

Mandelbrot & Julia-Mengen: Ein Fraktale-Visualisierungsprogramm

Tuan Vinh Nguyen

14. März 2024

1 Mandelbrot und Julia-Mengen

1.1 Definitionen

Das Bild der Mandelbrot-Menge soll jeder bekannt sein. Es taucht oft in der Mathematik auf, und manchmal sogar in der Popkultur. Was weniger bekannt

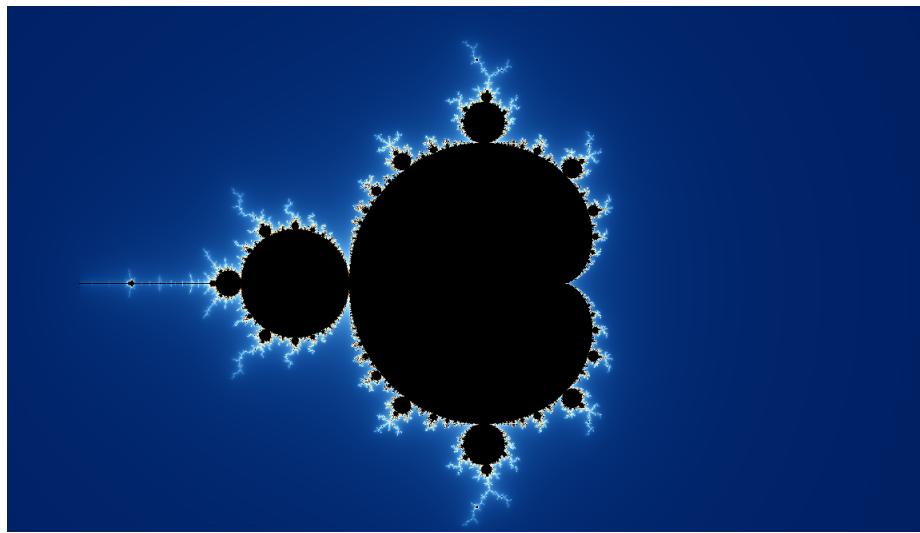


Abbildung 1: Die Mandelbrot Menge, generiert mit unserem Programm

ist, ist die Bedeutung des obigen Diagramms:

- Jeder Pixel (x, y) wird als eine komplexe Zahl $c = x + i.y$ interpretiert.
- Die Funktion $f(z) = z^2 + c$ wird auf den Startwert $z = 0$ angewendet, sodass eine Folge $z_0, z_1, \dots, z_n = 0, f(0), f(f(0)), \dots$ entsteht.

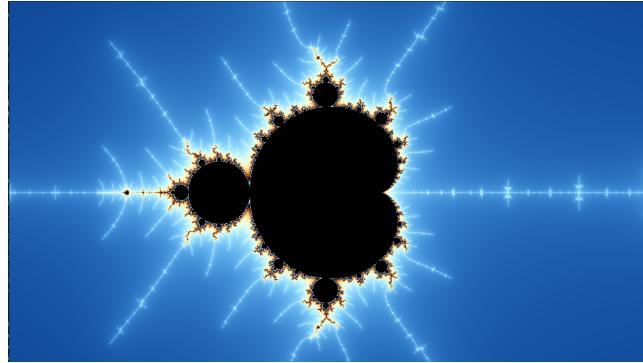


Abbildung 2: Ein Mini-Mandelbrot innerhalb des Mandelbrots

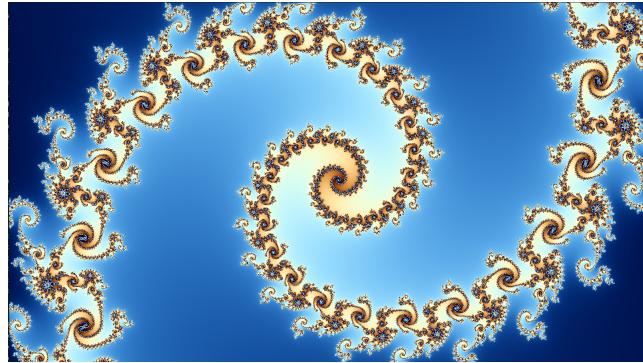


Abbildung 3: Unendliche Spiralen im Mandelbrot

- Man betrachtet dann die Beträge $|z_n|$ von Zahlen in der Folge. Die können bis ins Unendliche wachsen, z.B mit $c = 2$, die Folge ist: 0, 2, 6, 38, 1446, 2090918, ... Im Gegenteil können sie für bestimmte Werte von c immer unter einer Grenze bleiben. Man sagt dann, dass c in der Mandelbrot-Menge liegt, und die entsprechenden Pixel werden schwarz gefärbt. [7]

