

Aufgabe 2: Verzinkt

Team-ID: 00100

Team-Name: Team

Bearbeiter/-innen dieser Aufgabe:
Arne de Borman

21. November 2022

Inhaltsverzeichnis

1 Lösungsidee	1
2 Umsetzung	2
3 Beispiele	2
4 Quellcode	4

1 Lösungsidee

Die Position der Kristallkeime wird zufällig bestimmt. Die Wahrscheinlichkeit, dass ein Kristall an einer bestimmten Stelle entsteht, wird durch eine Gradient-Noise-Funktion bestimmt, zum Beispiel OpenSimplex Noise, sodass Kristalle an einigen Stellen eher entstehen als an Anderen. Außerdem wird diese Wahrscheinlichkeit mit einem Wert potenziert, sodass eingestellt werden kann, ob die Noise-Funktion einen höheren oder geringeren Einfluss auf die Simulation hat.

Im Bild sieht es aus, als hätten die Kristalle eine Richtung in die sie schneller Wachsen. Um diesen Effekt in der Simulation zu erreichen wird zuerst ein Richtungswinkel zwischen 0 bis 2π berechnet. Mithilfe der Sinus- und Kosinusfunktionen werden nun die Geschwindigkeiten in die vier Richtungen berechnet. Die Geschwindigkeiten werden mit einer zufälligen Magnitude multipliziert. Außerdem muss jede Richtung einen positiven Geschwindigkeitswert haben. Deshalb werden alle Geschwindigkeiten kleiner als ein zufälliger Wert auf den Wert gesetzt.

Wenn man die Geschwindigkeiten nun direkt benutzen würde und die Kristalle jeden Schritt um die jeweiligen Geschwindigkeiten in die verschiedenen Richtungen wachsen lassen würde, so könnte ein Kristall nebenan liegende Kristalle direkt überwachsen, sodass sie nie die Chance haben zu wachsen. Deshalb werden die Geschwindigkeiten in Wartezeiten umgewandelt, die abgewartet werden müssen, bis ein Kristall in eine bestimmte Richtung wachsen kann.

Um die Kristalle nach einer bestimmten Zeit wachsen zu lassen, wird ein Ringpuffer verwendet, in dem mehrere Listen mit möglichen neuen Positionen für Kristalle gespeichert sind. In jedem Schritt wird die nächste Liste mit möglichen Positionen gelehrt und es wird überprüft welche Positionen im Raster noch frei sind. Wenn ein Kristall auf eine neue Position "wächst", werden direkt neue mögliche Positionen für den Kristall bestimmt und je nach den Wartezeiten der jeweiligen Richtungen "näheröder" "weiterim Ringpuffer platziert. Hat ein Kristall zum Beispiel in positive x-Richtung eine Wartezeit von 5, dann wird die neue mögliche Position 5 Positionen weiter im Ringpuffer platziert. Das bedeutet aber auch, dass der Ringpuffer so groß sein muss wie die größte Wartezeit.

Im der Aufgabe sieht es teilweise so aus, als würden sich die Kristalle manchmal aufteilen und ihre Orientierung/Richtung verändern. Das kann auch in der Simulation geschehen, hier kann kontrolliert werden, wie oft die Kristalle "mutieren" und wie stark die Mutation ausfällt.

Insgesamt können also folgende Parameter kontrolliert werden:

- Die Höhe und Breite des erzeugten Bildes
- Die maximale Latenz
- Wie viele Kristallkeime erzeugt werden
- Die Mindestgeschwindigkeit neuer Kristalle
- Mindest- und Höchsthelligkeit
- Wie eingezoomt "die Noise-Funktion ist"
- Wie stark die Werte der Noise-Funktion die Simulation beeinflussen
- Die Wahrscheinlichkeit, dass Kristalle mutieren
- Wie stark die Helligkeit mutiert
- Wie stark die Wachstumsdauern mutieren

2 Umsetzung

Die Lösungsidee wird in rust implementiert. Für die Noise-Funktion wird die `opensimplex_noise_rs` Bibliothek, für das Erzeugen von zufälligen Zahlen die `rand` Bibliothek und für die Speicherung der Bilder die `image` Bibliothek verwendet. Das Raster wird durch eine Liste von Listen implementiert. Jedes Feld des Rasters enthält entweder Nichts oder eine Referenz zu einem Kristall.

```
type CellState = Option<Rc<Crystal>>;
struct Grid {
    pub rows: Vec<Vec<CellState>>,
    pub width: usize,
    pub height: usize,
}
```

Der Ringerpuffer für die Speicherung der neuen möglichen Positionen wird auch eine Liste mit mehreren Listen verwendet.

```
let mut poss_cells: Vec<Vec<(usize, usize, Rc<Crystal>)>>
```

Die Listen innerhalb werden immer wieder geleert, ihr Platz bleibt aber weiter reserviert.

