

Prozedurale Generierung von Landschaften

Praxisprojekt
im Studiengang Allgemeine Informatik (B.A.)
an der TH Köln

vorgelegt am: 09.06.2021
von: Luca Stamos und Stefan Steinhauer
aus: Köln, Gummersbach
Gutachter: Prof. Köhler

Abstrakt

Die prozedurale Generierung ist ein Gebiet der Informatik, welchem kürzlich viel Aufmerksamkeit zugetragen wurde. Dies ist unter Anderem auf Grund von beliebten Videospielen wie 'No Man's Sky' oder 'Minecraft' der Fall. Die Herausforderung vieler Entwickler besteht nun darin die vielfältigen Variablen und Stellschrauben auf eine einzigartige und gut aussehende Art und Weise einzustellen, so dass Spieler gefesselt werden können. Darüber hinaus ist es eine der wenigen Methoden, die unendlich große Spielwelten erzeugen kann ohne viel Aufwand auf Seiten der Entwickler zu beanspruchen und kann so, wenn gut angewendet, zu wortwörtlich grenzenlosem Spielspaß führen, was auch große Erfolge wie der von Minecraft beweisen.

Das Ziel der vorliegenden Arbeit ist es, eine gut aussehende Landschaft zu erzeugen, welche potenziell unendlich fortlaufend generiert wird. Zusätzlich soll der Workload so designt werden, dass er praktisch parallelisiert werden kann. Dieses Projekt soll ein Proof of Concept für die nachfolgende Arbeit darstellen, in der eine Verteilung auf mehrere Maschinen erreicht werden soll.

In der Arbeit hat sich gezeigt, dass selbst mit dem Aufwand von nur zwei Studenten eine unendlich fortlaufende, angemessen gut aussehende Landschaft generiert werden kann, welche die Grundlage für ein Spiel liefern könnte was selbst für ein erfahrenes Entwicklerteam bei manueller Arbeit mit der klassischen Design-Methode allein auf Grund der Größe einen wortwörtlich unendlichen Aufwand bedeuten würde.

Vorwort

Wir haben uns dazu entschieden unser eigenes Projekt zu verwirklichen anstatt in einer Firma eine praktische Arbeit zu unternehemen, weil dieses Thema schon vor dem Start des Projekts unser Interesse geweckt hatte. Uns fasziniert die Mathematik und die Algorithmen hinter dem Thema und dass zum Teil bloßes mathematisches Verständnis ausreicht um die manuelle Arbeit von ganzen Entwicklerteams zu ersetzen. Und nicht zuletzt verbindet uns eine gemeinsame Leidenschaft für Videospiele und virtuelle Welten. Wir benötigten nur noch einen begleitenden Professor, den wir mit Herrn Köhler nicht besser hätten finden können und wir möchten ihm an dieser Stelle nochmal ausdrücklich danken. Einige der verwendeten Fachbegriffe in dieser Arbeit stammen aus dem Buch von Tanya X. Short und Tarn Adams - 'Procedural Generation in Game Design' [3]. Die bei weitem größte Inspiration dieses Projekt anzufangen sind die vielen hervorragenden Videodokumentationen der Arbeit von Sebastian Lague [2].

Inhaltsverzeichnis

Abbildungsverzeichnis	III
Symbolverzeichnis	V
1 Einleitung	1
1.1 Architektur	2
1.2 Vorgehensweise	4
2 Heightmaps und Perlin Noise	4
3 Funktionalität	6
3.1 Erstellung der Landkarte	6
3.2 Verrechnung mehrerer Heightmaps	6
3.3 Bewegung auf der Landkarte	7
3.4 Verbesserung der Bewegungsmöglichkeit	8
3.5 Verwendung von Presets	8
4 User Interface	9
5 Mesh Generator	12
6 Verteilung von Objekten	13
6.1 3D Modelle	14
6.2 Wasser	16
7 Parallelisierung	18
7.1 Komplexität	18
7.2 Compute Shader und Threading (Unity Job System)	18
8 Ergebnisse	19
9 Erfahrung	21
9.1 Der Weg war lang	21
9.2 Debugging	39
9.3 Nice to Have	39
10 Code	40
11 Zusammenfassung und Ausblick	52
12 Literaturverzeichnis	VI
A Anhang	1
A.1 Programmcode	1

Abbildungsverzeichnis

1	Use Case Diagramm	3
2	Klassendiagramm	3
3	Visualisierung der Perlin Noise Funktion	5
4	Ansicht des User Interface (Version 8)	9
5	Modell eines Löwenzahns, auch 'Pusteblume' genannt	14
6	3D-Modell einer Sonnenblume	14
7	3D-Modell eines Baums (Version 3)	15
8	verschiedene Pflanzen in einer Landschaft	15
9	Landschaft mit Wasser Version 1	16
10	Landschaft mit Wasser Version 2	17
11	Vertex Shader Version 4	17
12	Landschaft Beispiel 1	19
13	Landschaft Beispiel 2	19
14	Landschaft Beispiel 3	20
15	Landschaft Beispiel 4	20
16	Erste generierte Landschaft	22
17	Erste kombinierte Heightmap	23
18	User Interface Version 1	24
19	Versuche mit Grobkörnigkeit 1	25
20	Versuche mit Grobkörnigkeit 2	25
21	RAM-Überflutung bei Bewegung über die Karte	26
22	Landschaft in Farbe Version 1	26
23	Mesh Shader Normalmap	27
24	Landschaft mit Skybox und Normalmap	27
25	Compute Shader Fail	28
26	Threading funktioniert, aber hohe Auslastung	29
27	Objekte werden in falscher Höhe platziert	29
28	Objekte werden in korrekter Höhe platziert	29
29	Erstes Modell einer Pusteblume	30
30	Erfolgreich die erste Pflanze platziert	30
31	Pusteblume Version 2	30
32	Pusteblume Invasion	31
33	Erste sinnvolle Verteilung	31
34	Erste Versuche einer alternativen Verteilung	31
35	Fliegende Pflanzen	32
36	Alternative Verteilung	32
37	User Interface Version 4	32
38	Überlappende Kartenabschnitte	33
39	Fehlerhafter Übergang der Kartenabschnitte	33
40	Ein schönes Zwischenergebnis	33
41	Prototyp der Blüte für die Sonnenblume Version 0	34
42	Blüte der Sonnenblume Version 1	34
43	3D-Modell der fertigen Sonnenblume	35
44	Sonnenblume zum ersten Mal in der Natur	35

