Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 13

TECHNISCHE UNIVERSITÄT DARMSTADT

Entwurf

Achtung: Dieses Dokument ist ein Entwurf und ist noch nicht zur Bearbeitung/Abgabe freigegeben. Es kann zu Änderungen kommen, die für die Abgabe relevant sind. Es ist möglich, dass sich **alle** Aufgaben noch grundlegend ändern. Es gibt keine Garantie, dass die Aufgaben auch in der endgültigen Version überhaupt noch vorkommen und es wird keine Rücksicht auf bereits abgegebene Lösungen genommen, die nicht die Vorgaben der endgültigen Version erfüllen.

Hausübung 13

Gesamt: 32 Punkte

Codecraft

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h13` und ggf. `src/test/java/h13`.

Verbindliche Anforderung für die gesamte Hausübung:

In dieser Hausübung sind alle Klassen aus der Java-Standardbibliothek erlaubt.

Einleitung

Im Jahr 2009 wurde das bahnbrechende Spiel *Minecraft*[1] auf den Markt gebracht. Es öffnete den Spielern eine schier unendliche Welt, ohne dabei übermäßige Ressourcen zu beanspruchen. Dies wurde durch die prozedurale Generierung der Spielwelt ermöglicht. Spiele wie *Minecraft* und *No Man's Sky*[2] setzen dazu auf komplexe Funktionen, deren Ergebnisse als vielfältige Elemente der Spielwelt interpretiert werden.

In dieser Aufgabe widmen wir uns einem prominenten Vertreter solcher Funktionen - dem *Perlin-Noise*[3], der 1982 von Ken Perlin für den Film *Tron*[4] entwickelt wurde.

Der Perlin-Noise erzeugt Pseudozufallszahlen, deren Generierung von bereits erzeugten Zahlen beeinflusst wird. Die Ergebnisse können beispielsweise als Höhenwerte einer Landschaft interpretiert werden und fließen nahtlos ineinander übe, anstatt - im Gegensatz zu herkömmlichen Zufallsgeneratoren - abruptes Rauschen zu erzeugen. Mit Perlin-Noise können unter anderem Texturen erstellt werden, die sowohl Struktur als auch zufällige Elemente vereinen, etwa in Form von Holzmaserungen.

[1] `https://www.minecraft.net/de-de`
[2] `https://www.nomanssky.com/`
[3] `https://de.wikipedia.org/wiki/Perlin-Noise`
[4] `https://de.wikipedia.org/wiki/Tron_(Film)`

Am Ende dieser Aufgabe werden wir ein Programm entwickeln, das es ermöglicht, mithilfe von Perlin-Noise und individuellen Konfigurationen eine „Welt“ zu generieren und darzustellen.

In der Abbildung 1 ist eine Vorschau der fertigen Anwendung zu sehen, das ähnlich zu Ihrem Ergebniss sein sollte. Auf der rechten Seite befinden sich die Konfigurationen für die Weltgenerierung und auf der linken Seite wird die generierte Welt dargestellt.

