Aufgabe 1 - Thread-Basierte Glühwürmchen

Zunächst wollte ich die Aufgabe um der alten Traditionen Willen in JavaScript lösen, leider ist mir dabei aber klar geworden, dass Multithreading und JavaScript noch schlechter zusammen passen als OpenCL und JavaScript.

Schlussendlich habe ich mich also dazu entschieden, die Aufgabe in Java zu realisieren, da es hier eine sehr komfortable Implementierung von Mutli-Threading gibt, und außerdem einige sehr ausgereifte UI-Bibliotheken.

Meine Implementierung besteht aus drei Klassen und einem Interface:

1. `class Main` JavaFX-Oberfläche, die die Glühwürmchen-Threads initialisiert und visualisiert.
2. `class Firefly` Implementiert `Runnable` und repräsentiert somit die Glühwürmchen-Threads. Sie erbt außerdem von `Rectangle` und kann sich somit direkt selbst in JavaFX visualisieren.
3. `class TorusTopology` Helfer-Klasse, die aus einem gegebenen 2-dimensionalen Array von Objekten die Nachbarn eines gegebenen Objektes in einem Torus zurückgibt.
4. `interface Topology` Interface, das die `getNeighbours()`-Methode enthält. Das Interface wird von der `TorusTopology`-Klasse implementiert und ist vorhanden, um einfacher weitere Topologien implementieren zu können.

Die Synchronisation der Glühwürmchen funktioniert nachrichtenbasiert: Immer wenn ein Glühwürmchen seine Phase von leuchtend auf nicht leuchtend oder umgekehrt wechselt, wird auf allen benachbarten Glühwürmchen die Methode `iAmFlashing(boolean isFlashing)` aufgerufen. Falls der Phasen-Zustand der beiden Glühwürmchen nicht übereinstimmt, wird die Phase des entsprechenden benachbarten Glühwürmchens um einen bestimmten Wert (`double coupling`) verschoben.

Die Main-Klasse erstellt ein Feld aus MxN Glührürmchen und visualisiert diese in einem JavaFX-Fenster. Dann werden mithilfe der `TorusTopology`-Klasse die Nachbarn jedes Glühwürmchens ermittelt und entsprechend in den `Firefly`-Objekten gesetzt.

Aufruf des Programms

Zum Bauen des Programms wurde das Build-Tool *Gradle* verwendet. Dieses muss nicht erst installiert werden, sondern kann direkt mithilfe des Gradle-Wrappers (`./gradlew` bzw für Windows `./gradlew.bat`) aufgerufen werden.

Folgender Befehl baut und startet das Programm:

```
./gradlew run
```

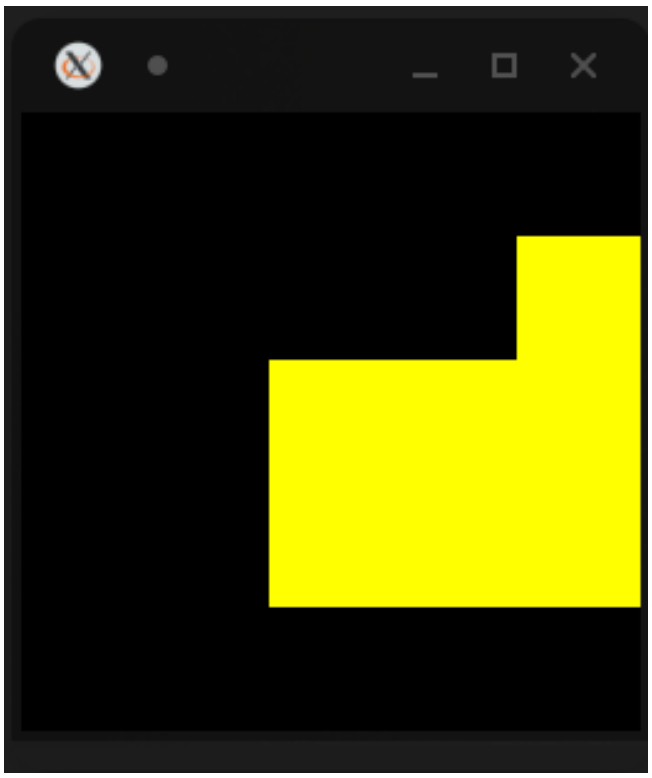

Abbildung 1: GUI der Threading-Implementierung

Besondere Vorkommnisse

Bei der Implementierung der Threading-Variante bin ich auf keine besonderen Schwierigkeiten gestoßen. Nachdem ich einmal verstanden habe, dass man den Algorithmus auch auf diese Weise (durch Benachrichtigung der Nachbarn beim Phasenwechsel) realisieren kann, war die Implementierung recht einfach.