45	God bless the Backup	36
46	Homogenes Blumenfeld	36
47	Blumen Schlachtfeld	36
48	Rotationsversuche	37
49	Blumenfeld nach Licht-Anpassungen	37
50	Blumenfeld Ergebnis	37
51	Einsatz verschiedener Pflanzen	38
52	Pflanzen werden nicht gelöscht	38
53	User Interface Version 5	38
54	Funktion zum Erstellen einer Heightmap	40
55	Funktion zum Erstellen einer Heightmap unter Verwendung des Compute Shaders	41
56	Funktion zum Erstellen einer Heightmap unter Verwendung von Threading	42
57	Kombination mehrerer Heightmaps	43
58	Inhalt des Perlin Noise Compute Shaders	44
59	Inhalt des Compute Shaders für den Mesh Generator	45
60	Funktion 'moveup'	46
61	Löschen (Dispose) eines Mesh Generators	47
62	Funktion zur Verteilung von Pflanzen	48
63	Funktion zum Erstellen einer Pflanze	49
64	Create Shape 1	50
65	Übergabe der Daten an den Compute Shader	51

Symbolverzeichnis

Symbol	Bedeutung
X :	Länge der zu generierenden Karte
Y :	Breite der zu generierenden Karte
h_i :	Heightmap i
H :	kombinierte Heightmap
$Pn()$:	Perlin Noise Funktion
s :	Skalierung der $Pn()$ -Funktion (Scale)
c :	Karte Gewichtung (Contribution)
κ	Karte Grobkörnigkeit (Coarseness)
s_p :	Pflanzenaufkommen Wahrscheinlichkeit (Plantscale)
o :	Pflanzenaufkommen Intensität (Occurance)
p :	3D-Vektor der generierten Karte (x,y,z)

1 Einleitung

Klassischerweise werden virtuelle Landschaften gezielt erstellt was zeitlich sehr ineffizient ist und gleichzeitig eine Menge menschlicher Ressourcen benötigt. Jeder Berg, jedes Tal und jede Pflanze wird willkürlich auf eine endliche Karte gesetzt und um eine realistische Bewegungsfreiheit zu gewährleisten muss die Karte möglichst groß sein.

In dieser Arbeit geht es darum, eine Landschaft zufällig und fortlaufend zu erstellen. Man soll sich endlos fortbewegen können während die Karte durch eine mathematische Funktion weiter generiert wird. Interessant sind dabei die einzelnen Stellschrauben mit denen verschiedene Landschaften berechnet werden können. Das Ziel ist es, unterschiedliche, realistische und nicht zuletzt auch schöne Landschaften erzeugen zu können welche den Betrachter zum Anschauen verleiten.

Vor allem in der Entwicklung von Videospielen nimmt die Gestaltung der virtuellen Umwelt eine große Rolle ein. Spielflächen sind typischerweise durch die Karte begrenzt und jeder Aspekt der Umwelt muss gestaltet werden. Die prozedurale Generierung von Landschaften bietet eine alternative Möglichkeit um dieses Problem anzugehen. Zusätzlich bietet es die Möglichkeit eine praktisch unendlich weite Umgebung zu liefern was zudem ganz neue Arten von Spielen ermöglicht.

1.1 Architektur

Zur Entwicklung der Software wird die Unity-Engine und die Programmiersprache C# (C-Sharp) verwendet. Es wurde sich für Unity entschieden, weil es die mit Abstand beliebteste 3D Engine ist und sich somit eine große Community um die Software gebildet hat. Dies hat zur Folge, dass viele sehr gute Ressourcen zum Verwenden der Software von der Community bereit gestellt werden und fast jede Frage bereits in einem Forum beantwortet wurde. Da kein Vorwissen über 3D Engines bestand war dies der ausschlaggebende Grund. Darüber hinaus benutzt Unity die Programmiersprache C# welche sehr ähnlich zu Java ist und in die sich aus dem Grund schnell eingearbeitet werden konnte. Zudem müssen mit Unity keine Abstriche in Umfang und Funktionalität gemacht werden, da die Engine sehr mächtig und umfangreich ist. Abbildung 1 zeigt eine grobe Übersicht über das Programm und dessen Funktionalität in einem Use Case Diagramm. Wie zu erkennen ist kann der Nutzer verschiedene Einstellungen treffen, welche starken Einfluss auf die Generierung der Landschaft nehmen. Außerdem kann er sich anschließend auf der Karte fortbewegen, was zu gegebenen Zeitpunkten zu der Generierung eines neuen Kartenabschnitts mit flüssigem Übergang führt. Abbildung 2 liefert einen Überblick über die Architektur des Projektes und wie dieses umgesetzt wurde. Die Anzahl der Klassen wird möglichst gering gehalten um unnötigen Overhead in der Unity Engine zu vermeiden. Allerdings führt dies dazu, dass die Klassen selbst viele Funktionen beinhalten und schnell sehr groß werden. Aus diesem Grund sind im Klassendiagramm nur die notwendigen und relevanten Funktionen eingetragen.

