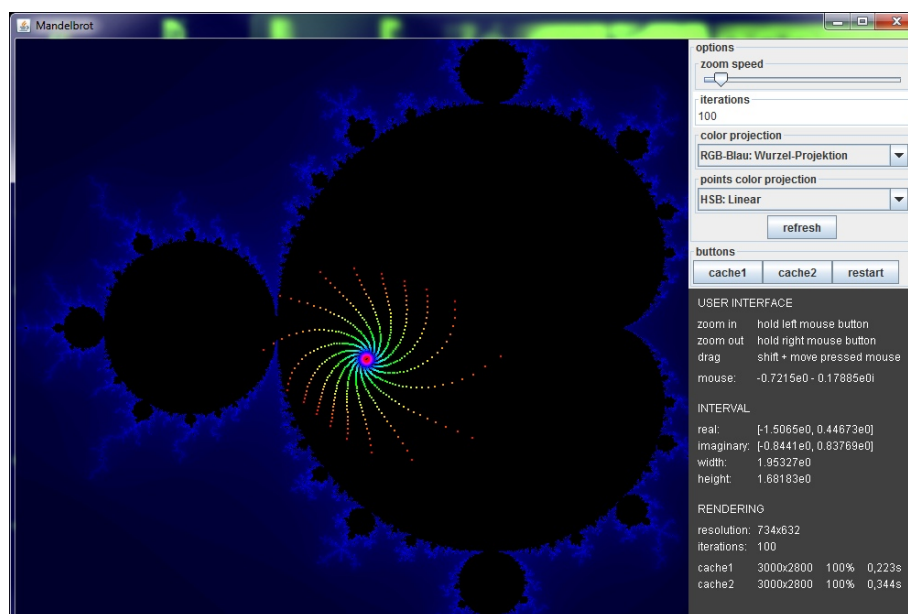


Fachbereich Mathematik/Informatik


Abschlussleistung

ABV-Kurs
Einführung in die Visualisierung



Projekt 04: Mandelbrot

Namen: Felicia Burtscher, Matthias Kupferschmidt,
Nora Rüländer und Eike Sommer

Matrikelnummern: 

Betreuer: Prof. Dr. Konrad Polthier, Konstantin Poelke

Datum der Abgabe: 02. Oktober 2015

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
2	Mathematischer Hintergrund	2
2.1	Grundlagen	2
2.2	Verhalten der Folgenglieder	3
3	Erste Implementierungen	6
3.1	Schwarz-Weiß-Darstellung	6
3.2	Farbabstufungen	8
3.2.1	Abstufungen in einer Farbe mittels eines RGB-Kanals	8
3.2.2	Vielfarbigkeit mittels HSB-Farbraum	8
3.3	Zoom-Funktion	9
3.4	Darstellung der Folgenglieder	9
4	Finale Version: Benutzeroberfläche und Optimierung	11
4.1	Allgemeine Struktur	11
4.2	Basisdefinitionen der Folge und Farbe	12
4.3	Rendering	13
4.3.1	Optimierung des Rendering-Prozesses durch TiledRendering	15
4.4	Herausforderungen bei der Modellierung von Vektoren, komplexen Zahlen und ihren Rechtecken	16
5	Erörterung verschiedener Parameter	18
5.1	Variation von z_0	18
5.2	Maximale Iterationen und Farbabbildung	18
5.3	Verschiedene Farbprojektionen	19
5.3.1	Linear vs. quadratisch	19
5.3.2	Linear mit Koeffizienten	20
6	Schlussbemerkungen	21
6.1	Ausblick	21
6.1.1	Mögliche Ergänzungen	21
6.1.2	Weitere Optimierungen	21
6.2	Persönliches Fazit	21
	Literaturverzeichnis	23

1 Einleitung

Die Autoren dieses Berichts hatten im Rahmen des ABV-Kurses *Einführung in die Visualisierung* am Fachbereich Mathematik/Informatik der FU Berlin die Aufgabe, die Mandelbrot-Menge in JavaView oder Python zu implementieren; alternativ konnte Java verwendet werden. Die Projektarbeit teilte sich auf in drei Bestandteile: Einarbeitung in das Thema und in die Sprache Java, praktische Umsetzung der Mandelbrot-Menge in Java in mehreren Etappen sowie Vorbereitung der Präsentationsfolien. Die folgende Ausarbeitung bezieht sich größtenteils auf den zweiten Teil, die Visualisierung der Mandelbrot-Menge in Java.

1.1 Motivation

Fraktale oder fraktal-ähnliche Objekte laufen uns im Alltag an vielen Stellen über den Weg. Betrachten wir z.B. einen Küstenverlauf vom Flugzeug aus, schaut dieser typischerweise nicht gerade, sondern sehr zerklüftet aus und weist viele Buchten und Halbinseln auf. Egal ob man die Küstenlinie von weit entfernt oder als Nahaufnahme am Strand beobachtet - sie hat immer die gleiche zerklüftete Form. Diese Ähnlichkeit der Form der Küstenlinie auf unterschiedlichen Skalen wird auch *Selbst-Ähnlichkeit* genannt und ist eine charakteristische Eigenschaft von vielen Fraktalen. [1, Seite 103 ff.]

Der Ausdruck *Fraktal* wurde von dem französischen Mathematiker Benoît Mandelbrot geprägt, der einer der Ersten war, der entdeckte, dass die Welt voll mit Fraktalen ist, d.h. viele Objekte in der Natur besitzen solch eine Struktur der Selbst-Ähnlichkeit, neben Küstenlinien z.B. Bergketten, Schneeflocken und Bäume. Nach Mandelbrot ist auch die berühmte *Mandelbrot-Menge* benannt, die der Mathematiker 1980 - als erstmals Computer mit der nötigen Rechenkapazität verfügbar waren - optisch darstellte. Mandelbrot stellte die Hypothese auf, dass sogar das Universum fraktal-ähnliche Eigenschaften besitze - man denke an die Verteilung der Galaxien, die Cluster an Galaxien, die Cluster von Clustern usw. Die visuelle Darstellung der Mandelbrot-Menge erfreute sich in der zweiten Hälfte des 20. Jahrhunderts wegen ihrer Ästhetik auch außerhalb der Mathematik großer Beliebtheit.

Im Allgemeinen meint ein *Fraktal* eine geometrische Form, die eine *feine Struktur auf jeder Skala* besitzt. Das besondere an Fraktalen ist, dass sie (meist) auf einfach zu formulierenden Rekursionsvorschriften basieren, die jedoch bei wiederholter Anwendung (Iteration) äußerst komplexe und formenreiche Objekte erzeugen. Es gibt unzählige Beispiele von Fraktalen oder fraktal-ähnliche Strukturen; im Folgenden wollen wir die Mandelbrot-Menge genauer untersuchen. [1, Seite 103 ff.]

2 Mathematischer Hintergrund

2.1 Grundlagen

Die Mandelbrot-Menge ist das Paradebeispiel für eine Menge mit nahezu unendlichem Formenreichtum. Ihr Rand bildet ein Fraktal, allerdings ohne exakte Selbstähnlichkeit. Sie ist von vielfältiger Struktur, kommt durch die wiederholte Anwendung einer simplen Regel zustande und ist eins der bekanntesten Beispiele für mathematische Visualisierung.

Die Mandelbrot-Menge lässt sich auf mehrere Weisen definieren. Wir bedienen uns der rekursiven Definition.