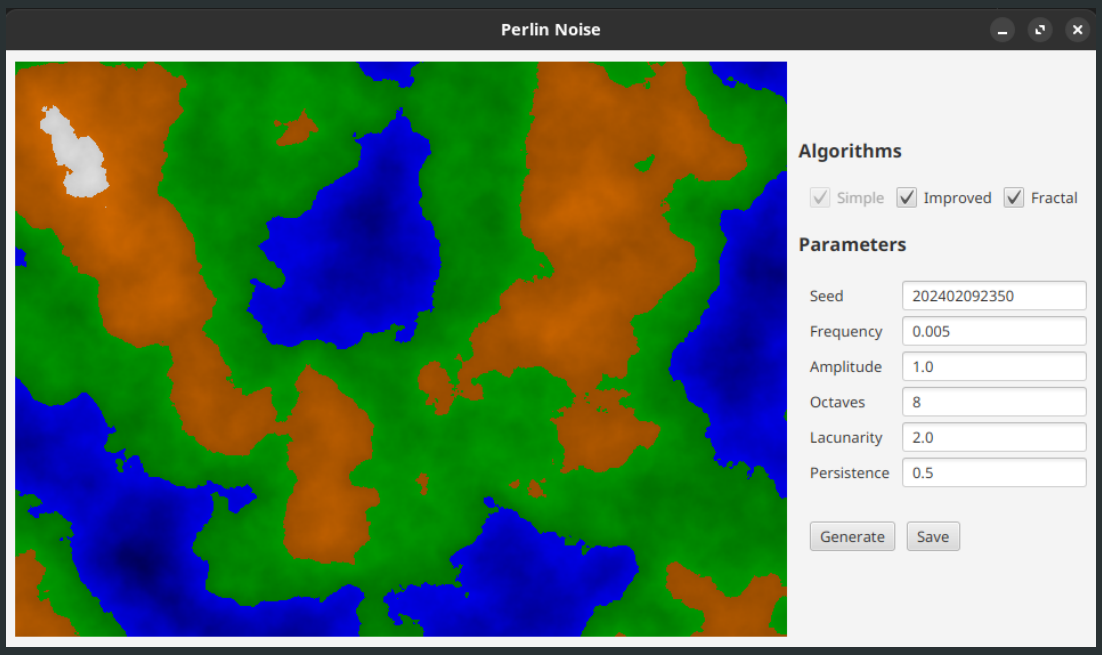


Abbildung 1: Vorschau der fertigen Anwendung auf Linux

Aufbau

Die Vorlage besteht aus zwei Hapt-Packages - `h13.noise` und `h13.ui`. Das Package `h13.noise` enthält die Implementierung der Perlin-Noise-Algorithmen. Wir werden in diesem Package drei verschiedene Perlin-Noise-Algorithmen implementieren.

Das Package `h13.ui` enthält die Implementierung der Benutzeroberfläche. Diese ist in drei Pakete unterteilt - `h13.ui.controls`, `h13.ui.layout` und `h13.ui.app`.

- `h13.ui.controls`: Dieses Package enthält die Implementierung des benutzerdefinierten `TextField` aus dem Package `javafx.scene.control`, die uns nur die Eingabe von Zahlen statt Text ermöglicht.
- `h13.ui.layout`: Dieses Package enthält die einzelnen visuellen Elemente der Benutzeroberfläche. Ein visuelles Element besteht aus einer `View` und einem `ViewModel`. Die `View` ist für die Darstellung der Benutzeroberfläche zuständig und das `ViewModel` für die Logik.
- `h13.ui.app`: Dieses Package enthält die konkrete Implementierung der Anwendung.

H1: Generating world... 7 Punkte

Wir beginnen mit der Erstellung des Grundgerüsts für unseren Algorithmus. Das Interface `PerlinNoise` im Package `h13.noise` stellt eine Funktion dar, die basierend auf Gitterkoordinaten einen Rauschwert erezugt und dient als Ausgangspunkt für die Implementierung verschiedener Perlin-Noise-Algorithmen.

Die relevanten Klassen befinden sich im Package `h13.noise`.

H1.1: Gradienten 3 Punkte

Um die Veränderungen der Rauschfunktion zu beschreiben, also wie sich der Rauschwert in benachbarten Bereichen ändert, verwenden wir sogenannte *Gradienten*. Ein Gradient ist in diesem Kontext ein Vektor im Einheitskreis, der dazu dient, die Richtung und Intensität der Veränderungen anzugeben. Diese Gradienten werden in einem Gitter gespeichert, das die Welt umspannt. Dies ist der Grund, warum Perlin-Noise als eine auf Gradienten basierende Rauschfunktion betrachtet wird.

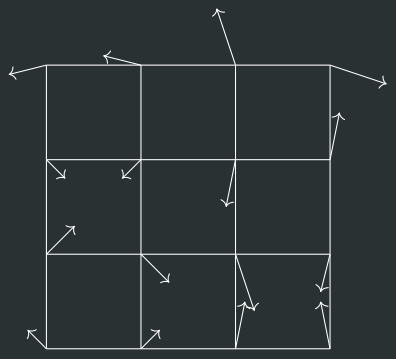


Abbildung 2: 4×4 Gitter mit Gradienten

In der Klasse `h13.noise.AbstractPerlinNoise` implementierenden Sie zwei Methoden, die Ihnen dabei helfen werden, die Gradienten zu berechnen.

Zuerst müssen wir einen zufälligen Gradienten im Einheitskreis erzeugen. Der Gradient $\vec{g} = \begin{pmatrix} x \\ y \end{pmatrix}$ mit $x, y \in [-1, 1]$ wird als `Point2D` dargestellt. Dafür implementieren Sie die Methode `createGradient()`, die einen zufälligen Gradienten zurückgibt und anschließend können wir mit Hilfe dieser Methode die Gradienten für das Gitter erzeugen. Dies erfolgt in der Methode `createGradients(int, int)`, die als Parameter die Weltgröße entgegennimmt und ein eindiemensionalen Array von Gradienten zurück. Die Größe des Arrays entspricht der Weltgröße. Da wir hier einen zweidimensionalen Array auf einen eindimensionalen Array abbilden, werden die Gradienten breitenweise gespeichert. Beispielsweise wäre das Array in einer 2×2 Welt foglendermaßen aufgebaut:

$$\{(0,0), (1,0), (1,0), (1,1)\}$$

Zum Schluss benötigen wir noch eine Funktion $g(x, y)$ mit $x, y \in \mathbb{Z}$, die uns zu einer gegebenen Position in der Welt den zugehörigen Gradienten findet. Dazu implementieren Sie die Methode `getGradient(int, int)`, die als Parameter die Weltkoordinaten entgegennimmt und den zugehörigen Gradienten zurückgibt.

