D = log_s a = ln(a)/ln(s)$$

Die Hausdorff- bzw. Selbstähnlichkeitsdimension der Linie ist also 1, des Quadrats 2 und des Würfels 3.

Wird nun der gleiche Prozess bei einem einfachen, selbstähnlichen Fraktal, wie zum Beispiel dem Sierpinski-Dreieck angewendet ergibt sich hier a = 3 und s = 2, da das Sierpinski-Fraktal nach der Teilung einer Seite in 2 Stücke aus 3 identischen Kopien von sich selbst besteht, welche jeweils eine halb so grosse Seitenlänge wie das ganze Fraktal aufweisen. Wird nun D mithilfe der bereits erwähnten Formel berechnet, ergibt sich ungefähr 1.58496.

Hausdorff-Dimensionen aus der reellen, nicht ganzzahligen Zahlenmenge sind typisch für Fraktale und auch der Grund weshalb Mandelbrot den Begriff von «frangere» bzw. «gebrochen» abgeleitet hat.

Die zweite Methode die Hausdorff-Dimension zu berechnen, nennt sich «Kästchenzählen» (Falconer 1993: 43). Sie funktioniert für jedes Objekt beliebiger Dimension und somit auch für nicht selbstähnliche Fraktale. Zuerst wird der Raum in gleichgrosse kleine Kästchen mit der möglichst kleinen Grösse $\varepsilon > 0$ aufgeteilt. Dann werden alle Kästchen N gezählt, welche das Fraktal berühren. Dies wird wiederholt für verschieden grosse ε, also verschiedene Skalierungen s. Werden diese Punkte auf einem Koordinatensystem mit N als y-Achse und s als x-Achse eingesetzt, kann eine exponentielle Funktion $N = cs^D$ mit dem Exponenten D, der Hausdorff-Dimension, und der Konstante c erkannt werden. Mithilfe des Logarithmus ergibt sich die Formel:

$$ln(N) = ln(c) + D * ln(s)$$

Somit ist D die Steigung einer linearen Funktion und kann für jedes Fraktal numerisch berechnet werden. Es gibt jedoch vereinzelt Fraktale, bei welchen die Dimension bei unterschiedlichen Skalierungen variiert (vgl. Sanderson 2017). Solche Fraktale werden multifraktale Systeme genannt.

Mithilfe der Hausdorff-Besicovitch-Dimension definierte Mandelbrot 1975 die Fraktale. Seine Definition lautet wie folgt:

> *«A fractal is by definition a set for which the Hausdorff Besicovitch dimension strictly exceeds the topological dimension.»*
>
> Mandelbrot 1982: 15

Die topologische Dimension bzw. Lebesguesche-Überdeckungsdimension ist die ganzzahlige Dimension des Objekts selbst. Eine Kurve hat zum Beispiel eine Lebesguesche-Überdeckungsdimension von 1, ein Quadrat hat eine von 2 et cetera. Sie ist nicht zu verwechseln mit der Dimension des Raumes. Die meisten Fraktale besitzen, wie bereits erwähnt eine nicht ganzzahlige Hausdorff-Dimension, welche somit grösser ist als deren Lebesguesche-Überdeckungsdimension. Es gibt vereinzelt Fraktale, welche ganzzahlige Dimensionen aufweisen. Solche Fraktale besitzen eine äusserst raue Oberfläche.

2.1.3 Fraktale in der Natur

Mandelbrot wollte mit seiner Definition ein Gebilde definieren, welches viel natürlicher ist als die typischen, perfekten und unendlich glatten Formen der Geometrie. Sein Ziel war es, wie es auch schon viele andere Mathematiker versucht haben, die Natur zu beschreiben. In seinem Buch «The Fractal Geometry of Nature», vereint er die Welt der Fraktale mit der Natur.

Ausserdem gibt es zahlreiche Beispiele von selbstähnlichen Fraktalen in der Natur. Bäume sind ein sehr typisches Beispiel für ein Fraktal in der Natur. Auch verschiedenes Gemüse, wie zum Beispiel Broccoli oder Romanesco weisen einen hohen Grad von Selbstähnlichkeit auf. Mandelbrot (1982:25) selbst nennt auch Küstenlinien als typisches Beispiel von natürlichen Fraktalen. Die Küste Britanniens hat zum Beispiel eine Hausdorff-Dimension von ungefähr 1.21.

2.2 Verschiedene Beispiele

In den folgenden Kapiteln werden verschiedene Beispiele von einfachen und komplexen Fraktalen erläutert, mit dem Ziel einen guten Überblick über die Fraktale zu verschaffen, ohne direkt von der enormen Vielfalt der Fraktale überwältigt zu werden.

Mit Ausnahme der Cantor-Menge sind alle diese Fraktale in der Applikation enthalten und deren Parameter sind durch den Benutzer verstellbar. Im 3. Kapitel wird noch genauer beschrieben, wie diese Fraktale programmiert wurden.

2.2.1 Cantor-Menge

Das wohl einfachste Fraktal ist die Cantor-Menge, eingeführt von Georg Cantor, Erfinder der Mengenlehre. Es wird oft als das erste Fraktal überhaupt und Grundlage der Fraktale von Mandelbrot bezeichnet.

Die Cantor-Menge beginnt mit einer Linie, welche in drei gleiche Stücke geteilt wird. Wird das mittlere Drittel entfernt, ergeben sich zwei neue Linien. Wird nun dieser Vorgang an den jeweils resultierenden Linien unendlich mal wiederholt, ergibt sich die Cantor-Menge.

Hat die ursprüngliche Linie eine Länge von 1, ergeben sich nach n Iterationen 2^n Linien mit einer Länge von jeweils 3^{-n} . Die Gesamtlänge ist also nach n Iterationen $(2/3)^n$ und da $\lim_{n \to \infty}(2/3)^n = 0$ hat die Cantor-Menge eine Länge von 0. Die Cantor-Menge ist jedoch keine leere Menge. Die Cantor-Menge besteht aus überabzählbar unendlich vielen Zahlen.

Da die Cantor-Menge selbstähnlich ist, kann ihre Selbstähnlichkeitsdimension berechnet werden. Da bei jeder Iteration zwei Stücke mit einer um den Faktor 1/3 verkleinerten Länge entstehen, ergibt sich für $D = \ln(2) / \ln(3) \approx 0.6309$. Die Lebesguesche-Überdeckungsdimension der Cantor-Menge ist 0 (vgl. Barile / Weisstein 2020).

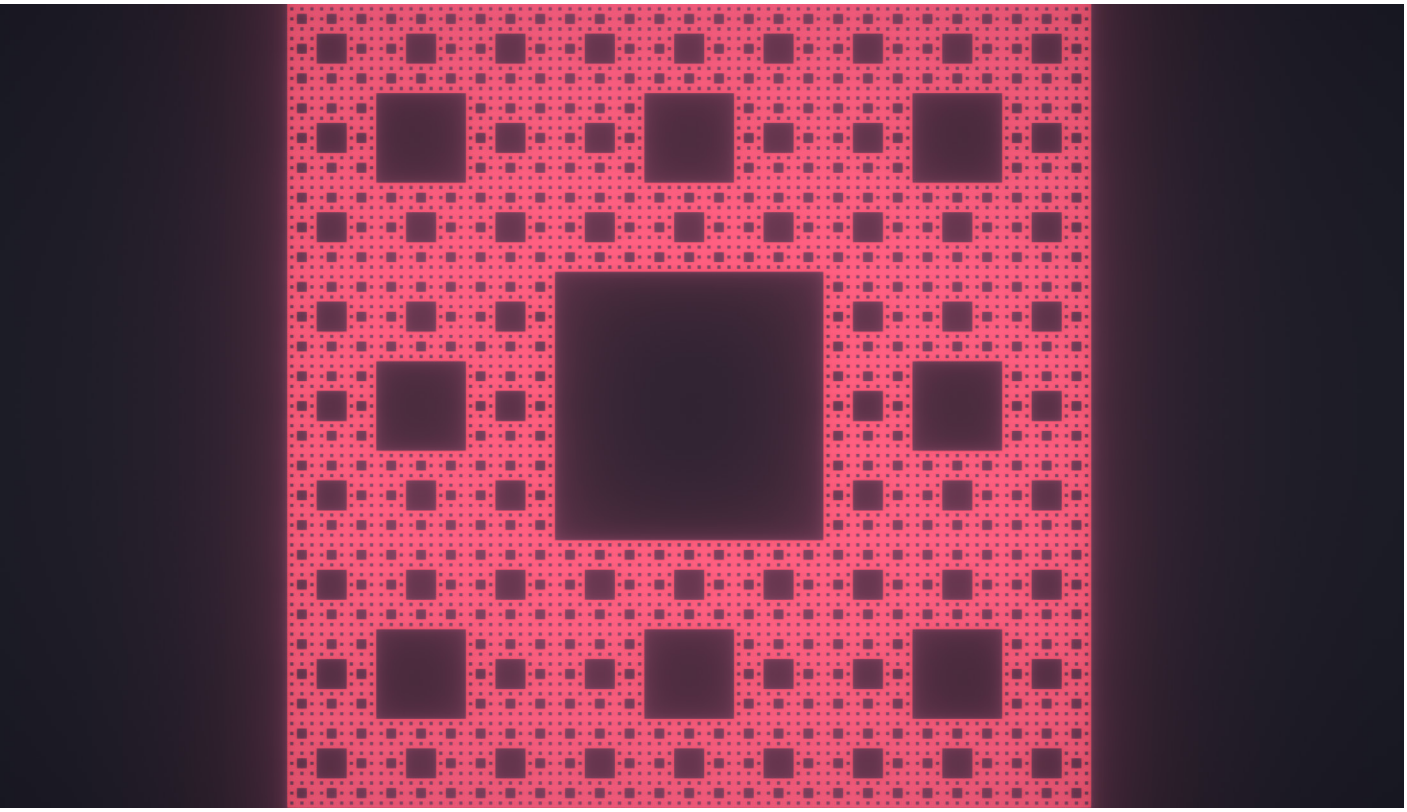
Die Cantor-Menge existiert auch in höheren Dimensionen. Die n-dimensionale Cantor-Menge nennt sich den Cantor-Staub und hat in jeder Dimension eine Lebesguesche-Überdeckungsdimension von 0 und eine Mass (n-dimensionales Analogon zu Volumen und Fläche) von 0. In der Ebene ist seine Hausdorff-Dimension 1.2619 und 1.8928 im Raum.

Das Cantor-Fraktal oder Variationen davon kommen immer wieder in anderen Fraktalen vor. So zum Beispiel im Sierpinski-Dreieck und Sierpinski-Teppich und deren 3-dimensionale Analoga, im Mandelbrot-Fraktal und sogar in der Mandelbox. Die Cantor-Menge ist eine so grundlegende Idee, dass sie sogar schon im antiken Ägypten auf einer Säule zu finden ist (vgl. Jollois, Jean-Baptiste P. / Devilliers, Édouard (1809): *Description d'Égypte*).

2.2.2 Sierpinski-Fraktale

Der polnische Mathematiker Waclaw Sierpinski ist der Erfinder des Sierpinski-Dreiecks und des Sierpinski-Teppichs. Das Sierpinski-Dreieck erfand er im Jahr 1915 und den Sierpinski-Teppich im darauffolgenden Jahr. Beide Fraktale basieren auf einem ähnlichen Prinzip wie die Cantor-Menge.

Für das Sierpinski-Dreieck wird ein gleichseitiges Dreieck benötigt. Das Fraktal kann zwar auch mit beliebigen Dreiecken erstellt werden, jedoch wird dies üblicherweise nicht gemacht. Wird dieses Dreieck in vier gleiche Stücke geteilt und das mittlere entfernt, wurde ein Iterationsschritt durchgeführt. Wird nun der gleiche Prozess, bei den drei jeweils resultierenden Dreiecken, unendlich mal wiederholt, so ergibt sich das Sierpinski-Dreieck.



Für den Sierpinski-Teppich wird hingegen mit einem Quadrat gestartet. Gleich wie beim Sierpinski-Dreieck wird die Form zuerst in gleiche Teile geteilt. Beim Sierpinski-Teppich sind es 9 Teile. Dann wird das mittlere Stück entfernt. Wird nun der gleiche Prozess, bei den acht jeweils resultierenden Quadraten, unendlich mal wiederholt, so ergibt sich der Sierpinski-Teppich. Hier wird die Ähnlichkeit zur Cantor-Menge noch einmal besonders klar. Das Fraktal wird auch als eine 2-dimensionale Version der Cantor-Menge bezeichnet.

Beide Fraktale haben eine Lebesguesche-Überdeckungsdimension von 1, da eine Kurve durch jeden seiner Punkte definiert werden kann (vgl. Mandelbrot 1982: 144). Das Sierpinski-Dreieck hat, wie schon einmal erwähnt, eine Hausdorff-Dimension von $D = \ln(3) / \ln(2) \approx 1.58496$, da es aus 3 identischen Kopien halber Seitenlänge besteht. Der Sierpinski-Teppich besteht aus 8 identischen Kopien, deren Seitenlängen 3-mal kleiner sind, und hat somit eine Hausdorff-Dimension von $D = \ln(8) / \ln(3) \approx 1.89279$.

Es existieren auch Versionen des Sierpinski-Fraktals, welche auf gleichmässigen Polygonen mit mehr Ecken basieren, jedoch sind diese nicht so perfekt, wie die dreieckige und die quadratische Version.

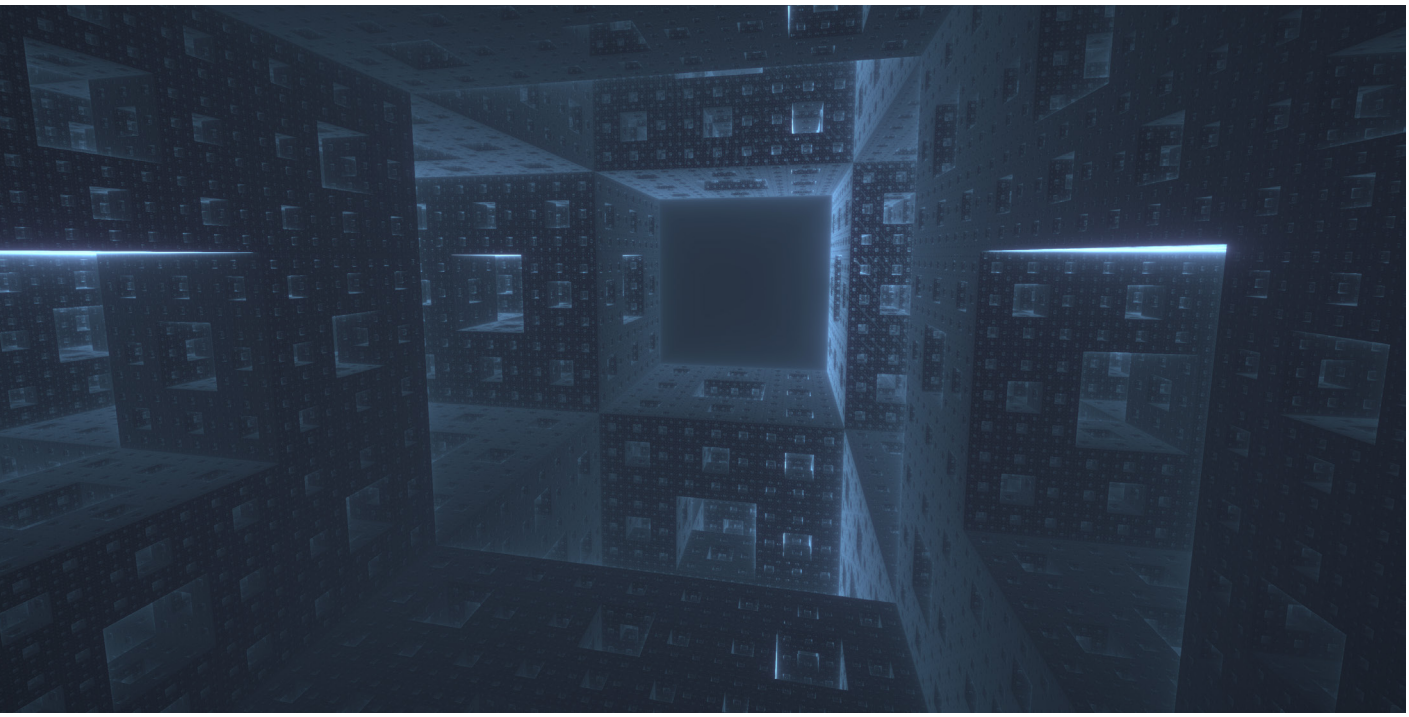
Für das Sierpinski-Dreieck gibt es zahlreiche 3-dimensionale Versionen. Die zwei intuitivsten sind die Sierpinski-Pyramide und der Sierpinski-Tetraeder. Sie unterscheiden sich im Anfangsobjekt. Die Sierpinski-Pyramide besteht aus einer gleichseitigen Pyramide mit einer quadratischen Grundfläche. Der Sierpinski-Tetraeder hingegen aus einem gleichmässigen Tetraeder. Ein weiteres 3-dimensionales Analogon des Sierpinski-Dreiecks ist das Sierpinski-Oktaeder, oder auch die Octahedron-Flake genannt. Auf dieses Fraktal wird später noch genauer eingegangen.

Das bekannteste 3-dimensionale Analogon des Sierpinski-Teppichs ist der Menger-Schwamm. Er wird im nächsten Kapitel noch genauer erläutert.

2.2.3 Menger-Schwamm

Der Menger-Schwamm wurde 1926 von Karl Menger eingeführt. Da eine Kurve durch jeden seiner Punkte definiert werden kann, hat er eine Lebesguesche-Überdeckungsdimension von 1, wie auch das Sierpinski-Dreieck und der Sierpinski-Teppich (vgl. Mandelbrot 1982: 144). Der Menger-Schwamm ist ein 3-dimensionales Analogon des Sierpinski-Teppichs und der Cantor-Menge.

Die Konstruktion startet mit einem Würfel. Zuerst wird der Würfel in 27 gleiche Stücke geteilt. Die Oberfläche wird also gleich unterteilt wie beim Sierpinski-Teppich. Dann werden jeweils die mittleren Würfel einer Seite und der Würfel im Inneren entfernt. Mit den jeweils verbleibenden 20 Würfeln wird der gleiche Prozess durchgeführt und nach unendlich vielen Iterationen ergibt sich der Menger-Schwamm.



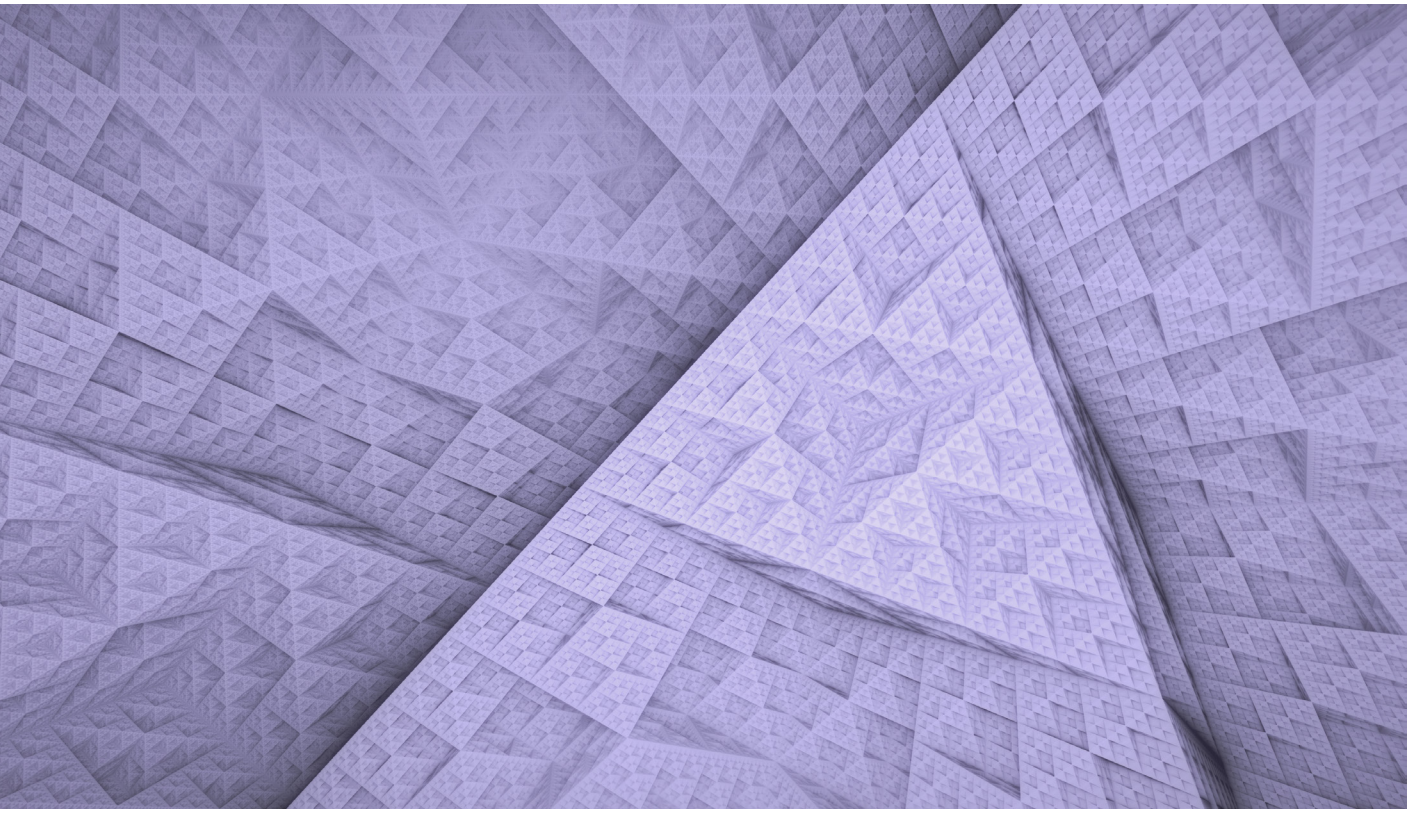
Da das Fraktal aus 20 identischen Kopien von sich selbst besteht, dessen Seitenlängen jeweils um den Faktor 1/3 verkleinert wurden, hat es eine Hausdorff-Dimension von $\ln(20) / \ln(3) \approx 2.72683$. Nach n Iterationen besteht der Menger Schwamm aus 20^n Würfeln. Hat das Fraktal zu Beginn eine Seitenlänge von 1 ist das Volumen nach n Iterationen $(20/27)^n$. Da $\lim_{n \to \infty}(20/27)^n = 0$ hat das Fraktal ein Volumen von 0. Die Oberfläche des Menger-Schwamms divergiert hingegen, da in jedem Iterationsschritt mindestens 4-mal so viel neue Oberfläche entsteht, als entfernt wird.