Obwohl die Funktion f an sich nicht so kompliziert ist, gibt es keine allgemeinen Formeln, die bestimmen können, ob eine Zahl in der Mandelbrot ist. Sehr kleine Änderungen bei dem Wert von c führen oft zu chaotischen Änderungen bei z_n , die nicht vorhersagen lassen. Das daraus entstehende Fraktal enthält deswegen sehr komplexe Muster, und manchmal auch sich selbst (siehe: Abb. 2, 3, 4)

Analog zu der Mandelbrot-Menge wird auch bei Julia-Mengen f_c iterativ auf z angewendet. Diesmal legt man c fest und ändert den Startwert z . Punkte, bei der kleinen Änderungen zu chaotischen Entwicklungen führen, gehören zur Menge. Grafisch liegen dann Punkte der Julia-Menge an den Grenzen (z.B. in Abb. 5: zwischen den schwarzen und blauen Bereichen)

Allgemein lassen Julia-Mengen sich durch eine Funktion $f(z) : C \rightarrow C$

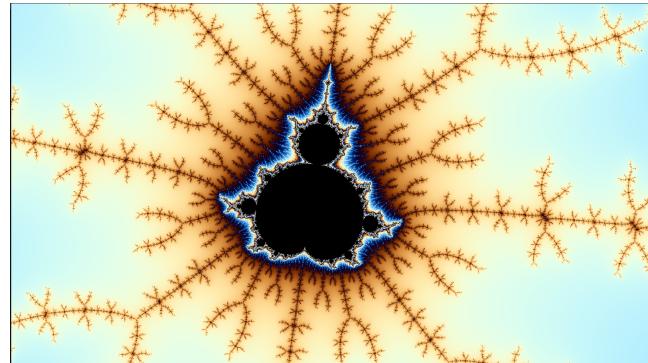


Abbildung 4: Ein Blatt mit Zweigen im Mandelbrot

beschreiben, wobei die Funktion nicht unbedingt der Form $f(z) = z^2 + c$ haben muss.

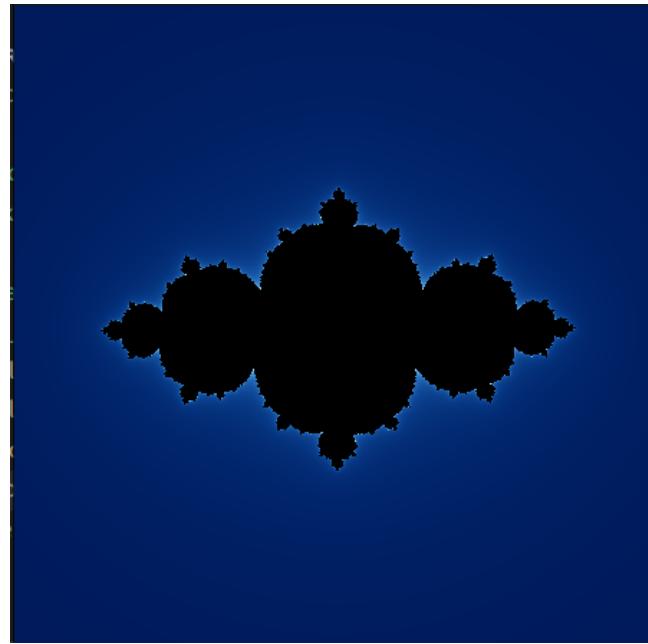


Abbildung 5: Die Julia Menge mit $f(z) = z^2 + c$, wobei $c \approx (-0.7022, 0.0089)$. Die Punkte der Menge sind an der Grenze zwischen Schwarz und Blau.

1.2 Ziele des Projekts

Um diese Mengen bzw. Fraktale zu untersuchen, ist ein Visualisierungsprogramm, bei dem man heran- und herauszoomen kann, meiner Ansicht nach am bestens geeignet. Der Zoomer soll auch in Echtzeit funktionieren, d.h: es muss mit ≈ 30 FPS laufen können. Obwohl Python nicht gerade als eine Performanz-Programmiersprache bekannt ist, können wir uns mithilfe einiger Optimierungen und des Xaos-Algorithmus diesem Ziel nähern. In nächster Sektion werden diese Algorithmen sowie deren Implementierungen im Programm genau erläutert.