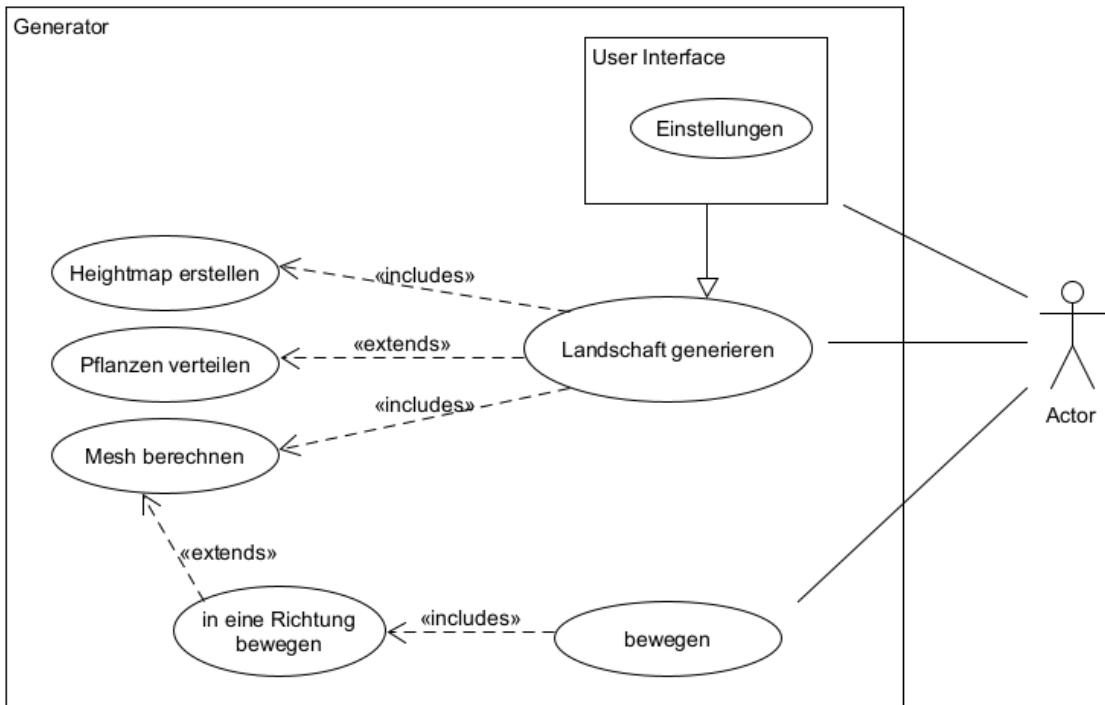


Abbildung 1: Use Case Diagramm

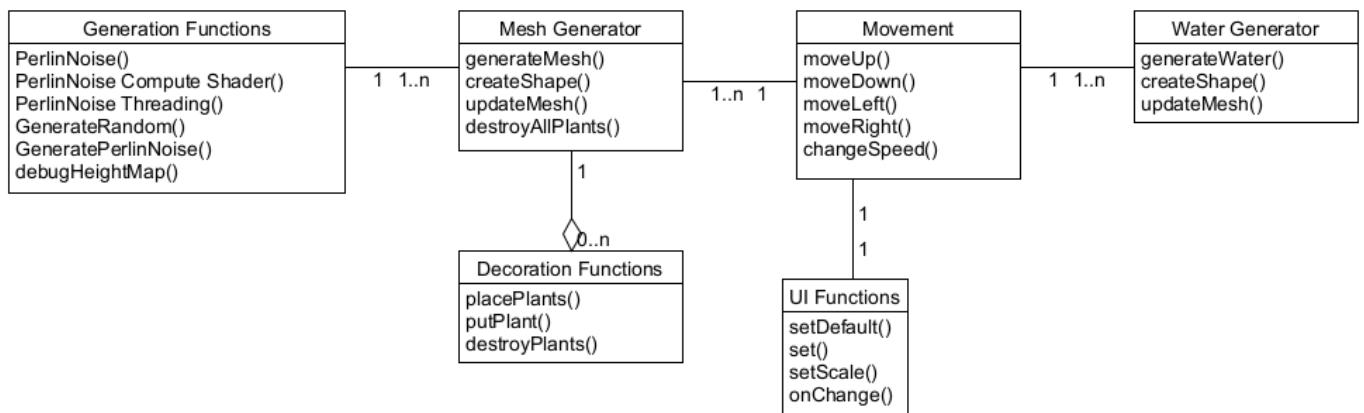


Abbildung 2: Klassendiagramm

1.2 Vorgehensweise

Das Projekt wird Schritt für Schritt iterativ durchgeführt. In jeder Iteration wird eine neue Funktion konzipiert und auf Funktionalität, Stabilität, sowie Performance getestet. Diese Vorgehensweise wurde gewählt damit möglichst agil gearbeitet werden kann. Das Projekt beinhaltet einen großen kreativen Teil und neue Ideen können während des Arbeitsprozesses jederzeit entstehen. Da zwei Personen gemeinsam entwickeln, herrscht ständige Kommunikation und Rücksprache. Die Daten werden online in einem Repository auf der Platform 'Github' gespeichert und fortlaufend aktualisiert. Das gewährleistet zum Einen die Möglichkeit, zu zweit gleichzeitig an verschiedenen Stellen des Codes arbeiten zu können, zum Anderen bietet es eine 'Version Control', sprich eine Möglichkeit die einzelnen Updates als neue Version unter Verwendung einer speziellen Beschreibung einzuordnen und gegebenenfalls wieder abrufen zu können.

2 Heightmaps und Perlin Noise

Um eine Landschaft generieren zu können wird eine Karte mit drei Dimensionen benötigt. Zu jeder Position muss der entsprechende Höhenwert berechnet und gespeichert werden. Diese Liste an Höhenwerten wird als Heightmap H definiert. Um die Werte zu berechnen und die Heightmap mit diesen zu füllen wird eine Funktion benötigt. Zu dem Zweck wird in diesem Projekt die sogenannte Perlin Noise Funktion verwendet, da sie zufällige Werte liefert welche allerdings flüssig ineinander übergehen, dass bedeutet der Unterschied der 'benachbarten' sprich aufeinander folgenden Werte ist nicht zu groß.

Zunächst wird ein zweidimensionales Array H mit der Größe $X * Y$ angelegt welches aus Float Werten besteht. Nun soll für jede Index-Kombination i, j ein Wert z zwischen 0 und 1 eingetragen werden, so dass gilt:

$$H[i, j] = z \mid 0 \leq z \leq 1 ; i, j \in \mathbb{N}; z \in \mathbb{R}$$

Hierbei ist H die Heightmap in der die von Perlin Noise zurückgegebenen Werte zwischen 0 und 1 gespeichert werden.

Die Perlin Noise Funktion $Pn()$ liefert zu jedem Tupel (x, y) einen bestimmten Wert zurück, welcher in $H[x, y]$ gespeichert wird. Der Grund warum in diesem Projekt Perlin Noise als Funktion benutzt wird um Werte für H zu generieren ist, weil diese für benachbarte Werte weiche Übergänge liefert (siehe Abbildung 3). Dies ist notwendig, weil Berge in der Regel in der Natur keine abrupten Änderungen in Höhenverhältnissen haben, welche zu harten Kanten und 'Stacheln' in der Landschaft führen würden.

