Aufgabe 5

Lösungsidee

Der Algorithmus wird hier von der Aufgabe ziemlich klar vorgegeben: ausgehend von bestimmten Koordinaten in einem kartesischen Koordinatensystem soll sich der Farbwert dieser Koordinate in x- und y-Richtung (jeweils positiv und negativ) so weit ausbreiten, bis auf den Bildrand oder einen anderen Farbwert getroffen wird.

Umsetzung

Als Programmiersprache habe ich Python und als Dateiformat für die Bilder Portable Graymap gewählt. Als veränderbare Parameter lässt mein Programm die Breite und Höhe des Bilds sowie die Anzahl und Ausbreitungsgeschwindigkeit der Keime zu, welche einfach als Kommandozeilenargumente in dieser Reihenfolge übergeben werden. Hierbei ist zu beachten, dass der Wert für die Geschwindigkeit die *niedrigste* und nicht die *höchste* erlaubte Geschwindigkeit angibt. Den Entstehungszeitpunkt großartig zu variieren habe ich für nicht sonderlich ertragreich befunden und deshalb rausgelassen. Daraufhin werden so viele Keime wie gewünscht mit "zufälligen" Werten für x und y, den Entstehungszeitpunkt sowie für die Ausbreitungsgeschwindigkeiten innerhalb der vorgegebenen Bereiche initialisiert. Dann werden die Keime so lange ausgebreitet, bis alle Pixel des Bilds von einem Keim erfasst wurden. Das ist dadurch sichergestellt, dass die für die Ausbreitungsgeschwindigkeit v eines Keims $v \in \mathbb{N}^+$ gilt und letzten Endes alle Pixel bedeckt werden müssen. Sobald der erste Keim entstanden ist, beginnt dieser mit jedem Schritt der Schleife, sich in die 4 Richtungen alle v Schritte einen Pixel auszubreiten. Daher ist die Ausbreitung auch mit höheren Werten langsamer, da nur bei jedem ganzzahligen Vielfachen von v ein Pixel hinzukommt und deren Abstände linear mit v zusammenhängen.

Beispiele

Das Namensschema der Bilddateien, die ich beigefügt habe lautet wie folgt:
`Kristall_<Breite>-<Höhe>-<Keimanzahl>-<Minimalgeschwindigkeit>_<Nummerierung>`

Um die Funktionsweise meines Programms zu demonstrieren habe ich die folgenden Beispiele gewählt:

Zuerst einmal ein Kristall mit relativ wenigen Keimen, aber relativ gleichmäßigem Wachstum (entweder 1 oder 2):

```
$ python aufgabe2.py 1920 1080 50 2
```

Bild 1 (sollte der Link nicht funktionieren, ist hier die Datei mit dem Namen `Kristall_1920-1080-50-2_1` verlinkt)

Bild 2 (gleiches Namensschema, nur mit einer 2 am Ende statt der 1)

Wenig überraschend kommen bei diesen Eingaben Kristalle mit relativ großen, gleichfarbigen Flächen als Ergebnis heraus.

Mit den nächsten Eingaben bin ich meiner Meinung nach ziemlich nah an die Vorgabe aus der Aufgabe herangekommen:

```
$ python aufgabe2.py 1920 1080 200 15
```

Bild 1

Bild 2

Zwar sind die Kristalle nicht ganz so homogen wie in der Vorgabe, allerdings finde ich die Verteilung und den Farbbereich ziemlich ähnlich.

Und dann noch eine Eingabe, die mehr Abwechslung verspricht:

```
$ python aufgabe2.py 1920 1080 1000 15
```

Bild 1

Bild 2

Wie zu erwarten war, gibt es hier deutlich mehr "Unordnung", die durch die vielen kleinen Farbsplitter kommt.