2 Implementierung

2.1 Allgemeines Verfahren

Im Grunde genommen brauchen wir 3 Schritte, um das Mandelbrot zu rendern:

- Pixels auf komplexe Zahlen abbilden (trivial)
- Iterationenanzahl `iter` berechnen
- Farbe aus `iter` berechnen

Der hier gegebene Code ist eine vereinfachte Version und kann vom tatsächlichen Programm abweichen. Es gibt auch einen Jupyter-Notebook, in dem Sie den Code selber ausführen können.

2.2 Iterationenanzahl berechnen

Es ist bekannt, dass keine komplexen Zahlen $z = x + yi$ mit Betrag $|z| = x^2 + y^2 \geq 2$ in der Mandelbrot-Menge liegen. D.h., wir müssen nur $z = z^2 + c$ anwenden, bis $|z|$ unsere Schranke überschreitet. Wenn nach `max_iter` Iterationen der Betrag $|z|$ noch nicht explodiert, sagen wir, dass c in der Mandelbrot-Menge ist.

Je größer `max_iter` ist, desto detaillierter wird das Fraktal, aber dafür müssen auch mehr gerechnet werden.

```
1 def find_iter(cx, cy):
2     zx, zy = 0.0, 0.0
3     iter = 0
4     while iter < max_iter and zx**2 + zy**2 < 4:
5         iter += 1
6         zx_old = zx
7         zx = zx**2 - zy**2 + cx
8         zy = 2*zx_old*zy + cy
9     return iter
```

Listing 1: Eine Schleife zur Berechnung der Iterationenanzahl

- Gemäß dem Formel ist der Realteil $zx = zx^2 - zy^2 + cx$, $zy = 2*zx*zy + cy$, wobei z mit Wert 0 startet und c unserer Parameter ist.

- Nach jeder Schleife wir geprüft, oft $\sqrt{zx^2 + zy^2} < 2$ bzw. $zx^2 + zy^2 < 4$

Färben wir nun die Punkte c in Graustufen, je nach wie schnell $|z|$ über den Fluchtradius gehen (`iter`), haben wir ein Bild vom Fraktal.

```

1 def render_numpy(width, height, max_iter):
2     image = np.full((width, height, 3), [0,0,0], dtype=np.uint32)
3     xcoords = np.linspace(-2, 2, width, dtype=np.float64)
4     ycoords = np.linspace(-1, 1, height, dtype=np.float64)
5     for x in range(width):
6         for y in range(height):
7             cx = xcoords[x]
8             cy = ycoords[y]
9             iter = find_iter(cx,cy,max_iter)
10            col = 0 if iter == max_iter else int(iter / max_iter *
11                255)
12            image[x,y] = [col,col,col]
13
14    return image
15
16 image = render_numpy(400,200, max_iter=100)
17 plt.imshow(np.swapaxes(image, 0, 1))
18 plt.show()

```

Listing 2: Ergebnis in Abb. 6

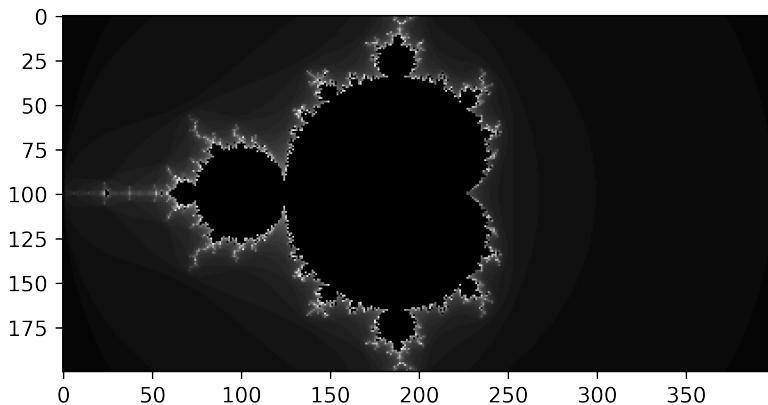


Abbildung 6: Ein Bild der Mandelbrot-Menge mit Auflösung 400x200, Laufzeit: 1.5 Sekunden

In weniger als 30 Zeilen haben wir die Mandelbrot-Menge visualisiert! Aber ein großes Problem ist zu lösen: Geschwindigkeit. Für nur ein 800x400 Auflösung

Bild brauchte mein Laptop 1.5 Sekunde. Das Ziel ist aber, mit 1920x1080 Auflösung, in $\sim 1/30$ einer Sekunde dies zu berechnen. Also **Parallelisierung** ist gebraucht.