Werden nicht nur die 7 mittleren Würfel, sondern auch die restlichen, mit Ausnahme der 8 Würfel in den Ecken, entfernt und wird dieser Vorgang unendlich mal iteriert, entsteht 3-dimensionaler Cantor-Staub oder auch Menger-Schneeflocke genannt. Auch Cantor-Staub hat ein Volumen von 0 und eine unendliche Oberfläche, hat aber eine Lebesguesche-Überdeckungsdimension 0, wie bereits erwähnt.

Ein weiterer interessanter Aspekt des Menger-Schwamms sind seine Querschnitte. Wird der Menger Schwamm entlang einer Diagonale mit einer Ebene geschnitten, welche normal zu dieser Diagonale ist, ergeben sich interessante 2-dimensionale Fraktale, welche an die Sierpinski-Fraktale erinnern.

2.2.4 Octahedron-Flake

Wie bereits erwähnt, ist das Sierpinski-Oktaeder oder auch die Octahedron-Flake ein 3-dimensionales Analogon des Sierpinski-Fraktals. Anders als die anderen 3-dimensionalen Sierpinski-Fraktale besteht es aus einem Oktaeder.



Das Startobjekt des Sierpinski-Oktaeders ist ein gleichmässiges Oktaeder. Ein Iterationsschritt besteht darin das Oktaeder durch 6 Oktaeder halber Seitenlänge zu ersetzen. Jeden der Oktaeder wird so in eine Ecke platziert, dass alle Oktaeder sich innerhalb des ursprünglichen Oktaeders befinden, sich jedoch gegenseitig nur berühren und nicht überlappen. Durch unendliche Iteration an den jeweils resultierenden Oktaedern ergibt sich das Sierpinski-Oktaeder.

Das Sierpinski-Oktaeder besteht aus 6 gleichen Oktaedern, welche jeweils um den Faktor 1/2 verkleinert wurden. Somit beläuft sich die Hausdorff-Dimension auf $D = \ln(6) / \ln(2). \approx 2.58496$. Ist das Startvolumen des Fraktals 1, hat es nach n Iterationen ein Volumen von $(6/8)^n$ und da $\lim_{n \to \infty}(6/8)^n = 0$ hat das Fraktal ein Volumen von 0. Die Oberfläche hingegen strebt nach Unendlich.

2.2.5 Komplexe Zahlen

Um die folgenden Fraktale zu verstehen, ist es notwendig zuerst ein gutes Verständnis der komplexen Zahlen zu haben. In diesem Kapitel werden die Grundlagen der komplexen Zahlen kurz und knapp erklärt.

Die komplexen Zahlen $\mathbb{C}$ ergänzen die Zahlenmenge der reellen Zahlen. Die Gleichung $x^2 + 1 = 0$ hat keine reelle Lösung, sondern nur eine in der Menge der komplexen Zahlen. Eine Lösung dieser Gleichung ist die imaginäre Zahl i. Die Zahl i hat daher die Eigenschaft, dass ihr Quadrat -1 ergibt. Eine komplexe Zahl z besteht jeweils aus einem reellen Teil und einem imaginären Teil. Es gilt:

$$z = a + bi,\ a = Re(z),\ b = Im(z)$$

Eine komplexe Zahl z kann auf der Gaussschen Zahlenebene als Punkt mit kartesischen Koordinaten dargestellt werden. Diese Zahlenebene besteht aus einer x-Achse, welche den Realteil von z trägt, und einer y-Achse, welche den Imaginärteil von z trägt.

Eine komplexe Zahl kann auch in der Polarform geschrieben werden. Die Polarkoordinaten einer komplexen Zahl z sind der Radius r und der Winkel φ und werden wie folgt berechnet:

$$r = |z| = \sqrt{a^2 + b^2}$$

$$\varphi = arg(z) = arctan(b/a)$$

Abhängig davon in welchem Quadranten sich z befindet, muss φ dementsprechend angepasst werden.

Die komplexe Zahl kann auch mithilfe der Polarform wie folgt geschrieben werden:

$$z = r(cos(\varphi) + i * sin(\varphi))$$

Für die Addition und die Multiplikation gelten das Kommutativgesetz, sowie das Assoziativgesetz. Auch das Distributivgesetz gilt bei den komplexen Zahlen. Die typischen Operationen sind auch in $\mathbb{C}$ recht intuitiv. Das Resultat einer Operation in $\mathbb{C}$ ist immer eine komplexe Zahl. Die Addition zweier komplexen Zahlen c und z funktioniert wie folgt:

$$c + z = (Re(c) + Re(z)) + i * (Im(c) + Im(z))$$

Die Subtraktion, analog zur Addition:

$$c - z = (Re(c) - Re(z)) + i * (Im(c) - Im(z))$$

Die Multiplikation ist etwas komplexer:

$$c * z = (Re(c) * Re(z) - Im(c) * Im(z)) + i * (Re(c) * Im(z) + Im(c) * Re(c))$$

Bei der Division muss der Bruch zuerst um das konjugiert komplexe erweitert werden:

$$\frac{c}{z} = \frac{c * (Re(z) - Im(z) * i)}{z * (Re(z) - Im(z) * i)}$$

$$= \frac{Re(c) * Re(z) + Im(c) * Im(z)}{Re(z)^2 + Im(z)^2} + i * \frac{Im(c) * Re(z) - Re(c) * Im(z)}{Re(z)^2 + Im(z)^2}$$

Die Division und Die Multiplikation können auch in der Polarform ausgedrückt werden und sind so wesentlich einfacher:

$$c * z = r_c r_z(cos(\varphi_c + \varphi_z) + i * sin(\varphi_c + \varphi_z))$$

$$c/z = (r_c/r_z)(cos(\varphi_c - \varphi_z) + i * sin(\varphi_c - \varphi_z))$$

2.2.6 Mandelbrot-Menge

Die Mandelbrot-Menge, benannt nach dem berühmten französischen Mathematiker Benoit B. Mandelbrot, ist eines der ersten komplexen Fraktale und zugleich das wahrscheinlich bekannteste Fraktal. Es ist bekannt für seine eindrückliche Form und bemerkenswerten farbigen Bildern, die durch Hineinzoomen entstehen.

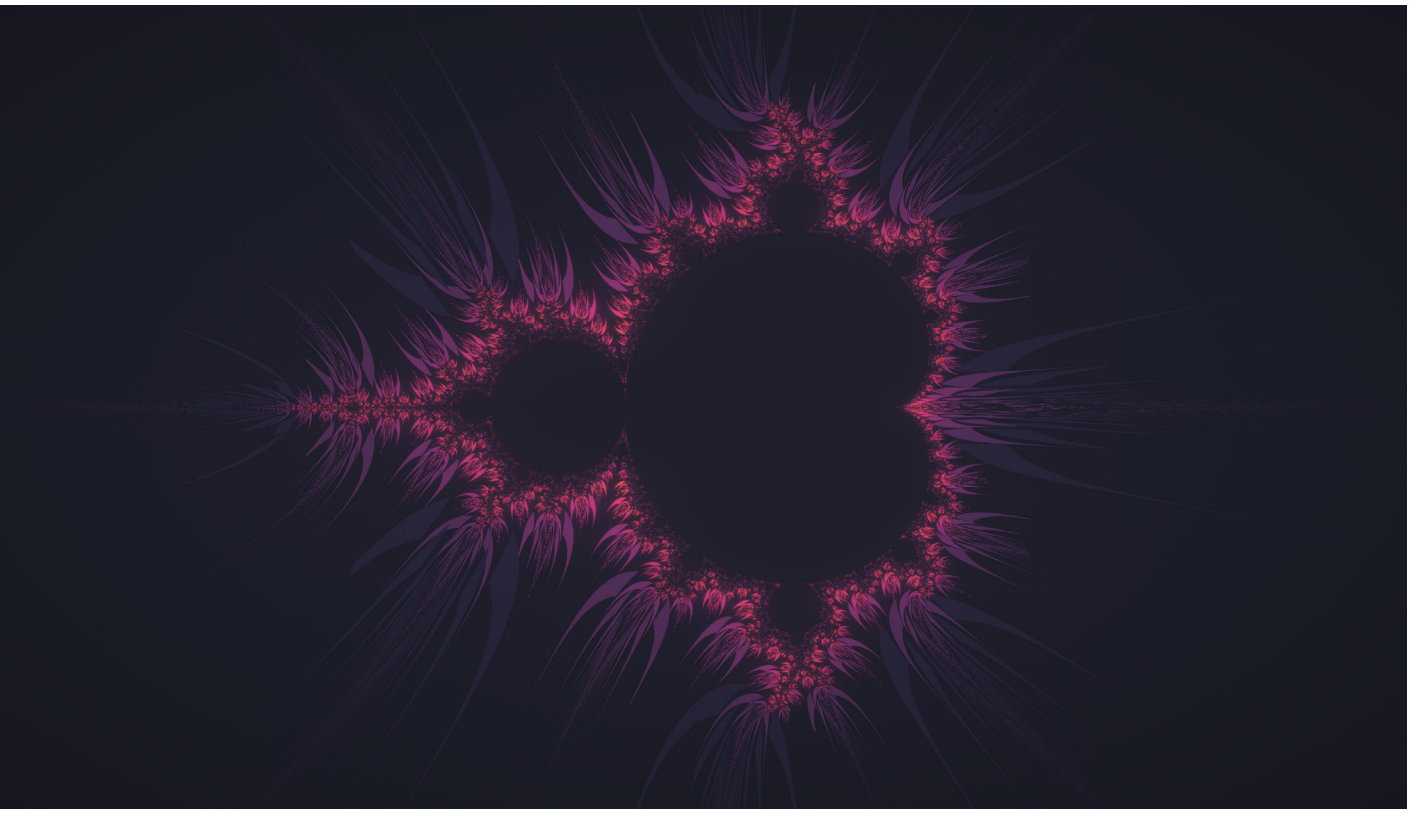
Die Definition der Mandelbrot-Menge lautet wie folgt (vgl. Falconer 1993: 236):

$$f_c(z) = z^2 + c$$

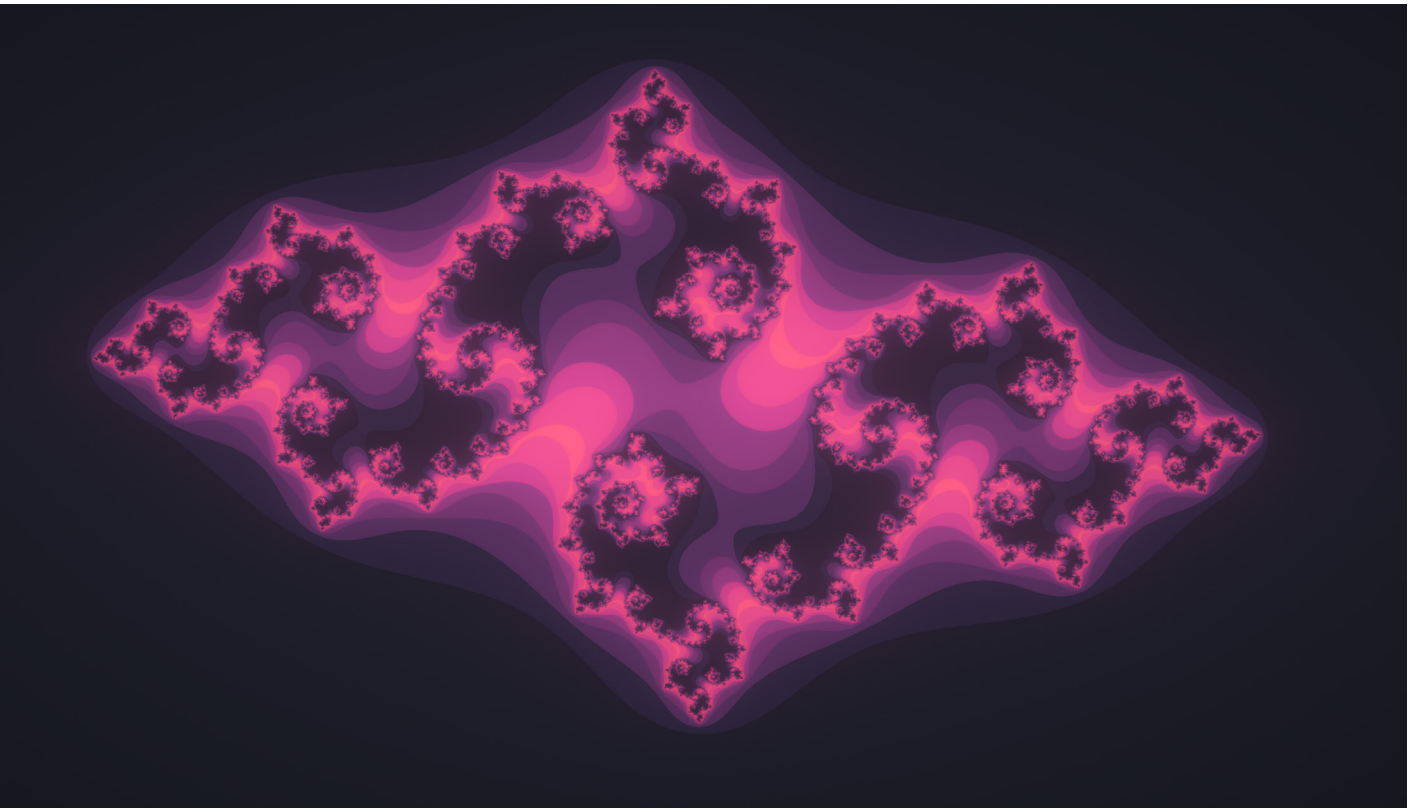
$$M = \{c \in \mathbb{C} : f_c^k(0) \nrightarrow \infty,\ k \to \infty\}$$

In anderen Worten, die Mandelbrot-Menge ist die Menge aller komplexen Zahlen c, für welche die Iteration $z_{n+1} = z_n^2 + c$, mit dem Startwert $z_0 = 0$, nicht divergiert.

Diese Definition bildet auch die Grundlage für computergenerierte Mandelbrot-Fraktale. Typische Mandelbrot-Fraktal-Bilder werden generiert, indem jeder Pixel der Bildebene einem Punkt c auf der Gaussschen Zahlenebene zugewiesen wird und $z_{n+1} = z_n^2 + c$, mit dem Startwert $z_0 = 0$, n mal iteriert wird.



Für jede komplexe Zahl c existiert zusätzlich noch eine Julia-Menge. Wird nämlich für z, anstatt für c den aktuellen Pixel bzw. Punkt auf der Gaussschen Zahlenebene verwendet und für c eine beliebige komplexe Zahl, so entsteht nach unendlicher Iteration der Mandelbrot-Funktion eine Julia Menge. Die Julia-Mengen sind benannt nach Gaston Maurice Julia, welcher sie mit Pierre Fatou im Jahr 1919, also vor dem Mandelbrot-Fraktal, erstmals erwähnt hat.



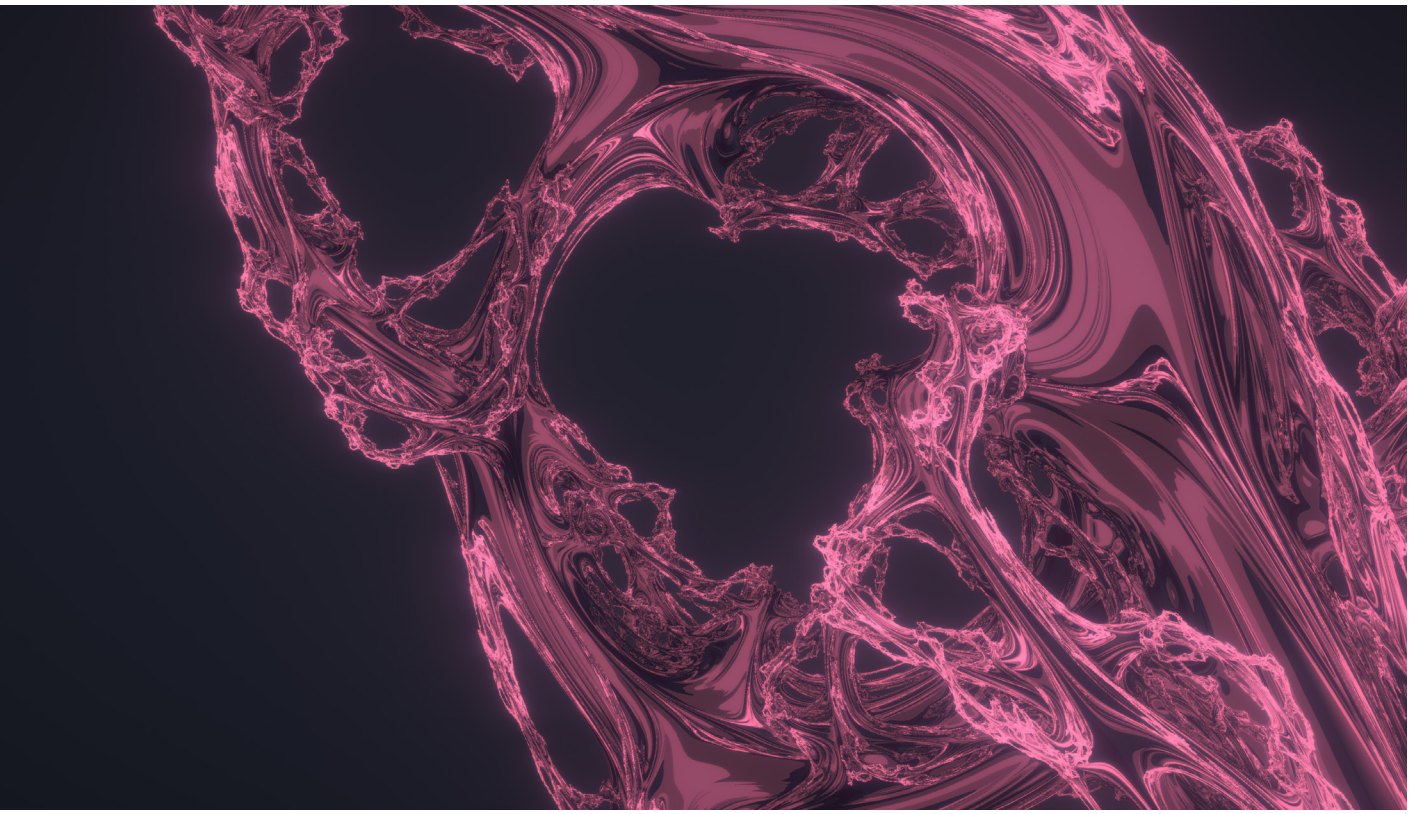
Das Mandelbrot-Fraktal ist eine Art Karte für alle Julia Mengen. Wird die Julia-Menge einer Konstante c mit der dazugehörigen Position der Mandelbrot-Menge verglichen, lassen sich starke Ähnlichkeiten feststellen. Befindet sich c innerhalb der Mandelbrot-Menge, ist die Julia-Menge zusammenhängend. Befindet sich c ausserhalb der Mandelbrot-Menge, ist die Julia-Menge unzusammenhängend.

Die Lebesguesche-Überdeckungsdimension des Randes der Mandelbrot-Menge ist 1 und die Hausdorff-Dimension des Randes ist 2, wie durch ein Ergebnis von Shishikura (1991:1) bestimmt wurde. Somit ist das Mandelbrot-Fraktal eines der wenigen Fraktale, welches eine ganzzahlige Hausdorff-Dimension aufweist.

3-dimensionale Analoga des Mandelbrot-Fraktals sind zum Beispiel das Mandelbulb-Fraktal oder die Mandelbox, beide werden später noch genauer erläutert. Sowie auch das Mandelbrot-Fraktal besitzen diese 3-dimensionalen Versionen auch ihre eigenen Julia-Mengen. Es gibt zudem noch weitere 3- und sogar 4- dimensionale Versionen, welche hier aber nicht mehr genauer erläutert werden. Es gibt jedoch keine perfekte 3-dimensionale Analoga, da es kein 3-dimensionales Analogon für die komplexen Zahlen gibt. Es gibt aber ein 4-dimensionales Analogon für die komplexen Zahlen, und zwar die Quaternionen. Diese werden entweder mithilfe einer Projektion oder einem 3-dimensionalen Querschnitt im 3-dimensionalen Raum dargestellt.
(vgl. en.wikipedia.org/wiki/Mandelbrot_set#Higher_dimensions [Stand: 03.08.2020])

2.2.7 Mandelbulb

Das 3-dimensionale Mandelbulb-Fraktal wurde 2007 von Daniel White erstmals eingeführt. Da es kein Analogon der komplexen Zahlen im 3-dimensionalen Raum gibt, basiert es auf «hyperkomplexen Zahlen» (Nylander 2009).