Definition 1 (Mandelbrot-Menge) Die Mandelbrot-Menge ist die Menge \mathbb{M} aller komplexen Zahlen c , für die die rekursiv definierte Folge

$$f(z_n) = z_{n+1} = z_n^2 + c \quad (2.1)$$

mit Anfangsglied

$$z_0 = 0.$$

beschränkt bleibt. [6, Seite 418]

Aus einem Startwert berechnet sich also ein Funktionswert, der wieder in dieselbe Funktion f eingesetzt wird usw. So erhalten wir die Glieder unserer Folge:

$$\begin{aligned} z_0 &= 0 \\ f(z_0) &= z_1 = z_0 + c = c \\ f(z_1) &= f(f(z_0)) = f(c) = c^2 + c \\ f(z_2) &= f(f(f(z_0))) = f(f(z_1)) = f(c^2 + c) = (c^2 + c)^2 + c \\ &\dots \end{aligned}$$

In anderen Worten: Eine komplexe Zahl c ist Teil der Mandelbrot-Menge, falls beginnend bei $z_0 = 0$ und nach wiederholten Iterationen (Anwenden obiger Operation) der Betrag von z_n beschränkt bleibt, egal wie groß n wird, d.h. der Betrag der Folgenglieder wächst nicht über alle Grenzen. Kurzum:

$$\mathbb{M} := \{c \in \mathbb{C} : \text{die Folge } (z_k) \text{ ist beschränkt}\}$$

Beispiel 1.

$$c = 0, 1 + 0 \cdot i$$

$$\begin{aligned}
 z_1 &= 0^2 + 0, 1 = 0, 1 \\
 z_2 &= (0, 1)^2 + 0, 1 = 0, 11 \\
 z_3 &= (0, 11)^2 + 0, 1 = 0, 1121 \\
 z_4 &= 0, 11256641 \\
 z_5 &= 0, 1126711966602881 \\
 &\dots \\
 z_{1000} &= 0, 11270166537925831 \\
 &\dots
 \end{aligned}$$

Beispiel 2.

$$c = 1 + 0 \cdot i$$

$$\begin{aligned}
 z_1 &= 0^2 + 1 = 1 \\
 z_2 &= 1^2 + 1 = 2 \\
 z_3 &= 2^2 + 1 = 5 \\
 z_4 &= 26 \\
 z_5 &= 257 \\
 &\dots
 \end{aligned}$$

Behauptung 1 Sei \mathbb{M} die Mandelbrot-Menge. Dann gilt:

$$\mathbb{M} \subset \{c \in \mathbb{C} : |c| \leq 2 \forall k \in \mathbb{N}\}.$$

Behauptung 2 Sei $c \in \mathbb{M}$ ein Element der Mandelbrot-Menge. Dann gilt für die entsprechende Folge (z_k) :

$$|z_k| \leq 2 \forall k \in \mathbb{N}.$$

Ein Beweis dieser beiden Behauptung ist in [2, Seiten 2 - 6] zu finden. □

Behauptung 3 \mathbb{M} ist achsensymmetrisch zur reellen Achse.

Beweis:

Sei $c = a + bi$ eine komplexe Zahl und $d = a - bi$ ihre komplex Konjugierte. Für c und d ergeben sich im nächsten Iterationsschritt:

$$\begin{array}{ll}
 z_{n-1} &= c = a + bi & z_{n-1} &= d = a - bi \\
 z_n &= (a + bi)^2 + (a + bi) & z_n &= (a - bi)^2 + (a - bi) \\
 &= a^2 + 2abi - b^2 + a + bi & &= a^2 - 2abi - b^2 + a - bi \\
 &= (a^2 - b^2 + a) + (2ab + b)i & &= (a^2 - b^2 + a) - (2ab + b)i
 \end{array}$$

Damit sind dann also auch alle Folgenglieder zueinander komplex konjugiert. □

2.2 Verhalten der Folgenglieder

Aus der Definition der Mandelbrot-Menge folgt, dass die Bahnen für Punkte innerhalb der Mandelbrot-Menge jeweils auf ein Gebiet beschränkt bleiben.

Es lassen sich verschiedene Verhalten der Folgenglieder beobachten:

- *Konvergenz*: Die Bahnpunkte streben auf einen Grenzwert zu.
- *Periodisches Grenzverhalten*: Die Bahnpunkte konvergieren gegen einen periodischen Zyklus.
- *Beschränktes Verhalten*: Die Bahnpunkte konvergieren weder, noch zeigen periodisches Verhalten, bleiben jedoch auf ein Gebiet beschränkt. Die Bahnpunkte einiger Zahlen c variieren chaotisch.
- *Wechselndes Verhalten*: Die Bahnpunkte zeigen abschnittsweise beschränktes, chaotisches oder periodisches Verhalten.

Für Zahlen auf der reellen Achse ist das Verhalten im Feigenbaum-Diagramm (Bild 2.1) dargestellt. Die c -Werte reichen von -2 bis zu 0.25 ; auf der Ordinate sind die Folgenglieder von z_{100} bis z_{200} aufgetragen. Darunter sehen wir zum Vergleich die Mandelbrot-Menge gestreckt. Für $c = -2$ sind die nächsten Folgenglieder $2, 2, 2, \dots$ 2 ist also ein Fixpunkt für dieses c . Zwischen -2 und -1.25 finden wir Punkte deren Bahnen chaotisch variieren und Bahnen, die gegen einen periodischen Zyklus konvergieren. Der Punkt $c = -1.3$ beispielsweise besitzt eine Bahn, die einen Zyklus der Länge 3 aufweist. Für alle $c \in \mathbb{R}$ mit $-1.25 \leq c \leq 0.25$ streben die Bahnen auf einen Grenzwert zu. [4]

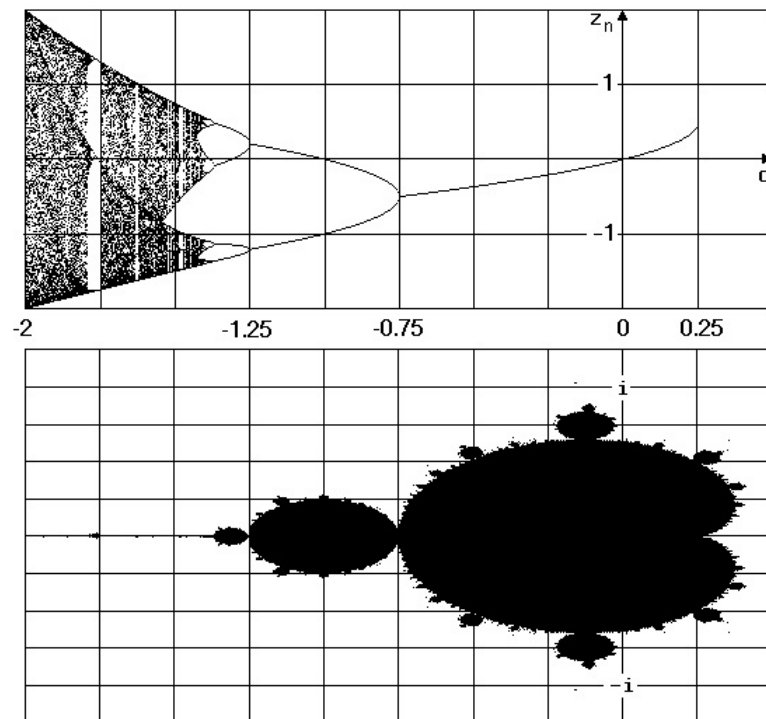


Abbildung 2.1: Das Feigenbaumdiagramm und die Mandelbrot-Menge [4, Bildquelle]

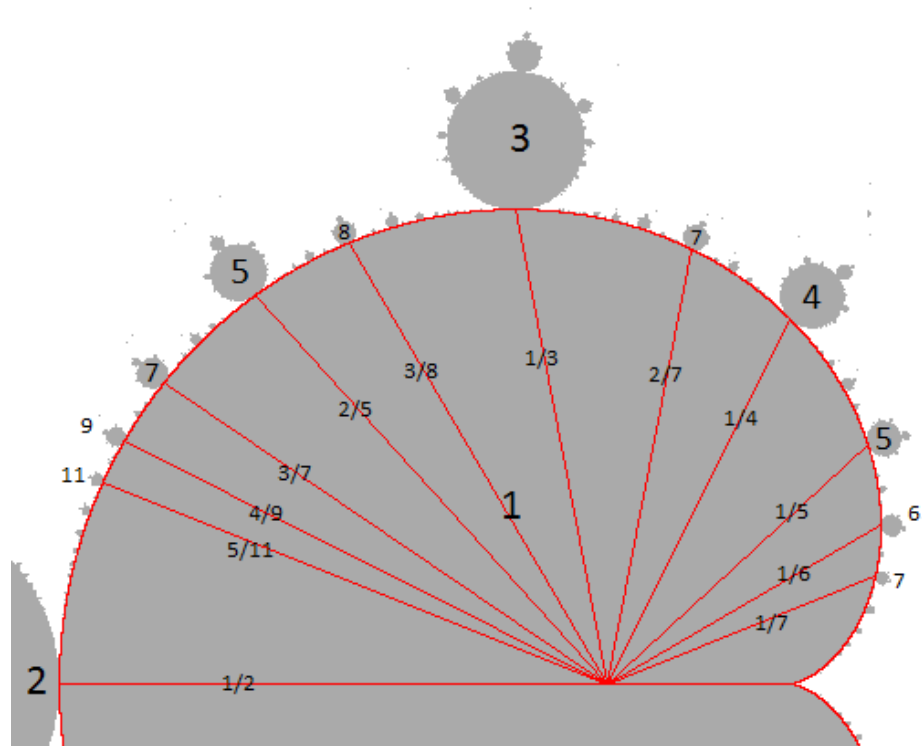


Abbildung 2.2: Vergrößerter Ausschnitt des Hauptkörpers der Mandelbrot-Menge mit anliegenden Knospen [3, Bildquelle]

