Aufgabe 2: Verzinkt

Inhaltsverzeichnis

Lösungsidee……1
Umsetzung……1 - 5
Beispiele……5 - 7
Quellcode……8 - 14

Lösungsidee

Die Simulation ist am einfachsten umzusetzen, wenn sie schrittweise geschieht. In einem Schritt wachsen alle Zink-Kristalle. Da in Pixeln simuliert wird, muss für jeden Zeitschritt entschieden werden, ob der Kristall in eine bestimmte Richtung sichtbar wächst oder nicht. Um also möglichst nah an die Realität heranzugelangen, sollte zuerst die bisherige Ausbreitung in diese Richtung berechnet werden.

Dafür wird die Wachstumsgeschwindigkeit (in Pixeln pro Schritt) mit der vergangenen Anzahl an Schritten multipliziert. Die Nachkommastellen dieses Wertes können in Pixeln nicht dargestellt werden. Daher wird gerundet. Wenn der gerundete Wert kleiner ist als der tatsächliche, so liegt die neue Position näher am bereits besetzten Pixel. In diesem Fall geschieht also kein Wachstum für den jeweiligen Schritt, ansonsten aber schon. Das gilt auch, wenn der gerundete Wert gleich dem tatsächlichen ist, solange die Wachstumsgeschwindigkeit ungleich 0 ist. Zur Vereinfachung darf keine Geschwindigkeit größer als 1 Pixel pro Schritt sein.

Wenn mehrere Kristalle gleichzeitig auf einen Pixel wollen, muss ein „Gewinner“ entschieden werden. Auch hier werden also die ungerundeten Wachstumswerte betrachtet. Der Kristall, der schon am weitesten in den jeweiligen Pixel vorgedrungen ist, gewinnt ihn für sich.

Natürlich sollte ein Kristall nur einen Pixel besetzen, wenn dieser Pixel frei ist. Das bedeutet, dass dieser existiert (also nicht außerhalb des Bildes liegt) und dass noch kein anderer Kristall ihn besetzt hat.

Sobald kein Kristall mehr weiter wachsen kann **und** keine verzögerten Kristalle mehr folgen, ist die Simulation abgeschlossen.

Umsetzung

Zur Umsetzung habe ich Python in der Version 3.10.7 verwendet. Der Quelltext muss mit einer Version **ab 3.7** kompiliert werden, um eine feste Ordnung von Dictionaries (Maps in Python) zu gewährleisten.

Achtung: Wenn in dieser Dokumentation von *Funktionen* die Rede ist, sind alleinstehende Funktionen gemeint, während *Methoden* klasseninterne Methoden beschreiben.

Ich habe folgende Module benutzt:

- ***numpy*** für die Erstellung und Verwaltung von Arrays / Matrizen
- ***random*** für das Erzeugen von Pseudo-Zufallszahlen und -Entscheidungen
- ***PIL*** (Python Imaging Library) für das Erstellen und Speichern von Bildern auf Grundlage von Arrays
- ***time*** zur Laufzeitmessung

Ich habe zunächst einige Standard-Bildauflösungen in einer statischen Klasse festgelegt.

Für die eigentliche Simulation der Kristalle habe ich eine separate Klasse *simulation* erstellt. Das hat den Vorteil, dass sich zur Laufzeit mehrere Simulationen erstellen und durchführen lassen und dass dieser Bereich dadurch klar abgekapselt ist. Auf ihren Aufbau wird später noch eingegangen.

Die Funktion „check_data“:

```
def check_data(img_size, start_pos, grayscales, speed_data, delays, background):
```

Diese Funktion überprüft übergebene Simulationsdaten auf ihre Gültigkeit. Z.B. werden zu lange Listen gekürzt, erwartete Ganzzahlen gerundet und andere Zahlen werden ggf. in ihren gültigen Bereich gezwungen. Wenn die Startpositionen nicht im Bereich des Bildes liegen, werden entsprechende *Exceptions* verursacht.

Zum Schluss werden die ggf. angepassten Daten ausgegeben.