Das Mandelbulb-Fraktal wird wie das Mandelbrot-Fraktal mit der Iteration $w_{n+1} = w_n^2 + c$, berechnet. Die «hyperkomplexe Zahl» w besteht hier jedoch aus 3 Komponenten (x, y, z). Anstatt w mithilfe der kartesischen Koordinaten zu quadrieren, was im 3-dimensionalen Raum, wie bereits erwähnt, nicht möglich ist, benutzt White die Formel für die Multiplikation mithilfe der Polarkoordinaten und passt sie für den 3-dimensionalen Raum an (vgl. Nylander 2009). Somit wird aus der Formel für das Quadrieren von komplexen Zahlen:

$$z^2 = r^2(cos(2\varphi) + i * sin(2\varphi))$$

eine Formel, welche eine «hyperkomplexe Zahl» w(x, y, z) quadriert:

$$w^2 = r^2(sin(2\theta)cos(2\varphi), sin(2\theta)sin(2\varphi), cos(2\theta))$$

wobei

$$r = \sqrt{x^2 + y^2 + z^2}$$
$$\theta = arccos(z/r)$$
$$\varphi = arctan2(y, z)$$

Diese Formel kann auch für beliebige Hochzahlen verallgemeinert werden:

$$w^p = r^p(sin(p\theta)cos(p\varphi), sin(p\theta)sin(p\varphi), cos(p\theta))$$

Und analog für die Iterationsfunktion:

$$w_{n+1} = {w_n}^p + c$$

Die Addition erfolgt analog wie bei den komplexen Zahlen:

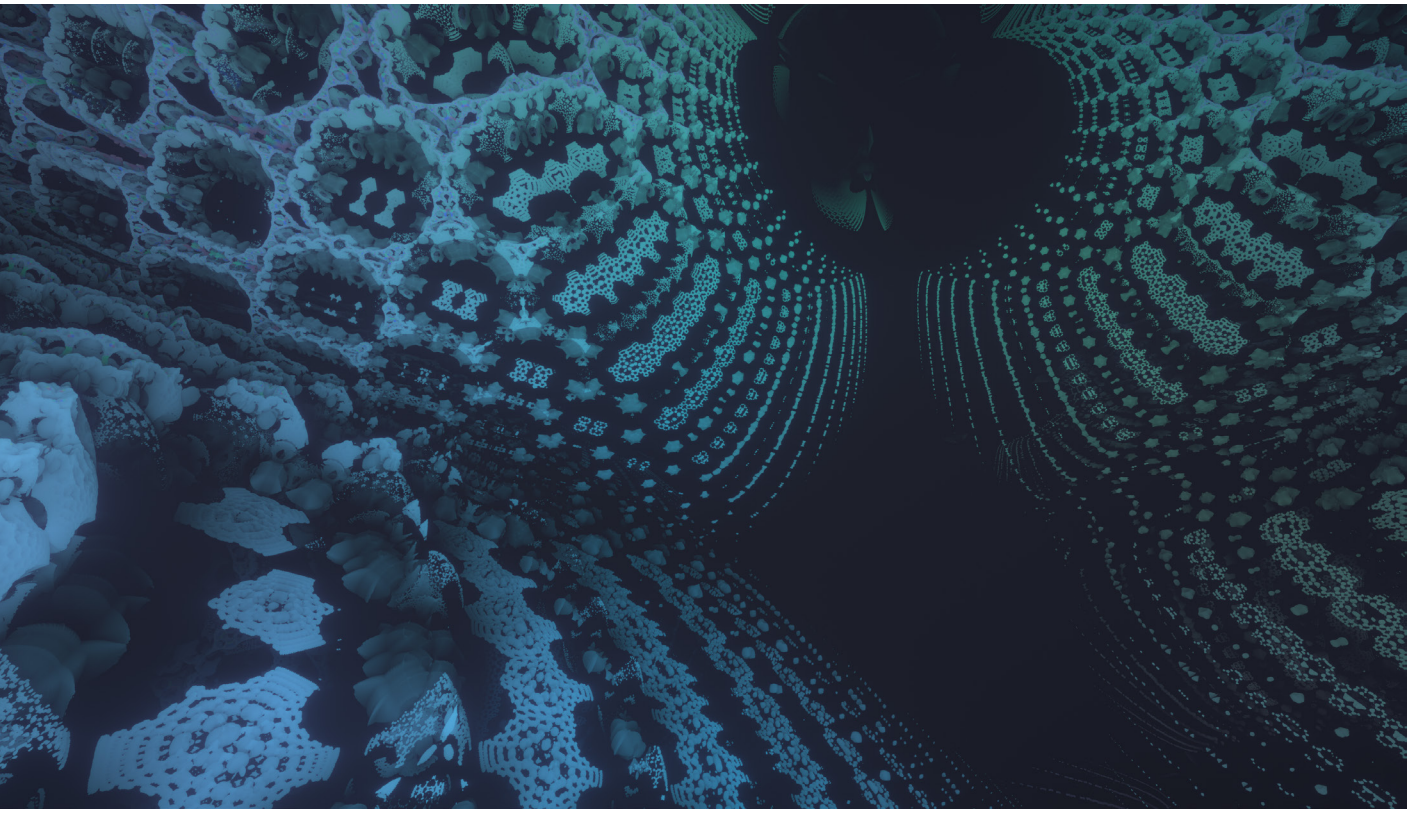
$$w + c = (x_w + x_c, y_w + y_c, z_w + z_c)$$

Das Mandelbulb-Fraktal ist ein sehr neues Fraktal und ist daher recht wenig erforscht. Es wird vermutet, dass das Mandelbulb-Fraktal eine Hausdorff-Dimension von 3 besitzt, dies ist jedoch noch nicht bewiesen (vgl. fractalforums.com/theory/hausdorff-dimension-of-the-mandelbulb [Stand 04.08.2020]).

Wie auch das Mandelbrot-Fraktal besitzt das Mandelbulb-Fraktal Julia-Mengen. Sie sind jedoch viel vielfältiger als die Mandelbrot-Julia-Mengen, da sie nicht nur durch zwei, sondern durch drei Parameter kontrolliert werden.

Das Mandelbulb-Fraktal weist starke Ähnlichkeiten mit seinem 2-dimensionalen Äquivalent auf. Auch die 3-dimensionalen Julia-Mengen gleichen stark deren 2-dimensionalen Äquivalenten.

2.2.8 Mandelbox



Das Mandelbox-Fraktal wurde von Tom Lowe im Jahr 2010 erstmals beschrieben. Es ist ein weiterer Versuch eine 3-dimensionale Version des Mandelbrot-Fraktals zu finden. Das Mandelbox-Fraktal löst das Problem des fehlenden Äquivalenten von komplexen Zahlen, in dem es ganz auf komplexe Zahlen verzichtet und die Mandelbrot-Funktion durch

$$v_{n+1} = s * f_{box}(f_{sphere}(v_n)) + c$$

ersetzt, wobei s ∈ $\mathbb{R}$, v ∈ $\mathbb{R}^3$, c ∈ $\mathbb{R}^3$.

f_{box} und f_{sphere} sind sogenannte «Folds». Folds werden im Kapitel 3.5.2 noch genauer beschrieben, in diesem Kapitel werden nur Box- und Sphere-Folds definiert.

Der Box-Fold $f_{box}(v)$ ist für v ∈ $\mathbb{R}^3$ folgendermassen definiert:

$$f_{box}(v) = (f_{box}(x_v), f_{box}(y_v), f_{box}(z_v)) \qquad v \in \mathbb{R}^3$$

$$f_{box}(a) = \begin{cases} 2 - a, & a > 1 \\ -2 - a, & a < -1 \\ a, & 1 \geq a \geq -1 \end{cases} \qquad a \in \mathbb{R}$$

Der Sphere-Fold $f_{sphere}(v)$ ist für $v \in \mathbb{R}^3$ folgendermassen definiert:

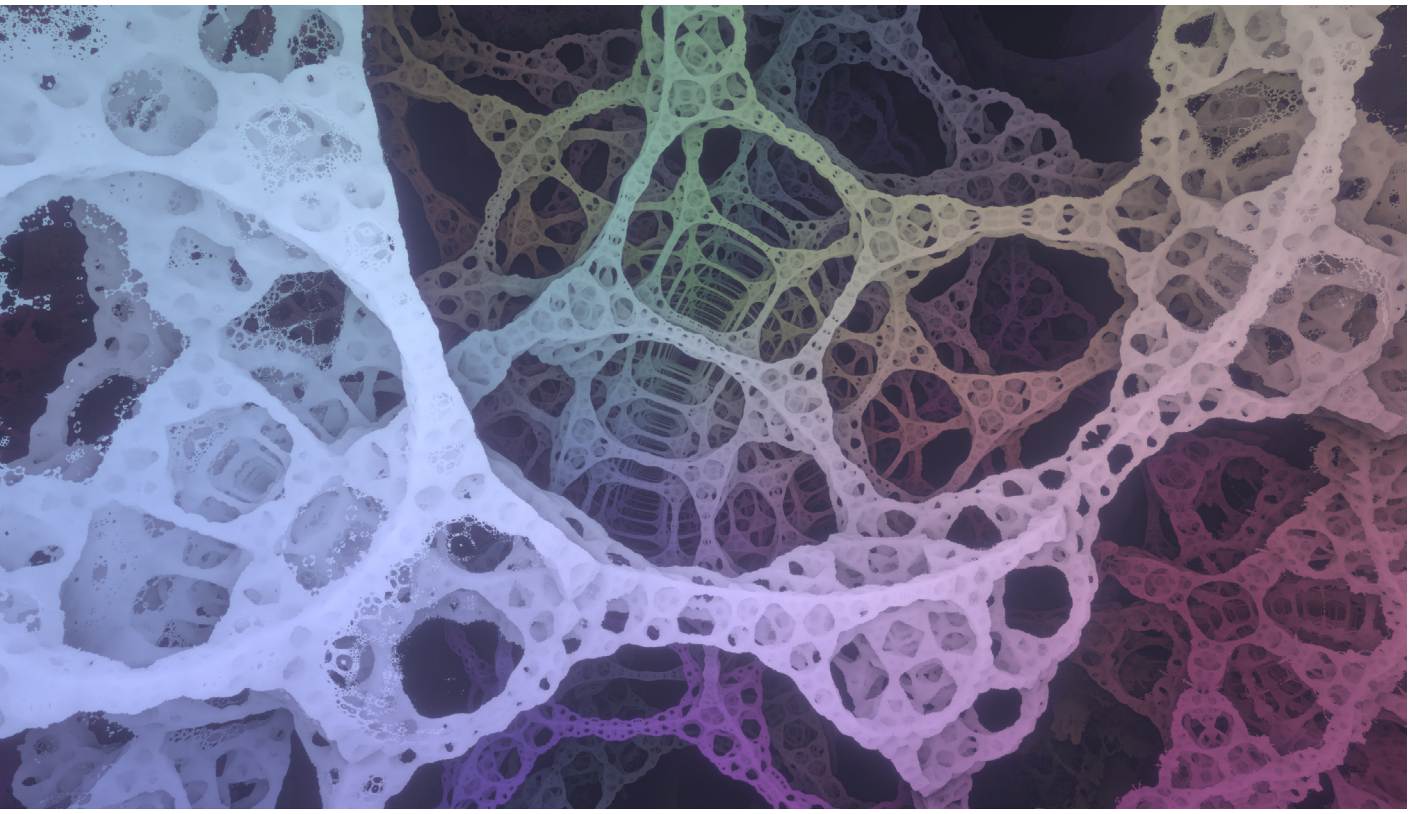
$$f_{sphere}(v) = \frac{v}{|v|} * f_{sphere}(|v|) \qquad v \in \mathbb{R}^3$$

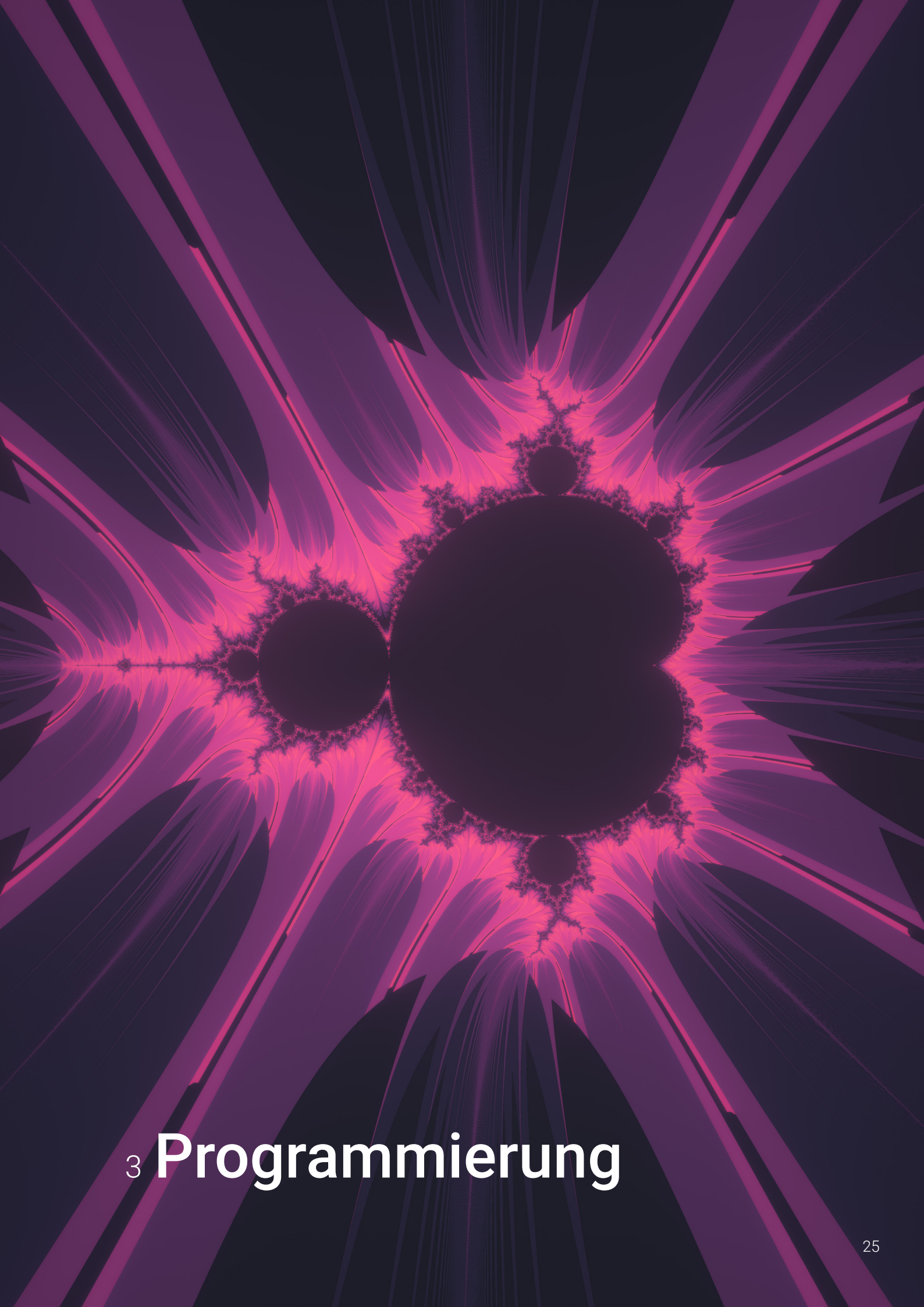
$$f_{sphere}(m) = \begin{cases} m/r^2, & m < r \\ 1/m, & m \leq |v| < 1 \\ m, & m \geq 1 \end{cases} \qquad m \in \mathbb{R},\ r = 0.5$$

Die Variable s, welche in der Iterationsfunktion vorkommt, wird auch Scale genannt und ist eines der interessantesten Variablen des Mandelbox-Fraktals. Es können aber auch andere Variablen, wie zum Beispiel r, verändert oder das Resultat der Funktion $f_{sphere}(v)$ um einen beliebigen Faktor multipliziert werden. Dadurch hat die Mandelbox eine enorme Vielfalt.

Das Mandelbox-Fraktal kann einfach auf n-Dimensionen verallgemeinert werden. Somit ist es dimensionsunabhängig. Da die Mandelbox ein multifraktales System ist, kann die Hausdorff-Dimension nicht eindeutig bestimmt werden. Es ist jedoch bewiesen (Rudi 2014), dass das Mandelbox-Fraktal für $1 < |s| < 2$ eine Hausdorff-Dimension, gleich der aktuellen räumlichen Dimension, besitzt.

Da die Mandelbox, wie das Mandelbrot-Fraktal, Julia-Mengen besitzt, kann sie auch durch die Julia-Mengen definiert werden: Das Mandelbrot-Fraktal ist die Menge aller $c \in \mathbb{R}^3$, für welche die zugehörige Julia-Menge zusammenhängend ist.





3 Programmierung

3.1 Umgebung

Für das Programmieren der Fraktale habe ich eine mir bereits bekannte Programmierumgebung gewählt. Ich habe mich für die Game-Engine Unity entschieden, da ich bereits gute Erfahrungen mit ihr gemacht habe. Eine Game-Engine wird normalerweise verwendet, um Games zu entwickeln, jedoch eignet sie sich auch perfekt für viele andere Bereiche. Der Vorteil einer Game-Engine ist, dass der Entwickler nicht alles von Grund aufbauen muss, sondern durch das umfangreiche Framework eine stabile Grundlage hat und sich so nur auf das Wichtigste konzentrieren kann.

3.1.1 Unity-Engine

Die wichtigsten Tools, welche Unity beinhaltet, sind:

- Eine umfangreiche Render-Engine, welche dem heutigen Standard entspricht
- Audio- und Video-Engine
- Animations- und Filmtools
- Scripting-Support mit der Programmiersprache C#
- Realistische Physik-Engine
- 2D-Support
- Plugins / Assets
- Multiplayer

Skripte in Unity werden mit der Programmiersprache C# erstellt. Skripte sind ein wichtiger Teil der Game-Engine, denn mit ihnen wird die ganze Spiellogik bzw. Programmlogik geschrieben und kontrolliert. Zusätzlich zu C# unterstützt Unity auch noch andere Programmiersprachen, wie zum Beispiel die Shadersprache HLSL, welche für diese Arbeit essenziell war. Die umfangreiche Unity API ermöglicht einen vollen Zugriff auf die Game-Engine und ist sehr übersichtlich und intuitiv gestaltet.

Das Hauptfenster von Unity besteht aus verschiedenen andockbaren Fenstern.
Die wichtigsten davon sind der Scene-View, die Hierarchie, der Inspektor und das Dateisystem. Der Scene-View ist der 3D-Arbeitsbereich, in dem die aktuelle Szene dargestellt ist.

Die Szene ist die wichtigste Einheit in Unity, wie bei vielen anderen 3D Programmen. Unity speichert sie als eine .unity Datei im Dateisystem. In einer Szene werden die verschiedenen Objekte, sogenannte «GameObjects» gespeichert. Die GameObjects einer Szene werden in der Hierarchie angezeigt. Wird eine neue Szene erstellt, erstellt Unity normalerweise ein Kamera GameObject und ein Licht GameObject. Aus einem GameObject kann ein sogenanntes Prefab erstellt werden, welches dann im Dateisystem gespeichert wird und alle Daten des GameObjects beinhaltet.

Ein Prefab ist eine Art Vorlage eines GameObjects und kann jederzeit manuell zu einer Szene hinzugefügt werden oder per Code zur Laufzeit:

```C#
GameObject obj = Instantiate(prefab, parent);
```

Wird ein GameObject in der Hierarchie ausgewählt, kann im Inspektor erkannt werden, dass ein GameObject aus verschiedenen Komponenten aufgebaut ist. Diese Komponenten sind, was ein GameObject ausmacht. Die Kamera hat zum Beispiel eine Kamera-Komponente und das Licht eine Licht-Komponente. Die meisten GameObjects haben eine Transform-Komponente, welche für die Position, Rotation und Skalierung des GameObjects im Raum verantwortlich ist.

Die Komponenten sind nichts anderes als Skripte bzw. Klassen, welche von der Klasse MonoBehaviour abstammen. Eine Komponente ist also ein MonoBehaviour. Ein MonoBehaviour-Objekt ist nicht statisch und kann zu beliebigen GameObjects hinzugefügt werden, bei welchen es dann unabhängig von anderen Instanzen ist (ausser statische Felder). In Unity können aber auch statische Klassen erstellt werden, diese können dann von allen anderen Klassen abgerufen werden. Das Hinzufügen von Komponenten zu GameObjects kann entweder manuell oder mithilfe von Code zur Laufzeit gemacht werden:

```C#
obj.AddComponent<Camera>();
```

Komponenten können auch ganz einfach abgerufen werden:

```C#
Camera cam = obj.GetComponent<Camera>();
```

Die zwei wichtigsten Methoden einer MonoBehaviour-Klasse ist Start und Update, wobei Start beim Programmstart aufgerufen wird und Update in jedem Frame. Es gibt jedoch noch viele weitere Methoden. Eine einfache MonoBehaviour-Klasse würde zum Beispiel so aussehen:

```C#
public class Example : MonoBehaviour {

    // Code, welcher zum Programmstart ausgeführt wird
    public void Start() { ... }

    // Code, welcher pro Frame ausgeführt wird
    public void Update() { ... }
}
```

3.1.2 Programmiersprachen

Für Unity wird hauptsächlich die objektorientierte Programmiersprache C# verwendet. C# ist ähnlich wie Java und wurde von Microsoft entwickelt. Die Syntax von C# ist fast gleich wie bei Java. C# ist jedoch umfangreicher und viel effizienter. Die Programmiersprache ist auf Windows, Mac, Linux und sogar Spielkonsolen verfügbar. Neben C und C++ ist sie die meistverwendete Sprache für Game-Development.

Viel wichtiger für diese Arbeit ist hingegen die Shadersprache HLSL bzw. High Level Shading Language. Shader-Code wird nicht wie bei den meisten Sprachen auf dem Prozessor, sondern auf der Grafikkarte ausgeführt. Die Vorteile einer Grafikkarte sind die enorme Parallelität und Geschwindigkeit. HLSL ist ähnlich wie C und ist nicht objektorientiert.