Aufgabe 2 - Jedes Glühwürmchen ein Prozess (IPC / gRPC)

Deutlich komplexer wurde die Implementierung der Glühwürmchen mit Inter-Prozess-Kommunikation anstatt Multithreading. Hier mussten nicht nur zwei unabhängige Programme geschrieben werden (eines simuliert das Glühwürmchen, eines dient der Beobachtung), sondern es mussten noch einige zusätzliche Hürden überwunden werden:

- Kommunikation untereinander mit gRPC
- Dynamische Visualisierung der vorhandenen Glühwürmchen.
- Geordnetes Starten von Glühwürmchen-Prozessen, sodass diese einen Torus repräsentieren

Im Folgenden werde ich erklären, wie ich diese Hürden überwunden habe.

Kommunikation zwischen den Glühwürmchen

Für die Kommunikation untereinander habe ich mich für das Remote-Procedure-Call-Protokoll *gRPC* entschieden. Dieses ermöglicht den Aufruf von Methoden in Java-Programmen, die in anderen Prozessen oder gar auf anderen Maschinen laufen, da es auf dem TCP-Protokoll aufbaut.

Dadurch steht fest, dass jedes Glühwürmchen auf einem anderen Port auf die Anfragen anderer Glühwürmchen lauschen muss. Somit kann jedes Glühwürmchen eindeutig durch seine Server-Portnummer identifiziert werden. Bei der Initialisierung wird einem Glühwürmchen über Kommandozeilen-Argumente zuerst die eigene Port-Nummer übergeben, und dann alle Portnummern der Nachbar-Glühwürmchen.

Zur Kommunikation muss zunächst ein Prototyp-File erstellt werden, in dem die verfügbaren Methoden sowie Parameter- und Return-Typen definiert sind. Diese befindet sich im Verzeichnis `./src/main/proto` und dort wird eine Methode `notifyFirefly` mit Parameter-Typ `FireflyRequest` und Return-Typ `FireflyReply` definiert. Der `FireflyRequest` enthält die wichtigen Informationen:

- `bool isFlashing` zeigt an, ob das sendende Glühwürmchen gerade leuchtet oder nicht.
- `int32 port` enthält die Information, welchen Server-Port das sendende Glühwürmchen hat. Dadurch kann es vom Firefly-Observer identifiziert und dargestellt werden.

Der Return-Typ ist im Grunde irrelevant, muss aber definiert werden.

Die Kommunikation über gRPC habe ich in die Klassen `FireflyServer` und `FireflyClient` gekapselt, um davon zu abstrahieren und mich nirgendwo sonst mit den Eigenheiten von gRPC herumschlagen zu müssen. Die Klasse `FireflyServer` bekommt abgesehen von einem Port auch ein Objekt vom Typ `FireflyCallable` übergeben, welches eine Methode `flashStatusChanged()` enthält, die aufgerufen wird, wenn der Server einen Methodenaufruf empfängt.

Der `FireflyClient` bekommt einen Port übergeben, der den Server identifiziert, an den dieser Client senden soll. Außerdem stellt er eine Methode `notifyFirefly()` bereit, über die der entsprechende Server angefragt wird.

Jedes Glühwürmchen hat also einen `FireflyServer`, über den andere Glühwürmchen es erreichen können, und eine Anzahl $n > 0$ an `FireflyClients`, die die Nachbarn des Glühwürmchens repräsentieren. Die restliche Logik konnte weitestgehend aus Aufgabe 1 übernommen werden.