Die Klasse „crystal“:

```
def __init__ (self, ID, start_pixel, grayscale, growth_speeds, delay):
```

Die Klasse für die Kristalle, die von der Simulationsklasse instanziiert und verwendet werden. Ihr Konstruktor übernimmt folgende Argumente:

- ID des Kristalls (Der Index des Kristalls in der Liste *crystals* in einem *simulation*-Objekt)
- Position des Keims / des ersten Pixels (= Startposition)
- Die aus der Ausrichtung resultierende Graustufe
- Wachstumsgeschwindigkeiten in alle Richtungen nach der Reihenfolge „Oben, rechts, unten, links“
- Eigene Startverzögerung bzw. der Simulationsschritt, in dem der Kristall beginnt zu wachsen

Die Klasse speichert Positionstupel in den Listen *pixels* und *growing_pixels*. Letztere beinhaltet Positionen von den Pixeln, die nach aktuellem Stand wachstumsfähig sind.

Die Methode real_growth_diff:

```
def real_growth_diff (self, d):
```

Gibt für eine bestimmte Richtung die Differenz aus dem gerundeten und dem tatsächlichen bisherigen Wachstum zurück, entnimmt die Daten aus den entsprechenden Dictionaries (Maps) der Klasse.

Wenn der Wert positiv ist, also aufgerundet wurde, dann gilt: Je niedriger der Wert ist, desto näher liegt der tatsächliche Wert am aufgerundeten, desto mehr ist also von der zweiten Hälfte des angrenzenden Pixels besetzt.

Die Methode update_growth:

```
def update_growth (self, step: int):
```

Aktualisiert die Wachstumsdaten des Kristalls, basierend auf der übergebenen Schrittzahl. In jede Richtung wird das Wachstum berechnet, indem die jeweilige Geschwindigkeit mit der vergangenen Schrittzahl **seit Entstehung des Kristalls** multipliziert wird. Von der übergebenen Zahl muss also die Verzögerung abgezogen werden.

Dieses und das gerundete Ergebnis werden in den entsprechenden Dictionaries (Maps) für die aktuelle Richtung gespeichert und die daraus berechnete reelle Wachstumsdifferenz wird ebenfalls in einem eigenen Dictionary gespeichert. Zum Schluss wird dieses Dictionary aufsteigend sortiert. Der Zweck dieser Sortierung wird im Abschnitt der *simulation*-Klasse unter „***pixel_requests***“ erklärt (gelb markiert).

Die Klasse „simulation“:

Diese Klasse ist der Kern des Programms. Sie simuliert, wie ihre Kristalle miteinander interagieren und wie das „Endprodukt“ aussieht. Im Folgenden die wichtigsten Methoden:

Der Konstruktor:

```
def __init__ (self, img_size:tuple, start_pos, grayscales, speed_data, delays,
random_placement_on_failure:bool = False, background = (0, 255, 0), name =
"Zink_Kristalle"):
```

Im Konstruktor dieser Klasse werden zuerst mittels der Funktion *check_data* die Daten überprüft. Startpositionen, Graustufenwerte und Geschwindigkeiten werden mit der *zip*-Funktion in eine Liste zusammengefasst, die aus Tupeln besteht, von denen jedes die Daten für einen Kristall beinhaltet.

Schließlich werden Kristall-Objekte mit den jeweiligen Daten und evtl. Verzögerungen initialisiert und in der Liste *crystals* referenziert. Diese Liste beinhaltet alle Kristalle der Simulation.

Zuletzt wird ein Dictionary *crystals_by_delay* erstellt, wenn es Delays gibt. Für jede Schrittzahl beinhaltet es, wenn vorhanden, die Kristalle, die zu diesem Zeitpunkt zu wachsen beginnen. Der aktuelle Schritt wird auf das kleinste angegebene Delay gesetzt. Zur Liste *growing_crystals* werden nun mit *place_start_pixel* die Kristalle hinzugefügt, die schon in diesem Schritt wachsen.

Wenn es keine Delays gibt, wird der Schritt auf 0 gesetzt und alle Kristalle werden in *growing_crystals* referenziert (Auch mit *place_start_pixel*, wird später erklärt).

Die Methode pixel_requests:

```
def pixel_requests(self, c: crystal):
```

Ermittelt für einen Simulations-Schritt die Positionen der neuen Pixel eines Kristalls. Wenn ein Pixel in einer bestimmten Richtung eine freie Position hat und in diese Richtung nicht mit der Geschwindigkeit 0 wächst, kann er aus den „toten“ (nicht mehr wachsenden) Pixeln ausgeschlossen werden.