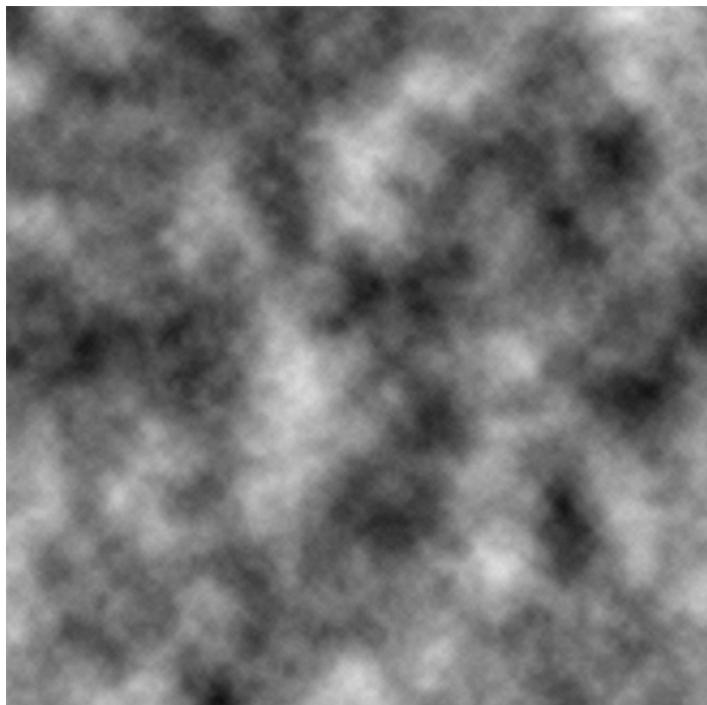


Abbildung 3: Visualisierung von benachbarten Werten der Perlin Noise Funktion. Die X, Y Achsen des Bildes repräsentieren die an die Funktion übergebenen x, y Werte. Die Farbgebung der Pixel repräsentiert den zurückgegebenen Wert z , wobei Schwarz für den Wert 0 steht und Weiß für den Wert 1.

3 Funktionalität

3.1 Erstellung der Landkarte

Das Programm bietet die Möglichkeit, die Größe der zu erstellenden Karte (X, Y) manuell festzulegen. $Pn()$ liefert nun für jede (x, y) -Koordinate den entsprechenden Höhenwert z und damit die fertige Heightmap welche in H gespeichert wird. Die an die Funktion übergebenen Werte werden durch verschiedene Stellschrauben beeinflusst. Zunächst werden zwei Float Werte x_s, y_s zufällig im Wertebereich $0 \leq x_s, y_s \leq 10^6$ bestimmt. Hierbei legt die obere Begrenzung die Anzahl von möglichen unterschiedlichen Landschaften unabhängig der anderen Stellschrauben und Variablen fest. Doch stellt sich in der Praxis heraus, dass diese Anzahl nur in der Theorie einen Unterschied macht, da die Werte die von Perlin Noise zurück gegeben werden unabhängig von der Sampling Position oberflächlich sehr ähnlich aussehen. So haben die anderen Stellschrauben eine proportional sehr viel größere Auswirkung auf das letztendliche Aussehen der Landschaft. Im nächsten Schritt werden alle Werte $H[i, j]$ algorithmisch wie folgt berechnet:

$$H(i, j) = Pn((x_s + i) * s, (y_s + j) * s)$$

wobei die Variable s eine weitere wichtige Stellschraube darstellt mit der die Grobkörnigkeit der Noise Map bestimmt werden kann. Grobkörnigkeit beschreibt die 'Rauheit' der Heightmap, ähnlich der Beschaffenheit eines Schmiegelpapiers bei dem der Körnungsgrad die Feinheit bestimmt, also wie groß der Unterschied der direkt benachbarten Werte ist.

3.2 Verrechnung mehrerer Heightmaps

Auf Grund der flüssig ineinander übergehenden Werte von Perlin Noise sieht die Landschaft jedoch zu sehr aus wie eine einfache 3-dimensionale mathematische Funktion statt wie eine echte Landschaft. Dies liegt daran, dass die Irrregularitäten bzw. die 'Ecken und Kanten' fehlen. Um diese Irrregularitäten künstlich in die Landschaft einzubauen werden mehrere Heightmaps miteinander verrechnet. Im Weiteren wird auf Grund dessen die kombinierte Heightmap als H bezeichnet, welche sich aus n einzelnen Heightmaps h_i berechnet. So ergibt sich folge Formel:

$$H(j, k, n) = \sum_{i=1}^n \frac{Pn((x_s + j) * s(i+1)^\kappa, (y_s + k) * s(i+1)^\kappa)}{ic + 1}$$

Der Faktor c bestimmt die Gewichtung der berechneten Teil-Heightmap h_i . Des Weiteren bestimmt der Exponent κ sowie der Faktor s die Rauheit der berechneten Teil-Heightmap h_i . Durch die Kombination dieser Faktoren kann die fraktalähnliche Struktur einer echten Landschaft zu einem gewissen Grad simuliert werden. Die Faktoren sind zwei weitere wichtige Stellschrauben und können deshalb im User Interface während des Programmbetriebes geändert werden. Um die oben definierten Begrenzungen einzuhalten die für Heightmaps gesetzt wurden muss H anschließend noch normalisiert werden, damit sich alle Werte weiterhin zwischen 0 und 1 aufhalten. Hierfür wird ein einfacher 'Inverse Lerp' verwendet:

$$InvLerp(min, max, n) = \frac{(n - min)}{(max - min)}$$

3.3 Bewegung auf der Landkarte

Anfangs wird die Bewegung auf der Landkarte auf eine sehr umständliche Art und Weise gelöst, um den verwendeten Algorithmus in der Tiefe zu testen. Dabei wird die Landschaft bei jeder Bewegung nach Vorne erneut berechnet aber mit einem Shift-Wert der alle Werte um eins in $-z$ Richtung bewegt. So kann zum Einen die Statik der benutzen Funktion getestet werden, ob diese für die selben Werte auch die selben Rückgabewerte liefert. Dies sollte auch für die Verschiebung der Fall sein. Sollte sich gegebenenfalls die Verschiebung auf h_i in unterschiedlichen Maße auswirken als auf h_{i+n} muss die Funktion überdacht werden. Zum Anderen kann so die Performance der Methode getestet werden, da diese bei jeder Bewegung ausgeführt werden muss. Die Methode funktioniert und liefert eine gut aussehende Landschaft, benötigt jedoch auf Grund der $O(n^2)$ Komplexität (siehe Kapitel 7.1) eine solch hohe Performance, dass keine flüssige Bewegung auf der Karte möglich ist.