Nun betrachten wir Zahlen c , die innerhalb der Knospen rund um den Hauptkörper liegen. Bild 2.2 zeigt einen Ausschnitt der Mandelbrot-Menge. Stammt c aus einer der Knospen, so gibt die Zahl in der Knospe die Länge des periodischen Grenzyklus an.

3 Erste Implementierungen

Um die Mandelbrot-Menge visuell darzustellen, haben wir die Sprache Java gewählt. Dabei war unsere Überlegung, dass Pixelgrafiken direkt mit den Standard-Bibliotheken von Java implementiert werden können.

3.1 Schwarz-Weiß-Darstellung

Die grundlegendste Methode, die Mandelbrot-Menge darzustellen ist zweifarbig. Man wählt eine maximale Anzahl an Iterationsschritten und das Intervall $I \subset \mathbb{C}$ in dem die Mandelbrot-Menge liegt.

Praktisch erzeugen wir dazu ein JFrame, dass dem Intervall I entspricht. Mittels zwei verschachtelter for-Schleifen geht unser Programm spaltenweise durch alle Pixel des Frames. Es berechnet, welcher komplexen Zahl c die Pixelkoordinaten entsprechen und führt die Iterationsvorschrift für dieses c aus.

Jedes $c \in I$, für das die Iterationsvorschrift in weniger als der maximalen Schrittzahl ein Folgenglied liefert, dessen Betrag größer als 2 ist, wird weiß eingefärbt und die Methode macht weiter mit dem nächsten Pixel. Jedes c , das alle Iterationsschritte durchläuft, wird schwarz gefärbt.

Genauer gesagt, erhält man damit natürlich nur eine Approximation der Mandelbrot-Menge, denn es gibt Punkte, die als Teil der Mandelbrot-Menge anerkannt werden, obwohl ihre Folge möglicherweise in einem späteren Iterationsschritt divergieren würde. Allerdings ist ab einer Iterationsobergrenze von 100 hier in der Gesamtdarstellung visuell kein Unterschied mehr zu erkennen.

Dabei benutzen wir eine eigene Klasse, die zwei (reelle) Double-Werte als komplexe Zahl auffasst (Real- und Imaginärteil) und die für diese komplexe Zahlen Addition, Multiplikation und die euklidische Norm implementiert.

```
public class Complex {
    private double r;
    private double i;
    public Complex(double r, double i) {
        this.r = r;
        this.i = i;
    }
    public Complex Plus(Complex c) {
        return new Complex(r + c.r, i + c.i);
    }
    public Complex Multiply(Complex c) {
```



```

        return new Complex(r * c.r - i * c.i, r * c.i + i * c.r);
    }
    public double AbsoluteValue() {
        return Math.sqrt(r * r + i * i);
    }
}

```

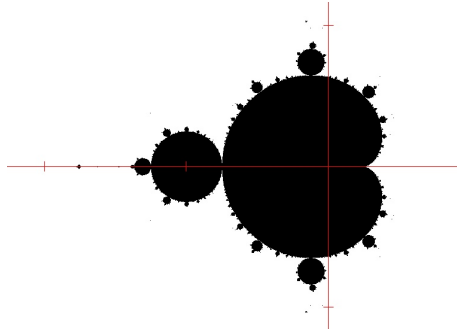


Abbildung 3.1: Die Mandelbrot-Menge in Schwarz-Weiß mit Achsen. Alle komplexen Zahlen c , für die die Folge z_k beschränkt ist, sind schwarz eingefärbt. Gezeigt wird der Ausschnitt der komplexen Zahlenebene: $-2,3 \leq \operatorname{Re}(c) \leq 1$ und $-1,2 \leq \operatorname{Im}(c) \leq 1,2$.

Herausforderungen

- Tatsächlich erhält man mittels `getWidth()` und `getHeight()` nicht die Koordinaten der Bildpixel, sondern die des gesamten Frames, inklusive des je nach Betriebssystem unterschiedlich dicken Rahmens. Zunächst lösen wir das händisch, indem wir das Intervall I nach oben etwas größer machen.
- Auf der reellen Achse bis -2 ist zunächst eine unverhältnismäßig dicke, schwarze Linie zu sehen. Denn obwohl diese Werte theoretisch alle zur Mandelbrot-Menge gehören (die Menge ist zusammenhängend), sollte diese Linie bei keiner Auflösung komplett zu sehen sein. Wir umgehen dieses numerische Artefakt, indem wir die Höhe des Frames statt auf 600 Pixel auf 601 Pixel setzen, sodass die reelle Achse nicht direkt ausgewertet wird.

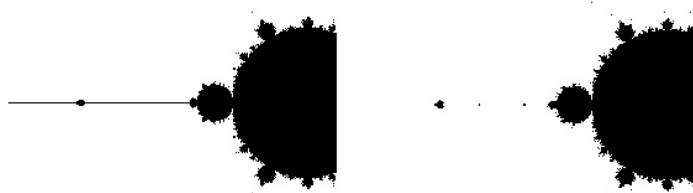


Abbildung 3.2: Links der Ausschnitt mit Artefakt bei einer Höhe von 600 Pixeln, rechts ohne Artefakt bei einer Höhe von 601 Pixeln

3.2 Farbabstufungen

Die tatsächliche Dynamik und Strukturfeinheit der Mandelbrot-Menge wird erst wirklich deutlich, wenn ihre Umgebung farblich abgestuft eingefärbt wird. Eine verbreitete Art der Färbung entsteht durch den Escape-Time-Algorithmus: Die Punkte, die alle Iterationsschritte durchlaufen, werden weiterhin schwarz gefärbt. Alle anderen Punkte erhalten abhängig vom Abbruch-Iterationsschritt einen Farbwert [5, Seite 41]. Dabei gilt der Ansatz, den Kontrast zwischen Rand und Menge zu maximieren. Zum Beispiel wird ein Pixel umso heller gefärbt, je mehr Iterationsschritte er benötigt um betragsmäßig größer als 2 zu werden.

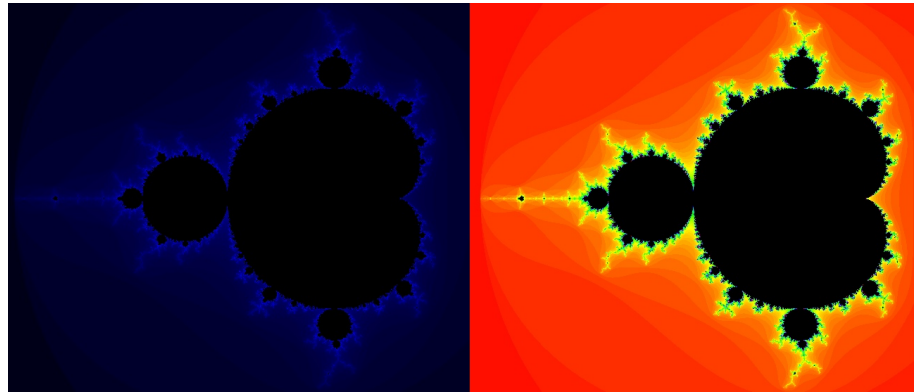


Abbildung 3.3: Links: RGB-Kanal blau, rechts: HSB-Raum

3.2.1 Abstufungen in einer Farbe mittels eines RGB-Kanals

Die offensichtlichste Variante ist, einen Kanal im RGB-Raum (Rot/Grün/Blau) zu wählen, zum Beispiel Blau. Dieser kann Integer-Werte zwischen 0 und 255 annehmen.