Wenn dann auch sein gerundetes Wachstum größer oder gleich dem realen Wachstum ist, also nicht abgerundet wurde, wird die Differenz daraus unter der neuen Position zum Request-Dictionary hinzugefügt. Durch die Sortierung in *update_growth* steht garantiert der

Wert vom neuen Pixel mit der **niedrigsten Differenz** (Also der **höchsten Besetzung des Pixels**) in den Requests, da er schneller als die anderen war, auch wenn diese im aktuellen Schritt hätten wachsen sollen.

Ein Pixel wächst **nicht** in eine Richtung, wenn das reelle Wachstum in dieser Richtung bereits vorher aufgerundet wurde, als die Zahl ohne Nachkommastellen dieselbe war. Sonst würde der Kristall z.B. bei einem Gesamtwachstum von 0,5 und auch anschließend von 0,8 Pixeln wachsen, weil in beiden Fällen aufgerundet wird.

Die Requests werden am Ende von der Methode zurückgegeben.

Die Methode evaluate_requests:

```
def evaluate_requests (self):
```

Bestimmt die neuen Pixel aller Kristalle für den aktuellen Schritt.

Es wird durch *growing_crystals* iteriert. Die Wachstumsdaten des jeweiligen Kristalls werden mit *update_growth* für den aktuellen Schritt berechnet. Dann werden die Positionen der neuen Pixel durch *pixel_requests* ermittelt und wenn es keine gibt, muss der Kristall später aus *growing_crystals* entfernt werden.

Die angefragten neuen Positionen werden jeweils mit dem bisher niedrigsten Differenzwert an dieser Position verglichen. Wenn der neue Wert kleiner ist, wird die Position in *allowed_requests* mit dem aktuellen Kristall und seinem Wert aktualisiert.

Nachdem also entschieden wurde, welche Pixel von welchem Kristall besetzt werden, werden alle Positionen durchgegangen und jeweils für den dort eingetragenen Kristall diese Position zu *growing_crystals* hinzugefügt und in der ID-Map die Position mit der ID des Kristalls markiert.

Die Methode place_start_pixel:

```
def place_start_pixel(self, c: crystal):
```

Trägt die Kristall-ID an der Stelle des Startpixels in das Array ein und fügt den Kristall zu *growing_crystals* hinzu, außer unter folgenden Ausnahmebedingungen:

Die Startposition ist durch einen anderen Kristall besetzt **und**:

- Der Kristall darf nicht zufällig platziert werden
  oder
- Er darf zufällig platziert werden, aber es gibt keinen freien Pixel mehr.

run_simulation und export_image:

```
def run_simulation (self, nsteps = None):
```

Führt die Simulation durch. Es kann angegeben werden, wie viele Schritte simuliert werden sollen, ansonsten wird simuliert, bis kein Kristall mehr wachsen kann **und** kein verzögerter mehr folgt.

In jedem Schritt werden ggf. neue Kristalle platziert und mit *evaluate_requests* ihr Wachstum ausgewertet. Zum Schluss wird die benötigte Zeit in der Konsole ausgegeben.

```
def export_image(self, name = None):
```

Will man das Ergebnis als Bild exportieren, wird das ID-Array durch *export_image* in ein Array aus RGB-Werten umgewandelt, daraus ein Bild erstellt und unter dem angegebenen Namen gespeichert. Wo kein Kristall ist, wird der gewählte Hintergrund angezeigt.

Die Funktion „create_random_simulation“:

```
def create_random_simulation (img_size = None, min_size = (1,1), max_size = (1500, 1500),
n_crystals = None, min_crystals = 1, max_c_proportion = randint(5, 15) * 10**(-5),
grayscale_range = (40, 215), min_growth_speed = 0, do_delays = True,
random_placement_on_failure = True, start_pos = [], grayscales = [], speed_data = [],
delays = [], background = (0, 255, 0), name = "Zink_Kristalle"):
```

Eine Wrapperfunktion für das Initialisieren einer Simulation. Alle Parameter, die hier nicht gegeben sind, werden zufällig bestimmt (Mit Ausnahme des Hintergrundes und des Namens).

Beispiele

Hier stelle ich einige Beispiele vor, vor allem zu Spezialfällen. Diese und 8 weitere Beispielbilder sind im Ordner „Beispielausgaben“ zu finden.