3.4 Verbesserung der Bewegungsmöglichkeit

Die naheliegende Möglichkeit ist, die Kamera zu versetzen um eine Bewegung zu erzeugen. Das Problem dabei ist allerdings, dass man so nach entsprechender Strecke an den Rand der Karte gelangt. Es muss also früh genug eine weitere Karte mit der selben Funktion erstellt und an der richtigen Stelle platziert werden. Die besondere Herausforderung dabei ist, die verschiedenen Kartenabschnitte nahtlos ineinander übergehen zu lassen. Dazu wird zunächst ein 'threshold' t festgelegt welcher bestimmt, zu welchem Zeitpunkt der neue Kartenabschnitt erzeugt wird. t wird für das Beispiel der Vörwärtsbewegung wie folgt definiert:

$$t = p.y + \frac{Y}{2}$$

Hierbei ist p definiert als der dreidimensionale Vektor der die Position der generierten Karte beschreibt. Des Weiteren ist Y definiert als die Größe der Karte in Y Richtung. Sobald t von der Kamara-Position überschritten wird, erzeugt das Programm eine weitere Version der Landkarte mit der passenden Startposition und erzeugt mit dieser den neuen Kartenabschnitt welcher an der entsprechenden Stelle eingefügt wird. Da sich nun unendlich lang in eine Richtung bewegen kann und fortlaufend neue Kartenabschnitte erzeugt werden, wird irgendwann der Arbeitsspeicher nicht mehr ausreichen. Um dem vorzubeugen gibt es einen Puffer von hinter dem ein Kartenabschnitt wieder gelöscht wird sobald dieser überschritten wird.

3.5 Verwendung von Presets

Um verschiedene Einstellungen für Landschaften speichern und abrufen zu können ist eine Preset-Funktion integriert. Eine jeweilige Einstellung der Parameter kann unter einem beliebigen Namen in einer Liste gespeichert werden. Die gesamte Presetliste kann zusätzlich in eine Datei exportiert- und an anderer Stelle importiert werden. Dadurch bieten sich gute Möglichkeiten für Anwender, eigene Ergebnisse und Ideen auszutauschen.

4 User Interface

Das User Interface dient zur Festlegung der Einstellungen und Parameter. Es besteht aus verschiedenen Inputfeldern, Slidern, Checkboxen, Buttons und einem Dropdownmenü. Während der Arbeit wurde das User Interface fortlaufend weiter entwickelt. Abbildung 4 zeigt das User Interface in seiner finalen Form.

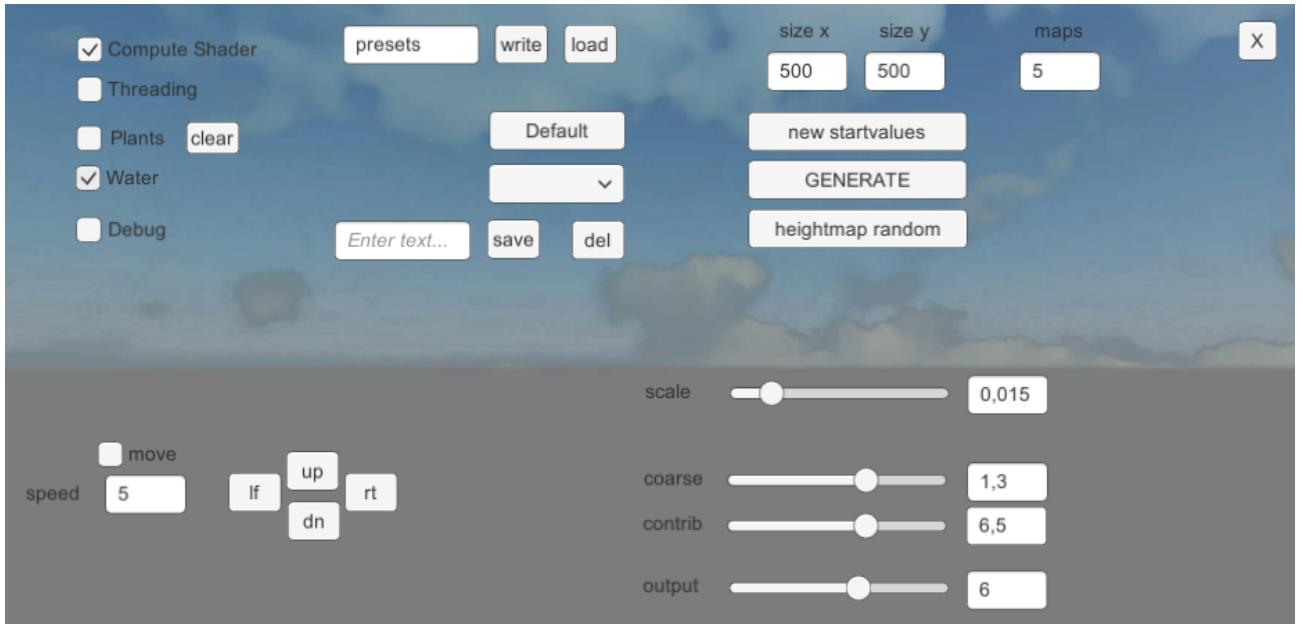


Abbildung 4: Ansicht des User Interface (Version 8)

Das User Interface beinhaltet folgende Bereiche:

Heightmap erzeugen:

- In den Feldern 'size x' und 'size y' wird die Größe der zu erstellenden Karte festgelegt.
- Im Feld 'maps' wird die Anzahl an zu berechnenden Heightmaps mitgegeben, welche miteinander kombiniert werden.
- 'GENERATE' erzeugt eine Perlinnoise Heightmap mit den vorhandenen Einstellungen.
- 'new startvalues' verändert die Startposition und berechnet die vorhandene Heightmap neu.

- 'heightmap random' erzeugt zufällige Werte für die einzelnen Parameter der Heightmap.

Heightmap Parameter:

- 'scale' multipliziert die Werte die an die Perlin Noise Funktion gegeben werden mit einem Wert zwischen 0 und 0,1, was die Grobkörnigkeit der Karte bestimmt.
- 'output' skaliert die z-Werte der Heightmap. Bei output=x liegen die z-Werte der Heightmap zwischen 0 und x.
- 'coarse' bestimmt den Unterschied der Grobkörnigkeit der verschiedenen Heightmap-Ebenen.
- Mit 'contrib' wird eingestellt mit welcher Gewichtung die verschiedenen Maps bei der Kombination berücksichtigt werden.

Movement:

- Die Buttons 'up', 'dn', 'lf' und 'rt' ermöglichen eine Bewegung auf der Map in die vier Himmelsrichtungen.
- Die Checkbox 'move' sorgt dafür, dass fortlaufend 'up' ausgeführt wird wodurch eine Vorwärtsbewegung über die Map erfolgt. Im Feld 'speed' wird die Geschwindigkeit festgelegt. Bei speed = x bewegt sich die Camera jede Sekunde um x nach vorne.