In Unity gibt es verschiedene Arten von Shader, die in HLSL programmiert werden. In dieser Arbeit wird hauptsächlich der auf DirectX 11 basierende Compute-Shader behandelt, da alle Fraktale in einem Compute-Shader programmiert wurden. Ein einfacher Compute-Shader, der den Bildschirm weiss färbt, sieht wie folgt aus:

```HLSL
#pragma kernel CSMain

RWTexture2D<float4> Texture;

[numthreads(8,8,1)]
void CSMain (uint3 id : SV_DispatchThreadID) {
   Texture[id.xy] = float4(1, 1, 1, 1);
}
```

Die wichtigste Zeile eines Compute-Shaders ist die Zeile «#pragma kernel CSMain», da sie den Namen des Kernels angibt. Da ein Compute-Shader ein Pixel-Shader ist, wird die Kernel Funktion in jedem Frame für jeden Pixel ausgeführt. Der Integer «id» gibt jeweils an für welchen Pixel die Funktion gerade ausgeführt wird. Die Variable «Texture» ist die Output Textur, auf welche der Compute-Shader schreibt. «numthreads» gibt die Anzahl von parallelen Prozessen innerhalb einer Prozessgruppe an.

HLSL unterstützt viele verschiedene Datentypen. Die wichtigsten davon sind bool, int, uint, float und double. Diese können aus bis zu vier Komponenten bestehen, zum Beispiel «float4». Buffer, Texturen, Vektoren und Matrizen werden auch unterstützt. Wie in C# können auch in HLSL eigene Structs definiert werden. HLSL-Shader haben keine Print-Funktion. Debugging wird daher meistens mithilfe der Output Textur gemacht.

3.1.3 IDE

Als IDE habe ich mich für JetBrains Rider entschieden, da ich mich mit diesem Code-Editor bereits gut auskenne und schon seit langer Zeit mehrere JetBrains Produkte verwende. Rider wurde für das .NET Framework entwickelt und unterstützt somit auch C#. Wie die meisten IDEs hat Rider Syntax-Highlighting, Codeanalyse, Debugging-Tools, Code-Vervollständigung und Refaktorierungen. Rider hat ausserdem eine umfangreiche Unity-Unterstützung, welche den Arbeitsablauf stark erleichtert.

3.2 Version-Control

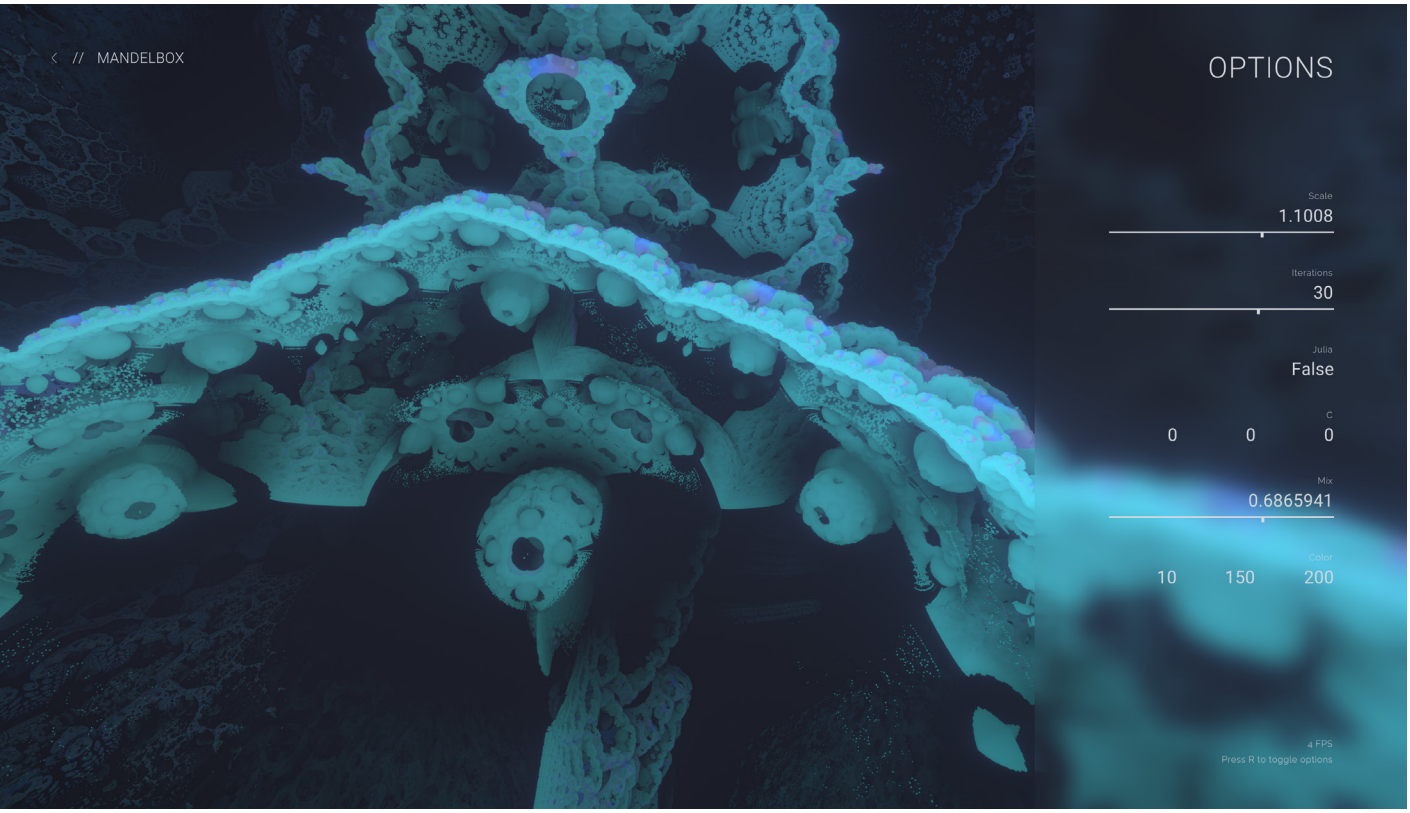
Bei grösseren Projekten ist es wichtig Version-Control zu verwenden. Die Versionsverwaltung hat den Vorteil, dass alle Änderungen protokolliert sind und jederzeit ältere Versionen des Projekts wiederhergestellt werden können. Ich habe mich für GitHub entschieden, welches auf der Versionsverwaltung Git basiert. Git ist derzeit die meistverwendete Versionsverwaltung und der Industriestandart. GitHub ist die bekannteste Open-Source-Software-Plattform. GitHub ist cloudbasiert und ermöglicht somit das Arbeiten auf mehreren Geräten von verschiedenen Orten. Kombiniert mit der Versionskontrolle Git ermöglicht dies einen effizienten und angenehmen Arbeitsablauf.

Wenn auf GitHub ein neues Projekt erstellt wird, dann wird eine sogenannte «Repository» erstellt. Repositories sind normalerweise öffentlich, das heisst open-source. Es können aber auch private Repositories erstellt werden. Neuer Code kann durch Commits hochgeladen werden. Der Code kann jederzeit zu einem früheren Commit zurückgesetzt werden. Es gibt auch die Möglichkeit mehrere Branches zu erstellen. Branches sind eine Art Abspaltung vom eigentlichen Code, welche separate Commits haben. Ist die Repository öffentlich, können andere Benutzer einen Pull Request erstellen. Somit können sie eigene Änderungen am Code vornehmen und diese mit der Erlaubnis des Besitzers mit dem originalen Code zusammenfügen.

3.3 Aufbau der Applikation

Die Applikation besteht aus verschiedenen Szenen, wobei eine Szene für das Hauptmenu und eine für den Loading-Screen sind. Die restlichen Szenen sind für die einzelnen Projekte, welche auch «Apps» genannt werden.

Die Szene eines Apps besteht jeweils aus einer Kamera und einem Canvas (GUI) und teilweise weiteren GameObjects. Das GUI zeigt dem User an, in welchem App er sich gerade befindet und ermöglicht das Zurückkehren in das Hauptmenu. Es zeigt ausserdem Informationen, wie zum Beispiel die aktuellen FPS an. Durch Drücken der Taste R werden die Optionen angezeigt. Hier können verschiedene Variablen der Apps (meistens Fraktal-Parameter), mithilfe intuitiver Bedienungsfelder, wie zum Beispiel Slider, Buttons und Textfelder, verändert werden.



Für jede App gibt es jeweils ein Skript, welches als Komponente an die Kamera der Szene angefügt ist, und einen Compute-Shader. Das Skript erbt eine abstrakte Basisklasse namens «App» und ist somit auch ein MonoBehaviour. Da die Basisklasse abstrakt ist, muss die abgeleitete Klasse die abstrakte Methode «Render» selbst definieren. Der Aufbau eines einfachen App-Skripts ist auf der nächsten Seite aufgezeigt.

```C#
using UnityEngine;

public class Example : App {

   private float variable;

   public void Start() {
      cameraType = CameraType.Orbit;
      // ... Code, welcher zum Programmstart ausgeführt wird
   }

   protected override void Render(RenderTexture s) {

      // shader
      shader.SetTexture(0, "Texture", tex);
      shader.SetTexture (0, "Source", s);
      shader.SetFloat("Variable", variable);

      shader.Dispatch(0, Mathf.CeilToInt(w / 8f),
                         Mathf.CeilToInt(h / 8f), 1);
   }

   public new void Update() {
      base.Update();
      // ... Code, welcher pro Frame ausgeführt wird
   }

   public float O_Variable {
      get => variable;
      set => variable = value;
   }
}
```

Wie jede MonoBehaviour-Klasse besitzt auch das App-Skript eine Start- und Updatemethode. Damit die Updatemethode der Basisklasse nicht überschrieben wird, muss sie das Keyword «new» enthalten und die Basismethode aufrufen. In der Startmethode wird normalerweise die Art von Kamera (Frei, Orbit, keine) gewählt. Die Kamerasteuerung wird in der Basisklasse berechnet.

Das Skript kann beliebig viele Variablen enthalten. Hier wurde einfachheitshalber nur eine Variable gewählt. Sollte die Variable durch die Optionen kontrolliert werden können, müssen Getter und Setter erstellt werden, wie das hier mit «O_Variable» gemacht wurde.

Die Methode «render» ist dafür verantwortlich den Shader aufzusetzen und auszuführen. Die Shader-Variable kommt von der Basisklasse und beinhaltet jeweils den zugehörigen Compute-Shader der App. Die Rendertexturen «s» und «tex» stammen von der Basisklasse. «s» ist die Source, also die Quelle, und beinhaltet, was der Standard-Unity-Renderer gerendert hat. In den meisten Apps ist dies die Hintergrundfarbe. Die zweite Rendertextur «tex» ist die Output-Textur, auf welche der Shader schreibt, und die Basisklasse dann an Unity weitergibt. Eine weitere Aufgabe der Render-Methode ist es, die Variablen des Shaders zu initialisieren. Der Compute-Shader akzeptiert die meisten einfachen Datentypen und Objekte. Für komplexere Objekte können auch Buffer erstellt werden, welche zum Beispiel mit einem Array von Structs gefüllt werden können.

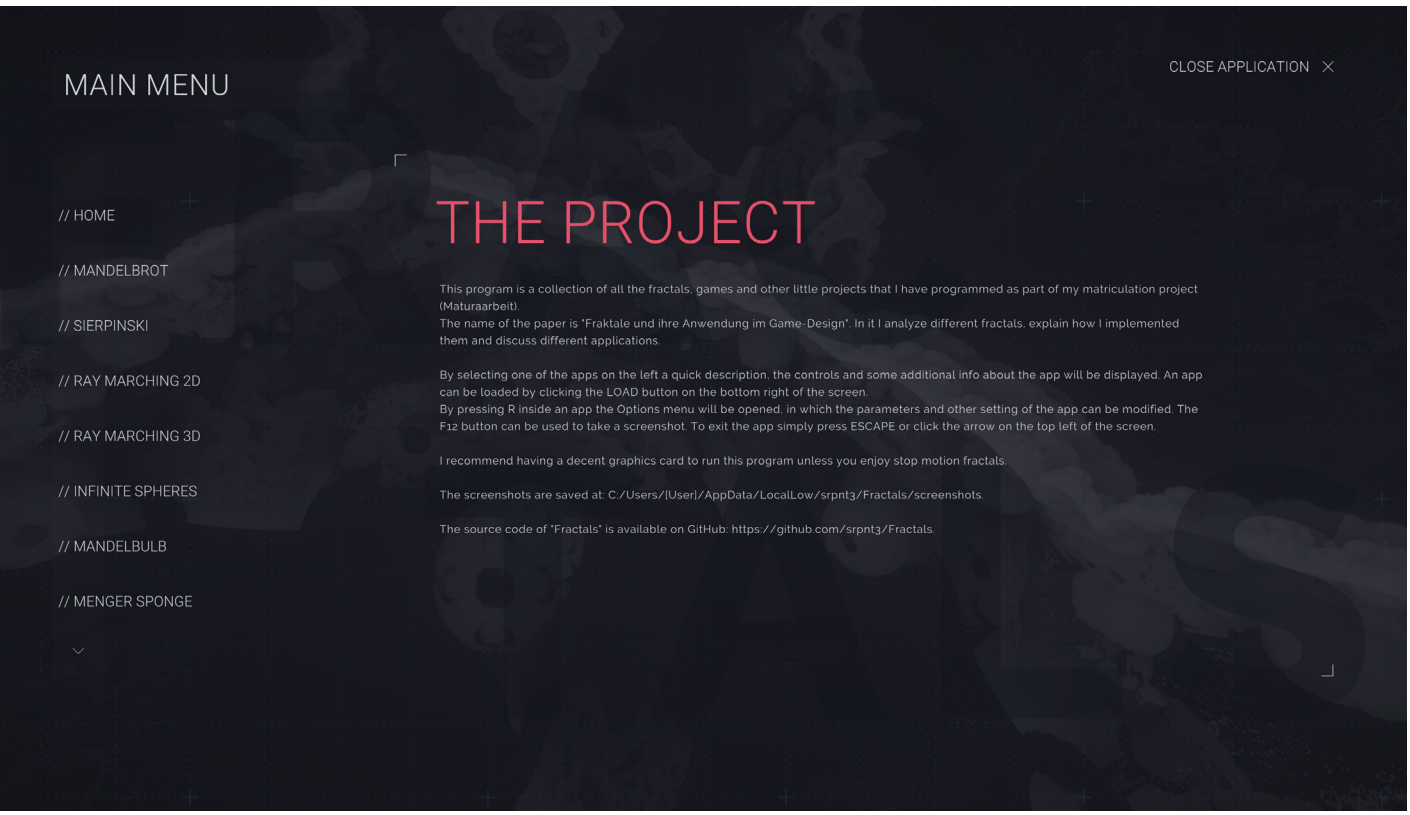
Ist der Shader fertig aufgesetzt, kann er mit der «dispatch» Funktion ausgeführt werden. Der erste Parameter der Funktion gibt den Kernel-Index an (Index oder Name des Kernels). Falls der Shader nur ein Kernel besitzt, ist dieser Index immer 0. Die drei weiteren Parameter geben die Anzahl Prozessgruppen an. Die Faktoren, durch welche die Breite «w» und die Höhe «h» der Rendertextur geteilt werden, sollten den «numthreads» des Compute-Shaders entsprechen, da die insgesamte Anzahl der Prozesse den Anzahl Pixel entsprechen sollte. Somit wird für jeden Pixel ein Prozess ausgeführt.

Die Basisklasse «App» ist verantwortlich für die Funktionen und Variablen, welche für alle Apps gleich sind. Sie erstellt und aktualisiert, falls nötig, die Rendertexturen. In ihr werden die Rendertexturen, Kameradaten, Auflösung und andere wichtige Daten gespeichert, damit sie von den abgeleiteten Klassen verwendet werden können.

Die «OnRenderImage» Methode ist in der Basisklasse enthalten und wird von Unity immer nach dem Rendern der Szene, durch den normalen Unity-Renderer, aufgerufen. Sie hat zwei Rendertexturen, «source» und «destination», als Parameter. Die Source-Textur beinhaltet, was der Standard-Unity-Renderer gerendert hat und die Destinations-Textur ist die Textur, welche später von Unity auf dem Bildschirm angezeigt wird. In dieser Methode kopiert die Basisklasse den Inhalt der Rendertextur «tex», welche das Resultat des Compute-Shaders beinhaltet, in die Destinations-Textur. Falls nötig wird von dieser Methode aus die abstrakte «render» Methode aufgerufen, mit «source» als Parameter, um das Bild neu zu berechnen. Die abstrakte «render» Methode ist wie bereits erwähnt von den einzelnen Apps definiert.

Die Basisklasse registriert und berechnet alle Controls (inklusive Kamerasteuerung) mithilfe der Klasse «ControlsHelper». Sie ist auch die Schnittstelle zwischen den Optionen im GUI und den Variablen innerhalb einer App. Des Weiteren beinhaltet sie verschiedene nützliche Methoden, wie zum Beispiel die Screenshot-Methode, welche ein Bildschirmfoto erstellt und speichert. Die Basisklasse «App» ermöglicht somit eine Entlastung der appspezifischen Klassen, damit in ihnen nur der Code, welcher für diese App wichtig ist, enthalten ist.

Mithilfe des «AppManager» Skripts wird das Hauptmenu erstellt und die einzelnen Apps verlinkt. Das Laden einer App wird von dem «SceneLoader» Skript kontrolliert.



3.4 2D-Fraktale

In diesem Kapitel wird erklärt, wie die 2-dimensionalen Fraktale, welche in der Applikation enthalten sind, programmiert sind. Das Mandelbrot-Fraktal und die zwei Sierpinski-Fraktale wurden mithilfe von Pixel-Iteration programmiert. Es ist auch möglich diese mit Distance-Functions zu rendern, welche im Kapitel 3.5.1 noch genauer erklärt werden.

3.4.1 Mandelbrot-Menge

Für das Mandelbrot-Fraktal werden zuerst alle Pixelkoordinaten in komplexe Zahlen der Gaussschen Zahlenebene umgerechnet. Bei dieser Umrechnung spielen auch der Zoom und die Verschiebung des Fraktals, die vom User kontrolliert werden, eine Rolle. Für jeden dieser Pixel bzw. für jede dieser komplexen Zahl wird dann die auf der nächsten Seite folgende Funktion angewendet.

```HLSL
int Iterate(double2 c) {
   double2 z = Julia ? c : double2(0, 0);

   for (int i = 0; i < Iterations; i++) {
      z = Multiply(z, z) + (Julia ? C : c); // z = z^2 + c
      if (length(z) > 2) { // bailout
         return i;
      }
   }
   return Iterations;
}
```

Dies ist die Mandelbrot-Funktion in Code umgesetzt. Der Parameter c ist der Punkt auf der komplexen Zahlenebene. Um möglichst genaue Resultate zu erhalten, wurden für alle komplexen Zahlen double anstatt float verwendet. Wenn eine Julia-Menge berechnet werden soll, dann ist z der Punkt auf der komplexen Zahlenebene, also c. Ansonsten ist z der Nullpunkt. Die variable «Iterations» gibt an, wie viel Mal die for-Schleife maximal wiederholt werden soll. In dieser Schleife wird z zuerst mit sich selbst multipliziert (also quadriert) und dann wird c addiert. Wird eine Julia-Menge berechnet wird die Konstante C, welche vom User bestimmt wird, anstatt c addiert. Da die Addition von komplexen Zahlen gleich ist, wie die Addition von Vektoren, kann hier einfach der Plus Operator verwendet werden. Für die Multiplikation hingegen muss die Funktion «Multiply» verwendet werden:

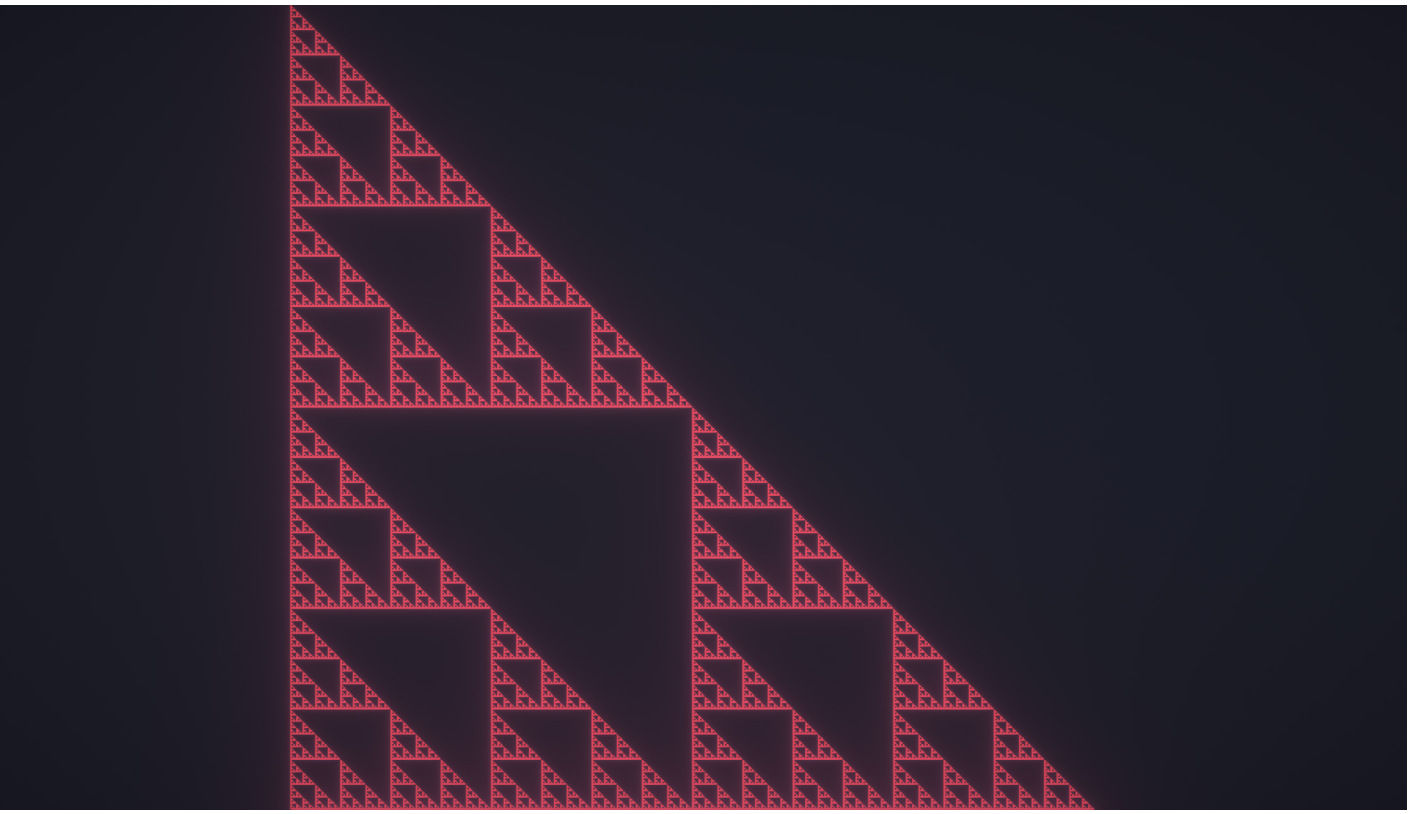
```HLSL
double2 Multiply(double2 a, double2 b) {
   double2 x;
   x[0] = (a[0] * b[0]) - (a[1] * b[1]);
   x[1] = (a[0] * b[1]) + (a[1] * b[0]);
   return x;
}
```

[0] ist hier der Realteil und [1] der Imaginärteil.