Rekonstruktion des Beispielbildes

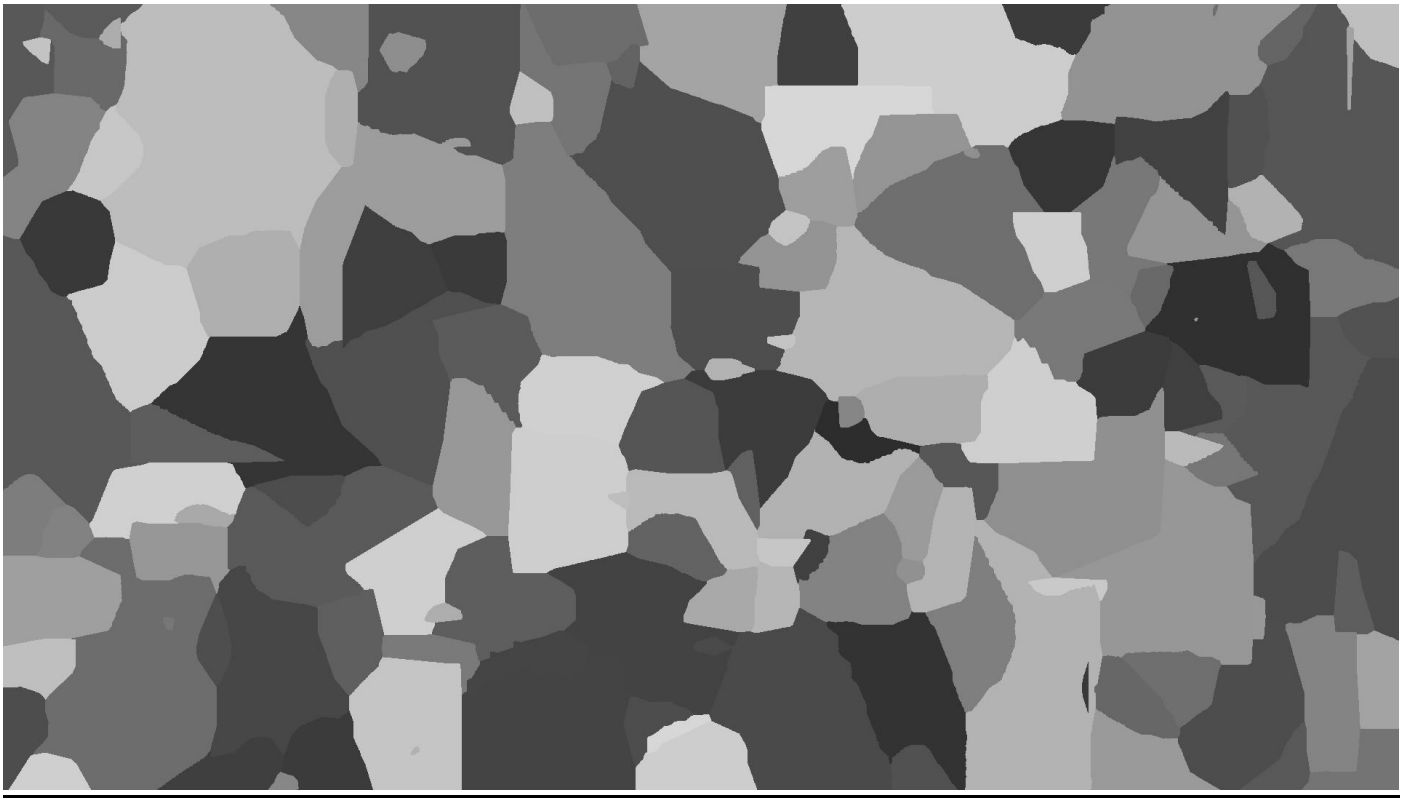
Um ein Bild zu erstellen, das dem gegebenen Bild aus der Realität ähnelt, habe ich das gegebene Bild in ähnlich gefärbte Bereiche aufgeteilt und jeweils den mittleren Graustufenwert und die ungefähre Position der Mitte dieses Bereiches bestimmt. Dann habe ich mit den Geschwindigkeitswerten und Verzögerungen experimentiert, bis ich zufrieden war.

