04.02.2021

“Conway’s Game of Life” in Processing

Programmcode & Dokumentation von Dennis Paust

1. Programmumfang

1.1 Das Game of Life

Das standardmäßige Spiel läuft nach Conways Regeln ab:

- Eine tote Zelle mit genau drei lebenden Nachbarn wird in der Folgegeneration neu geboren.
- Lebende Zellen mit weniger als zwei lebenden Nachbarn sterben in der Folgegeneration an Einsamkeit.
- Eine lebende Zelle mit zwei oder drei lebenden Nachbarn bleibt in der Folgegeneration am Leben.
- Lebende Zellen mit mehr als drei lebenden Nachbarn sterben in der Folgegeneration an Überbevölkerung.
- Das Spielfeld ist ein Torus.

1.2 Bedienung & Navigation

- es lassen sich durch Drücken der linken und rechten Maustasten lebendige (schwarz) bzw. tote Zellen (weiß) auf das Grid malen
- Drücken der mittleren Maustaste (Scrollrad) über einer Zelle druckt dessen x- / y-Koordinaten und Klassennamen in der Konsole aus
- die Play- / Pause-Schaltfläche in der Navigationszeile des Fensters oder das Drücken der Space-Taste startet / pausiert die automatische Evolution; standardmäßig beginnt diese mit 1 Hz, d.h. einer Evolution pro Sekunde
- die Evolutionsgeschwindigkeit kann durch den Schieberegler mit exponentiellen Skala von 1 Hz bis standardmäßig 2500 Hz geregelt werden
- ob der Computer solche Frequenzen überhaupt schafft, lässt sich anhand der FPS durch Drücken der “f”-Taste in der Konsole überprüfen
- soll das Spiel nur eine Generation auf einmal durchlaufen, geschieht dies durch die “Next Step”-Schaltfläche oder Drücken / gedrückt halten der “n”-Taste
- in welcher Generation sich das Spiel befindet wird mit der “g”-Taste in der Konsole angezeigt
- mit “q” wird das Fenster geschlossen

1.3 Benutzervariablen

Folgende Variablen (und keine anderen!) sind durch den Benutzer frei einstellbar und als solches in der ersten Zeile der Datei “Game_of_Life.pde” gekennzeichnet:

- **float res:** die jeweilige Größe der Zellen in Pixeln; diese Variable bestimmt im Zusammenhang mit der Größe des Programmfensters welche Dimensionen das Grid annimmt
- **float maxHz:** das Maximum des Schiebereglers in der Navigationsleiste

- **float margin:** eine rein ästhetische Variable in Pixeln, um den Rand zwischen Grid und Fenster einzustellen
- **float gridWeight:** eine rein ästhetische Variable in Pixeln, die die Linienstärke des Gitters bestimmt; sie sollte nicht unter 0.01 px eingestellt werden, um unerwünschte Erscheinungen beim Zeichnen von Zellen zu vermeiden

1.4 Speichern und Laden von Dateien

Es lässt sich zu jedem Zeitpunkt ein gegebenes Spielfeld abspeichern, wobei die Dimensionen des Feldes respektiert werden. Zu jeder einzelnen Zelle wird ein Objekt mit den Eigenschaften "alive", "x" und "y" einer JSON-Datei zugefügt. Den Namen und Speicherort kann der Benutzer frei wählen. Trivial ist jeglicher Versuch der Datei einen Typen zuzuschreiben, da der eingegebene Titel vom Programm auf ".json" geändert wird. Somit muss beim Speichern der Datei auch keine Endung zugefügt werden.

Beim Laden einer JSON-Datei überschreibt das Programm das bestehende Grid (und ggf. bereits eingezeichnete Zellen) und passt die Größe der einzelnen Zellen (*float res*) automatisch der Größe des Programmfensters an. Da nur die eben genannten drei Attribute einer Instanz der Klasse *Cell* beachtet werden, ist das Speichern und Laden von Spielfeldern auch für Modifikationen des Spiels (mit weiteren Klassen und Regeln) möglich, wobei Zellen mit neuen Eigenschaften oder Klassen lediglich als normale, lebendige / tote *Cell*-Objekte geladen werden.

2. Programmstruktur & Funktionsweise

Das Programm besteht aus insgesamt drei Dateien bzw. Tabs. Ihr Code ist nach ihren jeweiligen Funktionen unterteilt:

2.1 "Game_of_Life.pde" (ca. 210 Zeilen)

Der Code hierin ist ausschließlich für das Setup und die Handhabung von User-Inputs zuständig, weshalb hier nur globale Variablen, auf die Benutzeroberfläche zu zeichnende Elemente und Event-Listener zu finden sind.