Ist der Betrag von z (length) grösser als 2, divergiert der Punkt und ist kein Teil der Mandelbrot-Menge. Die Iterations-Funktion gibt die Iterationen zurück, anstatt einen boolean welcher angibt ob der Punkt divergiert oder nicht. Entspricht der Return-Wert den maximalen Iterationen, ist der Punkt ein Teil der Mandelbrot-Menge und kann dementsprechend eingefärbt werden. Ist der Return-Wert aber eine andere Zahl (zwischen 0 und den maximalen Iterationen), dann kann dieser Pixel zum Beispiel mit einem Farbverlauf eingefärbt werden. Damit sich der Farbverlauf übergangslos wiederholt wurde der Cosinus verwendet, mit den Iterationen als Argument. Der Funktionswert des Cosinus wird dann einer Farbe des Farbverlaufs zugewiesen. Um schönere Farbverläufe zu erstellen wurden die Farbverläufe im LCH-Farbraum berechnet, dazu mehr im Kapitel 3.7.1.

3.4.2 Sierpinski-Fraktale

Durch Untersuchen des Algorithmus habe ich festgestellt, dass der Sierpinski-Teppich und das Sierpinski-Dreieck mit dem gleichen Algorithmus gerendert werden können.



Da beide Fraktale quadratisch sind (bzw. in ein Quadrat passen), die Bildschirmauflösung und das Seitenverhältnis aber variabel sind, müssen sie zentriert werden und unabhängig von der Auflösung bzw. dem Seitenverhältnis gemacht werden. Das Quadrat, in dem sich die Fraktale befinden werden, hat die Seitenlänge der kleineren Seite des Bildschirms und ist zentriert. Zuerst werden die Koordinaten so verschoben, dass der Punkt (0, 0) mit dem unteren linken Eckpunkt des Quadrats, indem sich die Fraktale befinden sollten, übereinstimmt. Also um die Hälfte der Differenz der Seitenlängen des Bildschirms entlang der längeren Seite verschoben. Danach werden die Koordinaten durch die Seitenlänge des Quadrats dividiert, damit die Koordinaten, die sich im Quadrat befinden, zwischen 0 und 1 sind. In Code sieht dies folgendermassen aus:

```hlsl
    float ox = (w - min(w, h)) / 2; // calculate the x offset
    float oy = (h - min(w, h)) / 2; // calculate the y offset
    float s = 1 / min(w, h); // calculate scale
    float x = (id.x - ox) * s,
          y = (id.y - oy) * s; // center & normalize
```

Die Breite w und die Höhe h sind die Dimensionen des Bildschirms. Die aktuellen Koordinaten sind id.x und id.y und die neuen Koordinaten sind x und y.

Damit die Funktion, welche das Fraktal berechnet, keine negativen oder zu grossen Werte bekommt, wird sie nur für Pixel für x und y, welche zwischen 0 und 1 sind, ausgeführt:

```HLSL
if (min(x, y) > 0 && max(x, y) < 1) { // check boundaries
   if (InFractal(x, y)) {
      Texture[id.xy] = float4(0.8, 0.2, 0.2, 1);
   }
}
```

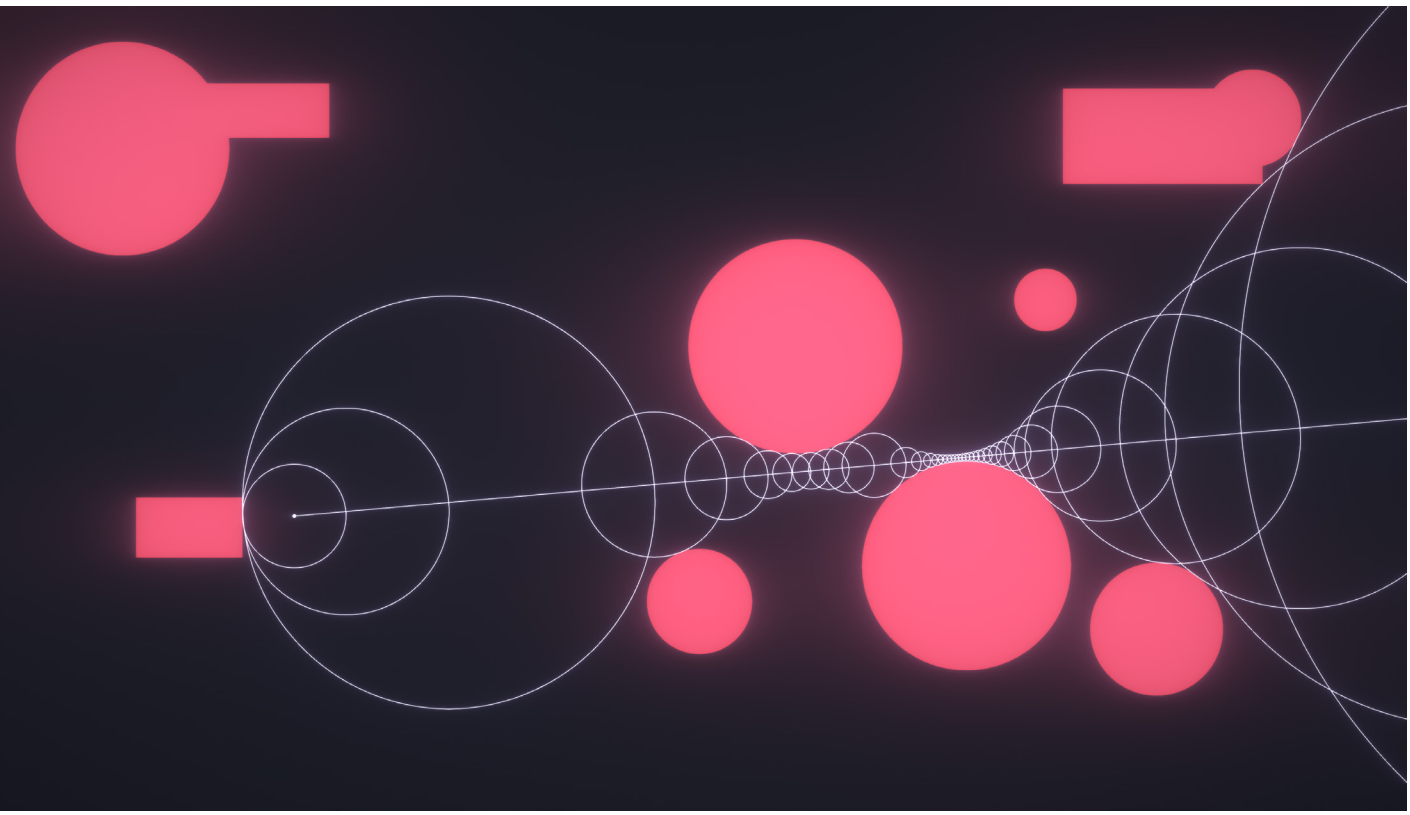
Somit bekommt die «InFractal» Funktion Koordinaten zwischen 0 und 1, welche im Bildschirm zentriert sind. Die Funktion gibt entweder true oder false zurück, je nachdem ob sich der Pixel, welcher dann dementsprechend eingefärbt wird, im Fraktal befindet. Die Funktion ist folgendermassen aufgebaut:

```HLSL
bool InFractal(float x, float y) {
   for (int i; i < Iterations; i++) {
      x *= N; y *= N; // scale to next level
      if (abs(x % N - pos) < size && abs(y % N - pos) < size) // check
         return false;
   }
   return true;
}
```

Wie beim Mandelbrot-Fraktal ist «Iterations» die maximale Anzahl an Iterationen. N bestimmt, welches Fraktal dargestellt werden soll. Für das Sierpinski-Dreieck ist N gleich 2 und für den Sierpinski-Teppich gleich 3. N bestimmt also in wie viele Teile das Fraktal in jedem Schritt geteilt werden soll. Das Fraktal wird durch Multiplikation in jedem Schritt in N Stücke «geteilt». Danach wird geprüft, ob sich der Punkt in dem 2. Quadrat von diagonal unten links befindet. Beim Sierpinski-Dreieck ergibt dies das Quadrat oben rechts und beim Sierpinski-Teppich das mittlere Quadrat. Ist dies der Fall, wird false zurückgegeben, da der Pixel kein Teil des Fraktals ist. Die Position des entfernten Quadrates wird durch «pos» bestimmt. Diese Position ist zwischen 0 (unten links) und N (oben rechts) und ist entlang einer Diagonale, da für x und y die gleiche Positionsvariable verwendet wird. Für die zwei Sierpinski-Fraktale ist die Position 1.5. Die Grösse wird durch «size» bestimmt und liegt zwischen 0 (nichts) und N/2 (alles). Für die zwei Sierpinski-Fraktale ist die Grösse 0.5.

Durch Verändern von der Position und der Grösse können ganz verschiedene Fraktale erstellt werden. Durch kleine Anpassungen am Algorithmus entstehen sogar noch mehr Möglichkeiten.

3.5 Ray-Marching



Um 3-dimensionale Fraktale zu rendern wird oft die Technik Ray-Marching verwendet. Daher wird sie in diesem Kapitel genau erklärt. Ausserdem ist eine Demonstration von Ray-Marching ist in der Applikation als «Ray Marching 2D» enthalten. Alle 3-dimensionalen Fraktale und Szenen der Applikation wurden mit Ray-Marching gerendert.

Ray-Marching, ähnlich wie Ray-Tracing, ist eine viel realistischere Rendering-Technik als Rasterung. Rasterung wird von den meisten real-time Renderern verwendet, da es schnell und einfach zu berechnen ist, dafür aber weniger realistisch. Es ist auch möglich real-time Renderer mit Ray-Marching zu programmieren, es braucht aber eine moderne und leistungsfähige Grafikkarte.

Ray-Marching simuliert die Lichtstrahlen (Rays), welche die Kamera treffen und ist somit schon viel näher an der Realität. Die Lichtstrahlen werden jedoch nicht von der Lichtquelle simuliert, sondern von der Kamera, da sonst auch Lichtstrahlen berechnet werden, welche die Kamera nicht treffen. Ein Ray hat jeweils eine Startposition (Positionsvektor) und eine Richtung (Richtungsvektor):

```hlsl
struct Ray {
    float3 position;
    float3 direction;
};
```

Für jeden Pixel wird dann ein Ray berechnet:

```hlsl
// convert into range [-1, 1]
float2 uv = id.xy / float2(w, h) * 2 - 1;

// create and march ray
Ray ray = CreateCameraRay(uv);
float res = March(ray);
```

Die Breite w und die Höhe h sind die Dimensionen des Bildschirms. Die Funktion «CreateCameraRay» berechnet die Richtung und Position des Rays mithilfe von Matrizen, welche von Unity bereitgestellt sind. Dieser Ray wird dann schrittweise marschiert, bis er auf ein Objekt trifft oder die maximale Anzahl von Schritten erreicht wird:

```hlsl
float March(Ray ray) {
    float d;
    float3 eye = ray.origin;

    int s = 0;
    while (s < steps) {
        d = DE(ray.origin); // calculate distance
        if (length(eye - ray.origin) > 100) s = steps; // to far away
        if (d < epsilon) break; // hit
        ray.origin += ray.direction * d; // march
        s++; // next iteration
    }

    return s / float(steps);
}
```

Die maximale Anzahl von Schritten ist durch «steps» bestimmt. Die Distanz, die ein Ray in jedem Schritt marschiert, ist d. Diese Distanz wird immer am Ende der Schleife mit der Richtung multipliziert und zur Ray-Position addiert, um vorwärts zu marschieren. Die Distanz ist aber nicht willkürlich gewählt, sondern die grösstmögliche Distanz, die der Ray marschieren kann, ohne ein Objekt zu überschiessen, also der Abstand zum nächsten Objekt. Sie wird von Distance-Functions berechnet. Somit überschreitet ein Ray niemals ein Objekt und falls sich ein Ray einem Objekt nähert, wird diese Distanz exponentiell kleiner. Ist sie kleiner als ein Richtwert Epsilon, dann hat der Ray ein Objekt getroffen und der Pixel wird dementsprechend gefärbt. Die Genauigkeit des Bildes verbessert sich mit kleineren Epsilons. Falls der Ray aber kein Objekt trifft, hört die Schleife auf, sobald der Ray zu weit (hier 100 Einheiten) entfernt ist. Zurückgegeben werden die Anzahl Schritte, da diese proportional zu der Menge von Licht sind, das an diesen Ort gelangen würde. Somit kann realistische Verschattung berechnet werden. Dies wird auch Ambient Occlusion oder Umgebungsverdeckung genannt.

3.5.1 Distance-Functions

Für jedes einfache 2- oder 3-dimensionale Objekt gibt es eine sogenannte Distance-Function (oder auch Distance-Estimator genannt). Der mathematische Begriff für eine Distance-Function wäre eine Metrik. Sie berechnet für einen beliebigen Punkt den Abstand zu diesem Objekt und gibt diesen zurück (falls sich der Punkt im inneren des Objekts befindet, ist der Abstand negativ). Eine einfache Distance-Function ist zum Beispiel die einer Kugel:

```hlsl
// sphere distance estimator
float DESphere(float3 p, float r) {
    return length(p) - r;
}
```
HLSL

Eine Distance-Function hat normalerweise ein Funktionsargument p für den Punkt, für den der Abstand berechnet werden soll und andere Argumente, welche die Eigenschaften des Objekts kontrollieren. In diesem Fall wäre das r, welches den Radius bestimmt. Der Abstand ist die Distanz von p zum Nullpunkt minus den Radius. Die Position der Kugel wird von p subtrahiert, bevor die Methode aufgerufen wird. So verschiebt sich die Kugel. Eine etwas komplexere Distance-Function ist zum Beispiel die eines Würfels (vgl. Quilez 2020a):

```hlsl
// box distance estimator
float DEBox(float3 p, float3 s) {
    float3 q = abs(p) - s;
    return length(max(q, 0.0)) + min(max(q.x, max(q.y, q.z)), 0.0);
}
```
HLSL

Die Funktion «DE», welche die zu marschierende Distanz des Ray berechnet, ist eine Distance-Function der ganzen Szene. Da die meisten Szenen aber aus mehreren Objekten bestehen, müssen verschiedene Objekte kombiniert werden. Werden zwei Objekte vereint, ist der Abstand zu beiden Objekten gleich dem Abstand zum näheren Objekt, also die kleinere. Um die Schnittmenge zu bilden, wird anstatt des kleineren Abstands der grössere benutzt. Wird eine der beiden Abstände negiert und der grössere davon als Abstand benutzt, ergibt sich eine Subtraktion des einen Objekts vom anderen. Diese Operationen funktionieren auch für mehrere Objekte und können in beliebiger Reihenfolge angewendet werden. Distance-Functions einzelner Objekte können auch aus anderen Distance-Functions aufgebaut sein. Zum Beispiel würde eine Distance-Function eines Würfels mit einem Loch aus der Distance-Function eines Würfels und aus der Distance-Function eines Zylinders bestehen. Distance-Functions sind also «stapelbar».

Das Mandelbulb-Fraktal und die Mandelbox besitzen ihren eigenen Distance-Estimator. Diese werden im Kapitel 3.6.3 und 3.6.4 erläutert.

3.5.2 Folds

Eine Eigenschaft von Ray-Marching bzw. Distance-Functions ist, dass das Positionsargument beliebig verändert werden kann. Dadurch kann der Raum beliebig transformiert werden. Er kann gekrümmt, verzerrt, skaliert, verschoben, rotiert und sogar gespiegelt werden. Solche Transformationen werden auch Folds genannt, da sie den Raum teilweise ähnlich wie ein Papier falten. Bei Ray-Marching wird also nicht das Objekt, sondern der Raum verändert

Mithilfe von Subtraktion bzw. Addition kann der Raum verschoben werden. Skalierung erfolgt mit der Multiplikation bzw. Division. Wird der Betrag der Position verwendet, spiegelt sich der Raum in alle Achsen. Rotationen werden mit Transformations-Matrizen gemacht.

Es sind aber auch komplexere Folds möglich, wie zum Beispiel die Repetition des Raumes, um eine Illusion von unendlichen Objekten zu erschaffen (vgl. Quilez 2020a):

```hlsl
// repeat space
float3 FoldRepeatSpace(float3 p) {
    float a = 3;
    float x = abs(p.x - a / 2) % a - a / 2;
    float y = abs(p.y - a / 2) % a - a / 2;
    float z = abs(p.z - a / 2) % a - a / 2;
    return float3(x, y, z);
}
```

Die Position ist in diesem Beispiel p und a ist der Abstand, in dem der Raum wiederholt werden soll. Mithilfe des Modulo Operator (hier %) wird die Position immer nach 3 Einheiten «zurückgesetzt». Wie in diesem Beispiel zu erkennen ist, wird die Berechnung für jede Koordinate einzeln durchgeführt. Bei den meisten Folds besteht die Möglichkeit den Fold nur auf einzelne Achsen anzuwenden. Des Weiteren können Folds auch auf einzelne Objekte angewandt werden. Ein weiterer wichtiger Fold ist die Spiegelung an einer Ebene mit Normale n (vgl. Christensen 2011):

```hlsl
// reflect on plane
float3 FoldReflectPlane(float3 p, float3 n) {
    return p - 2.0 * min(0.0, dot(p, n)) * n;
}
```

Einfache und selbstähnliche Fraktale können bereits mit einfachen wiederholten Folds, wie zum Beispiel Verschiebung, Spiegelung und Skalierung erstellt werden, wie das auch bei dem Menger-Schwamm und der Octahedron-Flake gemacht wurde. Das Mandelbox-Fraktal wird mit komplexeren Folds, dem Sphere- und Box-Fold, erstellt. Das Mandelbulb-Fraktal wird ohne Folds gerendert und hat eine eigene Distance-Function.

3.6 3D-Fraktale

In diesem Kapitel wird erklärt, wie die 3-dimensionalen Fraktale, welche in der Applikation enthalten sind, programmiert sind. Die Fraktale wurden mithilfe von Ray-Marching gerendert. Ray-Marching ist typisch für Fraktale, es gibt aber noch andere Methoden 3-dimensionale Fraktale zu berechnen.

3.6.1 Menger-Schwamm

Der Menger Schwamm wird mithilfe von mehreren Folds, welche mehrmals iteriert werden, gerendert. Seine Distance-Function sieht wie folgt aus:

```hlsl
// menger sponge distance estimator
float DE(float3 p) {
    float d = DEBox(p, Size);

    // iterate
    for (int i = 0; i < Iterations; i++) {
        d = max(d, -DECross(p, Size) / pow(3, i)); // subtract cross
        p *= 3; // scale down 1/3
        p = abs(p); // mirror
        p -= float3(Size, Size, Size); // move
        p = abs(p); // mirror
        p -= float3(Size, Size, Size); // move
    }

    return d;
}
```

HLSL

Zuerst wird die Distanz zum Startobjekt, einem einfachen Würfel, berechnet. Die Variable «Size» kontrolliert die Grösse des Fraktals. «Iterations» sind die Anzahl Iterationen, welche die Schleife durchläuft. Bei jedem Durchlauf wird vom Würfel ein Kreuz subtrahiert. Das Kreuz entspricht einem umgekehrten Menger Schwamm erster Iteration. Somit entfernt das Kreuz die sieben mittleren Würfel und es verbleiben 20 restliche Würfel. Danach wird der Raum mithilfe von Spiegelung und Verschiebung in 27 gleiche Teile geteilt. In der nächsten Iteration wird wieder das Kreuz von den 20 neuen Würfeln subtrahiert. Um ein Überschiessen des Rays zu verhindern, muss d bei jeder Verkleinerung des Raumes korrigiert werden. Wird d in jedem Schritt durch 3^i geteilt, wird das Fraktal wieder korrekt dargestellt.

Um Abwechslung in der Darstellung zu erzeugen, wird bei diesem Fraktal der Ambient Occlusion Wert umgekehrt. Somit leuchten die Spalten, während flache Oberflächen dunkel dargestellt sind.