Die verwendeten Daten können unter „Bild_Rekonstruktion.py“ angesehen werden. Hier das Resultat (Aus dem Ordner „Rekonstruiertes Bild“):

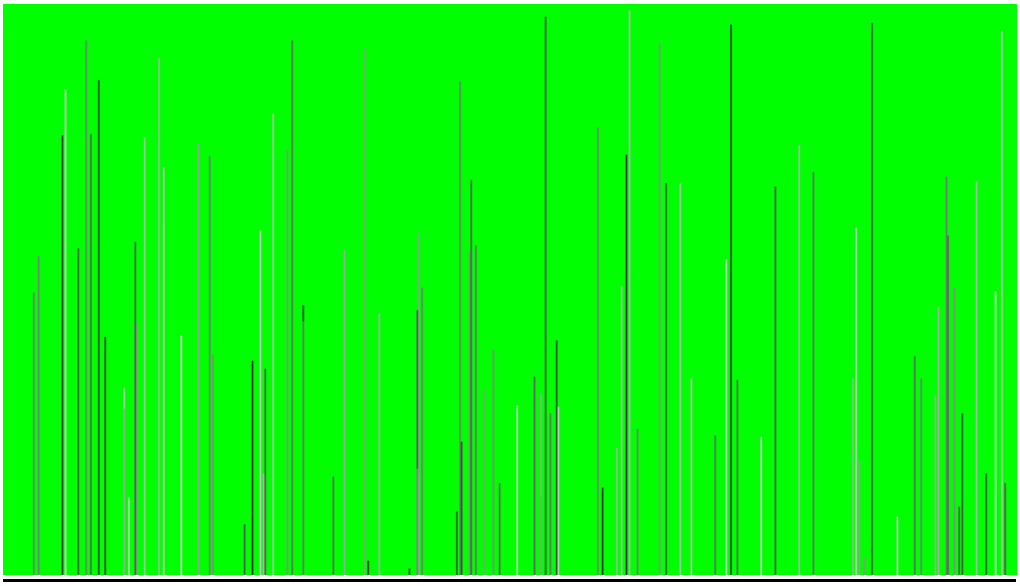
Komplett zufällige Simulation

Eine Simulation mit komplett zufälligen Parametern (bis auf den Hintergrund und den Namen):