> **Hinweis:**
>
> Beachten Sie bei der Implementierung von der Methode `getGradient(int, int)`, dass die Gradienten in einem Gitter gespeichert werden. Das Gitter umspannt dabei die Welt.

> **Verbindliche Anforderung:**
>
> Für die zufällige Generierung von Gradienten verwenden Sie das `java.util.Random`-Objekt aus der Klasse `AbstractPerlinNoise`.

H1.2: Lineare Interpolation und Fading-Funktion — 1 Punkt

Für die Berechnung des Rauschwerts benötigen wir eine Funktion $l(x_1, x_2, \alpha)$ mit $x_1, x_2, \alpha \in \mathbb{R}$, die es uns ermöglicht, zwischen zwei Werten zu interpolieren.

Implementieren Sie in der Klasse `h13.noise.SimplePerlinNoise` dazu die Methode `interpolate(double, double, double)`, die als Parameter zwei Werte x_1 und x_2 sowie einen Faktor α entgegennimmt und den interpolierten Wert zurückgibt.

$$l(x_1, x_2, \alpha) = x_1 + \alpha \cdot (x_2 - x_1)$$

Als zweite Funktion für die Berechnung des Rauschwerts benötigen wir eine sogenannte Fading-Funktion $f(t)$ mit $t \in \mathbb{R}$. Die Funktion f wird verwendet, um den Einfluss der Gradientenvektoren mit zunehmendem Abstand von der Eckposition zu reduzieren. Dieser Verblassungseffekt stellt sicher, dass der Gradienteneffekt näher an der Ecke stärker ist und sich beim Entfernen von der Ecke abschwächt.

Implementieren Sie dazu die Methode `fade(double)` in der Klasse `h13.noise.SimplePerlinNoise`.

$$f(t) = 6t^5 - 15t^4 + 10t^3$$

H1.3: Simpler Perlin-Noise Algorithmus — 3 Punkte

Nun haben wir alle Bausteine, um den simplen Perlin-Noise Algorithmus zu implementieren. Gegeben sei $(x, y)_{x,y \in \mathbb{R}}$ eine Position in einem Gitter. Wir berechnen zunächst die Eckpositionen $(x_0, y_0)_{x_0, y_0 \in \mathbb{Z}}$ und $(x_1, y_1)_{x_1, y_1 \in \mathbb{Z}}$ des Gitters, in dem sich (x, y) befindet.

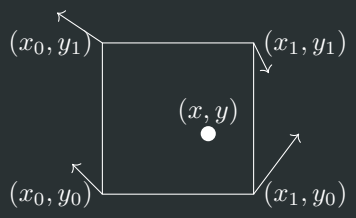


Abbildung 3: Punkt (x, y) und seine zugehörigen Gradienten an den Eckpositionen

Haben wir dies getan, können wir die vier Gradienten $\vec{g}_{(x_i, y_j)}$ mit $i, j \in \{0, 1\}$ bestimmen. Mit den Eckkoordinaten können wir die Distanzvektoren $\vec{d}$ zwischen dem Punkt (x, y) und der Gradienten-Ecke (x_0, y_0) berechnen.

Dann berechnen wir das Skalarprodukt zwischen dem Distanzvektor und dem Gradientenvektor der jeweiligen Ecke (refeq:scalar-products) und interpolieren die Ergebnisse des Skalarprodukts mit der Interpolationsfunktion. Die Parameter der Interpolationsfunktion sind die Skalarprodukte der gleichen x-Koordinate des Distanzvektors mit dem geglätteten $\vec{d}_x$ (2).

$$\begin{aligned} s_{(x_0,y_0)} &= (\vec{d} + (0,0)^T) \cdot \vec{g}_{(x_0,y_0)} \\ s(x_0, y_1) &= (\vec{d} + (0,-1)^T) \cdot \vec{g}_{(x_0,y_1)} \\ s(x_1, y_0) &= (\vec{d} + (-1,0)^T) \cdot \vec{g}_{(x_1,y_0)} \\ s(x_1, y_1) &= (\vec{d} + (-1,-1)^T) \cdot \vec{g}_{(x_1,y_1)} \end{aligned} \quad (1)$$

$$\begin{aligned} lx_0 &= l(s_{(x_0,y_0)}, s_{(x_0,y_1)}, f(\vec{d}_x)) \\ lx_1 &= l(s_{(x_1,y_0)}, s_{(x_1,y_1)}, f(\vec{d}_x)) \end{aligned} \quad (2)$$

Schließlich interpolieren wir die beiden interpolierten Werte erneut mit der Interpolationsfunktion. Als α verwenden wir die geglättete $\vec{d_y}$.

$$l = f(lx_0, l_{x_1}, f(\vec{d_y})) \tag{3}$$

Hinweis:

Sie können die Methode `javafx.geometry.Point2D#dotProduct(javafx.geometry.Point2D)` verwenden, um das Skalarprodukt zwischen zwei Vektoren zu berechnen.