Natürlich können wir die obige Funktion vektorisieren, sodass es auf den ganzen Array arbeitet, anstatt eine Schleife auf alle Elementen, aber ich möchte den Algorithmus minimal modifizieren und trotzdem dessen Performanz steigen. Mithilfe von **taichi** wurde der obige Code um fast 80-Mal beschleunigt. Demnächst muss aber erklärt werden, wie Taichi funktioniert.

2.2.1 Taichi: Eine schnelle Einführung

Taichi ist nicht nur ein Modul, sondern eine parallele Programmiersprache, die zusammen mit Python arbeiten kann. Unterschiedliche Backends können benutzt werden, z.B `ti.cpu` für die CPU oder `ti.gpu` für CUDA/Vulkan basierte Grafikkarten. Die Syntax von Taichi ist ähnlich wie die von Python, nun mit einigen kleinen Unterschieden und Konzepten::

- Taichi Kernel (bezeichnet mit `@ti.kernel`) sind Funktionen, die rechenintensive Aufgaben erledigen können, indem innere Schleifen automatisch parallelisiert werden. Eine Python-Funktion kann Taichi Kernel aufrufen aber nicht umgekehrt. Taichi Kernel können nur Taichi Funktionen aufrufen (`@ti.func`). Ein typischer Workflow wäre:

```

1  @ti.kernel
2  def taichi_kernel():
3      for i in range(1000000): # Parallelisiert
4          mach_etwas_schweres()
5
6  ti.init(arch=ti.gpu) # Benutzt das GPU
7  ... # Normaler Python-Code
8  taich_kernel()
9

```

- Analog zu `numpy` Arrays arbeitet Taichi Code oft mit Fields.

```

1  x = ti.Field(dtype=ti.float64, shape=(50,40))
2  # Ein 50x40 Feld mit 64-Bits-Gleitkommazahlen
3  x[0,30] = 4 # 0-te Reihe, 30-te Spalte
4  x[0,14] = x[0,13] + x[1,3]
5

```

- Globale Variablen sind bei Taichi-Kernen konstant, also mutable Variablen müssen explizit als Parameter übergeben werden. Außerdem ist der Typ von Variablen zu annotieren

```

1  x = 5
2  @ti.kernel
3  def print_wrong():
4      print(x)
5  x = 20
6  kernel_print(x) # Ausgabe: 5 (Fehler)
7

```

```

8     @ti.kernel
9     def print_right(x: int):
10        print(x) # Ausgabe:20 (Korrekt)
11

```

- Taichi Funktionen können keine Arrays/Fields erstellen

```

1 def python_func():
2     array = np.zeros(shape=(50,40), dtype=np.int32)
3     taichi_kernel(array)
4
5     @ti.kernel
6     taichi_kernel(ti.types.ndarray()):
7         for i in array:
8             array[i] = mach_etwas_schweres(i)
9

```

Für mehr Informationen: sehe Taichi-Docs [4]. Nun jetzt sind wir bereit, mithilfe von Taichi parallelen Code zu schreiben

```

1 ti.init(arch=ti.gpu) # Benutz GPU als Backend
2 def render_taichi(width, height, max_iter):
3     image_field = ti.Vector.field(n=3, shape=(width,height), dtype=ti
4         .uint32)
5     xcoords = np.linspace(-2,2,width)
6     ycoords = np.linspace(-1,1,height)
7     render_kernel(image_field, xcoords, ycoords, max_iter)
8     return image_field.to_numpy()
9
10    @ti.kernel
11    def render_kernel(image_field:ti.template(), xcoords: ti.types.
12        ndarray(), ycoords: ti.types.ndarray(), max_iter: int):
13        for x,y in image_field: # parallelisiert
14            cx = xcoords[x]
15            cy = ycoords[y]
16            iter = find_iter_taichi(cx,cy, max_iter)
17            col = 0 if iter==max_iter else int(iter / max_iter *255)
18            image_field[x,y] = [col,col,col]
19
20    # Der erster Lauf des Codes kann langsam sein
21    # Sequentielle Laeufe werden schneller
22    image = render_taichi(1600, 800, max_iter=100)
23    plt.imshow(np.swapaxes(image, 0, 1))
24    plt.show()