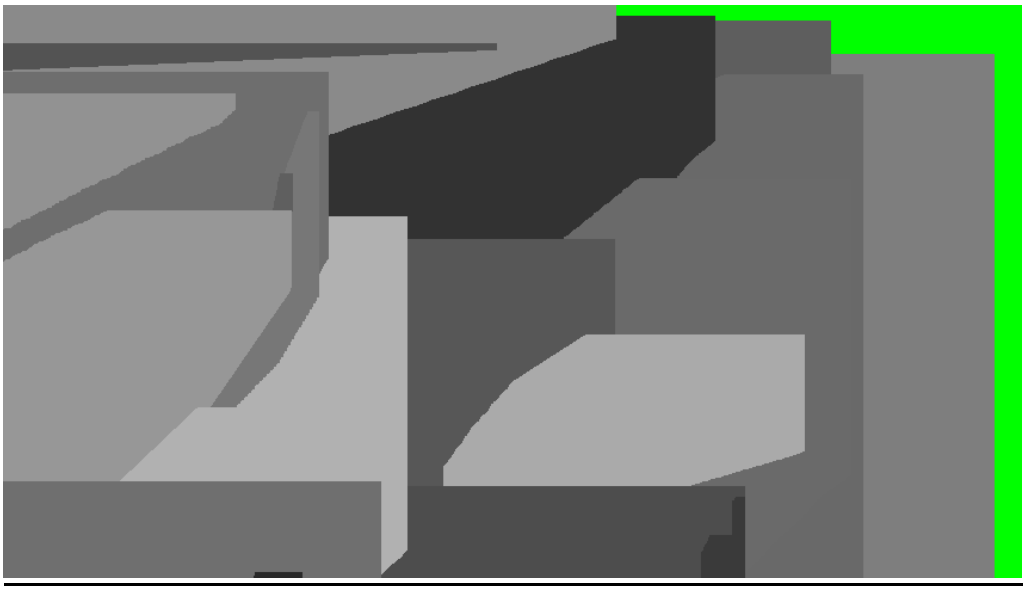
Simulation in Full-HD mit Verzögerungen



Simulation mit nur einer Wachstumsrichtung



Simulation mit nur zwei Wachstumsrichtungen



Simulation mit nur 2 Kristallen

Quellcode

Das ist der Quellcode ohne Kommentare. Die kommentierte Version ist „Verzinkt.py“.

```
import numpy as np
from random import randint, uniform, choice
from PIL import Image
from time import time

class resolutions:
    ULTRA_HD = (3840, 2160)
    FULL_HD = (1920, 1080)
    HD = (1280, 720)
    P_1440 = (2560, 1440)
    P_480 = (854, 480)
    P_360 = (640, 360)
    P_240 = (426, 240)

class crystal:
    DIRS = (UP, RIGHT, DOWN, LEFT) = \
    ((-1,0), (0,1), (1,0), (0,-1))
    def __init__ (self, ID, start_pixel, grayscale, growth_speeds, delay):
        self.ID = ID
        self.delay = delay

        self.set_start_pos (start_pixel)

        self.grayscale = grayscale

        (self.speed_up, self.speed_right, self.speed_down,
        self.speed_left) = self.growth_speeds = growth_speeds

        self.speed_from_dir = dict (zip ((self.UP, self.RIGHT, self.DOWN, self.LEFT),
        growth_speeds))
        self.real_growth_data = {self.UP: 0, self.RIGHT: 0, self.DOWN: 0, self.LEFT: 0}
        self.pixel_growth_data = self.real_growth_data.copy()
        self.diff_from_dir = self.real_growth_data.copy()

        self.last_rounded = {self.UP: -1, self.RIGHT: -1, self.DOWN: -1, self.LEFT: -1}

    def set_start_pos (self, pos):
        self.start_pixel = pos
        self.pixels = [self.start_pixel]
        self.growing_pixels = [self.start_pixel]

    def real_growth_diff (self, d):
        return self.pixel_growth_data[d] - self.real_growth_data[d]

    def update_growth (self, step: int):
        for d in self.DIRS:
            growth = self.speed_from_dir[d] * (step - self.delay)
            self.real_growth_data[d] = growth
```

```
            self.pixel_growth_data[d] = round(growth)
            self.diff_from_dir[d] = self.real_growth_diff(d)
        self.diff_from_dir = dict(sorted(self.diff_from_dir.items(),
        key = lambda item: item[1]))

class simulation:
    def __init__ (self, img_size:tuple, start_pos, grayscales, speed_data, delays = [],
    random_placement_on_failure:bool = False, background = (0, 255, 0), name =
    "Zink_Kristalle"):

        print("\nDaten werden überprüft...")
        img_size, start_pos, grayscales, speed_data, delays, background = check_data(
        img_size, start_pos, grayscales, speed_data, delays, background)

        self.crystals = []
        self.growing_crystals = []
        self.ID_map = np.asmatrix(np.full((img_size[::-1]), -1, int))
        self.width = img_size[0]
        self.height = img_size[1]

        if not delays:
            self.delays = []
        else:
            self.delays = list(delays)
        self.crystals_by_delay = dict()
        self.random_placement_on_failure = random_placement_on_failure

        self.name = name
        self.background = background

        start_pos = tuple(tuple(i) for i in start_pos)
        speed_data = tuple(tuple(i) for i in speed_data)
        self.crystal_data = list(zip(start_pos, grayscales, speed_data))

        delay = 0
        for i, data in enumerate(self.crystal_data):
            if self.delays:
                delay = self.delays[i]
            self.crystals.append(crystal(len(self.crystals), data[0], data[1], data[2],
            delay))

        if self.delays:
            unique_delays = set(self.delays)
            self.crystals_by_delay = dict(zip(unique_delays, [[] for _ in
            range(len(unique_delays))] ))
            for c in self.crystals:
                self.crystals_by_delay[c.delay].append(c)

            self.step = min(self.delays)
            for c in self.crystals_by_delay[self.step]:
                self.place_start_pixel(c)
        else:
```

```
            self.step = 0
            for c in self.crystals:
                self.place_start_pixel(c)

    def free_position (self):
        positions = np.argwhere (self.ID_map == -1)
        if positions.size:
            return tuple(choice(positions))
        return None

    def in_matrix_range (self, pos):
        if 0 <= pos[0] < self.height and \
           0 <= pos[1] < self.width:
            return True
        return False

    def pixel_requests(self, c: crystal):
        reqs = dict()
        dead_pixels = set(c.growing_pixels)

        for dir, diff in c.diff_from_dir.items():