H2: Erweiterte Perlin-Noise Algorithmen — 5 Punkte

In der H1 haben wir den simplen Perlin-Noise Algorithmus implementiert. In dieser Aufgabe werden wir weitere Perlin-Noise Algorithmen implementieren, die etwas komplexer sind als der simple Perlin-Noise Algorithmus.

Die relevanten Klassen befinden sich im Package `h13.noise`.

H2.1: Permutationstabelle — 2 Punkte

Die verbesserte Version von dem simplen Perlin-Noise Algorithmus verwendet eine Permutationstabelle und unterscheidet sich nur bei der Auswahl von Gradienten - Ecken.

Hier werden die Gradienten anhand einer Permutationstabelle $p(i)_{i\in[0,2\cdot256]}$ ausgewählt, die zufällig generiert wird. Die Permutationstabelle ist $2 \cdot 256$ groß und die ersten 256 sind aufsteigend geordnet. Die zweite Hälfte der Tabelle ist eine Kopie der ersten Hälfte, die zufällig permutiert wurde. Dies ist nur eine von vielen Möglichkeiten, eine Permutationstabelle zu generieren. Einer der klassischen Anordnung der Permutationstabelle finden Sie auf `https://en.wikipedia.org/wiki/Perlin_noise#Permutation`.

Implementieren Sie in dieser Aufgabe die Methode `createPermutation(Random)`. Die Methode befindet sich in der Klasse `h13.noise.ImprovedPerlinNoise`, nimmt als Parameter ein `java.util.Random`-Objekt entgegen und gibt die Permutationstabelle zurück. Die Anordnung der Elemente in der Permutationstabelle entnehmen Sie aus dem vorherigen Absatz.

Haben Sie dies getan, müssen wir nur noch die Methode `getGradient(int, int)` anpassen, um die Gradienten anhand der Permutationstabelle auszuwählen. Die Auswahl des Gradienten erfolgt anhand folgenden Schema für eine Permutationstabelle der Größe $2 \cdot 256$:

$$g(x, y) = p((x + p(y \ \& \ 255)) \ \& \ 255) \tag{4}$$

Das kaufmännisches Und-Zeichen stellt das das „bitweise Und“ dar.

Verbindliche Anforderungen:

(i) Verwenden Sie das `java.util.Random` Objekt von dem übergebenen `PerlinNoise`-Objekt für die zufällige Anordnung der zweiten Hälfte der Permutationstabelle.

(ii) Verwenden Sie die Konstante `PERMUTATION_SIZE` für die Größe der Permutationstabelle. Ggf. müssen Sie Ihre Berechnung anpassen, falls Sie eine andere Größe verwenden.

H2.2: Delegation 1 Punkt

Die nächsten zwei Implementierung von Perlin-Noise verwenden einen gebenen Perlin-Noise Algorithmus als Basis und erweitern diesen um zusätzliche Funktionen oder Berechnungen. Damit wir nicht jedes Mal alle Methoden neu implementieren müssen, verwenden wir hier das Prinzip eines *Delegation*[5].

Der Konstruktor von `h13.noise.DelgatePerlinNoise` erhält als Parameter ein `PerlinNoise`-Objekt als Delegaten (Basis Algorithmus) und leitet alle Methodenaufrufe von der Klasse `DelgatePerlinNoise` an den Delegaten weiter.

Delegieren Sie in dieser Aufgabe alle Implementierungen der Methoden des Interfaces `h13.noise.PerlinNoise` in der abstrakten Klasse `h13.noise.DelgatePerlinNoise` an den Delegaten weiter.

H2.3: Normalisierung 1 Punkt

Damit die Rauschwerte des Perlin-Noise-Algorithmus für verschiende Anwendungszwecke optimal genutzt werden können, implementieren wir einen Normalisierungs-Algorithmus, der einen Perlin-Noise-Algorithmus vom Werte-Bereich $[-1, 1]$ in den Wertebereich $[0, 1]$ transformiert.

Implementieren Sie dazu die Methoden in der Klasse `h13.noise.NormalizedPerlinNoise`.