Die Funktion *mySetup* existiert lediglich außerhalb der normalen *setup*-Funktion, sodass erstere beim Laden von Dateien erneut und mit etwas verschiedenen Einstellungen bzgl. der Darstellung von Zellen gerufen werden kann.

2.2 "Helper_Functions" (ca. 160 Zeilen)

Der Übersicht halber gibt es nur hier Code mit tatsächlicher Funktion für das Programm. Hierin befinden sich die Funktionen *fileImported*, *fileExported*, *slider*, *drawInitialGrid*, und *toggleLoop*. Der Code wird nicht näher betrachtet, da er mit dem eigentlichen Game of Life nichts zu tun hat. Jedoch befindet sich hier auch die Funktion *nextGeneration*, welche sich um die Evolution der Zellen kümmert. Das Verständnis dieses Codes ist essentiell, um dem Programm später neue Klassen und Regeln zuzufügen, weshalb er im Folgenden vollständig abgedruckt ist:

```
void nextGeneration() {
    nextGrid = new Cell[cols][rows];
    for (int i = 0; i < cols; i++) {
        for (int j = 0; j < rows; j++) {
            Cell clone = grid[i][j].clone().transition();
            nextGrid[i][j] = clone;
            clone.display();
        }
    }

    grid = nextGrid;
    gen++;
}
```

Zunächst wird *nextGrid* initialisiert, d.h. einem neuen, leeren Array zugeschrieben, sodass es später neue Werte bekommen kann. Bei dieser Variablen sowie *grid* handelt es sich um zweidimensionale Arrays, welche einen Datentypen der Klasse *Cell* speichern. In einer doppelten for-Schleife wird jede einzelne Zelle durchgangen. Es wird die Variable *Clone* erstellt, welche später *nextGrid* an den Koordinaten (i, j) zugeschrieben wird.

Zuvor wird jedoch das originale *Cell*-Objekt dupliziert (hier es handelt es sich um eine *deep copy*). Dies ist notwendig, weil ein jeweiliges Objekt unter keinen Umständen dem zweiten Grid zugewiesen werden darf: *nextGrid[i][j] = grid[i][j]* ist also **nicht erlaubt**. Grund dafür ist, dass ein solches Verfahren in beiden Arrays auf denselben Speicherplatz, d. h. das gleiche Objekt verweisen würde. Sodass das originale Array *grid* nicht laufend modifiziert wird, während folgende Zellen noch auf das ursprüngliche Objekt zugreifen müssten, wird eine exakte Kopie mit *grid[i][j].clone()* erstellt. Der Rückgabewert muss vom gleichen Typen (im Standardspiel nur *Cell*) sein und dieselben Werte für dessen Attribute haben wie das ursprüngliche Objekt.

Der erhaltene Klon wird noch in derselben Zeile verwendet, um die Evolution der jeweiligen Zelle (bzw. auf dessen Klon!) mit *.transition()*auszuführen. Auch *transition* hat einen Rückgabewert, welcher schlussendlich der lokalen Variable *clone* und dann mit *nextGrid[i][j] = clone* gespeichert wird. Extrem wichtig ist dabei, dass ***transition* immer den Typen *Cell* zurückgibt, dies jedoch auch einschließt, dass Tochterklassen von *Cell* Rückgabewerte sein können**, womit die ursprüngliche Zelle in der folgenden Generation ein komplett unterschiedliches Objekt von neuem Typen werden kann. Dies ist jedoch erst der Fall, sobald das Programm wie in später folgenden Beispielen modifiziert wird.

Zuletzt wird in den Schleifen die Methode *display* auf dem Klon gerufen, welche an der jeweiligen Position die neue Zelle im Grid übermalt.

Sobald alle Zellen durchlaufen sind, erhält das originale *grid* die neuen Werte und überschreibt dabei die alten. Das direkte Zuweisen mit dem Gleichzeichen ist nur möglich, weil zu Beginn von *nextGeneration nextGrid* wieder überschrieben wird, sie also nicht auf dieselben Speicher verweisen! Zuletzt wird der Zähler *gen* inkrementiert.

Obwohl diese Methode aufgrund der ständig neu erstellten Instanzen von Klassen sehr ineffizient für den Speicher sowie übermäßig umständlich ist, nur um einen Boolean-Wert zu speichern, ist dies die beste (und ggf.einzige) Methode, welche einfache Modifikationen an den Regeln und das Erstellen von Tochterklassen zulässt, ohne, dass am Aufbau und der Funktionsweise des Programms irgendetwas geändert werden muss.