Visualisierung der Glühwürmchen

Auch wenn die UI, wenn man das so nennen kann, aus Aufgabe 1 bereits viel von dem hätte tun können, was hier gebraucht wird, sind doch noch einige Erweiterungen und Optimierungen nötig.

Zunächst einmal ist die Anzahl der Glühwürmchen nicht mehr fest durch das Programm definiert, sondern die Aktivität der Glühwürmchen muss beobachtet werden und entsprechend viele Glühwürmchen visualisiert werden. Zum Zwecke der Beobachtung bekommt auch der *Observer*, der die Visualisierung übernimmt, einen `FireflyServer`, an den Glühwürmchen ihre Statusinformationen senden können. Daraus kann dann der Observer eine Visualisierung der Glühwürmchen erstellen.

Da die Glühwürmchen auch immer ihre Portnummer mitsenden, kann der Observer diese anhand des Ports identifizieren. So kann er, wenn er die erste Nachricht von einem Glühwürmchen bekommt, entsprechend ein Quadrat anzeigen, das dieses repräsentiert. Bei darauffolgenden Nachrichten wird dann jeweils der Status des Quadrats aktualisiert.

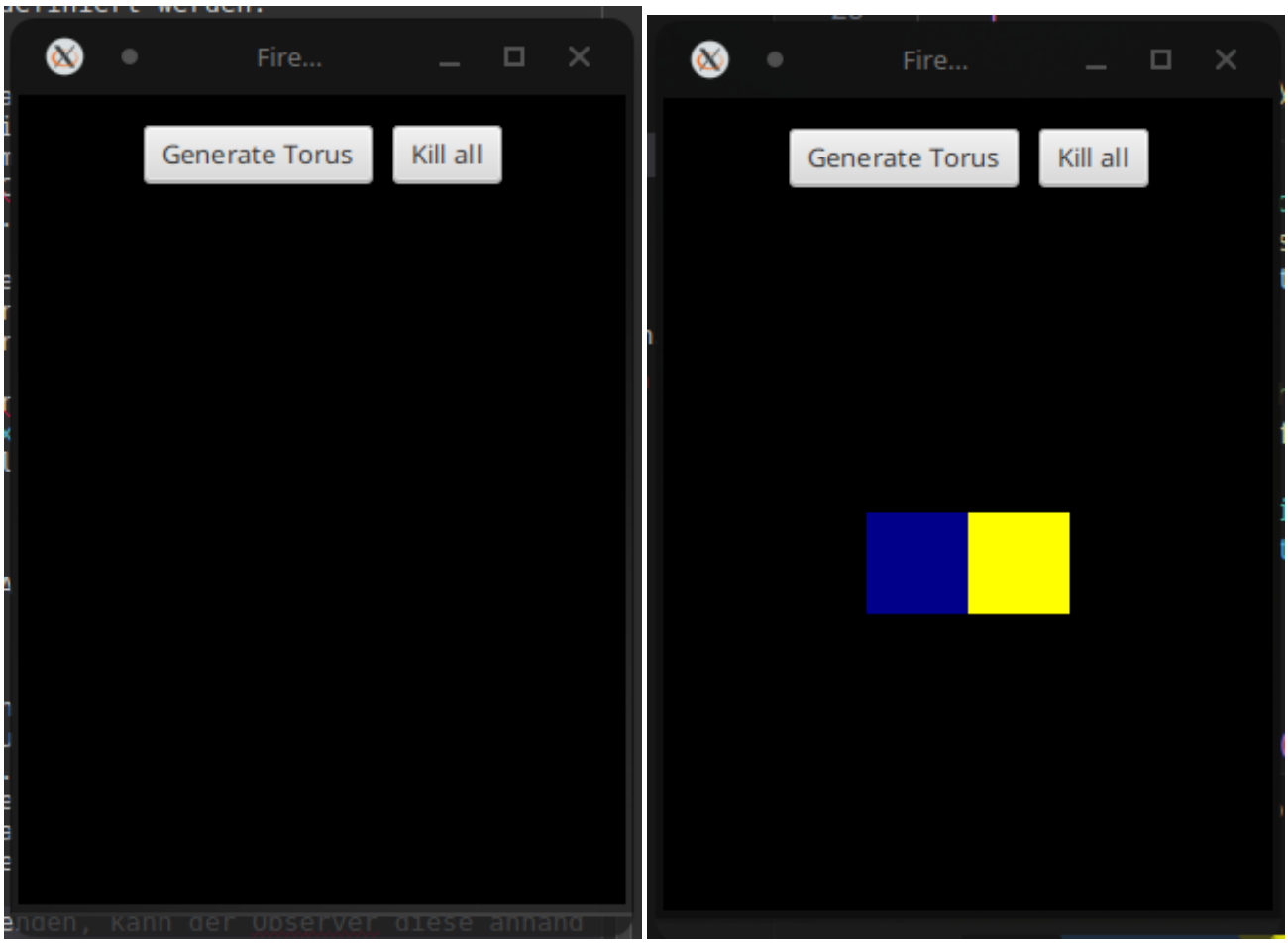

Abbildung 2: Observer-GUI mit und ohne aktive Glühwürmchen

Bei einer Anzahl von n Glühwürmchen werden diese in einem Quadrat der Kantenlänge $\lceil\sqrt{n}\rceil$ angeordnet, wobei das Quadrat von links nach rechts und von oben nach unten ausgefüllt wird. Damit man sieht, wo sich Glühwürmchen befinden, werden die nicht leuchtenden Glühwürmchen anders als in Aufgabe 1 dunkelblau anstatt schwarz dargestellt.

Erwähnenswert ist, dass die Glühwürmchen die Portnummer des Observers in der Liste ihrer Nachbarn haben müssen, damit sie die Statusmeldungen an den Observer senden und entsprechend dargestellt werden können.

Generierung und Visualisierung eines Torus