(a) Vor Normalisierung (b) Nach Normalisierung

Abbildung 4: 10 zufällige Gradienten im Koordinatensystem vor und nach der Normalisierung

H2.4: Fraktale 1 Punkt

In der Realität sieht die Welt nicht zu glatt aus und bewirkt einen unrealistischen Eindruck. Die Geländeformen in der realen Welt sind deutlich rauer. Um dies zu erreichen, müssen wir Rauschen hinzufügen. Hier kommen die Fraktale ins Spiel, die wir in der Methode `compute(double, double)` von der Klasse `h13.noise.FractalPerlinNoise` implementieren. Wir starten bei einer Grundfrequenz f_0 und -amplitude a_0, die Sie aus den Attributen der Klasse entnehmen können. Die Berechnung des Rauschwertes $N(x, y)$ erfolgt wie folgt:

$$N(x,y) = \sum_{i=0}^{o-1} U(x \cdot f_i, y \cdot f_i) \cdot a_i \tag{5}$$

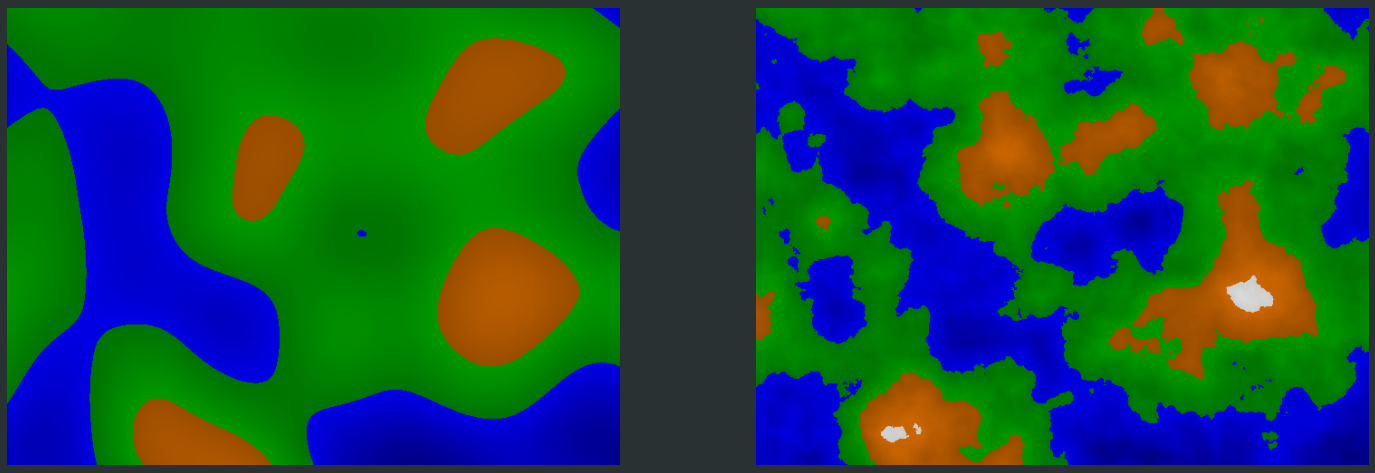
$$f_i = f_{i-1} \cdot l \tag{6}$$

$$a_i = a_{i-1} \cdot p, \tag{7}$$

wobei l für `lacunarity`, p für `persistence` und $U(x, y)$ für den Basialgorithmus stehen.

[5] https://de.wikipedia.org/wiki/Delegation_(Softwareentwicklung)

In der Abbildung 5 können Sie die Unterschiede zwischen dem simplen Perlin-Noise und dem Fraktalen Perlin-Noise sehen. Man kann deutlich erkennen, dass der Fraktale Perlin-Noise deutlich rauer ist als der simple Perlin-Noise.



(a) Simpler Perlin-Noise (b) Fraktal Perlin-Noise

Abbildung 5: Vergleich Perlin-Noise Algorithmus

H3: Controls 5 Punkte

In JavaFX gibt es Textfelder, die es ermöglichen, Texte einzugeben. In unseren Fall wollen wir aber nur Zahlen eingeben, weshalb wir unseren eigenen Textfeld implementieren, das nur Zahlen akzeptiert.

Für diese Aufgabe befinden sich alle Klassen im Package `h13.ui.controls`.

H3.1: Converter 2 Punkte

Damit unser Textfeld nur Zahlen akzeptiert, müssen wir einen `javafx.util.StringConverter` implementieren, der die Eingabe in eine Zahl umwandelt. Die Klasse `StringConverter` bietet zwei Methoden an, um einen String in Typ `T` bzw. in unseren konkreten Fall `Number` umzuwandeln und vice versa.

Implementieren Sie dazu die Klasse `NumberStringConverter` die Methode `toString(Number)`, die einen Zahl in einen `String` mittels dem `stringifier` zurückgibt. Falls die Eingabe `null` ist, so wird ein leerer String zurückgegeben.

Haben Sie dies getan, implementieren Sie anschließend die Methode `fromString(String)`, die einen `String` in eine Zahl mittels dem `numericizer` umwandelt. Falls die Eingabe `null` oder leer ist, so wird `null` zurückgegeben. Ansonsten wird der `String` in eine Zahl umgewandelt. Beachten Sie, dass die Eingabe `"-"` als `"-1"` interpretiert wird.