Dabei sei `num` der abbrechende Iterationsschritt, der wird durch die maximale Anzahl von Iterationsschritten geteilt, um einen Wert zwischen 0 und 1 zu erhalten und dies wird mit 255 multipliziert. Da die Iteration für „viele“ Zahlen schon früh abbricht, nehmen wir die Wurzel von `num`, um die Dynamik besser im Gesamtbild zu verteilen.

```
public Color num2col(int num) {
    double v = num / (double)maxIterations;
    return new Color(0, 0, (int)(Math.sqrt(v) * 255));
}
```

3.2.2 Vielfarbigkeit mittels HSB-Farbraum

Um mehrere Farben in der Darstellung zu erhalten, haben wir, anstatt auch die anderen RGB-Kanäle zu benutzen, den HSB-Raum (Hue, Saturation, Brightness) gewählt. Dabei verändern wir nur den Farbwert (Hue), der Double-Werte zwischen 0 und 1 annehmen kann und einen Farbverlauf von Rot-Orange-Gelb-Grün-Blau-Violett-Rot liefert. Weiterhin färben wir die Pixel, welche die Mandelbrot-Menge selbst repräsentieren, schwarz.

Dieser Farbraum ermöglicht genügend Vielfalt in der Darstellung, um auch andere Abbildungen, neben der hier demonstrierten Linearen, zu untersuchen (siehe 5.3).

```
public Color num2col(int num) {
    if (num == 0){
        return Color.BLACK;
    }else{
        double v = num / (double)maxIterations;
        return Color.getHSBColor((float)(v), 1, 1);
    }
}
```

3.3 Zoom-Funktion

Um den Formenreichtum des Rands der Mandelbrot-Menge zu erkennen, ist eine Zoom-Funktion notwendig. Zunächst entscheiden wir uns für einen diskreten Zoom: Per Rechtsklick mit der Maus auf einen Bildpunkt wird dieser Punkt als Mitte des Ausschnitts gewählt und mit einem fixierten Zoomfaktor wird darum ein neues, verkleinertes Intervall berechnet und dargestellt. Ein Linksklick führt dieselbe Methode aus, allerdings mit dem inversen Zoomfaktor, was effektiv einem Herauszoomen entspricht.

Letzten Endes wählen wir am Ende in der finalen Version doch eine kontinuierliche Zoomfunktion mit anpassbarem Faktor, weil uns dies effektvoller erscheint.

Herausforderungen

- Ab jetzt benötigen wir eine präzise Übereinstimmung zwischen Pixelkoordinaten und komplexen Zahlen, eine händische Anpassung des Intervalls ist keine Lösung mehr, weil ein Mausklick die exakte Mitte des neuen, verkleinerten Intervalls liefern muss. Wir lösten dies schließlich mit der Java-Methode `insets()`, welche dynamisch die Dicke der Ränder des Frames zurückgibt und eine pixelgenaue Übereinstimmung zwischen Bild und Intervall erlaubt.
- Durch den Zoom wird deutlich, dass eine Anpassung der maximalen Iterationstiefe nötig ist. Da eine Erhöhung davon aber neben der Rechenzeit auch die Farbbildung und damit die visuelle Darstellung beeinflusst, fanden wir keine Möglichkeit, sie dynamisch an die Zoomstufe anzupassen. Wir haben uns schließlich dazu entschieden, eine Option für die maximale Iterationstiefe auch in die Benutzeroberfläche aufzunehmen.

3.4 Darstellung der Folgenglieder

Als weiteren Schritt visualisieren wir die ersten 1000 Folgenglieder zu einer Zahl c aus dem Intervall, das die Mandelbrot-Menge beinhaltet. Diese Zahl c wird jeweils per Mausklick vom Nutzer ausgewählt, beziehungsweise ist später einfach die Stelle, an der der Mauszeiger ist. Anhand einer Farbgebung zeigt sich nun, wie sich die Folgenglieder

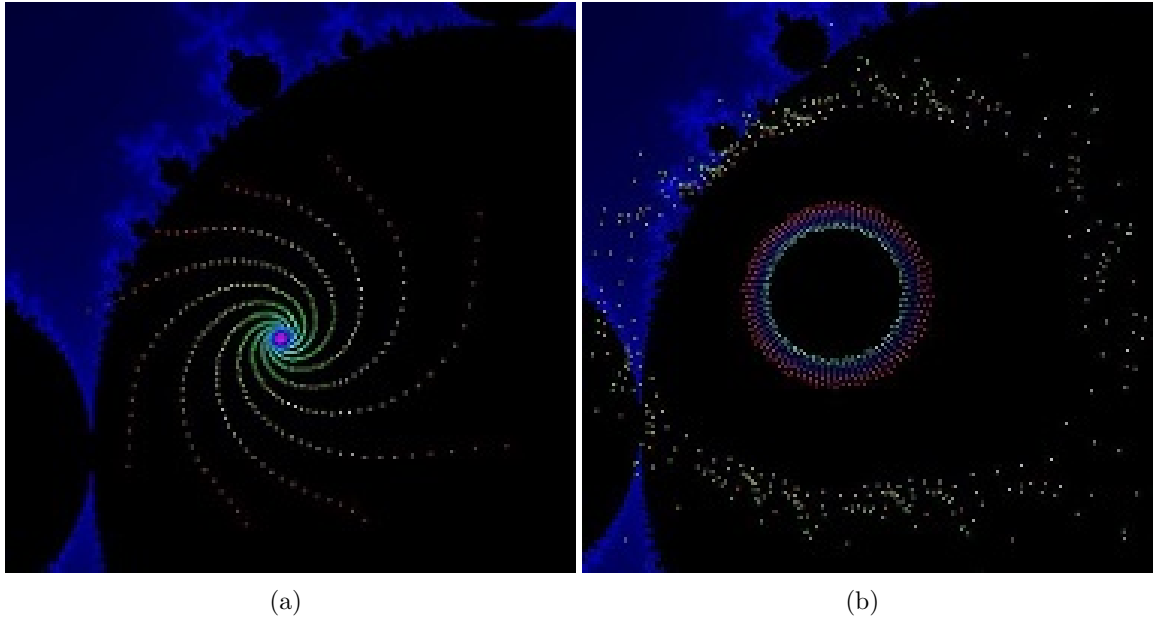


Abbildung 3.4: Zu sehen sind vergrößerte Ausschnitte der Mandelbrot-Menge, mit jeweils den ersten 1000 Folgengliedern für einen Punkt. Links sieht man die Bahn des Punktes $c = -0.6542 + 0.33322i$, der innerhalb des Hauptkörpers in der Kardioide liegt. Die Folgenglieder streben auf einen Grenzwert zu. Rechts ist die Bahn des Punktes $c = -0.6316 + 0.37961i$ dargestellt, der ebenfalls innerhalb der Kardioide liegt. Die Bahnpunkte streuen zunächst um ein Zentrum und scheinen sich dann auf einen Zyklus einzupendeln.

verhalten. Dabei werden die ersten Iterationswerte rot gefärbt, die folgenden orange, gelb, grün, türkis und blau, die letzten violett und purpur. In der finalen Version kann auch diese Farbgebung vom Nutzer angepasst werden.

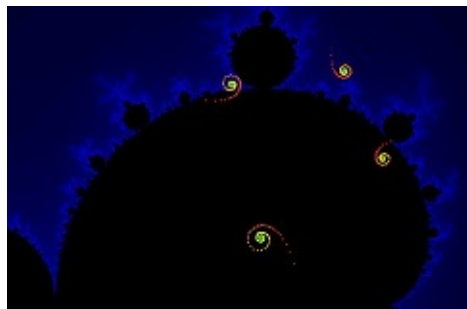


Abbildung 3.5: Der Punkt $c = 0.25687 + 0.49559i$ liegt in der Knospe mit der Nummer 4 (siehe Bild 2.2). Man kann beobachten, dass die Bahnpunkte als vier Spiralen angeordnet sind, was einem Grenzyklus der Länge vier entspricht.