2.3 "Cell.pde" (75 Zeilen)

Diese Datei enthält zunächst nur die Klasse *Cell* und ist ebenfalls für weitere, neue Klassen bei der erweiterung des Programms vorgesehen. Sie besitzt standardmäßig die Attribute *alive*, *x* und *y*. Die drei essentiellen Methoden der Klasse sind

- ***clone***, in welcher eine exakte Kopie der jeweiligen Instanz zurückgegeben wird,
- ***transition***, wo die Regeln für das verhalten der Zelle festgelegt werden, und
- ***display***, die die Darstellung und die visuelle Gestaltung der Zelle übernimmt.

Diese Methoden müssen in Tochterklassen überschrieben werden, um verschiedenes Verhalten bzw. Aussehen der Zellen zu erzielen. Weitere Methoden zur Erleichterung der Programmierung des Verhaltens der Zelle sind

- *className*, welche den Namen der (Tochter-) Klasse als String zurückgibt und
- *getNeighbours*, die alle acht Nachbarn (direkt und diagonal) einer Zelle als *Cell*-Array zurückgibt. Um das Feld zu einem Torus zu machen und die Spalte (*col*) und Reihe (*row*) zu bestimmen, wird der Modulo-Operator genutzt:

  ```
  int col = (x + i + cols) % cols;
  int row = (y + j + rows) % rows;
  ```

  Dabei sind *x* und *y* die Koordinaten der Zelle, deren Nachbarn zu finden sind, während *i* und *j* zu einer doppelten for-Schleife gehören, welche beide von -1 bis +1 gehen, um die jeweils umliegenden Zellen zu finden.

In der *transition*-Funktion werden alle lebenden Nachbarn summiert, wodurch Conways normale Regeln in nur zwei Zeilen abgehandelt werden:

```
if (!alive && sum == 3) alive = true;
else if (alive && (sum < 2 || sum > 3)) alive = false;
```

3. Modifizierung & Erweiterung des Originalprogramms

Wie zuvor beschrieben, sollten die Methoden *clone, transition* und *display* in den Tochterklassen überschrieben werden. Um mögliche Erweiterungen zu demonstrieren, werden die *Virus*- und *Tank*-Variation zur Hilfe genommen. Eventuell ist es hilfreich zuvor die Klassen und Regeln der Programme zu kennen, welche am Ende des Dokuments vorzufinden sind.

3. Hilfreiche Hinweise zur Erweiterung

1. Oft sind Objekte bereits per Definition lebendig oder tot, in welchem Fall dies nicht als Parameter des Konstruktors gebraucht wird:

   ```
   Carcass(int x, int y, int dur, float severity) {
           super(false, x, y); // carcass is dead by definition
   ...
   ```

2. Nutzt man Eigenschaften, die nur beim Initialisieren entfallen / gebraucht werden, kann dies durch optionale Parameter vereinfacht werden. Die drei Punkte stehen für ein Array des davorstehenden Typs mit beliebiger länge (weshalb es als letzter Parameter aufgeführt werden muss):

```
Tank(int x, int y, int maxHP, int...hp) {
        super(true, x, y);
        this.maxHP = maxHP;
        this.hp = hp.length > 0 ? hp[0] : maxHP;
    }
```

3. Nutzt man die die *className*-Methode lassen sich Fallunterscheidungen machen, sodass eine Zelle auf Nachbarn verschiedener Klassen unterschiedlich reagiert. Betrachtet man folgendes Beispiel, erkennt man, wie für jeden Nachbarn abgefragt wird zu welcher Klasse er gehört. Sollte er bspw. *"Infected"* sein, wird die Zelle der Basisklasse *Cell* aus *nbs* zur Tochterklasse *Infected* abgeleitet und dessen Attribut *severity* genutzt:

```
Cell transition() {
        ...

        Cell[] nbs = getNeighbours();
        int sum = 0;

        for (int i = 0; i < nbs.length; i++) {
            Cell c = nbs[i];
            sum +=c.alive ? 1 : 0;
            String n = c.className();
            ...

            // infection is exacerbated by other infected cells or carcasses
            if (random(1) < transferProb) {
                if (n.equals("Infected") && (c.x == x || c.y == y)) severity +=
((Infected)c).severity;
                else if (n.equals("Carcass")) severity += ((Carcass)c).severity;
            }
        }

       ...
```