H3.2: Nummerfeld 3 Punkte

Mittels dem Konverter aus H3.1 können wir nun unsere eigene Textfelder implementieren, das nur Zahlen akzeptiert. Dafür implementieren Sie die Methode `initBindings()` in der Basisklasse `NumberField`, die ein `TextField` erweitert und die Eingabe auf Zahlen beschränkt.

Das Attribut `value` muss zu jedem Zeitpunkt den Wert des Textfeldes als Zahl repräsentieren. Dazu muss eine Benachrichtigung an `value` gesendet werden, wenn sich der Wert des Textfeldes ändert und mittels dem `getConverter()` wird der Wert des Textfeldes in eine Zahl umgewandelt und vice versa.

Zum Schluss können wir in den Klassen `IntegerField`, `LongField` und `DoubleField` die abstrakte Methode `getConverter()` aus der Basisklasse implementieren. Die Methode soll uns einen passenden Konverter für den jeweiligen Typ zurückgeben. Den Typ können Sie aus dem Klassennamen entnehmen. Verwenden Sie hierzu die Klasse `NumberStringConverter` aus H3.1.

H4: Konfigurations-Menü 7 Punkte

Bevor wir den Perlin-Noise Algorithmus visualisieren können, benötigen wir eine Möglichkeit, den Algorithmus zu konfigurieren, bspw. Parameter und die Art des Algorithmus.

Für diese Aufgabe befinden sich alle Klassen im Package `h13.ui.layout`.

H4.1: Algorithmusauswahl 1 Punkt

Wir möchten einer Person ermöglichen, einen Perlin-Noise Algorithmus auszuwählen. Dazu verwenden wir die Klasse `h13.ui.layout.ChooserView`.

Die Auswahlmöglichkeiten werden in einer `javafx.collections.ObservableMap` gespeichert mit jeweils den Namen des Algorithmus als Schlüssel und die `javafx.scene.control.CheckBox` und soll bei einer Veränderung der Map die Auswahlmöglichkeiten in der Benutzeroberfläche aktualisiert werden. Implementieren Sie die Methode `initialize()`, die die Auswahlmöglichkeiten in der Benutzeroberfläche aktualisiert, falls sich die `Map` ändert. Das heißt, falls ein Element aus der `Map` eingefügt oder entfernt wird, soll das Element in `root` eingefügt werden.

Die Position im `javafx.scene.layout.GridPane` wird mittels der Attribute `nextColumn` und `nextRow` bestimmt, wobei unsere `ChooserView` eine fixe Spaltenanzahl (`columnSize`) besitzt. Falls die Spaltenanzahl überschritten wird, soll die nächste Zeile verwendet werden.

Verbindliche Anforderung:

Aktualisieren Sie die Attribute `nextColumn` und `nextRow` bei einer Änderung der `Map`.

Hinweis:

Eine `ObservableMap` ist eine `Map`, die es ermöglicht, Änderungen an der Map zu beobachten. Mittels der Methode `addListener(MapChangeListener)` können wir eine `MapChangeListener` hinzufügen, der bei einer Änderung der Map aufgerufen wird.

Schauen Sie sich am besten das funktionale Interface `javafx.collections.MapChangeListener` an, die Ihnen hilfreiche Methoden zur Verfügung stellt.

H4.2: Parameter 1 Punkt

Analog zu H4.1 soll es einer Person möglich sein, die Parameter des Perlin-Noise Algorithmus auszufüllen. Dazu verwemdem wir die Klasse `h13.ui.layout.ParameterView`.

Die Nummernfelder werden in einer `javafx.collections.ObservableMap` gespeichert mit jeweils den Namen des Parameters und das zugehörige Label und Nummernfeld.

Implementieren Sie die Methode `initialize()`, die die Parameterauswahl in der Benutzeroberfläche aktualisiert, falls sich die `Map` ändert. Das heißt, falls ein Element aus der `Map` eingefügt oder entfernt wird, soll das Element in `root` eingefügt werden.

Die Position im `javafx.scene.layout.GridPane` wird mittels des Attributes `nextRow` bestimmt. Das `Label` und `NumberField` werden auf derselben Zeile eingefügt.

Verbindliche Anforderung:

Aktualisieren Sie das Attribut `nextRow` bei einer Änderung der `Map`.

H4.3: Welche Parameter für welchen Algorithmus? 4 Punkte

Unser simplen Perlin-Noise Algorithmus benötigt nur einen Startwert (`seed`) und die Frequenz (`frequency`). Der fraktale Perlin-Noise Algorithmus benötigt zusätzlich noch die Amplitude (`amplitude`), die Lacunarity (`lacunarity`) und die Persistence (`persistence`).

Das heißt, wir müsssen die Parameterauswahl passend zum ausgewählten Algorithmus verfügbar machen. Dies wird mittels der Methode `addVisibilityListener(Map<String, Set<String>> configurations)` in der Klasse `h13.ui.layout.SettingsViewModel` realisiert, die Sie implementieren werden.