4 Finale Version: Benutzeroberfläche und Optimierung

4.1 Allgemeine Struktur

Zur Visualisierung sollte ein interaktives Java-Programm entwickelt werden, das Aspekte der Mandelbrot-Menge visualisiert, aber auch vom Quelltext im Wesentlichen einfach verständlich ist. Unser Programm teilt sich daher in drei Teile:

1. Das Paket **mandelbrot** enthält unsere Hauptklasse *Mandelbrot* (enthält die *main* Methode), sowie die komplette Geschäftslogik, die den Kern unserer Software bildet inklusive der Optimierung des Renderings.
2. In **mandelbrot.view** werden dann die Views der Anwendung *RenderingView*, *RenderingCacheView* und *MetaView* definiert, sowie die Klasse *MandelbrotWindow*, in der alle Teile zusammengefügt werden.
3. Das Paket **util** schließlich enthält die zusätzlich benötigten allgemeinen Definitionen eines Vektors beziehungsweise einer komplexen Zahl (Klasse *Vector*, basierend auf der in 3.1 erörterten Klasse *Complex*) und eines Rechtecks daraus (Klasse *VectorRectangle*), sowie einer Formatierungsmethode für lange Gleitkommazahlen (*Util.format*) und einer Projektion zwischen der kanonischen zweidimensionalen Vektorebene bzw. komplexen Zahlenebene und der Bildschirmenebene in Java-Koordinaten (Klasse *ScreenProjection*). Außerdem ist hier noch eine Hilfsklasse *AnimatedJComponent* definiert, mit der automatisch aktualisierende und animierte, doppelt gepufferte JComponents implementiert werden.

Verwendet wird ausschließlich pures Java ohne zusätzliche Bibliotheken, um die Verständlichkeit des Programmtextes zu verbessern. Außerdem kommen wir mit dem Sprachumfang von Java JDK 5 aus, sodass das Programm mit alten Java-Installationen wie in dem zur Vorführung vorgesehenen Computerraum funktioniert.

Geachtet wurde auf eine Model View Controller artige Implementierung (MVC-Pattern), wobei wir aus Zeit- und Optimierungsgründen Abstriche beim Geheimnisprinzip gemacht haben und eine passive Model-Struktur implementiert haben, die nicht mitbekommt, wenn Daten geändert werden, sodass zum Beispiel im Model sämtliche Variablen öffentlich zugreifbar sind und nicht den Umweg über setter- und getter-Methoden benötigen.

Auch die Views werden nicht ereignis-, sondern frameratenbasiert aktualisiert. Da bei kontinuierlichen Operationen, wie dem Zoomen, in der Regel sogar mehr ereignisbasierte Aktualisierungen pro Zeit auftreten würden als mit dem Auge wahrgenommen

werden können, sparen wir somit Aktualisierungsvorgänge. Es kann so außerdem auf sämtliche Änderungs-Benachrichtigungs-Infrastrukturen verzichtet werden, was das Programm leichtgewichtiger macht. Die Views zeichnen das unterliegende Model mit einer konstanten Framerate.

Da das Paket **util** nur ein paar selbsterklärende (mathematische) Grunddefinitionen enthält, die nicht einmal miteinander verstrickt sind, konzentrieren wir uns jetzt ab hier auf die beiden Pakete **mandelbrot** und **mandelbrot.view**.

4.2 Basisdefinitionen der Folge und Farbe

Zentral in unserem Programm ist natürlich die Definition eines Werkzeugs zur Berechnung von Mandelbrot-Folgen, nämlich der Klasse **ProgressionCalculator**. Dies geschieht anhand folgender Klasse, die eine Methode *void begin(Vector)* zur Initialisierung der Berechnung einer neuen Folge mit einem Startwert bereitstellt, sowie eine Methode *void calculate()*, die das jeweils nächste Folgenglied berechnet. *int calculate(int)* berechnet mehrere Folgeglieder, bis ein Konvergenzverhalten erkennbar ist und liefert die Anzahl der Schritte bis zu diesem Folgeglied zurück, beziehungsweise 0, falls kein Konvergenzverhalten erkannt wurde.

ProgressionCalculator.java

```

1  import util.Vector;
2
3  /**
4   * Definiert das Werkzeug zur Berechnung von Mandelbrot-Folgen.
5   */
6  public class ProgressionCalculator {
7      public Vector characteristicValue;
8      public Vector currentValue;
9
10     /**
11      * Beginnt eine neue Berechnung.
12      *
13      * @param characteristicValue charakteristischer Wert, der für die
14      * Berechnung der Folge verwendet werden soll
15      */
16     public void begin(Vector characteristicValue) {
17         this.characteristicValue = characteristicValue;
18         currentValue = new Vector(0, 0);
19     }
20
21     /**
22      * Berechnet das nächste Glied der Folge.
23      */
24     public void calculate() {
25         currentValue = currentValue
26             .complexMultiply(characteristicValue)
27             .plus(characteristicValue);
28     }
29
30     /**
31      * Berechnet Folgeglieder, bis ein Konvergenzverhalten erkennbar ist.

```

```

32      *
33      * @param iterations Anzahl der maximal zu berechnenden Folgeglieder
34      * @return Anzahl der berechneten Folgeglieder, bis Konvergenzverhalten
35      * erkennbar wurde oder 0 sofern bis zur maximalen Anzahl der zu
36      * berechnenden Folgeglieder kein Konvergenzverhalten erkannt wurde.
37      */
38      public int calculate(int iterations) {
39          for (int i = 0; i < iterations; ++i) {
40              calculate();
41              if (currentValue.getLength() > 2) return i + 1;
42          }
43
44          return 0;
45      }
46  }

```

Letztlich dient das gesamte restliche Programm ausschließlich der Darstellung der hier eben definierten Berechnung.

In der Klasse **ColorProjection** werden zudem Abbildungen natürlicher Zahlen in den Farbraum definiert, die zur Darstellung einer dritten Dimension in Form von Farbe genutzt werden. Jede Instanz der Klasse repräsentiert eine spezielle solche Abbildung, sodass zu verwendende Farbprojektionen zur Laufzeit ausgetauscht werden können. (Strategy-Pattern)

4.3 Rendering

Für die Berechnung ebenfalls zentral ist die Definition der Klasse **Rendering**. Ein Rendering ist ein Pixelbild fester Größe, das ein festes Intervall der komplexen Zahlenebene darstellen soll. Diese Objekte beginnen als ein schwarzes Bild und werden bei ihrer Ausführung ganz einfach Pixel für Pixel von links nach rechts von oben nach unten berechnet und dabei entsprechend eingefärbt.

Rendering.java

```

1  package mandelbrot;
2
3  import java.awt.image.BufferedImage;
4  import util.ScreenProjection;
5  import util.VectorRectangle;
6
7  /**
8   * Beschreibt ein Pixelbild fester Größe, das eine unter Umständen noch nicht
9   * fertig berechnete Darstellung eines festen Intervalls der durch
10  * ProgressionCalculator definierten Funktion ist und für die eine fest
11  * gewählte Farbprojektion, sowie eine festgewählte Iterationszahl verwendet
12  * wird.
13  * Objekte der Klasse sind ausführbar, wobei bei der Ausführung die
14  * entsprechende Darstellung berechnet wird.
15  * Ein Objekt der Klasse sollte immer nur genau einmal ausgeführt werden. Dazu
16  * gibt es einen synchronisierten Reservierungsmechanismus, sodass im Falle der
17  * Verwendung eines Threadpools ein thread threadsafe die Reservierung eines
18  * Rendering-Objektes mit der Methode boolean tryToReserve() versuchen kann,