3 Beispiele

Das Programm wird mit dem `"$ cargo run --release` Befehl ausgeführt die `-release` Endung ermöglicht Optimisierungen die erlauben, dass das Programm in einer akzeptablen Zeit läuft. Alternativ kann auch die vorkompilierte Datei für die jeweilige Plattform verwendet werden, zum Beispiel `"$ verzinkt-windows.exe"`. Das Programmargument ist der gewünschte Name des erzeugten Bildes.

```
$ cargo run --release out.png
simulierte Wachstum...
fertig berechnet, speichere Bild in out.png
finished after 19.165862294s
```

Die Parameter die Simulation müssen im Quellcode verändert werden. Verschiedene Parameter führen zu verschiedenen Bildern, hier ein paar Beispiele:

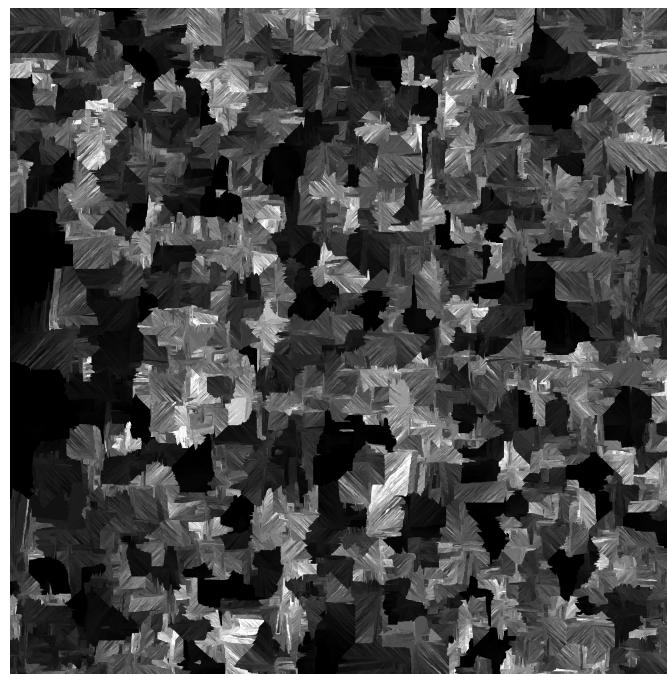


Abbildung 1: Hohe Mutationswahrscheinlichkeit, geringere Helligkeit



Abbildung 2: Geringe Maximalwartezeit, geringe Mutationsraten, Noise Funktion hat relativ viel Wichtigkeit (5.0)



Abbildung 3: Geringe Mutationswahrscheinlichkeit, viele Kristall keime, ähnelt dem Bild aus der Aufgabe am ehesten

4 Quellcode

```

struct Crystal {
    pub brightness: u8,
    //Wartezeiten in der folgenden Reihenfolge: HOCH, RECHTS, RUNTER, LINKS
    pub update_delays: [u8; 4],
}
//Die verschiedenen Richtungen
const DIRS: [(i32, i32); 4] = [(0, 1), (1, 0), (0, -1), (-1, 0)];
#[derive(Clone, Debug)]
struct Crystal {
    pub brightness: u8,
    //Wartezeiten in der folgenden Reihenfolge: HOCH, RECHTS, RUNTER, LINKS
    pub update_delays: [u8; 4],
}

impl Crystal {
    //Der Konstruktor
    fn new(max_speed: u8, rng: &mut ThreadRng) -> Self {
        let brightness = rng.gen_range(BRIGHT_MIN..BRIGHT_MAX);
        let mag = /*max_speed as f64; */rng.gen_range(MAG_RANGE);
        let dir = rng.gen_range(0.0..PI * 2.0);
        let update_rates: [f64; 4] = [
            dir.sin() * mag, //HOCH
            dir.cos() * mag, //RECHTS
            -dir.sin() * mag, //RUNTER
            -dir.cos() * mag, //LINKS
        ];
        let update_delays: [u8; 4] = update_rates
            .iter()
            //Stellt sicher, dass die Geschwindigkeiten mindestens
            //im Wertebereich MIN_SPEED_RANGE sind
            .map(|rate| (max_speed as f64 / rate.max(rng.gen_range(MIN_SPEED_RANGE))) as u8)
    }
}