Die Methode erhält als Parameter eine `Map` mit den Namen des Algorithmus als Schlüssel und die Namen der Parameter, die für den jeweiligen Algorithmus benötigt werden. Die Auswahl eines Algorithmus wird im Attribut `options` gespeichert und die Sichtbarkeiten der Parameter werden im Attribut `parameters` gespeichert.

Die Aufgabe von dem Listener ist es einen `BooleanBinding` für alle Parameter zu erstellen, das angibt, wann ein Parameter aktiviert bzw. deaktiviert werden soll. Haben Sie die `BooleanBinding`s korrekt erstellt, so müssen Sie diese an die Sichtbarkeit Eigenschaft (`BooleanProperty`) der jeweiligen Parameter anbinden.

Hinweise:

(i) Speichern Sie die einzelnen `BooleanBinding`s in einer `Map` mit dem Namen des Parameters als Schlüssel, damit Sie die `BooleanBinding`s später an die Sichtbarkeit Eigenschaft anbinden können.

(ii) Die Methode `Bindings#createBooleanBinding(Callable, Observable...)` ermöglicht es Ihnen, einen `BooleanBinding` zu erstellen, der sich automatisch aktualisiert, falls sich die übergebenen `Observable`s ändern. Initialisieren Sie diesen mit einem `BooleanProperty` mit dem Startwert `false`.

(iii) Sie können nun den `BooleanBinding` anpassen, dass dieser den Wert `true` zurückgibt, falls der Algorithmus ausgewählt ist und der Parameter in der Liste der benötigten Parameter enthalten ist.

H4.4: Konfigurations - Ansicht 1 Punkt

N Nun haben wir alle Bausteine, um die Konfigurationen in der Benutzeroberfläche zu integrieren. Dafür implementieren Sie die Methode `initialize()` in der Klasse `h13.ui.layout.SettingsView`.

Die Methode soll zuerst die Algorithmenauswahl und dann die Parameterauswahl in die `GridPane` einfügen, wobei der Label zuerst und anschließend die Ansicht für die Auswahl eingefügt werden soll.

Zum Schluss soll die Button-Gruppe (`buttonGroup` eingefügt werden. In die Button-Gruppe müssen Sie die `Button` `generate` und `save` einfügen, wobei der `generate` Button links und der `save` Button rechts eingefügt werden soll.

Jetzt fehlt nur noch die Sichtbarkeit der Parameter zu den zugehörigen Algorithmen. Rufen Sie dazu die Methode `addVisibilityListener(Map<String, Set<String>> configurations)` von dem `viewModel` auf und übergeben Sie die `Map configurations`.

H5: Algorithmus-Ansicht 6 Punkte

Jetzt fehlt uns nur noch die Ansicht für die Visualisierung des Perlin-Noise Algorithmus. Alle relevanten Klassen befinden sich im Package `h13.ui.layout`.

Die Klasse `AlgorithmViewModel` fungiert als Basisklasse, um verschiedene Perlin-Noise-Algorithmen auf einem `Canvas` darzustellen. Sie bietet Funktionen zur Generierung und Darstellung der Welt. Die spezifischen verwendeten oder verfügbaren Algorithmen werden in den Unterklassen festgelegt.

H5.1: P-I-C 3 Punkte

Die Klasse `AlgorithmViewModel` dient als Basisklasse für die Darstellung verschiedener Perlin-Noise-Algorithmen auf einem `Canvas`. Sie stellt Funktionen bereit, um die Welt zu generieren und anzuzeigen. Die konkreten Algorithmen werden in den Unterklassen festgelegt.

Ihre Aufgabe ist es, die Methode `createImage` zu implementieren. Diese Methode nimmt einen Algorithmus als Eingabe und erzeugt ein Bild von den Koordinaten (x, y) bis $(x + w, y + h)$, basierend auf den Startkoordinaten und der Größe.

Beginnen Sie damit, ein `javafx.scene.image.WritableImage` mit den Abmessungen (w, h) zu erstellen. Dieses Bild wird zur Zeichnung verwendet und am Ende zurückgegeben. Um auf dem Bild zu zeichnen, benötigen Sie einen `PixelWriter`, den Sie über die Methode `getPixelWriter()` von `WritableImage` erhalten. Verwenden Sie die Methode `setColor(int, int, Color)`, um die Farbe eines Pixels festzulegen. Zeichnen Sie dann die (Teil-)Welt und verwenden Sie die `colorMapper`-Funktion, um aus einem Rauschwert die entsprechende Farbe zu generieren.

Die Methode `draw` verwendet den übergebenen Algorithmus und zeichnet diesen auf dem `Canvas` über den `GraphicsContext`. Falls der Algorithmus `null` ist, gibt es nichts zu tun. Wenn der Algorithmus nicht normalisiert ist, normalisieren Sie ihn. Speichern Sie nach jedem Wechsel des Algorithmus den aktuellen Algorithmus in der Variable `lastAlgorithm` und zeichnen Sie die (Teil-)Welt mit der Methode `drawImage` von `GraphicsContext`.