```

4 Finale Version: Benutzeroberfläche und Optimierung

```
19  * die nur bei ihrem ersten Aufruf überhaupt true liefern wird, was dann als
20  * Übertragung der exklusiven Ausführungszuständigkeit an den reservierenden
21  * Thread interpretiert werden soll.
22  */
23  public class Rendering extends BufferedImage implements Runnable {
24      public final VectorRectangle interval;
25      public final ColorProjection colorProjection;
26      public final int iterations;
27
28      public int time = 0; // Ausführungsdauer in Millisekunden
29
30      /**
31       * Erzeugt ein neues Rendering.
32       *
33       * @param width Breite des Renderings in Pixeln
34       * @param height Höhe des Renderings in Pixeln
35       * @param interval darzustellendes Interval
36       * @param colorProjection zu verwendende Farbprojektion
37       * @param iterations zur Berechnung zu verwendende Iterationstiefe
38       */
39      public Rendering(int width, int height, VectorRectangle interval,
40                      ColorProjection colorProjection, int iterations) {
41          super(width, height, BufferedImage.TYPE_INT_RGB);
42          this.interval = interval;
43          this.colorProjection = colorProjection;
44          this.iterations = iterations;
45      }
46
47      private boolean isReserved = false;
48
49      /**
50       * Versucht das Objekt zu reservieren und somit die exklusive
51       * Ausführungszuständigkeit zu erlangen.
52       * @return true sofern die Reservierung geglückt ist, ansonsten false
53       */
54      public synchronized boolean tryToReserve() {
55          if (isReserved) return false;
56          return isReserved = true;
57      }
58
59      @Override
60      public void run() {
61          long t0 = System.currentTimeMillis();
62
63          ScreenProjection sp = new ScreenProjection(interval,
64              getWidth(), getHeight());
65          ProgressionCalculator pc = new ProgressionCalculator();
66
67          for (int x = 0; x < getWidth(); ++x) {
68              for (int y = 0; y < getHeight(); ++y) {
69                  pc.begin(sp.projectFromScreen(x, y));
70                  setRGB(x, y, colorProjection
71                      .project(pc.calculate(iterations), iterations).getRGB());
72              }
73          }
74      }
```



```

74
75     time = Math.max(1, (int)(System.currentTimeMillis() - t0));
76 }
77 }

```

4.3.1 Optimierung des Rendering-Prozesses durch TiledRendering

Mit den genannten Klasse lässt sich prinzipiell bereits alles berechnen, was wir möchten, jedoch ist vor allem für den Vorgang des kontinuierlichen Hereinzoomens und Verschiebens eine Optimierung des Rendering-Prozesses erforderlich. Die Klasse **TiledRendering** definiert daher ein übergeordnetes Rendering, das aus einer Anzahl an Kacheln von den bereits oben definierten Rendering-Objekten besteht. Dies bietet Möglichkeiten zur Parallelisierung und zur gezielten Vorberechnung voraussichtlich später benötigter Bildinhalte. (Pipelining)

Das Kachelsystem ist so aufgebaut, dass im Hintergrund ständig ein Threadpool nach zu berechnenden Kacheln sucht und diese dann ggf. berechnet. Die Kacheln werden priorisiert berechnet. Die Priorisierung beginnt dabei bei der Kachel, die in der Mitte des Bildes ist und geht spiralförmig nach außen, sodass der Betrachter schnell einen Eindruck des Bildes bekommt und die optionalen Bereiche am Rand zuletzt berechnet werden.

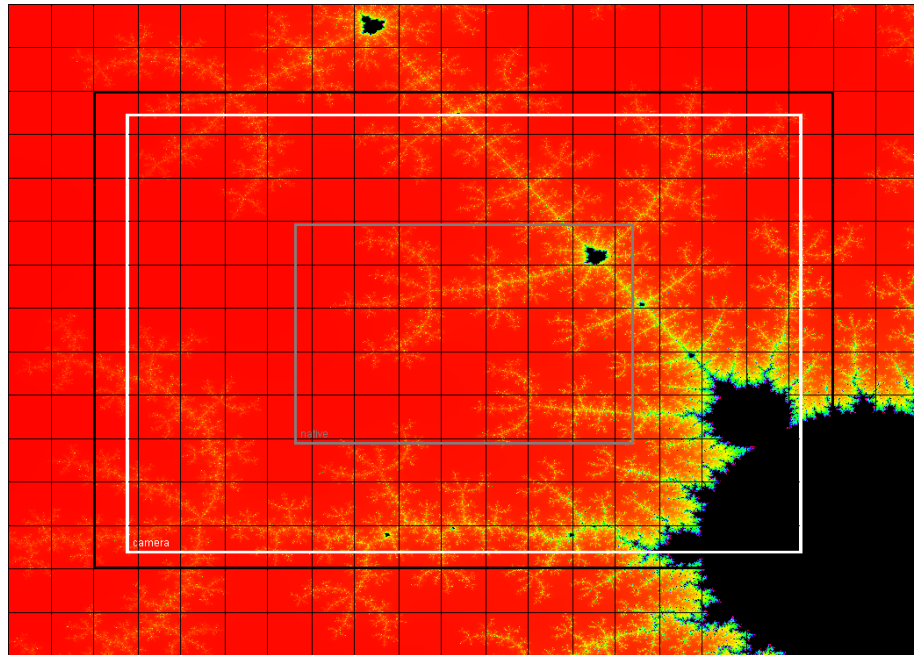


Abbildung 4.1: Kachelsystem

Ausgangspunkt ist eine eigentlich zu berechnende Pixel-Bildgröße, die ganz einfach der Fenstergröße unserer Anwendung entspricht. Berechnet wird dann ein Bild, das einen etwas größeren Bildausschnitt als den geforderten hat und doppelt so hoch wie benötigt aufgelöst ist. Dazu wird die geforderte Auflösung zunächst verdoppelt, berechnet, wie

viele Kacheln vordefinierter Größe dafür benötigt werden, dies aufgerundet und dann außerdem ein zusätzlicher Rahmen von einer Breite von zwei Kacheln rundherum gelegt.

In der Abbildung 4.1 ist der darzustellende Bildausschnitt mit dem weißen Rahmen und der Beschriftung *camera* markiert. Der äußere schwarze Rahmen markiert den zwei Kacheln breiten äußeren Rahmen. Der graue, innere Rahmen mit der Beschriftung *native* zeigt die Größe, die im berechneten Bild gerade die eigentlich geforderte Auflösung umfasst.

Das Kachelsystem beschleunigt nun das Verschieben des Bildausschnittes erheblich. Wird der Bildausschnitt verschoben, so verschiebt sich zunächst der weiße Rahmen. Dies geht in kleinem Umfang also sogar ganz ohne jede Neuberechnung. Jedoch wird nach jeder Verschiebung die Operation *recenter* aufgerufen. Diese prüft wie viele ganze Kacheln sich außerhalb des weißen Bereichs am Rand des Bildes befinden und versucht diese durch ein stabiles Verfahren anzugleichen. Dazu müssen ggf. alle Kacheln um ein Feld in eine Richtung verschoben werden, wodurch an einer Seite Kacheln gelöscht werden und an der gegenüberliegenden Seite Kacheln neu berechnet werden müssen.

Durch die Priorisierung der Kachelberechnung aus der Mitte heraus werden zuerst die am weitesten am sichtbaren Bildausschnitt liegenden Kacheln berechnet. Somit wird beim Verschieben immer genau an der richtigen Seite im Vorhinein gerendert, was die Zeit in Anspruch nehmen kann, die der Benutzer benötigt, um in den neuen Bildausschnitt zu scrollen. Dies geht somit kontinuierlich, ohne dass etwas vom Bildaufbau gemerkt wird.

Ebenso optimiert das Kachelsystem den Vorgang des Zoomens. Auf Grund der höheren, bereits berechneten Auflösung, kann in kleinem Umfang das bestehende Bild einfach hochskaliert werden. Dies geht genau solange, bis die geforderte Auflösung die tatsächlich berechnete übersteigt, denn ab diesem Punkt würde die Hochskalierung auffallen. Dieser Punkt ist erreicht, wenn der weiße *camera*-Rahmen kleiner als der graue *native*-Rahmen wird. Da diese Rahmen durch komplexe Koordinaten verwaltet werden kann das Programm dies einfach feststellen. Passiert dies, so muss ein neues Bild berechnet werden.