3.6.2 Octahedron-Flake

Wie der Menger Schwamm wird auch die Octahedron-Flake mithilfe von Folds gerendert. Hier sind vor allem die Spiegelungen an Ebenen wichtig. Die Distance-Function ist wie folgt aufgebaut:

```hlsl
// octahedron flake distance estimator
float DE(float3 p) {
   for (int i = 0; i < Iterations; i++) {
      p *= 2; // scale down 1/2
      p = FoldReflectPlane(p, normalize(float3(0, 1, 1)));  // mirror
      p = FoldReflectPlane(p, normalize(float3(0, 1, -1))); // mirror
      p = FoldReflectPlane(p, normalize(float3(1, 1, 0)));  // mirror
      p = FoldReflectPlane(p, normalize(float3(-1, 1, 0))); // mirror
      p -= float3(0, 1, 0) * Size; // move
   }
   return DEOctahedron(p, Size) / pow(2, i);
}
```
HLSL

Auch hier werden die Anzahl Iterationen von «Iterations» kontrolliert und die Grösse von «Size». In jeder Iteration werden der Raum bzw. das Oktaeder zuerst skaliert. Danach wird der Raum, mithilfe der Funktion «FoldReflectPlane», mehrmals an verschiedenen Ebenen gespiegelt, um sechs Kopien des ursprünglichen Oktaeders zu erhalten. Danach werden die Oktaeder noch korrekt verschoben. Nachdem der Raum korrekt gefaltet wurde, wird das eigentliche Oktaeder berechnet. Das ganze Fraktal besteht also nur aus einem Oktaeder, welcher mehrmals gespiegelt und verschoben wurde. Die Octahedron-Flake ist ein Musterbeispiel von Fraktalen, welche mit Folds berechnet werden.

3.6.3 Mandelbulb

Das Mandelbulb-Fraktal wird ohne Folds berechnet. Seine Distance-Function basiert auf der Formel (vgl. Christensen 2011):

$$dst = 0.5 * ln(r) * r/dr$$

Die Distanz dst zum Fraktal wird mithilfe von r und dr berechnet. r ist der Betrag der hyperkomplexen Zahl z und dr ist die iterierte Ableitung der Mandelbulb-Funktion (vgl. Christensen 2011). Mithilfe dieser Formel wäre auch ein Distance-Estimator für das Mandelbrot-Fraktal möglich.

Der Distance-Estimator des Mandelbulb-Fraktals sieht wie folgt aus (vgl. Christensen 2011):

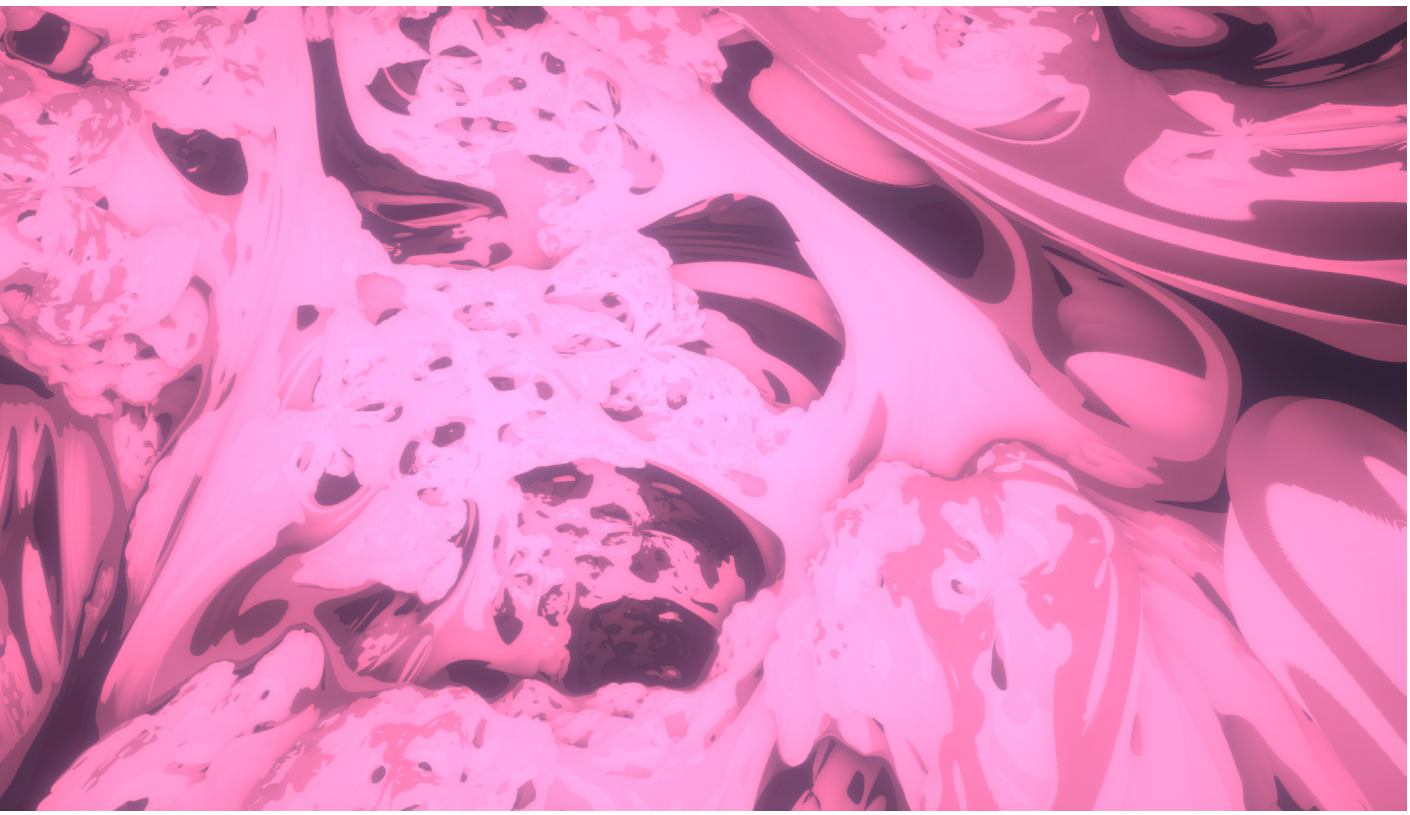
```hlsl
// mandelbulb distance estimator
float2 DE(float3 p) {
   float3 z = p;
   float r, dr = 1;

   for(int i = 0; i < Iterations; i++) {
      r = length(z);
      if (r > 3) break; // bailout

      // convert to polar coordinates
      float theta = acos(z.z / r);
      float phi = atan2(z.y, z.x);
      dr = pow(r, Power - 1.0) * Power * dr + 1.0;

      // scale and rotate the point
      float zr = pow(r, Power);
      theta = theta * Power;
      phi = phi * Power;

      // convert back to cartesian coordinates
      z = zr * float3(sin(theta) * cos(phi),
                      sin(theta) * sin(phi), cos(theta));
      z += (Julia * C) + ((1 - Julia) * p);
   }

   float dst = 0.5 * log(r) * r / dr; // calculate distance
   return float2(dst, sin(i));
}
```

HLSL

Die hyperkomplexe Zahl z wird als float3 gespeichert. Die Variable «Iterations» bestimmt, wie viel Mal die Schleife durchlaufen werden soll. In jeder Iteration wird zuerst geprüft, ob der Betrag r von z grösser als 3 ist, denn dann würde der Punkt divergieren und ist kein Teil der Mandelbulb-Menge. Danach wird z in Polarkoordinaten umgerechnet und die iterierte Ableitung dr wird berechnet. Die hyperkomplexe Zahl wird, wie im Kapitel 2.2.7 beschrieben, mit «Power» potenziert. Die Variable Power wird vom Benutzer kontrolliert.

Schlussendlich wird z wieder in kartesische Koordinaten umgewandelt und c wird addiert. Normalerweise ist c die aktuelle Position p. Falls aber eine Julia-Menge berechnet wird, ist c die Konstante C, welche vom Benutzer definiert ist. Bei diesem Fraktal ist die Variable Julia kein Boolean, sondern ein Float, um einen kontinuierlichen Übergang zwischen dem Mandelbulb-Fraktal und seinen Julia-Mengen zu ermöglichen. Dieser Effekt kann auch beim Mandelbrot-Fraktal und bei der Mandelbox verwendet werden.

Nachdem die Schleife durchlaufen oder abgebrochen wurde, wird die Distanz dst mit der oben erwähnten Formel berechnet. Sie wird zusammen mit dem Sinus der Iterationen zurückgegeben. Der Sinus der Iterationen dient der Färbung des Fraktals.

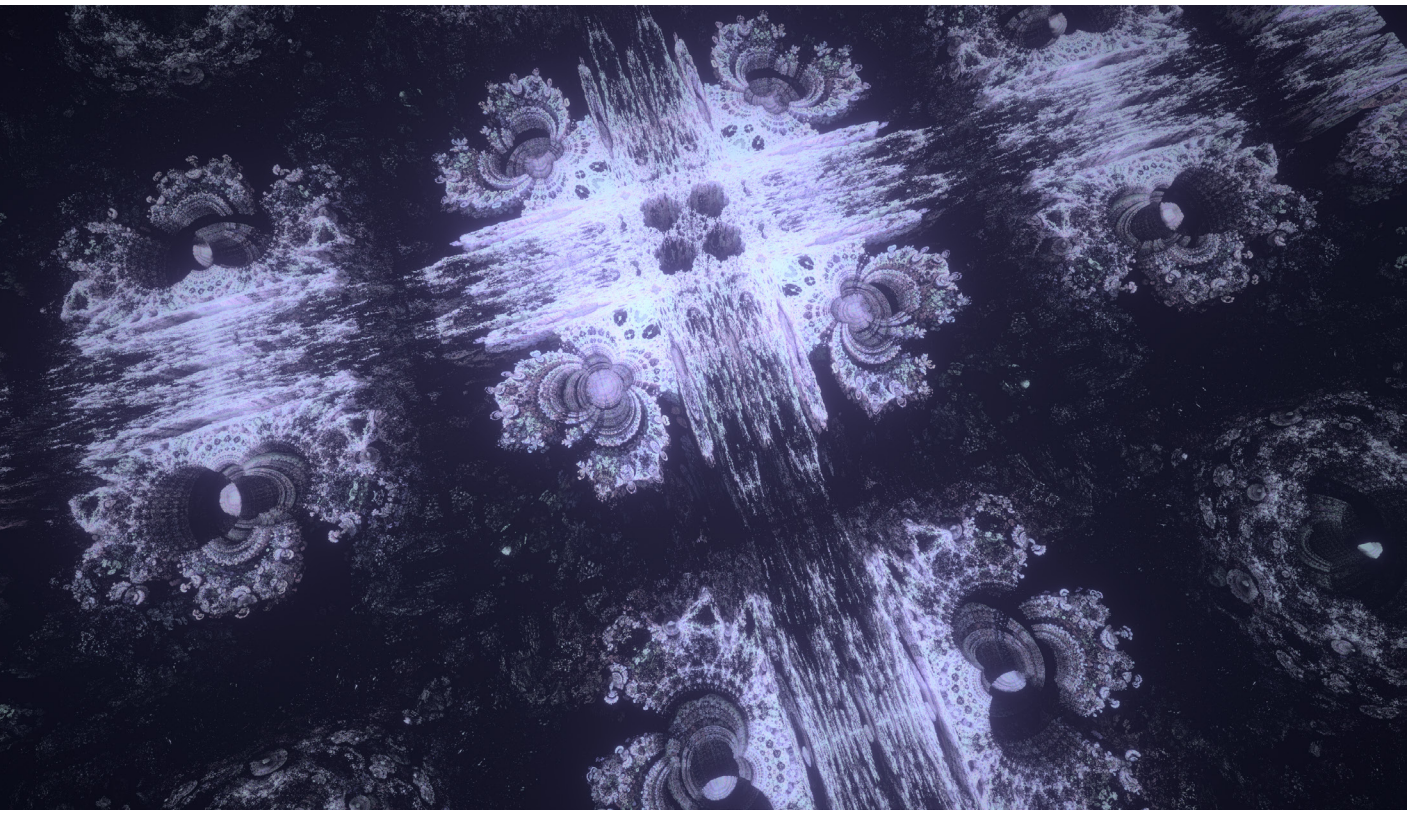


3.6.4 Mandelbox

Die Distance-Function des Mandelbox-Fraktals basiert, wie die des Mandelbulb-Fraktals, auf einer iterierten Ableitung dr (vgl. Christensen 2011). Die Formel für die Distanz und die Berechnung von dr unterscheidet sich jedoch. Die Formel für die Distanz dst lautet wie folgt (vgl. Christensen 2011):

$$dst = r/|dr|$$

r ist die Länge des Punktes z, welcher, wie im Kapitel 2.2.8 erklärt, mithilfe von Box- und Sphere-Folds berechnet wird.



Die ganze Distance-Function ist wie folgt aufgebaut (vgl. Christensen 2011):

```HLSL
// mandelbox distance estimator
Data DE(float3 p) {
   float3 z = p, trap = p;
   float dr = 1.0;

   for (int i = 0; i < Iterations; i++) {
      BoxFold(z); // box fold
      SphereFold(z, dr); // sphere fold

      z *= Scale; // scale
      z += Julia ? C : p; // c
      dr = dr * abs(Scale) + 1.0; // dr

      Trap(z, trap); // trap
   }

   return DataConstr(length(z) / abs(dr), trap);
}
```

Die Anzahl Iterationen wird durch «Iterations» bestimmt. In jedem Durchlauf der Schleife werden zuerst ein Box-Fold und ein Sphere-Fold für z bzw. dr berechnet.

Der Box-Fold ist wie folgt definiert (vgl. Christensen 2011):

```HLSL
// box fold
void BoxFold(inout float3 z) {
   float foldingLimit = 1;
   z = clamp(z, -foldingLimit, foldingLimit) * 2.0 - z;
}
```

Aufgrund des inout Keywords muss z nicht zurückgegeben werden und kann direkt verändert werden. Die Variable «foldingLimit» kann beliebig verändert werden, was für noch mehr Variabilität des Fraktals sorgt. Der Sphere-Fold ist etwas komplexer und wie folgt definiert (vgl. Christensen 2011)::

```HLSL
// sphere fold
void SphereFold(inout float3 z, inout float dr) {
   float fixedRadius = 1,
         minRadius = 0.5;
   float fixedRadius2 = pow(fixedRadius, 2),
         minRadius2 = pow(minRadius, 2);
   float r2 = dot(z, z);

   if (fixedRadius < minRadius2) {
      float temp = (fixedRadius2 / minRadius2);
      z *= temp, dr *= temp;
   } else if (r2 < fixedRadius2) {
      float temp = (fixedRadius2 / r2);
      z *= temp, dr *= temp;
   }
}
```

Auch hier sind die inout Keywords wichtig, da das Zurückgeben zweier Variablen umständlicher ist. Auch hier können die Variablen «fixedRadius» und «minRadius» beliebig verändert werden. Somit ist die Mandelbox überraschend vielfältig.

Nachdem der Box- und Sphere-Fold berechnet wurde, wird der Punkt z um den Faktor «Scale» skaliert. Scale ist der wichtigste und interessanteste Parameter des Mandelbox-Fraktals. Wird eine Julia-Menge berechnet wird die Konstante C, welche vom User bestimmt wird, addiert, ansonsten wird die aktuelle Position addiert.

Nach der Berechnung von dr wird der Punkt mit der Funktion «trap» gefangen (vom englischen «to trap», «einfangen»). Sogenannte Orbit-Traps sind eine weitere Art Fraktale zu färben. Da beim Mandelbox-Fraktal die Anzahl Iterationen für jeden Punkt gleich ist, können die Iterationen nicht zum Färben des Fraktals verwendet werden. Ambient Occlusion allein ist nicht sehr interessant und darum werden oft Orbit-Traps verwendet. Sie berechnen wie nahe ein Punkt einem Objekt jemals kam. Bei diesem Beispiel wurde als Objekt der Ursprung gewählt. Die berechnete nächste Position wird später als Farbe verwendet. Die Trap-Funktion sieht wie folgt aus:

```HLSL
void Trap(float3 z, inout float3 trap) {
   if (length(z) < length(trap))
      trap = z;
}
```

Für das Einfärben wird später der normalisierte Betrag der Position verwendet, um negative oder zu grosse Werte zu vermeiden. Es ist auch möglich Farbverläufe zu verwenden.

Nachdem die Schleife durchlaufen ist, wird die Distanz berechnet und mit der Orbit-Trap in einem Struct zurückgegeben.

3.7 Weiteres

Im Verlaufe des Programmierens der Applikation kam ich vor viele spannende Herausforderungen und experimentierte viel. Daher entstanden viele interessante Skripte und Methoden, wovon einige dieser in den folgenden Kapiteln genauer beschrieben sind.

3.7.1 LCH-Farbverläufe

Um möglichst ästhetische Fraktale zu erzeugen, habe ich als Teil des Programms eine LCH-Farbverlauf Library programmiert. Farbverläufe im LCH-Farbraum sind visuell einiges interessanter als in anderen Farbräumen.

Bei digitalen Anwendungen wird meistens der RGB (Rot, Grün, Blau) -Farbraum verwendet, da auch die Pixel auf Rot, Grün und Blau basieren. Weitere bekannte Farbräume sind zum Beispiel HSV (Farbton, Sättigung, Hellwert), HSL (Farbton, Sättigung, Luminanz) und CMYK (Cyan, Magenta, Gelb, Schwarz).

Der LCH (Helligkeit, Sättigung, Farbton) -Farbraum wird selten verwendet und ist daher auch weniger genau dokumentiert. Er ist ein zylindrischer Farbraum. Es gibt zwei Arten von LCH-Farbräumen, welche sich in der Berechnung unterscheiden; LCH(ab) und LCH(uv). Für dieses Projekt wurde LCH(ab) verwendet.

Die Umwandlung von RGB zu LCH(ab) erfolgt über mehrere andere Farbräume (Berechnungen für die Umwandlung von Lindbloom (2013), die Interpolation ist selbst hergeleitet). Zuerst wird vom RGB-Farbraum in den XYZ-Farbraum umgerechnet. Dann wird vom XYZ-Farbraum in den LAB-Farbraum umgerechnet. Und zum Schluss vom LAB-Farbraum in den LCH-Farbraum. Diese Umrechnung funktioniert auch in die andere Richtung, falls von LCH zu RGB umgewandelt werden soll.

Um einen LCH-Farbverlauf zu berechnen, werden zuerst die vorgegebenen RGB-Werte in LCH umgerechnet und dann interpoliert (angeglichen). Die Interpolation ist ähnlich wie im RGB-Farbraum, mit dem Unterschied, dass H ein Winkel zwischen 0 und 360 Grad aufweist und somit anders interpoliert werden muss. Dadurch entsteht für H ein «längerer» und ein «kürzerer» Weg.

Interpoliert wird indem eine Anzahl von Schritten (Auflösung des Verlaufs) definiert wird. Für jeden dieser Schritte wird dann die entsprechende Zwischenfarbe berechnet. Für die Berechnung der Zwischenfarbe muss zuerst für jede Komponente, in diesem Fall L, C und H, ein Delta berechnet werden. Delta bestimmt um wieviel sich die Farbe in jedem Schritt verändert. Für L und C kann Delta wie folgt berechnet werden:

```hlsl
float deltaL = (b.L - a.L) / (s + 1);
float deltaC = (b.C - a.C) / (s + 1);
```

Die Anfangsfarbe ist a, die Endfarbe b und s die Anzahl Schritte (exklusive Anfang und Ende). Die Berechnung von Delta H ist wie bereits erwähnt etwas komplexer und wird daher nicht erläutert.

Nachdem Delta für jede Komponente ausgerechnet wurde, wird in einer Schleife für jeden Schritt die Farbe ausgerechnet. Diese Farbe ist definiert als der Schritt mal Delta plus die Anfangsfarbe. Um zu kleine bzw. zu grosse H zu vermeiden, muss H angepasst werden ($0 \leq H < 360$). Die eigentliche Interpolation ist also wie folgt aufgebaut:

```hlsl
for (int i = 0; i < s; i++) {
   result[i] = new Colors.LCHColor(
      a.L + deltaL * (i + 1f),
      a.C + deltaC * (i + 1f),
      a.H + deltaH * (i + 1f)
   );

   // adjust H
   if (result[i].H ≥ 360f) result[i].H -= 360f;
   if (result[i].H < 0) result[i].H += 360f;
}
```

Es ist auch möglich mehr Farben als nur die Anfangs- und Endfarbe zu definieren, indem die Interpolation jeweils zwischen den einzelnen definierten Farben berechnet wird und die Resultate zusammengefügt werden.

3.7.2 Z-Buffer

Im Z-Buffer (Depth-Buffer, Tiefenbuffer) werden die Tiefeninformationen einer Szene für jeden Pixel gespeichert. Der Unity-Renderer hat einen eingebauten Depth-Buffer, welcher gelesen werden kann. Der Ray-Marching Compute-Shader verfügt auch über Tiefeninformationen, indem die Startposition eines Rays von der Endposition subtrahiert wird.

Der Depth-Buffer ist vor allem nützlich, wenn die von Unity gerenderte Szene mit der des Ray-Marching-Shaders kombiniert werden soll. Da beide Renderer mit den gleichen Einheiten rechnen sind die Tiefeninformationen vergleichbar. Da der Unity Depth-Buffer normalisiert (zwischen 0 und 1) vorliegt, muss er zuerst entschlüsselt werden. Um die Szenen zu kombinieren müssen die Tiefenwerte für jeden Pixel verglichen werden. Der Tiefere der beiden Werte ist das nähere Objekt und somit wird die Pixelfarbe vom näheren Objekt bestimmt. In Code ist das wie folgt umgesetzt:

```hlsl
1    if (DecodeDepth(Source[id.xy].w) > z)
2       Texture[id.xy] = color;
3    else
4       Texture[id.xy] = Source[id.xy];
```

Source ist die Rendertextur, welche Unity schon gerendert hat und die w-Komponente ist der Depth-Buffer dieser Textur, die Funktion DecodeDepth entschlüsselt diesen Depth-Buffer. Dann wird er mit der Tiefe der Ray-Marching-Szene verglichen und der Pixel wird dementsprechend eingefärbt. Somit wurden die beiden Szenen kombiniert. Diese Technik wird auch von dem Projekt «F.R.A.X.» verwendet. Dieses Projekt ist ein Anwendungsbeispiel von Fraktalen im Game-Design Bereich, dazu später noch mehr.