Eine weitere Herausforderung war die Koordination von vielen Glühwürmchen-Prozessen, um so einen Torus zu generieren, und diesen dann schlussendlich auch so zu visualisieren, wie die Glühwürmchen tatsächlich angeordnet sind.

Glücklicherweise bietet Java mit der Methode `Runtime.exec()` und der `Process`-Klasse gute Werkzeuge, um Prozesse zu verwalten. Gradle wurde so konfiguriert, dass beim Ausführen des `runObserver`-Tasks eine ausführbare Jar-Datei erstellt wird, die das Glühwürmchen enthält. Dieses kann dann über den Befehl `java -jar` direkt ausgeführt werden.

Um die vielen Prozesse zu verwalten, habe ich die Klasse `ProcessManager` angelegt. Diese bietet zunächst die grundlegenden Methoden `createFireflyProcess()` und `killProcesses()`, die

zum erstellen einzelner Glühwürmchen-Prozesse und zum killen aller zuvor erstellten Prozesse verwendet werden können.

Zusätzlich habe ich dort eine weitere Methode `createTorus()` hinzugefügt. Diese weist zunächst allen Glühwürmchen in einem MxN-Gitter einen Port zu und generiert mithilfe einer abgeänderten `TorusTopology`-Klasse aus Aufgabe 1 für jedes Glühwürmchen die Liste seiner Nachbarn (inklusive Port des Observers). Schlussendlich werden die Glühwürmchen-Prozesse gestartet.

Um die Glühwürmchen korrekt als Torus zu visualisieren, ist es nötig, dass diese im Gitter bereits erstellt wurden, bevor der Observer die erste Nachricht von ihnen bekommt: ansonsten würden sie in der Reihenfolge visualisiert werden, in der die erste Nachricht beim Observer ankommt. Zu diesem Zweck erhält der `ProcessManager` eine Referenz auf das Firefly-Gitter und fügt die generierten Glühwürmchen dort bereits hinzu, bevor die dazugehörigen Prozesse gestartet werden. Die Glühwürmchen sind dann zunächst weiß, bis schließlich der Observer Nachrichten von ihnen erhält und sie korrekt visualisiert.