```

Die fundamentale Schleife ist fast nicht geändert, aber wir können ein Bild, das 16 Mal detaillierter ist, in ein Fünftel der Zeit berechnen. (Abb. 7)

2.3 Farbe

Natürlich sind Schwarz-Weiß Bilder ausreichend, aber die sind noch zu langweilig. Außerdem, wenn wir den `max_iter` zu hoch setzen, z.B 2000, bekommen wir ein nahezu komplett schwarzes Bild. Die meisten Punkten divergieren schon mit einer niedrigen Iterationenanzahl, also deren Helligkeit `iter/max_iter` geht gegen 0. (Abb. 8)

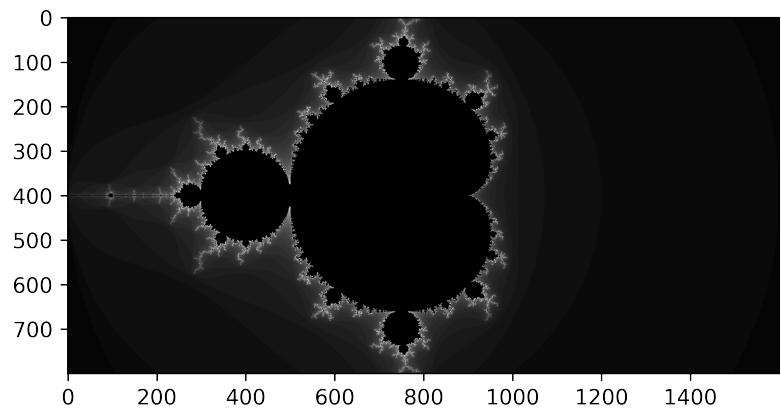


Abbildung 7: Ein Bild der Mandelbrot Menge mit Auflösung 1600x800, Laufzeit:
0.3 Sekunden mit Taichi

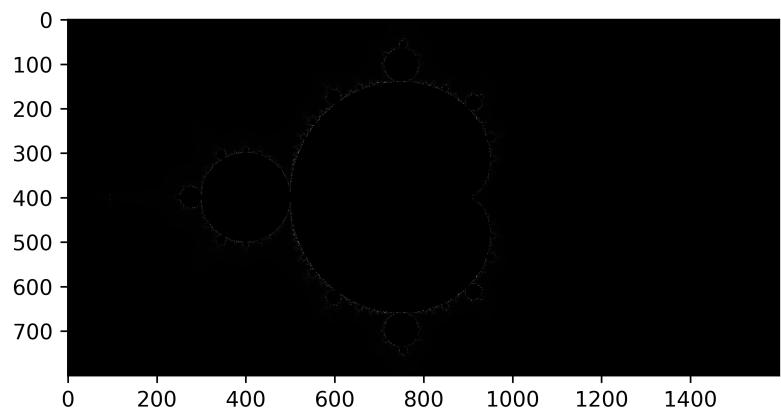


Abbildung 8: Mandelbrot mit `max_iter = 2000`

2.3.1 Palette automatisch generieren

Eine mögliche Implementation wäre die Benutzung einer Palettendatei bzw. Palettenarray. Um die Farbe von einem Punkt zu bekommen, muss man nur den entsprechenden Eintrag `palette_array[iter]` zugreifen, was sehr effizient ist. Allerdings kann diese Vorgehensweise eine nicht-ganzzahlige Iterationenanzahl nicht behandeln (*die bekommt man durch Normalisierung, mehr im nächsten Kapitel*). Stattdessen können wir mithilfe einer simplen Formel eine kontinuierliche Palette generieren:

$$color(t) = a + b \cdot \cos[2\pi(c \cdot t + d)]$$

a, b, c, d und Ergebnis $color(t)$ sind RGB-Vektoren mit Werten im Bereich (0,1). Geht t von 0 bis 1, schwingt der Cosinus-Teil c -Mal mit Startphase d , a und b sind der Bias- bzw Skalierungsfaktor [3].