Der Unity Depth-Buffer wird auch von dem Unity-Postprocessing verwendet. Dies stellt ein Problem dar, da somit nicht alle Effekte mit dem Ray-Marching-Shader funktionieren werden. Eines dieser Effekte ist die Tiefenschärfe, welche hauptsächlich auf dem Depth-Buffer basiert. Eine mögliche Lösung dieses Problems wäre die Implementierung eines eigenen Tiefenschärfe-Effekts. Durch Experimentieren hat sich herausgestellt, dass dies möglich ist, jedoch nicht wirklich realistisch aussieht. Eine andere Möglichkeit wäre das Überschreiben des Unity Depth-Buffers mit den zusammengefügten Depth-Buffers. Dies ist jedoch nicht einfach, falls es überhaupt möglich ist. Aus diesen Gründen ist zu diesem Zeitpunkt in der Applikation noch keine Tiefenschärfe vorhanden.

3.7.3 Lichtkrümmung

Ein exklusiver Vorteil von Ray-Marching ist die volle Kontrolle über Rays. Somit sind auch Effekte wie das Krümmen der Rays fast ohne Leistungsverlust möglich. Den Effekt habe ich durch Experimentieren entdeckt. Es entsteht die Illusion von gekrümmtem Licht bzw. gekrümmtem Raum. Dieser Effekt ist ganz einfach umzusetzen. Es wird zuerst ein Vektor festgelegt, der die Krümmung kontrolliert. In jeder Iteration der Marschier-Funktion wird die Richtung des Rays um diesen Vektor verändert. Rays, welche näher an Objekten liegen, haben in der Regel mehr Iterationen und würden somit stärker gekrümmt werden. Um dies zu verhindern wird der Krümmungsvektor mit dem Abstand d, welcher der Ray marschiert, multipliziert. Dies ist zwar nicht perfekt, löst das Problem aber ziemlich gut. Dieser Effekt wird auch in «F.R.A.X.» verwendet, wobei der Krümmungsvektor noch zusätzlich animiert wird.

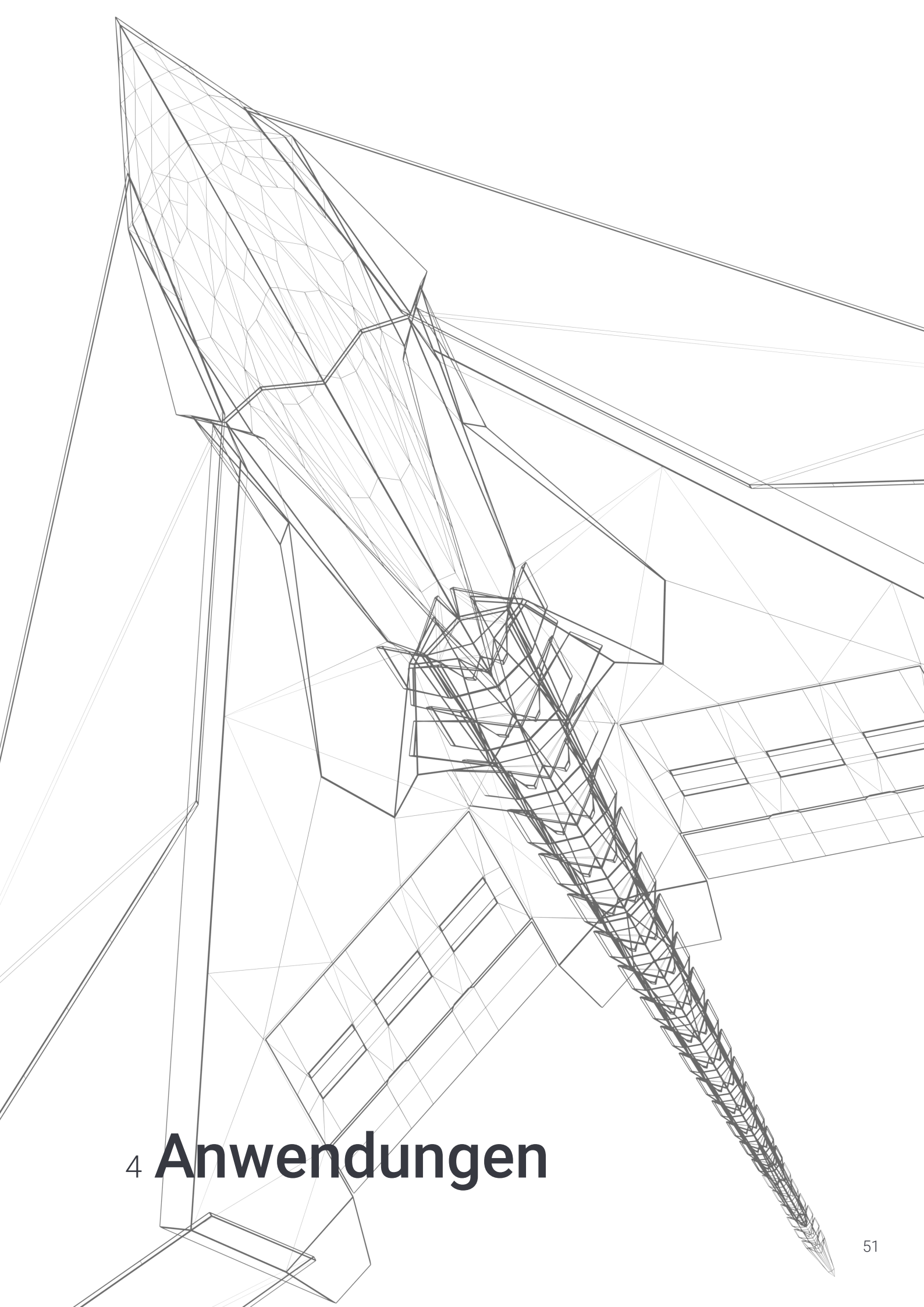
3.7.4 Flight-Controls

«F.R.A.X.» besteht darin mit einem Raumschiff durch animierte Fraktale zu fliegen. Es wurde also auch eine realistische Flugsteuerung (3rd-Person Kamera) implementiert. Zu einer solchen Steuerung gehören Roll, Pitch, Yaw und Geschwindigkeit. Werden nun zu jeden dieser vier Steuerungen zwei Tasten zugewiesen, ergibt sich das Problem, dass die Steuerung recht grob daherkommt. Dies liegt daran, dass Tasten meist kein analoger Input sind, sondern ein digitaler und das Raumschiff sich somit auch nur dreht oder nicht dreht.

Eine einfache Lösung für dieses Problem ist, dass die Tasten nicht die Geschwindigkeit der Bewegung, sondern die Beschleunigung kontrollieren. Dadurch wird bei einem Tastendruck das Raumschiff nicht direkt gedreht, sondern drehend beschleunigt. Um die Drehung wieder zu einem Stillstand zu bringen wurde ein «Luftwiderstand» bzw. eine «natürliche Entschleunigung» eingebaut, welche über Zeit die Geschwindigkeit verkleinert. Um noch eine sanftere Steuerung zu erreichen, bewegt sich das Raumschiff schneller als die Kamera.

3.7.5 Animation

Da Fraktale so viele verschiedene Parameter besitzen, welche bei kleinen Veränderungen zu ganz unterschiedlichen Resultaten führen, ist es üblich diese zu animieren. Oft sind diese Parameter begrenzt, das heisst ab einem gewissen Wert ist das Fraktal nicht mehr interessant oder gar nicht mehr sichtbar, für solche Parameter eignen sich Animationen basierend auf den trigonometrischen Funktionen Sinus oder Cosinus. Wird das Funktionsargument animiert (in jedem Frame vergrössert), ergibt sich eine harmonische Animation. Der umgerechnete Funktionswert kann dann als Parameter verwendet werden. Die Geschwindigkeit der Animation kann angepasst werden.

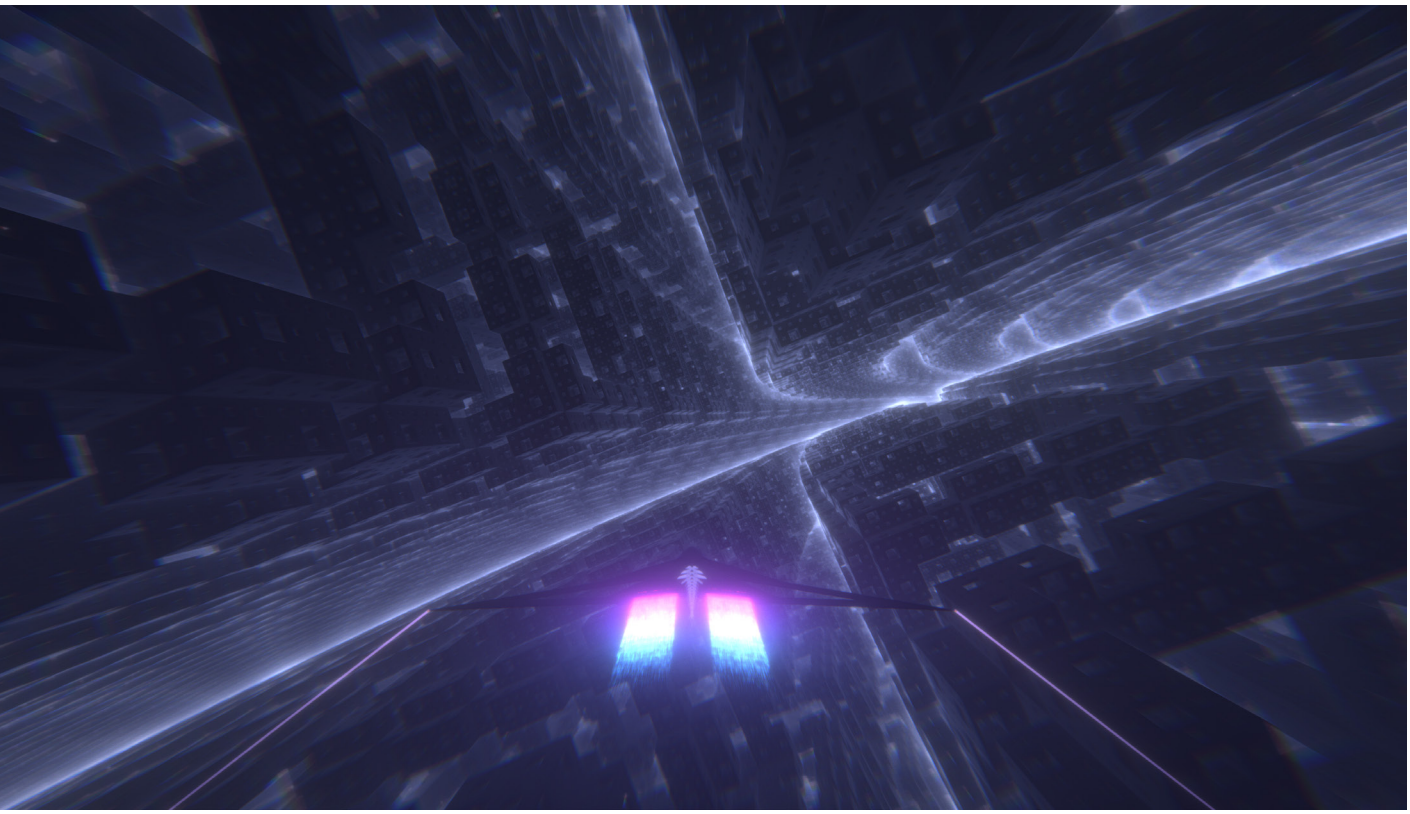


4 Anwendungen

4.1 Game-Design

Game-Design beschreibt das Konzipieren von Games. In dieser Arbeit wird der Begriff aber etwas breiter verwendet und umfasst somit auch das Entwickeln und das grafische Designen eines Games.

Um die Anwendungen im Bereich Game-Design zu untersuchen habe ich selbst ein Game konzipiert und programmiert. Dieses Anwendungsbeispiel von Fraktalen im Game-Design heisst «F.R.A.X.» und ist enthalten in der Applikation.



4.1.1 F.R.A.X.

Das Game besteht darin ein Raumschiff durch eine abstrakte Welt aus Fraktalen zu fliegen. Diese Welt besteht aus unendlichen Menger-Schwamm-Fraktalen, dessen Parameter mithilfe der Sinusfunktion (siehe Kapitel 3.7.5) animiert sind. Der eine Parameter, welcher animiert wird, ist die Anzahl der Iterationen. Auch die Grösse des Startwürfels des Menger Schwamms wird animiert, was dazu führt, dass das Fraktal «atmet». Zusätzlich zu den Fraktal Parameter wird auch der Krümmungsvektor des Lichts animiert (siehe Kapitel 3.7.3). Durch die Animation dieser Variablen entsteht eine sehr interessante und abstrakte Umwelt für das Game.

Das Raumschiff selbst ist sehr futuristisch designt und wird im 3rd-person Modus gesteuert. Die Controls wurden im Kapitel 3.7.4 genauer erläutert.

Nachdem das Konzept des Games erstellt war, musste im ersten Schritt das Raumschiff designt werden. Die ersten Skizzen wurden auf Papier erstellt. Aus diesen Skizzen wurde dann in der 3D-Modellierungssoftware «Blender» das eigentliche Modell erstellt. Dann musste das Modell nur noch in die Unity-Engine importiert werden. Dieser Prozess ist recht einfach, da Unity eine umfangreiche Unterstützung von Blender beinhaltet.

Der zweite Schritt war es, die Umwelt für das Game zu erstellen. Den Code für die Menger-Schwämme und die Wiederholung des Raumes hatte ich bereits. Ein Problem war jedoch die Genauigkeit der float Variablen, da sie bei grösseren Werten abnimmt (siehe Kapitel 4.5). Da die Position des Raumschiffes und somit auch der Kamera eine float Variable ist, wird sie mit grösserer Entfernung vom Ursprung ungenauer. Dies führt zu Artefakten im Bild. Eine Lösung ist es, nicht nur den Raum innerhalb des Distance-Estimators zu wiederholen, sondern auch bevor der Ray marschiert wird. Dies kann mit einem leicht modifizierten Repeat-Fold (siehe Kapitel 3.5.2) umgesetzt werden.

Im dritten Schritt wurde die Steuerung implementiert. Die ersten Versuche ergaben eine recht grobe Steuerung. Nach Experimentieren entstand eine sehr angenehme Steuerung, welche im Kapitel 3.7.4 genauer erläutert ist. Bei einem Game ist es auch wichtig verschiedene Peripherie-Geräte, unter anderem Ps4 und Xbox Controller, zu unterstützen. Dies geht dank der Unity-Engine recht einfach.

Als letztes wurden verschiedene Effekte implementiert, um das ganze Game abzurunden und einen gewissen Stil zu setzen. Dies sind einfache Postprocessing-Effekte von Unity, Triebwerke mithilfe von Partikelsystemen und weiteres.

«F.R.A.X.» hat zwar sehr vielen gefallen, es ist jedoch ein sehr spezifisches Anwendungsbeispiel der Fraktale im Game-Design. Viele Anwendungen für Fraktale in diesem Bereich zu finden ist schwierig und sie haben vor allem einen dekorativen oder ästhetischen Zweck.

Das grundlegende Konzept der Fraktale, die unendliche Rauigkeit, könnte jedoch auch auf eine abstrakte Art implementiert werden. Dies wäre eine nicht (nur) dekorative Art Fraktale im Game-Design anzuwenden.

Interaktive Games sind auch möglich, indem Kollisionen berechnet werden. Ein gutes Beispiel für ein solches Game ist «Marble Marcher» von «CodeParade» (vgl. github.com/HackerPoet/MarbleMarcher).

4.1.2 Ray-Marching / Ray-Tracing

Im Verlauf dieser Arbeit ist mir aufgefallen, dass neben den Fraktalen auch die Technik Fraktale zu rendern, Ray-Marching, ihre Anwendung im Game-Design hat. Ray-Marching ist gegenüber von konventionellen Render-Techniken (Rasterung) viel realistischer. Da der Realismus heutzutage in der Entwicklung von Games immer wichtiger wird, ist Ray-Marching gut für Games geeignet.

Ähnlich wie Ray-Marching ist Ray-Tracing. Ray-Tracing basiert auf der Berechnung von Überschneidungen des Rays mit den Objekten der Szene. Ray-Tracing wird bereits von vielen Game-Studios verwendet. Der bekannte Grafikkartenhersteller Nvidia hat im Jahr 2018 erstmals eine Grafikkartenserie herausgegeben deren Schwerpunkt auf Ray-Tracing liegt. Diese Grafikkarten sind dazu in der Lage Games, welche mit Ray-Tracing oder Ray-Marching gerendert werden, mit über 60 FPS zu rendern.

Der Vorteil von Ray-Marching gegenüber von Ray-Tracing ist, dass Licht und Schatten Berechnungen nicht nur am Schnittpunkt des Rays mit dem Objekt, sondern in jedem Schritt des Rays berechnet werden. Dies führt zu noch realistischeren Resultaten und ermöglicht viele Effekte, welche mit Ray-Tracing sehr kompliziert zu implementieren wären. Ein solcher Effekt wäre zum Beispiel die Lichtkrümmung oder einfache Leuchteffekte.

Ray-Marching braucht jedoch meistens mehr Rechenleistung als Ray-Tracing. Da Objekte in Games normalerweise aus Dreiecken aufgebaut sind, ist Ray-Tracing besser geeignet, denn die Berechnung von Schnittpunkten mit Dreiecken ist viel effizienter als den Abstand zu einem Dreieck zu berechnen. Um ein Game mit Ray-Marching effizient zu rendern, müssen die Objekte nicht aus Dreiecken, sondern aus primitiven Objekten aufgebaut sein, für welche es einfache Distance-Functions gibt.

4.2 Andere Bereiche

Neben dem Game-Design gibt es auch noch viele andere Bereiche, in denen Fraktale ihre Anwendungen haben. Vor allem im Druckdesign und im digitalen Design sind sie aufgrund ihrer Ästhetik sehr gut geeignet. Im Verlauf dieser Arbeit habe ich auf mehrere Arten versucht Fraktale im Design zu integrieren. Besonders geeignet erschienen das Posterdesign und das Webdesign, da auf diese Weise die Fraktale deutlich zur Geltung kommen und sehr kreativ integriert werden können.

Der Vorteil von digitalem Design ist, dass die Fraktale auch animiert werden können. Dies ist nicht nur interessanter, sondern führt zu ganz neuen Möglichkeiten. Ein Beispiel wäre ein Fraktal, dessen Parameter nicht durch die Zeit, sondern durch Musik animiert sind. Ein solcher Audio-Visualizer ist in der Applikation als «Audio» enthalten, wobei es sich hier um Julia-Mengen des Mandelbulb-Fraktals handelt, welche auf Musik reagieren. Ein Fraktal-Visualizer wäre auch bei einem Konzert sehr interessant.

4.3 Feedback

Im Verlauf der Arbeit wurden viele Zwischenversionen des Programms von verschiedenen Personen getestet und kommentiert, um schon früh Rückmeldungen zu erhalten. Nachdem das Programm fertiggestellt war, wurde es nochmals ausführlich von ausgewählten Personen getestet. Diese durften dann einen Fragebogen ausfüllen.

Der Schwerpunkt des Fragebogens lag bei den Bereichen Funktionalität, Intuitivität, Performance und Design. Der Fragebogen ist in folgende Teile gegliedert: Allgemein, Fraktale, Performance, Anwendungen, PC-Spezifikationen und Gesamteindruck.

> *«Tolle Arbeit, ich bin beeindruckt»*
>
> Etienne Schorro, Programmierer & CEO, 2020

Der erste Eindruck war bei allen Testpersonen sehr positiv. Die Bedienung wurde grösstenteils als intuitiv empfunden, mit ein paar kleinen Ausnahmen. Das Design hatte auch einen sehr guten Eindruck hinterlassen. Die meisten Testpersonen hatten bereits ein grobes Vorwissen, was sich als genügend erwies, um das Programm und die Fraktal-Parameter zu verstehen und zu bedienen.

Sehr positiv stachen vor allem die 3-dimensionalen Fraktale und das Game heraus. Generell schienen die Mandelbrot-Basierten Fraktale, Mandelbulb und Mandelbox, sehr fesselnd und faszinierend zu sein. Auch das Game «F.R.A.X.» bekam äusserst positive Rückmeldungen. Aber auch einfachere Projekte, wie die Ray-Marching-Demonstrationen kamen sehr gut an.

Ein Grosser Wunsch war die Animation aller Fraktale in der Applikation und der Kamera, unter anderem auch mit der Hilfe von Sinuskurven, wie es im Kapitel 3.7.4 beschrieben ist. Ausserdem gab es vereinzelt Fehler. Einige davon sind Resultat gewisser Limitationen, welche im Kapitel 4.5 beschrieben sind. Bei der Steuerung der Kamera und des Raumschiffs waren die Meinungen gespalten, da dies ein eher subjektives Thema ist. Trotzdem konnte die Steuerung dank des Feedbacks verbessert werden. Die Zweifarbigkeit der meisten Fraktale schien teilweise etwas einschränkend zu sein. Dafür stachen die Fraktale, welche von diesem Farbschema abweichen, noch positiver heraus. Oft erwähnt wurde, dass die Parameter nicht immer ganz verständlich waren, und daher eine kurze Beschreibung der einzelnen Parameter / Optionen angebracht wäre.

Die Testpersonen wurden auch gefragt, ob sie andere Anwendungsbereiche von Fraktalen bzw. ein Potenzial sehen. Die meisten Testpersonen sahen vor allem ein ästhetisches Potenzial der Fraktale, unter anderem auch im Design und in der Kunst:

> *«Neben Lernapplikationen sehe ich dein Konzept zur dreidimensionalen Darstellung von Fraktalen […] ganz klar in der Kunst, wie du sehr vielseitig bewiesen hast. Wie andere der Natur zugrundeliegenden Algorithmen sind Fraktale faszinierend, weil sie mit wenig Information extrem komplexe Welten erschaffen.»*
>
> Tom Kuhn, Ton- & Lichttechniker, 2020

Aber auch in der Filmindustrie:

> *«[Ich sehe ein Potenzial von Fraktalen] in der Filmindustrie, wenn man bei Spielfilmen oder 3D-Animationen Wälder, Bäume und Natur nachahmen möchte, ohne dass alles von Hand programmiert werden soll.»*
>
> Etienne Schorro, Programmierer & CEO, 2020

Der Audio Visualizer braucht hingegen noch einige Verbesserungen. Viele Testpersonen empfanden ihn als zu hektisch und gewisse Frequenzen sind weniger reaktiv als andere. Diese Probleme können mit Feintuning verbessert werden.