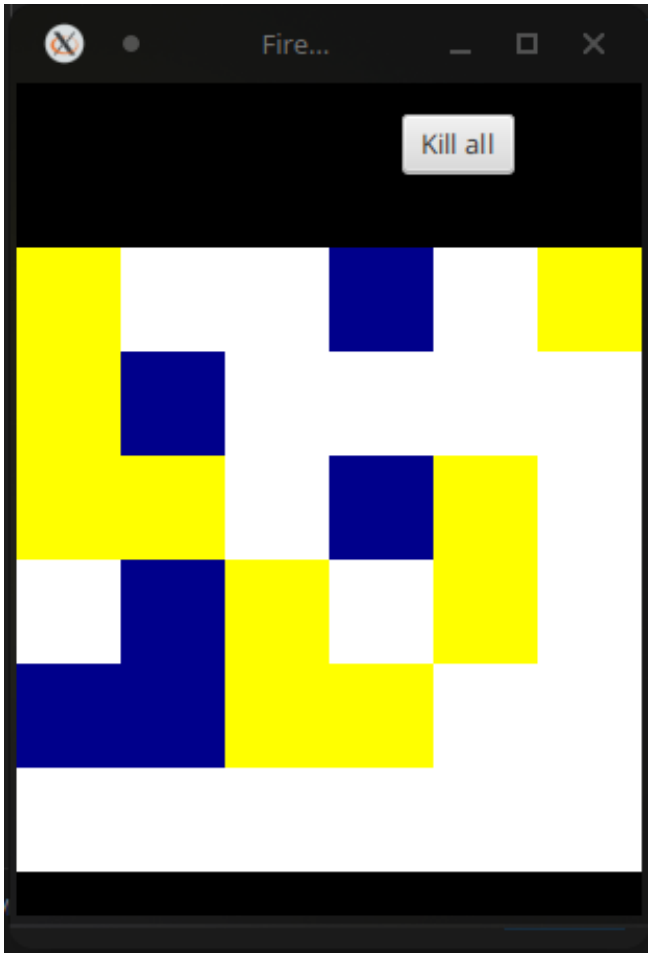

Abbildung 3: Erste Glühwürmchen im Torus werden vom Observer erkannt

Aufruf der Programme

Auch hier kam Gradle als Build-Tool zum Einsatz, wofür dank des Wrappers keine weitere Installation notwendig ist. Um den Observer zu starten, muss folgender Befehl verwendet werden:

```
./gradlew runObserver [observerPort (default 50051)]
```

Dieser Befehl kompiliert nicht nur den Observer und führt ihn aus, sondern kompiliert auch das Firefly-Programm und packt es in einer ausführbaren Jar-Datei. Diese kann dann beim Klick auf den *Generate Torus*-Button vom `ProcessManager` aufgerufen werden. Um diese Funktion aufrecht zu erhalten, sollte der Port auf dem Default belassen werden ;)

Will man dagegen nur das Firefly-Programm starten, kann dies auf zwei Wegen geschehen.

1. Direkt über Gradle mit dem Befehl
   `./gradlew runFirefly --args 'serverport [clientPorts...]'`
2. Alternativ kann zunächst die Jar-Datei erstellt werden über
   `./gradlew jar`
   Dann kann das Programm direkt aufgerufen werden über
   `java -jar ./build/libs/firefly-grpc.jar serverPort [clientPorts]`

Sonstige Schwierigkeiten

Lange Zeit habe ich mit Gradle gekämpft, bis es mir eine ausführbare Jar-Datei erstellt hat, die ich dann direkt über `Runtime.exec("java -jar ...")` ausführen konnte. Wenn ich über diese Methode versuche, den Prozess über Gradle zu starten (`gradle runFirefly`), schlägt dies aus mir nicht bekannten Gründen fehl. Letztendlich hat es aber über die Jar-Datei dann geklappt.

Ein Problem, das immer noch besteht, ist, dass im Torus manchmal Glühwürmchen einfach sterben, bzw. zumindest die Kommunikation mit dem Observer abbricht. Die Prozesse laufen dann zwar immer noch, aber die Glühwürmchen verharren in der Visualisierung in einem Zustand. Ich konnte das Problem an der Stelle noch nicht identifizieren.

Demo-Video

Das Demo-Video kann hier eingesehen werden: `https://youtu.be/WSfmdKwF6VY`