Abbildung 9: Regenbogen-Palette mit $a = (0.5, 0.5, 0.5)$, $b = (0.5, 0.5, 0.5)$, $c = (1.0, 1.0, 1.0)$, $d = (0.00, 0.33, 0.67)$

Setzen wir `t=iter/max_iter` in den Formel, bekommen wir ein gefärbtes Bild der Mandelbrot in Abbildung 10

2.3.2 Kontinuierlich Farbverläufe

Zoomen wir ein bisschen in die obige Abbildung, fällt etwas auf: separate “Farbstreifen” entstehen im Bild (Abb. 11). Grund dafür ist, dass Punkte nur ganzzahlige Farbstufen (`iter`) haben. Deswegen gibt es eine abrupte Transition z.B: zwischen Regionen mit `iter = 1` und Regionen mit `iter = 2`. Dieses Problem lässt sich lösen, wenn dazwischen mit Punkten mit `iter = 1.1, 1.2, 1.3, ...` gefüllt wird. Dies schafft man durch **Normalisierung** der Iterationenanzahl mit der unteren Formel:

$$mu = n + 1 - \frac{\log(\log(|z_n|)) / \log(exponent))}{\log(bailout)}$$

, wobei mu der normalisierte Wert, n die Iterationenanzahl, z_n ist 0 nach n Iterationen, $exponent$ der Exponent der Mandelbrot-Funktion $z = z^n + c$, und $bailout$ unserer Fluchtradius ist. [6]

Geben wir anstatt n den Wert mu der Palettenfunktion weiter, bekommen wir ein geglättetes Bild des Mandelbrots (Abb. 12)

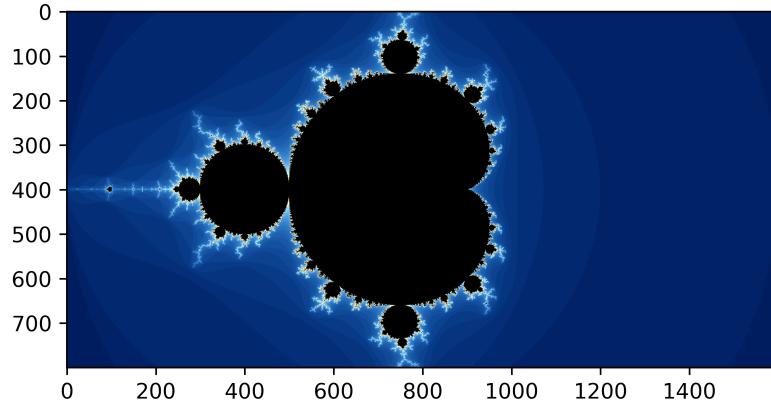


Abbildung 10: Mandelbrot mit Palette $a = (0.5, 0.5, 0.5)$, $b = (0.5, 0.5, 0.5)$, $c = (1.0, 1.0, 1.0)$, $d = (0.5, 0.60, 0.70)$

2.4 Xaos-Algorithmus

Benutzen wir die obigen Funktionen, können wir einen simplen Mandelbrot-Zoomer bilden mit folgender Schleife:

- Skalierungsfaktor `scale` vergrößern (um herauszuzoomen) oder verkleinern (um heranzuzoomen)
- `xcoords` (die Realteile / Spalten) und `ycoords` (die Imaginärteile / Reihen) aus `scale` berechnen
- Pixel (x, y) repräsentiert dann die Nummer $xcoords|x| + i \cdot ycoords|y| \rightarrow$ Iterationenanzahl \rightarrow Farbe
- Wiederholen

Wir merken aber, dass beim Zoomen der Bildschirm nicht so stark sich ändert. Viele Punkte bzw. Pixel vom letzten Frame tauchen im neuen Frame wieder auf. Betrachtet man die Spalten z.B. beim 2x Zoomen:

$$\frac{O_1 \quad O_2 \quad O_3 \quad O_4}{X_1 \quad X_2 \quad X_3 \quad X_4 \quad X_5 \quad X_6 \quad X_7 \quad X_8}$$

, wobei $O_1 \dots O_4$ alte Spalten und $X_1 \dots X_8$ neue Spalten sind. Schon lässt sich