            rounded = False
            for pos in c.growing_pixels:
                new_pos = (pos[0] + dir[0], pos[1] + dir[1])
                if self.in_matrix_range(new_pos):
                    if self.ID_map[new_pos] == -1 and c.speed_from_dir[dir] > 0:
                        dead_pixels.discard(pos)
                        if diff >= 0:
                            if int(c.real_growth_data[dir]) > c.last_rounded[dir]:
                                rounded = True
                                if new_pos not in reqs.keys():
                                    reqs[new_pos] = diff
            if rounded:
                c.last_rounded[dir] = int(c.real_growth_data[dir])
        for pos in dead_pixels:
            c.growing_pixels.remove(pos)
        return reqs

    def evaluate_requests (self):
        allowed_requests = dict()
        blocked_crystals = []
        for c in self.growing_crystals:
            c.update_growth(self.step)
            reqs = self.pixel_requests(c)
            if not c.growing_pixels:
                blocked_crystals.append(c)
                continue

            for pos, growth_diff in reqs.items():
                if pos in allowed_requests.keys():
                    if growth_diff < allowed_requests[pos][1]:
                        allowed_requests[pos] = (c, growth_diff)
```

```
                else:
                    allowed_requests[pos] = (c, growth_diff)

        for pos, (c, _) in allowed_requests.items():
            c.growing_pixels.append(pos)
            self.ID_map[pos] = c.ID

        for c in blocked_crystals:
            self.growing_crystals.remove(c)

    def place_start_pixel(self, c: crystal):
        if self.ID_map[c.start_pixel] == -1:
            self.ID_map[c.start_pixel] = c.ID
        elif self.random_placement_on_failure:
            rand_pos = self.free_position()
            if rand_pos:
                c.set_start_pos(rand_pos)
                self.ID_map[c.start_pixel] = c.ID
            else:
                return
        else:
            return
        self.growing_crystals.append(c)

    def run_simulation (self, nsteps = None):
        print("\nDie Simulation hat begonnen.")
        if nsteps:
            print(str(nsteps) + " Schritte werden simuliert...")
        else:
            print("Alle Schritte werden simuliert...")
        t_start = time()

        for c in self.growing_crystals:
            self.ID_map[c.start_pixel] = c.ID
        max_delay = 0
        if self.crystals_by_delay:
            max_delay = max(self.crystals_by_delay)
        while self.growing_crystals or self.step < max_delay:
            self.step += 1

            if self.step in self.crystals_by_delay.keys():
                for c in self.crystals_by_delay[self.step]:
                    self.place_start_pixel(c);


            self.evaluate_requests()
            if nsteps:
                print(str(round(self.step / nsteps * 100)) + "% der Schritte sind fertig",
                end = "\r")
                if self.step >= nsteps:
                    break
            else:
```

```
                print(str(self.step) + " Schritte durchgeführt", end = "\r")
        t_res = time() - t_start
        print(f"\n\nSimulation abgeschlossen. Es wurden {t_res} Sekunden benötigt.")

    def ID_to_grayscale_map (self):
        height = self.ID_map.shape[0]
        width = self.ID_map.shape[1]

        def convert(ID):
            if ID >= 0:
                return self.crystals[ID].grayscale
            else:
                return self.background

        gs_map = np.ndarray(shape = self.ID_map.shape + (3,), dtype = np.uint8)
        for y in range(height):
            for x in range(width):
                gs_map[y, x, [0,1,2]] = convert(self.ID_map[y, x])
        return gs_map

    def export_image(self, name = None):
        print("Bild wird gespeichert...")
        gs_map = self.ID_to_grayscale_map()
        image = Image.fromarray(gs_map, "RGB")
        if not name:
            name = self.name
        image.save(name + ".png")
        print(f"\nBild wurde als \"{name}\" gespeichert")

def check_img_size(img_size):
    if img_size:
        img_size = round(img_size[0]), round(img_size[1])
        if img_size[0] <= 0 or img_size[1] <= 0:
            raise ValueError("Bilddimensionen müssen Ganzzahlen größer als 0 sein")
    return img_size

def check_data(img_size, start_pos, grayscales, speed_data, delays, background):