Quellcode

```
from math import floor, sin
from datetime import datetime
from random import randint
from sys import argv
from itertools import chain
from typing import TypedDict


class Pixel:
    color: int
    x: int
    y: int

    def __init__(self, values: PixelValues) -> None:
        self.color, self.x, self.y = values.values()


class Sprout(Pixel):
    draw_at: int
    up: int
    down: int
    right: int
    left: int

    def __init__(self, values: SproutValues) -> None:
        super().__init__({k: v for k, v in values.items() if k ==
"color" or k == "x" or k == "y"})
        self.color, self.x, self.y, self.draw_at, self.up, self.down,
self.right, self.left = values.values()


class Spread(Pixel):
    draw_at: int
    parent_sprout: Sprout

    def __init__(self, values: PixelValues, draw_at: int, parent_sprout:
Sprout) -> None:
        super().__init__(values)
        self.draw_at = draw_at
        self.parent_sprout = parent_sprout


class Picture:
    width: int
    height: int
    pixels: list[list[int | None]]
    sprouts: list[Sprout]
```

```
    filename: str
    active_edges: filter

    def simulate(self, update=False) -> None:
        future_sprouts = self.sprouts.copy()
        step = 0
        while not all(chain.from_iterable(self.pixels)):
            if update:
                self.draw()
                # erlaubt das Setzen eines Breakpoints des Debuggers, um
nach jedem Draw anzuhalten
                pass
            # sortiert bereits gezeichnete Pixel aus
            future_sprouts = [sprout for sprout in future_sprouts if
sprout.draw_at >= step]
            # filtert nach den Pixeln, die in diesem Schritt gezeichnet
werden
            self.active_edges = filter(lambda s: s.draw_at == step,
future_sprouts)
            for sprout in self.active_edges:
                # falls der betroffene Pixel bereits einen Wert hat,
wird dieser nicht überschrieben
                if self.pixels[sprout.y][sprout.x] is not None:
                    continue
                self.pixels[sprout.y][sprout.x] = sprout.color
                # bereitet die nächste Ausbreitungsstufe vor
                if sprout.y != 0 and self.pixels[sprout.y - 1][sprout.x]
is None:
                    if type(sprout) == Sprout:
                        up = Spread({"color": sprout.color, "x":
sprout.x, "y": sprout.y - 1}, step + sprout.up,
                                    parent_sprout=sprout)
                    else:
                        up = Spread({"color": sprout.color, "x":
sprout.x, "y": sprout.y - 1},
                                    step + sprout.parent_sprout.up,
parent_sprout=sprout.parent_sprout)
                    future_sprouts.append(up)
                if sprout.y < self.height - 1 and self.pixels[sprout.y +
1][sprout.x] is None:
                    if type(sprout) == Sprout:
                        down = Spread({"color": sprout.color, "x":
sprout.x, "y": sprout.y + 1}, step + sprout.down,
                                      parent_sprout=sprout)
                    else:
                        down = Spread({"color": sprout.color, "x":
sprout.x, "y": sprout.y + 1},
                                      step + sprout.parent_sprout.down,
parent_sprout=sprout.parent_sprout)
```

```
                    future_sprouts.append(down)
                if sprout.x < self.width - 1 and self.pixels[sprout.y]
[sprout.x + 1] is None:
                    if type(sprout) == Sprout:
                        right = Spread({"color": sprout.color, "x":
sprout.x + 1, "y": sprout.y}, step + sprout.right,
                                       parent_sprout=sprout)
                    else:
                        right = Spread({"color": sprout.color, "x":
sprout.x + 1, "y": sprout.y},
                                       step +
sprout.parent_sprout.right, parent_sprout=sprout.parent_sprout)
                    future_sprouts.append(right)
                if sprout.x != 0 and self.pixels[sprout.y][sprout.x - 1]
is None:
                    if type(sprout) == Sprout:
                        left = Spread({"color": sprout.color, "x":
sprout.x - 1, "y": sprout.y}, step + sprout.left,
                                      parent_sprout=sprout)
                    else:
                        left = Spread({"color": sprout.color, "x":
sprout.x - 1, "y": sprout.y},
                                      step + sprout.parent_sprout.left,
parent_sprout=sprout.parent_sprout)
                    future_sprouts.append(left)
            step += 1
        self.draw()

    def draw(self) -> None:
        with open(self.filename, "w") as file:
            file.write("P2 {} {} 65535\n".format(self.width,
self.height))
            for row in self.pixels:
                for pixel in row:
                    file.write(str(pixel if pixel is not None else 0) +
" ")
                file.write("\n")


def sprout_randomizer(width: int, height: int, amount: int,
min_velocity: int) -> list[Sprout]:
    values: list[SproutValues] = [{"color": 0,
                                   "x": randint(0, width - 1),
                                   "y": randint(0, height - 1),
                                   "draw_at": randint(0, 15),
                                   "up": randint(1, min_velocity),
                                   "down": randint(1, min_velocity),
                                   "right": randint(1, min_velocity),
                                   "left": randint(1, min_velocity)
```

```
                                    } for _ in range(amount)]
    for value in values:
        # berechnet einen Grauwert in Abhängigkeit der Richtung
        value["color"] = floor(sin(
            (value["up"] + value["down"] + value["right"] +
value["left"]) / (min_velocity * 4)) * 65535)
    return [Sprout(value) for value in values]
```