Damit auch dies komplett kontinuierlich funktioniert, wird dieses Bild durch ein Backbuffersystem ebenfalls bereits im Vorhinein im Hintergrund berechnet. Zur Berechnung steht dann genau die Zeit zur Verfügung, die der Benutzer braucht, um so weit herein-zuzoomen, bis der genannte Punkt erreicht ist.

4.4 Herausforderungen bei der Modellierung von Vektoren, komplexen Zahlen und ihren Rechtecken

Eine komplexe Zahl hat viel mit einem zweidimensionalen Vektor gemein. Es würde daher naheliegen, die Definition eines Vektors mittels Vererbung um ein entsprechendes komplexes Produkt und eine verschönerte *toString()*-Methode zu erweitern. Dies würde auch Vorteile für die Typsicherheit bringen, da dann genauer spezifiziert werden könnte, ob ein Vektor oder sogar eine komplexe Zahl an einer entsprechenden Stelle erwartet wird.

Da wir es – vor allem zur Repräsentation von Bildschirmausschnitten – oft mit durch

Vektoren oder komplexe Zahlen definierten Rechtecken zu tun haben ist ebenfalls eine Klasse wünschenswert, die solche Rechtecke repräsentiert (*VectorRectangle*) und hierfür entsprechende Operationen zur Verfügung stellt. Dies stellt uns aber vor das Problem, dass wir uns entscheiden müssten, in dieser Klasse entweder mit Vektoren oder mit komplexen Zahlen zu arbeiten. Beides würde das andere ausschließen und dann im verwendenden Code Konvertierungen erfordern.

Ein Ausweg wäre, zwei Rechteck-Klassen zu definieren, eine für Vektoren und eine für komplexe Zahlen. Das wäre allerdings unelegant, da dies erstens eine doppelte Definition zur Folge hätte und zweitens die Verwendung komplexer Zahlen auch als Vektoren zuließe, was zu Inkonsistenzen führen könnte.

Ein anderer Ausweg wäre die Klasse generisch mit einem Typparameter *T extends Vector* zu versehen. In diesem Fall ließen sich aber Rückgabewerte nicht mehr schön produzieren, da Objekte vom Typ generischer Typparameter in Java nicht direkt ohne Reflection erzeugt werden können. Der Einsatz von Reflection an einer so einfachen Stelle würde das Programm für unsere Zwecke erheblich unleserlicher machen und auch Rechenzeit kosten. Diese Situation ist ein Beispiel für eine Designeinschränkung durch Java, die schon durch ein relativ einfaches Problem erreicht wurde. Wir haben uns schließlich dafür entschieden, auch für jeden Vektor bereits die Operationen zu definieren, die eigentlich nur mit komplexen Zahlen sinnvoll sind.

5 Erörterung verschiedener Parameter

5.1 Variation von z_0

Bei der Mandelbrot-Menge ist der Startwert für jede Iteration $z_0 = 0 + 0i$. Uns fiel auf, dass dies selten verändert und erörtert wird und planten, dies als einen unserer anpassbaren Parameter zu wählen. Letzten Endes haben wir uns aber dagegen entschieden, denn die entstehenden Figuren wirken wie verzerrte Ausschnitte der Mandelbrot-Menge mit deutlich weniger Strukturreichtum. Insbesondere ist nicht mehr der komplette Rand fraktal. Dies führt daher, dass der Ursprung der einzige *kritische Punkt* für die Iterationsvorschrift ist, das heißt, die einzige Stelle, an der die Ableitung Null wird [5, Seite 40].

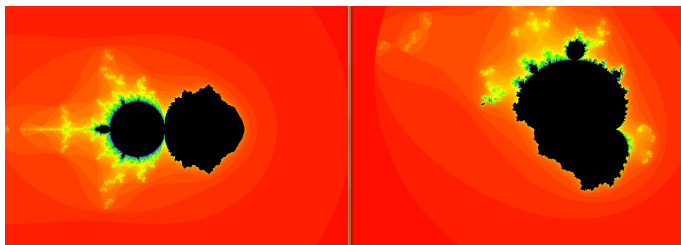


Abbildung 5.1: Links: $z_0 = -1 + 0i$, rechts: $z_0 = 0.5 + 0.5i$

5.2 Maximale Iterationen und Farbabbildung

Wie schon erwähnt, beeinflusst eine Änderung der maximalen Iterationsschritte `maxIterations` die Farbabbildung: Je höher `maxIterations` gewählt wird, desto stärker wird ein Großteil des Farbraums an die Mandelbrot-Menge gequetscht.

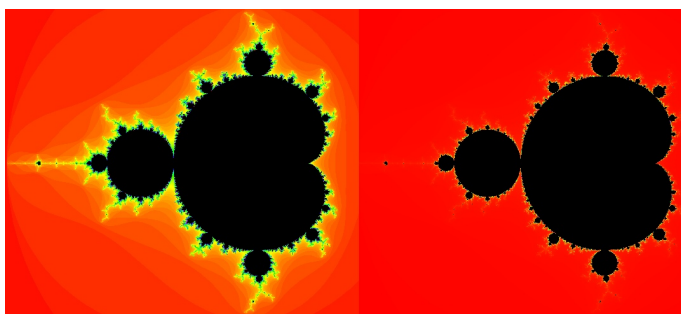


Abbildung 5.2: Lineare Abbildung in den HSB-Raum, links mit `maxIterations = 100`, rechts mit `maxIterations = 1000`

Folgerungen:

- Es gibt keine *ideale* Iterationstiefe: Eine, die für höhere Zoomstufen funktioniert, liefert ein undynamisches Bild in niedrigen Zoomstufen. Andererseits *muss* die Iterationstiefe erhöht werden, je tiefer hineingezoomt wird, weil sonst Details in der Darstellung fehlen.
- Die Iterationstiefe dynamisch an die Zoomstufe anzupassen bleibt problematisch, denn dies verändert die Darstellung des gesamten Farbspektrums - Man erhält das Gefühl, dass man nicht in die Darstellung hineinzoomt, sondern dass man ein anderes Bild sieht.

5.3 Verschiedene Farbprojektionen

Ebenso wenig wie es eine ideale Iterationstiefe gibt, existiert eine ideale Farbdarstellung. Im Folgenden werden verschiedene Abbildungen in den HSB-Raum erörtert.

5.3.1 Linear vs. quadratisch

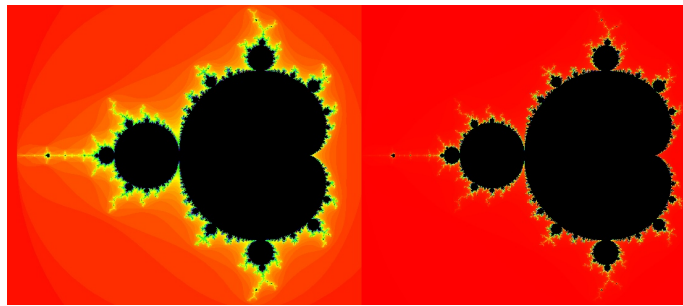


Abbildung 5.3: Gesamtansicht: links eine lineare Abbildung in den HSB-Raum, rechts eine quadratische Abbildung

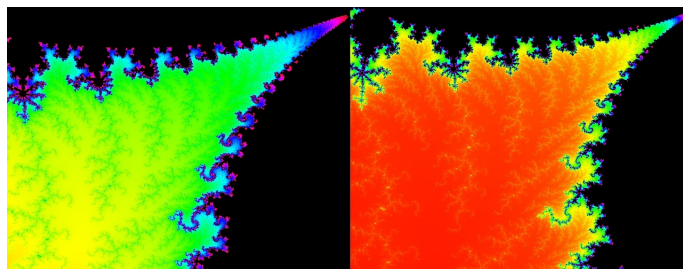


Abbildung 5.4: Dieselben Projektionen mit Zoom auf die gleiche Stelle: Bei der linearen Abbildung sieht man kein Rot mehr, bei der Quadratischen ist noch das gesamte Spektrum sichtbar.

Die lineare Projektion liefert eine ansprechende Ansicht bei niedrigen Zoomstufen, bei Höheren verliert sie jedoch an Dynamik - Die gesamte nähere Umgebung der Mandelbrot-Menge enthält nur noch Blau- und Violett-Töne.