Presets: Ein Preset beinhaltet gespeicherte Werte für die einzelnen Parameter.

- Der 'Default' Button setzt die Parameter auf zuvor festgelegte Standartwerte.
- Über das Preset Dropdown-Menü kann ein vorher gespeichertes Preset ausgewählt werden.
- 'save' speichert das Preset unter dem im Feld eingegebenen Namen.
- 'del' löscht das aktuell ausgewählte Preset.
- Mit 'write' und 'load' kann die Presetliste dauerhaft gespeichert und geladen werden.

Optionen:

- 'Compute Shader' aktiviert die Berechnung auf der GPU
- 'Threading' aktiviert die Berechnung in mehreren Threads / Kerne der CPU.
Compute Shader und Threading können nicht gleichzeitig aktiviert werden.
- 'Plants' aktiviert die Verteilung von Pflanzen auf der Karte.
- Der Button 'clear' löscht alle Pflanzen auf der aktuellen Karte.
- 'Water' aktiviert die Generierung von Wasser.
- Die 'Debug' Checkbox erzeugt verschiedene Ausgaben auf der Konsole, hauptsächlich zur Fehlersuche.

Sonstige: Der Menübutton ermöglicht es, die Einstellungen im laufenden Programm ein- und auszublenden.

5 Mesh Generator

Der Mesh Generator ist für die grafische Darstellung der Landschaft zuständig. Er bekommt die vorher von `Pn()` generierten Höhenwerte übergeben, welche zusammen mit den entsprechenden x,y-Positionen die Vertices ergeben. Die Vertices werden zur Berechnung von Dreiecken in einer großen Liste an die GPU übergeben. Dieser Vorgang wird als 'Triangulation' bezeichnet. So entsteht das fertige 'Mesh'. Hierbei ist die Reihenfolge in der die Vertices der Liste hinzugefügt werden sehr wichtig, da Unitys automatisiertes Backfaceculling darauf basiert in welcher Richtung die Vertices in der Liste stehen. Backfaceculling ist eine Methode mit der bestimmt wird welche Dreiecke wirklich von der GPU in einem bestimmten Frame berechnet werden müssen um den Rechenaufwand bei Runtime minimal zu halten. Deshalb müssen alle Vertices im Uhrzeigersinn der Liste hinzugefügt werden, was mehr Rechenaufwand bei der Initialisierung erfordert. Zusätzlich kriegen alle Vertices eine Vertex Color auf Basis des Höhenwertes des bestimmten Vertex zugewiesen. Hierzu kann in Unity ein Gradient bestimmt werden von dem im Skript anschließend gesampled wird. Um die Landkarte etwas detaillierter zu machen ohne noch mehr Vertices hinzufügen zu müssen kann eine Normalmap verwendet werden. Diese beeinflusst für ein gegebenes Mesh wie Licht von diesem reflektiert werden soll. So können zum Beispiel Unebenheiten oder Unterschiede in der Oberflächenbeschaffenheit simuliert werden. Um die Normalmap richtig darstellen zu können müssen zusätzlich UVs (U,V-Koordinaten) für das Mesh bestimmt werden. Diese UVs werden anschließend mit einem Faktor multipliziert, welcher dafür sorgt, dass sich die Normalmap öfter auf dem Mesh wiederholt statt in einem Stück auf die komplette Karte projiziert zu werden. Später im Verlauf des Projekts wurde die Triangulation mit Hilfe eines Compute Shaders auf die GPU ausgelagert um die Performance weiter zu erhöhen.

6 Verteilung von Objekten

Zur Verteilung von Objekten wird ein neues ebenfalls mit Perlin Noise erzeugtes Array verwendet. Hierzu wird die selbe Funktion verwendet wie für eine einzelne Ebene der Landkarte. Da Perlin Noise immer Werte zwischen 0 und 1 zurück gibt bietet diese sich stark als eine Funktion an, mit der Werte für eine Probability Distribution D berechnet werden können. Diese wird mit der vorhandenen Heightmap verrechnet um das Aufkommen eines Objekts zu bestimmen, wobei die Höhe der aktuellen Position die Wahrscheinlichkeit verringert, dass dort eine Pflanze platziert wird (abhängig vom Planzentyp auch umgekehrt). Dadurch dass für die Probability Distribution die Perlin Noise Funktion verwendet wird, bilden sich auf der Landkarte einzelne Blumenfelder statt vereinzelte zufällig verteilte Blumen. Dadurch wirkt die Landschaft noch ein Stück realistischer. Hierbei ist die oben beschriebene Probability Distribution D mit der Grobkörnigkeit s_p wie folgt definiert:

$$D(i, j) = Pn((x_s + i) * s_p, (y_s + j) * s_p)$$

Nun wird die Wahrscheinlichkeit für eine Blume an stelle j, k wie folgt definiert:

$$P_b(j, k) = (1 - h[j, k]^{\frac{1}{4}}) * D[j, k]^2 * o$$

Mit o ('occurrence') existiert ein weiterer Faktor der mit der Wahrscheinlichkeit für eine Pflanze verrechnet wird, wodurch die generelle Wahrscheinlichkeit für das Aufkommen dieser Pflanze genauer bestimmt werden kann. Diese Funktion liefert so für jede Koordinate j, k auf der Landkarte einen statischen Wahrscheinlichkeitswert ob an der Stelle ein Objekt erzeugt werden soll oder nicht. Auf Grund dessen, dass für jede Koordinate die Möglichkeit besteht, eine Blume zu platzieren muss die Wahrscheinlichkeit generell sehr gering gehalten werden und mit steigender Auflösung der Karte weiter verringert werden.

6.1 3D Modelle

Die zu platzierenden Objekte sind mit der Software 'Blender' selbst erstellte 3D-Modelle verschiedener Pflanzen. Zum Repertoire gehören Löwenzahn (Pusteblume), Sonnenblumen und Bäume zu sehen auf den Abbildungen 5 bis 8.