    img_size = check_img_size(img_size)

    data = [start_pos, grayscales, speed_data]
    if delays:
        data.append(delays)
    for i in range(len(data)):
        data[i] = data[i][:min(len(seq) for seq in data)]

    start_pos = list(list(i) for i in start_pos)
    for i, v in enumerate(start_pos):
        start_pos[i] = v = (round(v[0]), round(v[1]))
        if v[0] < 0 or v[1] < 0:
            raise ValueError("Startpositionen müssen Ganzzahlen größer als 0 sein")
        if img_size:
            if v[0] > img_size[1] - 1 or v[1] > img_size[0] - 1:
```

```
                raise ValueError("Startposition geht über die Bilddimensionen hinaus")

    grayscales = list(grayscales)
    for i, v in enumerate(grayscales):
        grayscales[i] = min(max(round(v), 0), 255)

    speed_data = list(list(i) for i in speed_data)
    for i, seq in enumerate(speed_data):
        for i2, v in enumerate(seq):
            speed_data[i][i2] = min(max(v, 0), 1)
    if delays:
        delays = list(delays)
        for i, v in enumerate(delays):
            delays[i] = max(round(v), 0)
    delays = tuple(delays)

    background = list(background)
    for i, v in enumerate(background):
        background[i] = min(max(round(v), 0), 255)
    return img_size, tuple(tuple(i) for i in start_pos), tuple(grayscales), tuple(tuple(i)
    for i in speed_data),\
    delays, tuple(background)

def create_random_simulation (img_size = None, min_size = (1,1), max_size = (1500, 1500),
    n_crystals = None,
    min_crystals = 1, max_c_proportion = randint(5, 15) * 10**(-5), grayscale_range = (40,
    215), min_growth_speed = 0, do_delays = True, random_placement_on_failure = True,
    start_pos = [], grayscales = [], speed_data = [], delays = [], background = (0, 255,
    0), name = "Zink_Kristalle"):

    def rand_crystal_number (min_crystals, max_c_proportion, total_pixels):
        max_c_proportion = min (max_c_proportion, 1)

        if min_crystals:
            min_crystals = min(min_crystals, total_pixels)
        else:
            min_crystals = min(2, total_pixels)
        max_crystals = max(1, round(total_pixels * max_c_proportion), min_crystals)

        n_crystals = randint(min_crystals, max_crystals)
        return n_crystals


    print("Zufällige Simulation wird erstellt. Zufällige Parameter:")
    param_names = ["Bildgröße", "Startpositionen", "Graustufen",
    "Wachstumsgeschwindigkeiten", "Verzögerungen", "Hintergrundfarbe"]
    for i, p in enumerate((img_size, start_pos, grayscales, speed_data, delays,
    background)):
        if not p:
            print(param_names[i])
    img_size = check_img_size(img_size)
    if not img_size:
```

```
        img_size = (randint(min_size[0], max_size[0]), randint(min_size[1], max_size[1]))
    total_pixels = np.product(img_size)

    if not (n_crystals or start_pos or grayscales or speed_data):
        n_crystals = rand_crystal_number (min_crystals, max_c_proportion, total_pixels)
    elif not n_crystals:
        for seq in (start_pos, grayscales, speed_data):
            if len(seq):
                n_crystals = len(seq)

    if not start_pos:
        while len(start_pos) < n_crystals:
            pos = (randint(0, img_size[1]-1), randint(0, img_size[0]-1))
            if pos not in start_pos:
                start_pos.append(pos)

    if not grayscales:
        grayscale_range = (max(0, round(grayscale_range[0])),
        min(255,round(grayscale_range[1])))
        for _ in range(n_crystals):
            scale = randint(*grayscale_range)
            grayscales.append(scale)

    if not speed_data:
        min_growth_speed = min(min_growth_speed, 1)
        for _ in range(n_crystals):
            speed_data.append(list(uniform(min_growth_speed, 1) for _ in range(4)))

    if do_delays or delays:
        if not delays:
            delays.append(0)
            max_delay = max(1, round(min(img_size) / 3))
            for _ in range (n_crystals - 1):
                delays.append(randint(1, max_delay))
            delays.sort()
    else:
        delays = []

    return simulation (img_size, start_pos, grayscales, speed_data, delays,
    random_placement_on_failure, background, name)
```

Hier der Quellcode zum Erstellen einer (fast) komplett zufälligen Simulation:

```
from Verzinkt import create_random_simulation

sim = create_random_simulation()
sim.run_simulation()
sim.export_image("Beispielausgaben\Komplett zufällig")
```