```

```

.collect::<Vec<u8>>()
.try_into()
.unwrap();
Self {
    brightness: brightness as u8,
    update_delays,
}
}
fn mutate(&self, rng: &mut ThreadRng) -> Rc<Self> {
    let brightness = self.brightness as i32 + rng.gen_range(BRIGHT_MUT_RANGE);
    //Stellt sicher, dass die Helligkeit mindestens BRIGHT_MIN und höchstens BRIGHT_MAX ist
    let brightness = brightness.max(BRIGHT_MIN as i32).min(BRIGHT_MAX as i32) as u8;
    let update_delays = self
        .update_delays
        .iter()
        .map(|d| {
            let new = *d as i32 + rng.gen_range(LATENCY_MUT_RANGE);
            //Die Wartezeit ist mindestens eins und höchstens LATENCY_MAX
            new.max(1).min(LATENCY_MAX as i32) as u8
        })
        .collect::<Vec<u8>>()
        .try_into()
        .unwrap();
    Rc::new(Self {
        brightness,
        update_delays,
    })
}
fn add_to_poss_cells(
    self: &Rc<Crystal>,
    poss_cells: &mut Vec<Vec<(usize, usize, Rc<Crystal>)>>, //Der Ringpuffer
    x: usize,
    y: usize,
    t: usize, //Die momentane Position des Ringpuffers
    rng: &mut ThreadRng,
) {
    let mut mutated = false;
    let crystal = if rng.gen_range(0.0..1.0) < MUT_PROB {
        //den Kristall mutieren
        mutated = true;
        self.mutate(rng)
    } else {
        self.clone()
    };
    for (i, dir) in DIRS.iter().enumerate() {
        //Mutierte Kristalle schneller erzeugen, sodass weniger "Linien" entstehen
        let delay = if mutated { 1 } else { self.update_delays[i] };
        let new_x = x as i32 + dir.0;
        let new_y = y as i32 + dir.1;
        let len = poss_cells.len();
        //die neue Position in den Ringpuffer delay Stellen weiter einfügen
        poss_cells[((t + delay as usize) % len) as usize].push((
            new_x as usize,
            new_y as usize,
            crystal.clone(),
        ));
    }
}

```

```

}

//Ein Feld des Rasters
type CellState = Option<Rc<Crystal>>;
//Das Raster
struct Grid {
    pub rows: Vec<Vec<CellState>>,
    pub width: usize,
    pub height: usize,
}

//Generiert die Kristall Keime
fn gen_kristal_seeds(
    grid: &mut Grid,
    rng: &mut ThreadRng,
) -> Vec<Vec<(usize, usize, Rc<Crystal>)>> {
    let mut poss_cells = vec![Vec::new(); LATENCY_MAX as usize];
    let mut n_gen_start = 0;
    //Die Noise-Funktion initialisieren
    let noise_fn = OpenSimplexNoise::new(Some(rng.gen_range(0..10000)));
    while n_gen_start < N_SEEDS {
        let x = rng.gen_range(0..WIDTH);
        let y = rng.gen_range(0..HEIGHT);
        match grid.get_unchecked(x, y) {
            //nach neuer Position suchen, falls die Position schon besetzt ist
            Some(_) => continue,
            None => {
                //Wert zwischen -1 und 1
                let p = noise_fn.eval_2d((x as f64) * NOISE_SCALE, (y as f64) * NOISE_SCALE);
                let p = (p + 1f64) / 2f64; //Wert zwischen 0 und 1
                if rng.gen_range(0.0..1.0) > p.powf(NOISE_IMPORTANCE) {
                    continue;
                }
                n_gen_start += 1;
                let crystal = Rc::new(Crystal::new(LATENCY_MAX as u8, rng));
                //die neuen möglichen Positionen der Kristalle berechnen
                crystal.add_to_poss_cells(&mut poss_cells, x, y, 0, rng);
                grid.set(x, y, crystal);
            }
        }
    }
    poss_cells
}

//Wachstum der Kristalle simulieren
fn run_simulation(
    poss_cells: &mut Vec<Vec<(usize, usize, Rc<Crystal>)>>,
    mut grid: Grid,
    mut rng: ThreadRng,
) -> Grid {
    let mut is_empty = false; //gibt an, ob es noch mögliche neue Positionen gibt

    //Platzhalter für eine Liste im Ringpuffer
    //Wird aufgrund von rusts "Borrow Checker" benötigt
    let mut cells = Vec::new();
    while !is_empty {
        is_empty = true;
        //Geht jede Liste im Ringpuffer durch
        for t in 0..LATENCY_MAX {

```

```
//tauscht die aktuelle Liste mit dem Platzhalter aus
std::mem::swap(&mut poss_cells[t], &mut cells);
if cells.is_empty() {
    continue;
}
is_empty = false;
while let Some((x, y, crystal)) = cells.pop() {
    if !grid.contains(x as i32, y as i32) {
        continue;
    }
    if grid.get_unchecked(x, y).is_some() {
        continue;
    }
    //Berechnet neue mögliche Positionen und platziert sie in dem Ringpuffer
    crystal.add_to_poss_cells(poss_cells, x, y, t, &mut rng);
    grid.set(x, y, crystal);
}
//Platziert die Liste zurück in den Puffer,
//sodass der beriets reservierte Platz wiederverwendet werden kann
std::mem::swap(&mut poss_cells[t], &mut cells);
}
grid
}
```