Andererseits liefert eine quadratische Projektion, die das Spektrum in Richtung der höheren Iterationsschritte verschiebt, eine schöne Dynamik im Zoom aber nicht in der Gesamtansicht.

5.3.2 Linear mit Koeffizienten

Man kann diverse verschiedene Effekte erzielen, wenn man die lineare Abbildung in den HSB-Raum mit einem Koeffizienten n verändert. Während dadurch größere Strukturen sichtbar werden, beinhaltet es auch, dass Pixel nahe an der Mandelbrot-Menge eine scheinbar zufällige Farbe erhalten und somit dort keine Struktur deutlich wird. Generell erhält man statt einer einzelnen Projektion des Farbspektrums eine zyklische, n -fache Wiederholung der Farben, was am deutlichsten für $5 \leq n \leq 15$ wird.

Wählt man stattdessen ein großes n , das ein Teiler von `maxIterations` ist, zum Beispiel $n = \text{maxIterations}/2$, erhält man `maxIterations/n`-viele, alternierende Farben. Ein sehr interessantes Ergebnis erhält man, wenn man n etwas größer als `maxIterations/2` wählt, zum Beispiel für `maxIterations` = 1000 sei $n = 510$. Dies erzeugt einen alternierenden Gradienten: Zwei Farben die sich abwechseln und die langsam in der Tiefe des Bildes ihre Farbe verändern.

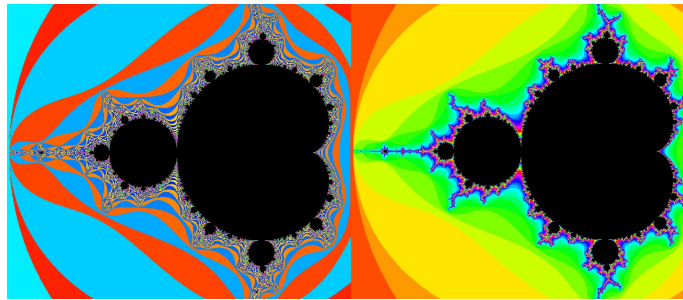


Abbildung 5.5: Gesamtansicht: links mit alternierendem Gradienten, rechts mit dem Koeffizientenwert $n = 15$

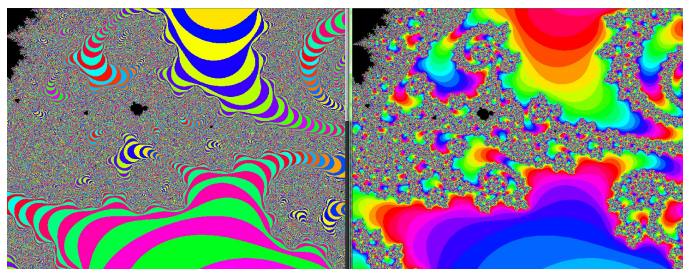


Abbildung 5.6: Dieselben Projektionen mit Zoom auf die gleiche Stelle: Links sind die alternierenden Farben deutlich zu sehen, rechts die zyklischen Regenbögen.

6 Schlussbemerkungen

6.1 Ausblick

6.1.1 Mögliche Ergänzungen

Es gibt einige Möglichkeiten, die Funktionalität unseres Programms zu erhöhen. Wir haben zum Beispiel eine systematische Variation des Startwerts nicht weiterverfolgt. Ebenso gibt es zahlreiche weitere interessante Farbprojektionen sowie ganz andere Färb-Algorithmen, zum Beispiel mittels des Newton-Verfahrens.

Ein zentraler Aspekt, den wir nicht bearbeitet haben und der ebenfalls einen hohen visuellen Reiz bietet, sind die zu jedem Punkt $c \in I$ gehörigen Julia-Mengen, wozu ein zweites Display-Fenster notwendig wäre [6, Seite 415 ff.].

Außerdem gibt es auch diverse andere Iterationsvorschriften, um Mengen zu erzeugen, die in ihrem Strukturreichtum der Mandelbrot-Menge gleichen, zum Beispiel $z_{n+1} = c \cdot \sin(z_n)$ mit dem Startwert $z_0 = \frac{\pi}{2}$ [5, Seite 41].

6.1.2 Weitere Optimierungen

Wir benutzen keine tiefergehenden mathematischen Ansätze, um die Berechnung des Bildes zu beschleunigen. Beispielsweise gibt es Programme, die Symmetrien in der Mandelbrot-Menge oder bei den Iterationsfolgen ausnutzen, um die Berechnungszeit zu verkürzen oder eine (vermutete, aber nicht bewiesene) Eigenschaft der Mandelbrot-Menge: Wenn sie tatsächlich einfach zusammenhängend ist, braucht man das Innere einer geschlossenen Kurve, die vollständig Teil der Menge liegt, gar nicht mehr zu berechnen.

Ein anderer Aspekt, den wir nicht ändern konnten, ist die maximale Zoomtiefe: Sobald das Intervall eine Breite von circa 10^{-13} hat, werden nebeneinanderliegende Pixel zu Blöcken zusammengefasst - die Maschinengenauigkeit für Double-Werte ist erreicht, aber eine Nutzung von Variablen mit höherer Präzision würde natürlich wieder die Rechenzeit erhöhen.

6.2 Persönliches Fazit

Ziel der vorliegenden Arbeit war es, aufzuzeigen, wie sich die Mandelbrot-Menge in Java implementieren lässt und wie sich Eigenschaften der Menge mit Java visualisieren lassen.

Zu diesem Zweck haben wir eine Benutzeroberfläche entwickelt, die es ermöglicht, die Mandelbrot-Menge und ihre Strukturvielfalt sowie das Iterationsverhalten komplexer Zahlen außerhalb und innerhalb der Mandelbrot-Menge zu visualisieren.

6 Schlussbemerkungen

Ein wesentlicher Aspekt dabei ist die Möglichkeit des Herein- und Herauszoomens. Sie erlaubt es, die Vielgestaltigkeit der Mandelbrot-Menge und die der Bahnen der iterierten Folgenglieder zu visualisieren. Eine anpassbare Iterationstiefe spielt dabei jedoch eine wesentliche Rolle, ebenso wie eine veränderbare Farbgebung, weil sonst immer Details verloren gehen. Eine ideale Kombination, die für alle Zoomtiefen befriedigende Ergebnisse liefert, scheint es nicht zu geben.

Es war uns deshalb besonders wichtig, dass einzelne Parameter wie unterschiedliche Farbprojektionen, maximale Iterationstiefe und Zoomgeschwindigkeit vom Nutzer flexibel eingestellt werden können, um jeweils unterschiedliche Eigenschaften der Mandelbrot-Menge sichtbar zu machen.

Insgesamt kann festgestellt werden, dass sich die Mandelbrot-Menge in Java, im Hinblick auf den zeitlichen Rahmen des Kompaktseminars, zufriedenstellend implementieren ließ. Das von uns entwickelte Programm bietet die Möglichkeit, diverse Eigenschaften der Mandelbrot-Menge und ihre Ästhetik zu visualisieren und könnte basierend auf den in Abschnitten 6.1.1 und 6.1.2 erörterten Ergänzungs- und Optimierungsideen weiterentwickelt werden.

Literaturverzeichnis

- [1] Melanie Mitchell. *Complexity. A guided tour* [Oxford University Press, 2009].
- [2] Die Mandelbrotmenge, Fakultät für Mathematik, KIT.
<http://www.math.kit.edu/iana2/~mandel/seite/schnupperkurs/media/vl6.pdf>
- [3] *Der quadratische Iterator $z \rightarrow z^2 + c$ und periodische Zyklen.*
<http://a137.de/Iterationen/Perioden-3.htm>
- [4] *Mandelbrot-Menge - immer wieder Apfelmännchen.*
<http://katgym.by.lo-net2.de/c.wolfseher/chaos/mandel3.htm>
- [5] *Flohmarktreport II* in Spektrum der Wissenschaft Dossier 2/2004, Spektrum der Wissenschaft Verlagsgesellschaft, 2004.
- [6] Hartmut Jürgens, Dietmar Saupe, Heinz-Otto Peitgen. *Fractals for the classroom. Part two. Complex Systems and Mandelbrot set*, Springer-Verlag, 1992.