4.4 Performance

Ein weiterer wichtiger Aspekt der Applikation ist die Leistung (engl. Performance), ein Schwerpunkt des Fragebogens lag bei diesem Thema. Jede Testperson musste verschiedene Fragen zu der Performance der Applikation beantworten und zusätzlich ihre Systemspezifikationen auflisten. Daraus kann erkannt werden, wie flüssig sich die Applikation auf verschiedener Hardware verhält und somit können Aussagen über die Performance getroffen werden.

Fraktale sind bekannt dafür, dass sie viel Rechenleistung benötigen. Dies ist aufgrund des nicht-linearen Leistungsverhalten der Algorithmen, die Fraktale rendern. Das Verhalten (die Laufzeit) von Algorithmen bei unterschiedlicher Anzahl von Inputs n lässt sich mit der O-Notation beschreiben. Ein Algorithmus, dessen Laufzeit sich linear zu den Anzahl Inputs n verhält, kann mit O(n) beschrieben werden. Solche Algorithmen haben normalerweise eine recht gute Performance. Algorithmen, dessen Laufzeit quadratisch zu n wächst, $O(n^2)$, sind bereits drastisch weniger performant.

In der Praxis kann das Verhalten der Laufzeit auch mithilfe der Anzahl und Verschachtelung der for-Schleifen geschätzt werden. Besitzt der Algorithmus eine einzelne for-Schleife, kann der Algorithmus mit O(n) beschrieben werden, da sich die Laufzeit linear zu den Anzahl Wiederholungen der for-Schleife verhält, welche n ist. Enthält der Algorithmus zwei verschachtelte for-Schleifen, so kann er mit $O(n^2)$ beschrieben werden. Da die Unterschiede von linearen und quadratischen Laufzeiten so drastisch sind, kann der ganze Algorithmus mit $O(n^2)$ beschrieben werden auch wenn nur ein kleiner Teil des Algorithmus $O(n^2)$ ist.

Verschachtelte for-Schleifen sind typisch für Fraktale. Somit lässt sich auch deren Performance einfach erklären. Allein schon die Iteration für jeden Pixel besteht aus zwei verschachtelten for-Schleifen und ist somit $O(n^2)$. In unserem Fall lässt sich dies aber nicht so genau sagen, da die Iteration der Pixel vom Compute-Shader gemacht wird, also auf der Grafikkarte ausgeführt wird, und somit Parallel verläuft. Des Weiteren ist n hier nicht nur eine Zahl, sondern die horizontale und vertikale Auflösung des Bildschirms.

Alle Fraktale in dieser Arbeit, welche mit Ray-Marching gerendert werden, sind mindestens $O(n^2)$, da der Algorithmus des Fraktals aus mindestens einer for-Schleife besteht und sich diese innerhalb der for-Schleife des Marschier-Algorithmus befindet. Auch hier ist n zwei Zahlen, und zwar die Anzahl der Schritte des Ray-Marching Algorithmus und die Anzahl Iterationen. In der Informatik ist die O-Notation also nur eine Approximation bzw. Schätzung der Laufzeit. Die Algorithmen der Fraktale, welche nicht mit Ray-Marching gerendert werden, sind meistens O(n), wobei n die Anzahl der Iterationen ist.

Daraus lässt sich folgern, dass die Leistung von verschiedenen Parametern abhängig ist:

- Die Anzahl Schritte der Ray-Marching-Funktion
- Die Anzahl der Iterationen
- Der Aufbau der Fraktal-Funktion bzw. Distance-Function
- Die Auflösung des Bildschirms
- Die Leistung und Parallelität der Grafikkarte

Selbstverständlich gibt es noch viele weitere Parameter, welche die Performance beeinflussen, diese werden aber hier nicht behandelt.

Dank verschiedenen Tests und des Fragebogens konnte das bestätigt werden. Die Grafikkarte spielte hier die wichtigste Rolle. Ein Laptop mit integrierter Grafikkarte erreichte knapp 20 FPS, während die meisten Computer mit durchschnittlicher Gaming-Grafikkarte über 100 FPS erzielen konnten. Auch die Auflösung des Bildschirms hatte einen spürbaren Einfluss. So konnte die Performance auf einem Laptop fast verdoppelt werden, wenn die Auflösung stark reduziert wurde. Die Iterationen und Schritte hatten natürlich auch einen grossen Einfluss.

Unterschiede in der Performance gab es nicht nur zwischen verschiedener Hardware, sondern auch zwischen verschiedenen Fraktalen. Während einfache Fraktale, wie zum Beispiel die 3-dimensionalen Sierpinski-Analoga über 200 FPS erzielten, gerieten die komplexeren Mandelbrot-Basierten-Fraktale teilweise unter 60 FPS. Im Fragebogen wurde auch oft erwähnt, dass sich die Mandelbox und das Mandelbulb-Fraktal nicht immer sehr flüssig verhielten.

4.5 Limitationen

Wie bereits erwähnt, ist die Leistung des Computers definitiv eine Limitation. Um eine fliessende Anzahl von FPS (> 60 FPS) zu erzielen, braucht es mindestens eine gute Grafikkarte.

Nebst Hardware Limitationen war die Präzision von float-Variablen oft ein Problem bzw. eine Limitation. Dies kommt vor allem auf zwei Arten zum Vorschein. Zum Beispiel ist ab einem gewissen Zoomlevel die Positionsvariable beim Mandelbrot-Fraktal nicht mehr präzise genug und die Bedienung wird stark erschwert. Bei weiterem Zoomen kann die Kamera nicht mehr verschoben werden und das Fraktal wird verpixelt. Die Berechnung des Fraktals erfolgt mit double-precision-Variablen, um genauere Resultate zu erhalten. Aber auch diese haben ihre Grenze, welche durch die Verpixelung sichtbar wird. Die Positionsvariable kann hingegen nicht als double gespeichert werden, da sie auf Unity-Vektoren basiert, welche nur float Präzision haben. Eine Umstellung auf double wäre zwar möglich, jedoch sehr umständlich.

Eine weitere Art wie die float-Präzision sichtbar wird, ist bei grosser Entfernung vom Ursprung. Mit grösseren Werten nimmt die Präzision der Kommastellen ab, aufgrund der Art wie eine float-Variable gespeichert wird. Auch hier wird die Position mit Unity-Vektoren gespeichert und hat somit dieses Problem, die Präzision der Position nimmt also bei grösseren Werten, einer grösseren Entfernung vom Ursprung, ab.

Auch aufgrund der Umgebung, welche für diese Arbeit gewählt wurde, entstanden mehrere Einschränkungen, welche hier aber nicht genauer besprochen werden.

Limitierend war auch, dass die selbstgemachten Render-Engines nicht von vielen Unity Funktionalitäten profitieren konnten, da sie auf Compute-Shader basieren. Ein Beispiel davon wären die Postprocessing-Effekte, welche grösstenteils auf dem Depth-Buffer basieren und somit nicht mit den eigenen Render-Engines funktionierten, da diese nicht denselben Depth-Buffer verwenden (vgl. Kapitel 3.7.2).

4.6 Schlussfolgerung

Fraktale haben in sehr vielen Bereichen ihre Anwendung. Eine davon ist das Game-Design. Um diesen Anwendungsbereich zu untersuchen, habe ich ein Game namens «F.R.A.X.» programmiert. Das Game besteht darin, ein Raumschiff durch eine abstrakte Welt aus animierten Menger-Schwämmen zu fliegen. Das Resultat war ein fesselndes Game, das allen Testpersonen sehr gut gefallen hat. Neben «F.R.A.X» gibt es auch noch andere Games, welche Fraktale verwenden. Eines davon ist der «Marble Marcher» von «CodeParade» (vgl. github.com/HackerPoet/MarbleMarcher). Mit Hilfe von Kollisionsberechnung sind auch Games mit interaktiven Fraktalen möglich.

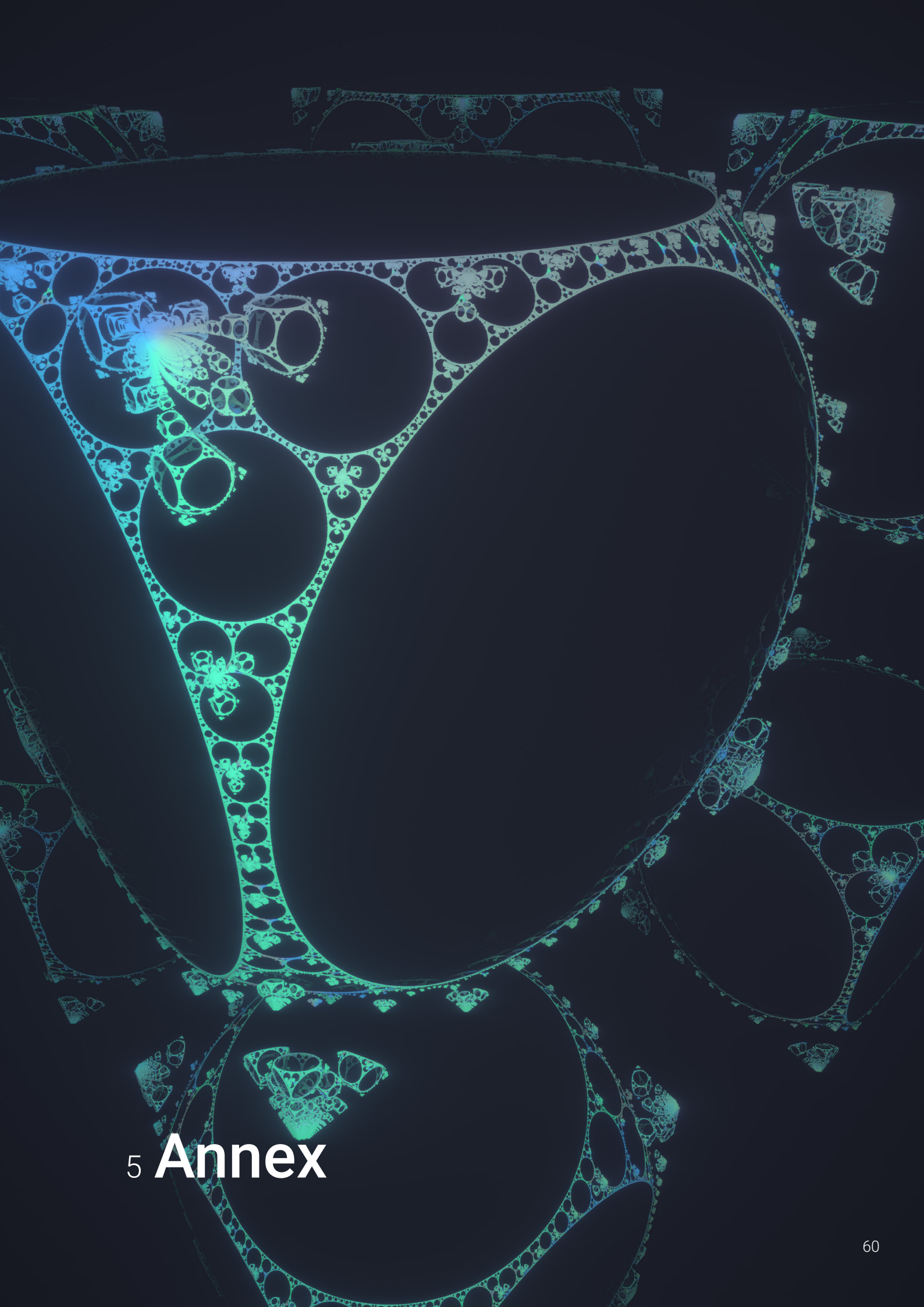
Da die meisten 3-dimensionalen Fraktale mit Ray-Marching gerendert sind, sind auch die meisten Games mit Fraktalen ganz oder teilweise mit Ray-Marching gerendert. Ray-Marching erwies sich im Vergleich zu der konventionellen Render-Technik der Rasterung als viel realistischer und unkomplizierter. Ray-Marching ermöglicht es, viele verschiedene Effekte viel einfacher und fast ohne erhöhte Rechenleistung zu implementieren, welche mit der Rasterung sehr viel aufwändiger wären.

Eine ähnliche Render-Technik wie Ray-Marching ist Ray-Tracing. Ray-Tracing basiert auf der Berechnung von Überschneidungen des Rays mit den Objekten einer Szene. Ray-Tracing hat bereits ihre Anwendung in der Game-Entwicklung und es gibt bereits Grafikkarten, welche für Ray-Tracing optimiert sind. Auch Ray-Marching wäre sehr gut geeignet für die Game-Entwicklung, jedoch müssten die Objekte nicht aus Polygonen, wie das normalerweise gemacht wird, sondern aus primitiven Objekten aufgebaut sein, für welche es einfache Distance-Functions gibt. Wie auch bei Ray-Tracing wäre eine Grafikkarte essenziell, da sonst nur sehr wenige FPS erreicht werden könnten, da beide Techniken sehr rechenaufwendig sind. Dafür ergeben sich einiges realistischere Resultate.

Neben dem Game-Design gibt es auch noch viele andere Bereiche, in denen Fraktale ihre Anwendungen haben. Vor allem im Druckdesign und im digitalen Design sind sie aufgrund ihrer Ästhetik sehr gut geeignet. Im Rahmen dieser Arbeit habe ich mehrere Objekte designt, in welchen ich versucht habe, Fraktale zu integrieren. Besonders geeignet erschienen das Posterdesign und das Webdesign, da auf diese Weise die Fraktale deutlich zur Geltung kommen und sehr kreativ integriert werden können. Das digitale Design hat ausserdem noch den Vorteil, dass die Fraktale animiert werden können. Ein Beispiel wäre ein Fraktal, dessen Parameter nicht durch die Zeit, sondern durch Musik animiert sind. Dies könnte auch bei einem Konzert als Audiovisualizer verwendet werden.

Den Testpersonen haben auch die Design-Bereiche sehr gut gefallen, darunter vor allem das Poster- und Webdesign. Die Testpersonen wurden auch gefragt, ob sie andere Anwendungsbereiche von Fraktalen bzw. ein Potenzial sehen. Die meisten Testpersonen sahen vor allem ein ästhetisches Potenzial der Fraktale, unter anderem auch im Design und in der Kunst aber auch in der Filmindustrie.

Generell musste ich feststellen, dass sich die Anwendungen von Fraktalen im Bereich Game-Design vor allem auf das Ästhetische belaufen, dafür aber als sehr schön empfunden werden.



5 Annex

5.1 Schlusswort

Im Rahmen dieser Arbeit möchte ich gerne meinem Betreuer und Mathematiklehrer, Yves Gärtner, welcher mich ausgiebig unterstützt und inspiriert hat, den Testpersonen Tom Kuhn, Etienne Schorro, David Frank, Zara Franco, Noel Engeler und Yannis Rutishauser, welche sich die Zeit genommen haben, das Programm zu testen und kommentieren, Olaf Prinz für das Durchlesen und Korrigieren der Arbeit, meinen zwei neugierigen Cousinen, welche in dieser Arbeit erwähnt sein wollten, meinen Eltern und meinen beiden Katzen ganz herzlich danken.

5.2 Deklaration

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt habe,
- dass ich auf eine eventuelle Mithilfe Dritter in der Arbeit ausdrücklich hinweise,
- dass ich vorgängig die Schulleitung und die betreuende Lehrperson informiere, wenn ich
  - diese Maturaarbeit bzw. Teile oder Zusammenfassungen davon veröffentlichen werde
  - Kopien dieser Arbeit zur weiteren Verbreitung an Dritte aushändigen werde
- dass mir der Inhalt des «Merkblatts Plagiat» sowie auch die Konsequenzen eines Plagiats bekannt sind.

Luzern, 02.12.2020

5.3 Glossar

3rd-Person:
Kameraperspektive in Computerspielen, bei der sich die Kamera hinter dem Spieler befindet

API:
Programmierschnittstelle, Schnittstelle für Software, ermöglicht das Integrieren von einem Programm in einem anderen Programm

Applikation:
Computerprogramm

Artefakt:
Darstellungsfehler

Buffer:
Temporärer Zwischenspeicher für Daten

Controls:
Steuerung in einem Computerspiel / -programm

Feintuning:
Präzise, kleine Anpassungen

FPS:
Bilder pro Sekunde

Framework:
Programmiergerüst, Rahmen, in welchem Programme erstellt werden können

Game:
Computerspiel

Game-Engine:
Software für das Entwickeln von Computerspielen, beinhaltet nützliche Tools

GUI:
Benutzeroberfläche

IDE:
Integrierte Entwicklungsumgebung, beinhaltet nützliche Tools

Kernel:
Hier: Untershader innerhalb eines Shaders

Laufzeit:
Die Zeit, in der das Programm operiert oder wie viel Zeit das Programm für etwas benötigt

Library:
Bibliothek von nützlichen Funktionen und Klassen

Loading-Screen:
Ladebildschirm

Multiplayer:
Mehrspielermodus, mehrere Spieler spielen miteinander

Open-Source:
Software mit offengelegtem Quellcode, welcher für jeden sichtbar ist

Plugins & Assets:
Erweiterungen für eine Software

Postprocessing:
Effekte, welche nach dem Rendern berechnet werden

Rendern:
Berechnung des Bildes unteranderem mit Hilfe von Shader

Roll, Pitch, Yaw:
Achsen, in welche sich ein Flugzeug dreht (Rollen, Neigen und Gieren)

Shader:
«Schattierer» Computer Programm, welches auf der Grafikkarte ausgeführt wird und Bildeffekte berechnet bzw. Bilder rendert

Struct:
Ein Datentyp, welcher aus verschiedenen anderen Datentypen bestehen kann (ähnlich wie ein Objekt)

Umgebung:
Programmierumgebung (Betriebssystem, Code-Editor, Programmiersprachen, weitere Software)

User:
Benutzer

5.4 Quellenverzeichnis

5.4.1 Literatur

Mandelbrot, Benoit B. (1982):
The Fractal Geometry of Nature. New York: W.H. Freeman and Company.

Mandelbrot, Benoit B. (1977):
Fractals. Form, Chance, and Dimension. San Francisco: W.H. Freeman and Company.

Lauwerier, Hans (1992a):
Fraktale verstehen und selbst programmieren. Band 1. Einführung. 2. Auflage. Hückelhoven: R. Wittig Fachbuchverlag.

Lauwerier, Hans (1992b):
Fraktale verstehen und selbst programmieren. Band 2. Vertiefung. Hückelhoven: R. Wittig Fachbuchverlag.

Jürgens, Hartmut / Peitgen, Heinz-Otto (1990):
Fraktale. Gezähmtes Chaos. München: Carl Friedrich von Siemens Stiftung.

Falconer, Kenneth J. (1993):
Fraktale Geometrie. Mathematische Grundlagen und Anwendungen. Heidelberg / Berlin / Oxford: Spektrum Akademischer Verlag.

5.4.2 Internetquellen

Shishikura, Mitsuhiro (1991).
The Hausdorff dimension of the boundary of the Mandelbrot set and Julia sets. URL: arxiv.org/pdf/math/9201282.pdf [Stand: 03.08.2020]

Sanderson, Grant (2017):
Fractals are typically not self-similar. URL: youtube.com/watch?v=gB9n2gHsHN4 [Stand: 02.08.2020]

White, Daniel (2009):
The Unravelling of the Real 3D Mandelbrot Fractal. URL: skytopia.com/project/fractal/mandelbulb.html [Stand: 04.08.2020]

Nylander, Paul (2009):
Hypercomplex Fractals. URL: bugman123.com/Hypercomplex/index.html [Stand: 04.08.2020]

Lowe, Tom (2010):
What is a Mandelbox. URL: sites.google.com/site/mandelbox/what-is-a-mandelbox [Stand: 04.08.2020]

Chen, Rudi (2014):
The Mandelbox Set. URL: digitalfreepen.com/mandelbox370 [Stand: 04.08.2020]

Christensen, Mikael H. (2011):
Syntopia. URL: blog.hvidtfeldts.net [Stand: 20.08.2020]

Kuri, David (2018):
GPU Ray Tracing in Unity – Part 1. URL: blog.three-eyed-games.com/2018/05/03/gpu-ray-tracing-in-unity-part-1 [Stand: 20.08.2020]

Quilez, Inigo (2020a):
Distance Functions. URL: iquilezles.org/www/articles/distfunctions/distfunctions.htm [Stand: 20.08.2020]

Quilez, Inigo (2020b):
2D Distance Functions. URL: iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm [Stand: 20.08.2020]

Wikipedia (2020):
List of color spaces and their uses. URL: en.wikipedia.org/wiki/List_of_color_spaces_and_their_uses [Stand: 26.08.2020]

Lindbloom, Bruce J. (2013):
Useful Color Equations. URL: brucelindbloom.com/index.html?Math.html [Stand: 26.08.2020]

Wikipedia (2020):
Big O notation. URL: en.wikipedia.org/wiki/Big_O_notation [Stand: 6.09.2020]

Barile, Margherita / Weisstein, Eric W. (2020):
Cantor Set. URL: mathworld.wolfram.com/CantorSet.html [Stand: 27.09.2020]

5.4.3 Dokumentationen

Unity Technologies (2020):
Unity User Manual. URL: docs.unity3d.com/Manual/index.html [Stand: 08.08.2020]

Unity Technologies (2020):
Unity Scripting Reference. URL: docs.unity3d.com/ScriptReference/index.html [Stand: 08.08.2020]

Microsoft (2020):
C# Documentation. URL: docs.microsoft.com/dotnet/csharp [Stand: 08.08.2020]

Microsoft (2020):
HLSL Documentation. URL: docs.microsoft.com/windows/win32/direct3dhlsl [Stand: 08.08.2020]

5.4.4 Bilder

Alle Bilder und Grafiken in dieser Arbeit sind Eigenkreationen und wurden hauptsächlich mit der Applikation «Fractals» erstellt.