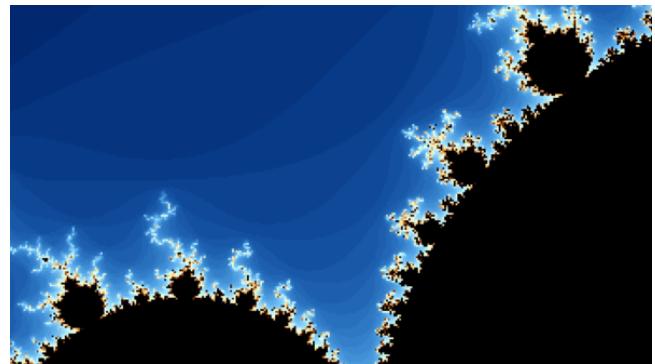


Abbildung 11: Die verschiedenen Farbstreifen im Mandelbrot

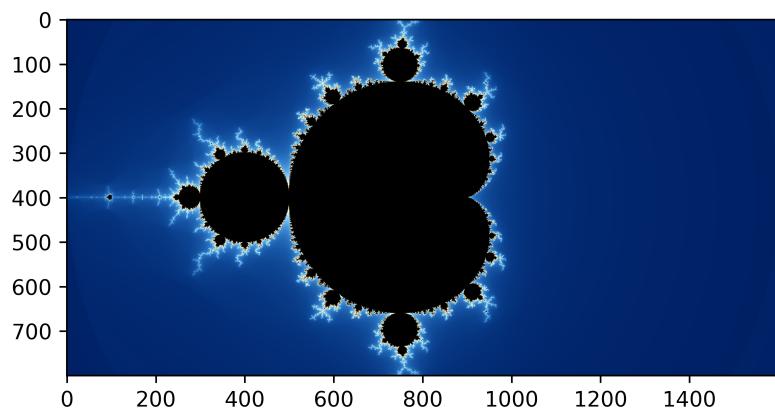


Abbildung 12: Ein Bild des Mandelbrots ohne Farbstreifen

sehen, dass wir die Spalten X_1, X_3, X_5, X_7 nicht erneut berechnen müssen. Stattdessen können dafür die Pixel von den entsprechenden Spalten O_1, O_2, O_3, O_4 "wiederverwendet" werden. Das ist auch die grundlegende Idee des Xaos-Algorithmus: wir versuchen, beim Rendern eines neuen Frames neue Spalten/Reihen durch alte Spalte/Reihen zu approximieren. Nur Punkte mit keinen guten Approximationen sind erneut zu berechnen [1].

Die Implementierung im Programm folgt der von MandelbrotSSE auf GitHub [5]:

1. Sucht für jede neue Zeile/Spalte eine alte Zeile/Spalte, die sie am bestens approximiert:

```

1  # WIDTH: Anzahl der Spalten
2
3  for i in range(WIDTH):
4      diff_best = infinity
5      id_best = -1
6      # Sucht nur bei seinen Nachbarn
7      for j in range(i - 40, i + 40):
8          diff = abs(xcoords[i] - old_xcoords[j])
9          if diff < diff_best:
10             id_best = j
11             diff_best = diff
12
13     pairs[i].id_orig = i
14     pairs[i].id_best = id_best
15     pairs[i].dist = diff_best
16

```

Mach das gleiche für Zeilen.

2. Füllt die Lookup-Tabellen aus:

```

1  # Sortiert die Paare danach, wie gut die Approximierung ist
2  # Die schlechtesten Appr. sind am Anfang.
3  pairs.sort_on('dist').reverse()
4
5  for i in range(WIDTH):
6      id_orig = pairs[i].id_orig
7      id_best = pairs[i].id_best
8
9      if i < REDRAW_PERCENT / 100 * WIDTH:
10         # Die schlechtesten Paare bzw. Spalten sind erneut zu
11         # berechnen
12         xlookup[id_orig] = -1
13     else:
14         # Sonst: approximiere neue Spalte id_orig mit alter
15         # Spalte id_best
16         xlookup[id_orig] = id_best
17         xcoords[id_orig] = old_xcoords[id_best]

```

Mach das gleiche für Spalten

3. Rendert den neuen Frame mithilfe der Lookup-Tabellen und des alten Frames:

```

1  for x in range(WIDTH):
2      for y in range(HEIGHT):
3          x_best = xlookup[x]
4          y_best = ylookup[y]
5          if x_best != -1 and y_best != -1:
6              # Eine Appr. fuer die Spalte und Reihe ist gefunden
7              # Benutzt den entsprechenden Pixel vom alten Frame
8              frame[x,y] = old_frame[x_best, y_best]
9          else:
10             real = xcoords[x]
11             imag = ycoords[y]
12             frame[x,y] = calculate_mandelbrot(real, imag)
13

```

Durch Anwendung des Xaos-Algorithmus wurde das Programm mindestens um 2-3 Mal beschleunigt. Was aber überraschend ist, dass obwohl er bei der CPU sehr gut funktioniert, scheiter der Algorithmus komplett bei der GPU. Es liegt wohl daran, dass die GPU sehr wenig RAM hat, um schnell auf den Array `old_frame` zuzugreifen.