Abbildung 5: Modell eines Löwenzahns, auch 'Pusteblume' genannt

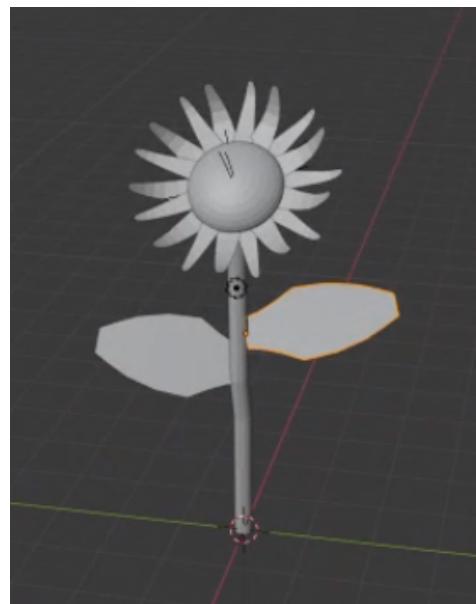


Abbildung 6: 3D-Modell einer Sonnenblume



Abbildung 7: 3D-Modell eines Baums (Version 3)

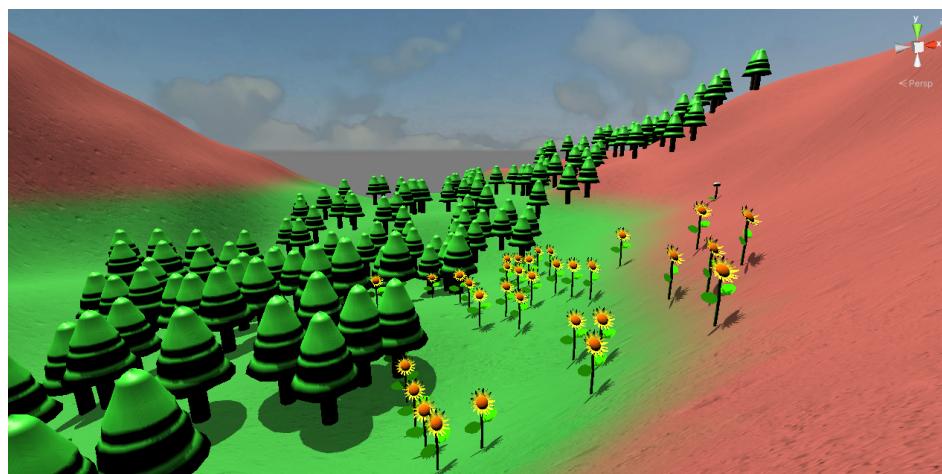


Abbildung 8: verschiedene Pflanzen in einer Landschaft

6.2 Wasser

Um Wasserbereiche auf der Landkarte zu erzeugen wurde zunächst ein Quaderobjekt (cube) verwendet. Das Objekt wird in der Größe des Kartenabschnitts über die generierte Landschaft gelegt so dass anhand eines vorher festgelegten Wasserspiegels eine Schnittfläche entsteht. Der Bereich unterhalb dieser Schnittfläche wird nun von dem Quader überdeckt und so entsteht der Eindruck verschiedener Wasseransammlungen wie Seen, Flüssen oder auch einem Meer. Um das Wasser realistischer erscheinen zu lassen wird es mit Farbe und einer Normalmap versehen (Abbildung 9).

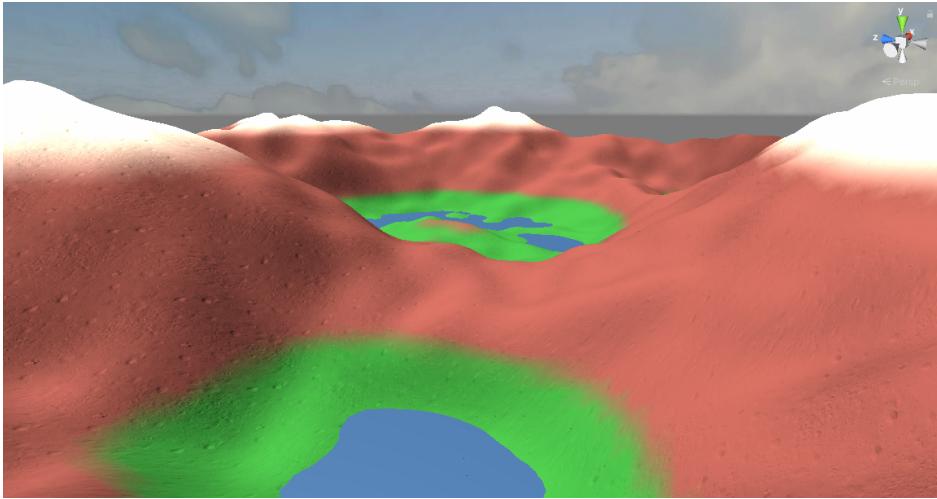


Abbildung 9: Landschaft mit Wasser Version 1

Da das Quaderobjekt aus zu wenig Vertices besteht um es mit einer realistischern Oberflächenbewegung zu versehen wird es durch einen weiteren Mesh Generator ersetzt welcher um einen selbst konzipierten Vertex Shader (Abbildung 11) erweitert wird. Die Positionen der Vertices werden mittels eines einfachen Rauschens (simple noise) so verändert, dass der Eindruck eines Seegangs entsteht. Um die monotone Erscheinung der Wasseroberfläche aufzubrechen wird auch hier zusätzlich eine Normalmap verwendet. Diese wird zeitabhängig fortlaufend verschoben so dass der Eindruck von bewegten Wellen hinzukommt (Abbildung 10).