4. Auch Tochterklassen können weiterhin spezialisiert werden, wie z. B. *Cell* → *TribeMember* → *Warrior*

5. Um den Zellen visuelle Information zu geben, können in *display* Klassenattribute, Text, Zahlen oder auch Bilder, welche im "data"-Ordner hinterlegt wurden (hier SVGs), verwendet werden:

```
void display() {

        float p = severity / maxSeverity;
        float r = pow(10, p * log(maxSeverity * 255) / log(10)) + 50;
        fill(r, 0, 0);
        stroke(0);
        strokeWeight(gridWeight);
        rect(x * res + offsetX, y * res + offsetY + 40, res - gridWeight, res -
gridWeight);

        fill(255);
        textSize(textS);
        text(str(health), x * res + offsetX, y * res + offsetY + 40, res - gridWeight, res
- gridWeight);
    }
```

```
void setup() {
    helmet = loadShape("helmet.svg");
...

void display() {

        ...

        shape(helmet, x * res + offsetX, y * res + offsetY + 40, res - gridWeight, res -
gridWeight);
    }
```

3.2 Bedingungslos einzuhaltende Regeln

1. Das originale Array *grid* darf niemals verändert werden.
2. Im Array *nextGrid* darf niemals ein Wert außerhalb von *nextGeneration* (d. h. nur für die aktuell in den for-Schleifen betrachteten Zellen) verändert werden.
3. Die Attribute *alive, x,* und *y* der Basisklasse *Cell* dürfen niemals verändert werden
4. Der Rückgabetyp von *clone* muss immer derselbe sein wie der der Tochterklasse
5. Der Rückgabetyp von *transition* muss immer *Cell* sein, sodass hier die Rückgabe nicht nur auf einen Typ beschränkt ist, sondern *Cell* und alle seine Tochterklassen als Wert akzeptiert werden. So entwickeln sich Zellen im Laufe der Evolution zu anderen Arten.
6. Rein kausal kann es der ausgedachte Sachzusammenhang nahelegen, dass eine Zelle einen Effekt auf ihre umliegenden Nachbarn hat, wie z. B. eine infizierte Zelle, die ihren Virus an alle ihre Nachbarn übergibt. Allerdings darf in der *transition*-Methode der *Infected*-Klasse niemals versucht werden, Nachbarzellen zu verändern, da dies die Integrität bereits behandelte Zellen in *nextGrid* und zukünftig noch zu behandelnde Zellen in *grid* kompromittieren würde (vgl. Regeln 1 und 2). Stattdessen müssen Regeln für die angezielten Zellen in Abhängigkeit von anderen Zellen definiert werden. In diesem Beispiel bedeutet dies, dass gesunde Zellen infiziert werden, weil ihre Nachbarn den Virus tragen, und nicht, dass infizierte Zellen umliegende, gesunde Zellen anstecken. Hierbei ist es wichtig nicht davor zurückschrecken, die originale *transition*-Methode in *Cell* zu verändern. Weiterführen des Auszugs aus Punkt 3.1.3 zeigt dies:

```
class Cell {
    ...

    Cell transition() {
        ...

        // RULES
        if (!alive && sum == 3) alive = true;
        else if (alive && (sum < 2 || sum > 3)) alive = false;

        if (alive) {
            if (severitySum > 0)
                return new Infected(x, y, severitySum, spawnDur);
            else if (random(1) < infectionProb)
                return new Infected(x, y, spawnSev, spawnDur);
        }

        return this;
    }
```

4. Programmcode mit beispielhaften Variationen

Das Originalprogramm mit Conways Standardregeln sowie einige Beispiele mit neuen Klassen und modifizierten Regeln sind auf GitHub unter dem “main”-Zweig bzw. den anderen Zweigen verfügbar:

https://github.com/dango301/Game-of-Life
https://github.com/dango301/Game-of-Life/tree/Tank
https://github.com/dango301/Game-of-Life/tree/Virus
https://github.com/dango301/Game-of-Life/tree/Tribe

Es empfiehlt sich die README-Dateien durchzulesen, welche die neuen Regeln und Klassen exakt beschreiben. Weiterhin befindet sich dort ein Hinweis zur Umbenennung der Ordner, wenn die Dateien als ZIP heruntergeladen werden: Leider erlauben es die Vorgaben zur Benennung von Git-Hub-Pfaden sowie Processing-Dateien nicht sie gleichermaßen zu benennen. Daher ist es wichtig beim **Entpacken der ZIP den Namen des Ordners an den der jeweiligen Processing-Datei anzupassen!**