Unbewertete Verständnisfrage:

Wir haben für die Parameterwerte nur eine untere Schranke definiert. Je nachdem, wie wir die Parameter wählen, könnte es bei der Berechnung und Konvertierung des Rauschwertes vorkommen, dass die Werte außerhalb des Farbraums liegen.

(i) Was würde in diesem Fall passieren?

(ii) Wie könnte man dies in JavaFX der nutzenden Person mitteilen? Sie müssen dies nicht implementieren, können dies aber gerne tun. Dies würde aber nicht bewertet werden. Achten Sie darauf, dass die Implementierung trotzdem für valide Parameterwerte funktionieren muss.

H5.2: Zoom in, zoom out — 3 Punkte

Wir haben nun eine Möglichkeit einen Bild zu generieren bzw. speichern und darzustellen. Was uns jetzt fehlt ist diese Aktion an den `Button` zu binden, damit dieser den aktuell ausgewählten Algorithmus (`getAlgorithm()`) auf der kompletten Größe des `Canvas` zeichnet.

Implementieren Sie dazu die Methode `initializeButtons()` und `initializeSize()` in der Klasse `h13.ui.layout AlgorithmView`.

Die Methode `initializeButtons()` soll beim Drücken des `Buttons generate` das Bild mittels der Methode `draw(PerlinNoise, GraphicsContext, int, int, int, int)` auf die Zeichenfläche zeichnen. Wählen Sie hierzu den aktuellen Algorithmus aus und zeichnen Sie über die komplette verfügbare Fläche des `Canvas`.

Falls der `Button save` gedrückt wird, soll das Bild mittels der Methode `save(int, int)` mit der Größe der Zeichenfläche gespeichert werden.

Zum Schluss müssen wir noch angeben, was passieren soll, wenn sich die Größe des `Canvas` ändert bzw. des `BorderPanes`.

Die Zeichenfläche ist immer so groß wie die verfügbare Fläche des `BorderPanes` minus die Fläche für die Konfigurationen. Falls sich die Zeichenfläche ändert, soll mittels der Methode `draw(PerlinNoise, GraphicsContext, int, int, int, int)` die **neue** Zeichenfläche gezeichnet werden.

Hinweise:

(i) Die `GraphicsContext2D` erhalten Sie mittels der Methode `getGraphicsContext2D` von Canvas.

(ii) Beachten Sie bei der Breite die `SettingsView` und bei der Höhe das `Padding` des `BorderPanes`.

H6: App 2 Punkte

Die Anwendung ist fast abgeschlossen. Es fehlt nur noch die Implementierung der Methode `getAlgorithm`.

Zunächst überprüfen wir den `cacheSimpleNoise`, um festzustellen, ob bereits ein Algorithmus mit dem gleichen `seed` gespeichert ist. Wenn sich kein Algorithmus im Cache befindet, erstellen wir einen neuen Algorithmus und speichern ihn im Cache, zusammen mit den aktuellen Display-Abmessungen, der Frequenz und dem Seed.

Als nächstes prüfen wir, ob der improvisierte Algorithmus ausgewählt wurde. Wenn ja, suchen wir im Cache `cacheImprovedNoise`, ob bereits ein entsprechender Algorithmus gespeichert ist. Wenn nicht, erstellen wir einen neuen Algorithmus und speichern ihn im Cache.

Zum Schluss überprüfen wir, ob der fraktale Algorithmus ausgewählt wurde. Falls ja, erstellen wir einen neuen fraktalen Algorithmus mit den angegebenen Parametern: Amplitude, Octaves, Lacunarity und Persistence.

Wenn der ausgewählte Algorithmus dem zuletzt ausgeführten Algorithmus entspricht und die Frequenz gleich ist, geben wir `null` zurück, um keinen neuen Algorithmus zu zeichnen. Andernfalls aktualisieren wir den zuletzt ausgeführten Algorithmus auf den aktuellen Algorithmus und setzen die Frequenz entsprechend.

Haben Sie dies getan, so können Sie die Anwendung ausführen und die verschiedenen Algorithmen ausprobieren.

Hinweise:

1. Mittels `computeIfAbsent` können Sie einen Wert aus einer Map abrufen. Falls der Wert nicht vorhanden ist, wird der Wert mit der übergebenen Funktion berechnet und zurückgegeben.
2. Schauen Sie sich die Enums `Parameter` und `Algorithm` an bw. ebenfalls die Methoden `getParameter` und `getOption` an, um die Parameter und Algorithmen zu erhalten.
3. Die Display-Abmessungen können über `Screen.getPrimary().getVisualBounds()` abgerufen werden.