2.5 Andere Optimierungen

2.5.1 Orbit-Erkennung

Wenn beim Iterieren z einen vorherigen Wert wieder trifft, heißt es, dass z in einer Schleife ist und daher immer begrenzt bleibt. Z.B:

$$z_0 \rightarrow z_1 \rightarrow z_2 \rightarrow z_1 \rightarrow z_2 \rightarrow z_1 \rightarrow z_2 \rightarrow \dots$$

Wenn wir dies erkennen können, können wir die Iterationsschleife abbrechen und die Rechenarbeit sparen . Bei meisten Punkten im Mandelbrot geschieht dies auch sehr früh (Benutzen Sie den Orbit-Modus des Programms, um dies zu sehen). Im Folgenden ist eine Implementierung davon, mit Credits an Lockless-Inc [2]:

```

1 def find_iter_with_orbit_detection(c, max_iter):
2     period = 8
3     iter = 0
4     while period != max_iter:
5         mem = z
6         period = period*2
7         if period > max_iter: period = max_iter
8
9         while iter < period:
10            iter = iter + 1
11            z = z**2 + c
12
13        if abs(z) > 4: return iter
14        if z == mem: return max_iter
15

```

Der obige Code prüft auf Perioden mit Länge 8, dann 16, 32, ...

2.5.2 Potenz durch Quadrieren

Da die eingebauten Taichi Funktionen für komplexe Zahlen viel zu langsam sind, muss leider Potenzierung z^n (n ist ganzzahlig & positiv) selber implementiert werden. Ein naiver Approach wäre, n Mal mit z mit sich selbst zu multiplizieren (n Schritte). Wir können aber stattdessen Potenzierung wie folgt rekursiv definieren:

- $z^n = (z^{n/2})^2$ für gerade n
- $z^n = (z^{n-1}).z$ für ungerade n

Beispielsweise wäre $z^{17} = (((z^2)^2)^2).z$, mit vier Mal Quadrieren und ein Mal Multiplizieren, anstatt 17 Multiplikationen. Allgemein wird die Anzahl der Schritte von n auf $\log n$ reduziert. Für große n bleibt dies allerdings noch zu langsam, aber es ist für dieses Projekt hinreichend, wobei maximal nur z^{10} berechnet wird.

3 Schwachstellen & Verbesserungen

Alles in allem bin ich zufrieden mit der Performanz des Programms. Dennoch gibt es auch einige Probleme, die wegen Mangel der Zeit oder technischer Fähigkeit nicht gelöst wurden:

3.1 Optik

- Beim Zoomen sieht man oft einen glitzernden Effekt. Dies ist leider ein Nebeneffekt des Renderns mit dem Xaos-Algorithmus.
- Bei bestimmten Regionen des Mandelbrots ist das Bildrauschen sehr stark ausgeprägt, da keine Antialiasing implementiert wurde.

3.2 Geschwindigkeit

- Trotz der Orbit-Erkennung verlangsamt sich / laggt das Programm erheblich beim Rendern von Punkten innerhalb der Mandelbrot-Menge.
- Statt taichi kann **pyopengl** benutzt werden, um direkt mit Shader bzw. der GPU zu arbeiten für erhöhte Leistung.

4 Fazit

Zusammenfassend lässt sich sagen, dass dieses Projekt eine gute Lektion in Sachen der Parallelisierung, Computergrafik und Python im Allgemeinen. Ohne großen Verlust der Codelesbarkeit kann Python ziemlich leistungsfähig sein. Zukünftig wird eventuell noch Bildnachbearbeitung implementiert und ein neues Modul gesucht, um die GUI zu verbessern.

Literatur

- [1] Jan Hubicka. *Developer's Guide: Algorithms*. URL: <https://github.com/xaos-project/XaoS/wiki/Developer's-Guide#algorithms>.
- [2] Lockless Inc. *The Mandelbrot Set*. URL: locklessinc.com/articles/mandelbrot/.
- [3] Inigo Quilez. *Palettes*. URL: <https://iquilezles.org/articles/palettes/>.
- [4] Taichi. URL: <https://docs.taichi-lang.org>.
- [5] ttsiodras. *MandelbrotSSE*. URL: <https://github.com/ttsiodras/MandelbrotSSE>.
- [6] Linas Vepstas. *Renormalizing the Mandelbrot Escape*. URL: <https://linas.org/art-gallery/escape/escape.html>.
- [7] Wikipedia. *Mandelbrot Set*. URL: https://en.wikipedia.org/wiki/Mandelbrot_set.