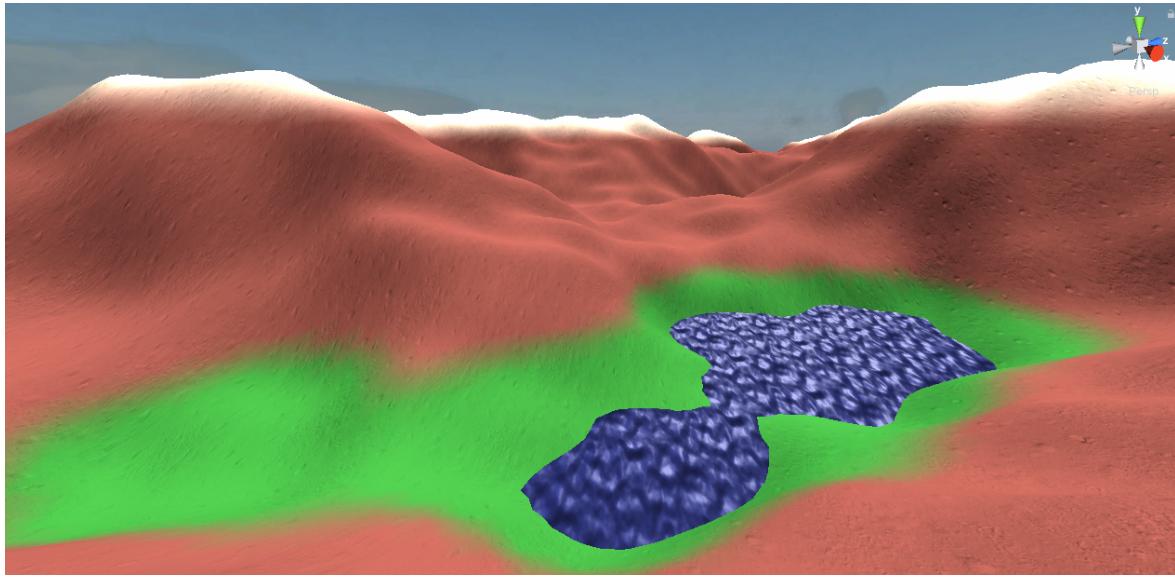


Abbildung 10: Landschaft mit Wasser Version 2

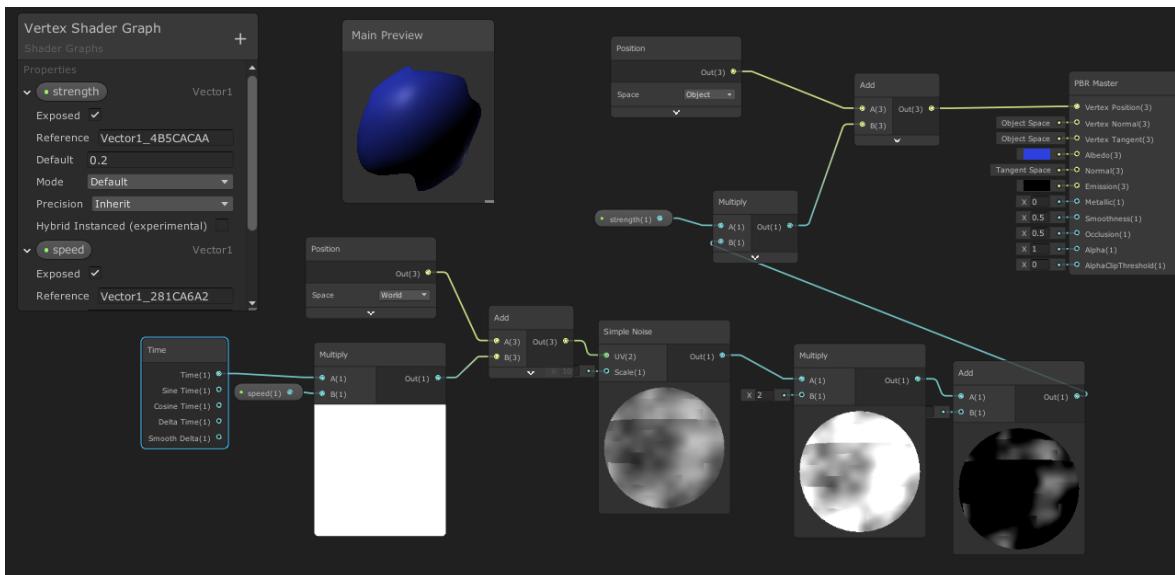


Abbildung 11: Vertex Shader Version 4

7 Parallelisierung

7.1 Komplexität

Der Algorithmus hat die Komplexität $O(n^2)$ da der Rechenaufwand mit zunehmender Kartengröße exponentiell steigt. Um dem Herr zu werden kann die Bewegung auf eine Richtung beschränkt werden, wodurch die Breite der Karte nicht steigt und die Komplexität auf $O(n)$ herunter gesetzt wird. Da aber eine Auflösung der Karte festgelegt werden muss dessen Komplexität weiterhin mit $O(n^2)$ bestimmt ist und nicht reduziert werden kann bietet sich hier die Nutzung einer Parallelisierung an. Mit steigender Auflösung wird die Parallelisierung ab einem bestimmten Punkt unabdinglich.

7.2 Compute Shader und Threading (Unity Job System)

Der Zweck eines Compute Shaders ist es, die Rechenkapazität der GPU auszunutzen. Der Große Vorteil gegen über der Berechnung auf der CPU ist, dass die GPU standardmäßig auf Parallelisierung ausgelegt ist. Dies ist der Fall, da die Rasterisierung von 3D Umgebungen einen sehr hohen Rechenaufwand erfordert doch im Gegenzug stark parallelisierbar ist. Aus diesem Grund eignet sich eine GPU auch für Anderweitige Rechnungen sehr gut, gesetz dem Fall, dass diese auch parallelisierbar sind. Und diese Rechenleistung lässt sich in Unity mit einem Compute Shader nutzbar machen. Hierfür muss die Berechnung unserer Karte in kleinstmögliche und unabhängige Teilrechnungen unterteilt werden, was auf Grund der Nutzung der Perlin Noise Funktion einfach umzusetzen ist. So werden die Daten vorher für den Compute Shader prepariert und in einem Buffer geschrieben aus dem der Compute Shader die Daten lesen kann. Nachdem der Compute Shader fertig mit seiner Berechnung ist schreibt er das Ergebnis in einen weiteren Buffer welcher dann von Unity ausgelesen werden kann. In dieser Sektion war ein Artikel von David Kuri [1] äußerst hilfreich da dieser das komplexe Thema der Parallelisierung auf der GPU gut auf den Punkt bringt. Trotz der hohen Rechenkapazität der GPU sollte die Rechenkapazität der CPU auch nicht ungenutzt bleiben. Außerdem kann die CPU durch Ihre Mehrzahl an Kernen auch eine beträchtliche, wenn auch nicht mit der GPU gleichen, Paralellisierungsleistung erbringen. Um diese Leistung nutzbar zu machen muss für die CPU Threading verwendet werden. Da dies ein sehr Häufiges Unterfangen von Spieleentwicklern ist stellt Unity eine einfache Möglichkeit

des Threadings bereit. Diese nennt sich das Unity Job System. Mit diesem kann in einem Struct alles Definiert werden was für die Berechnung benötigt wird und welche Berechnung ausgeführt werden soll. Den restlichen Teil, inklusive Aufgabenverteilung an Threads und Vermeidung von Race Conditions wird von dem Job System selbst geregelt.

8 Ergebnisse

Dieser Abschnitt führt einige Beispiele der Möglichkeiten auf, welche Ergebnisse mit dem Programm generiert werden können (Abbildungen 12-15).

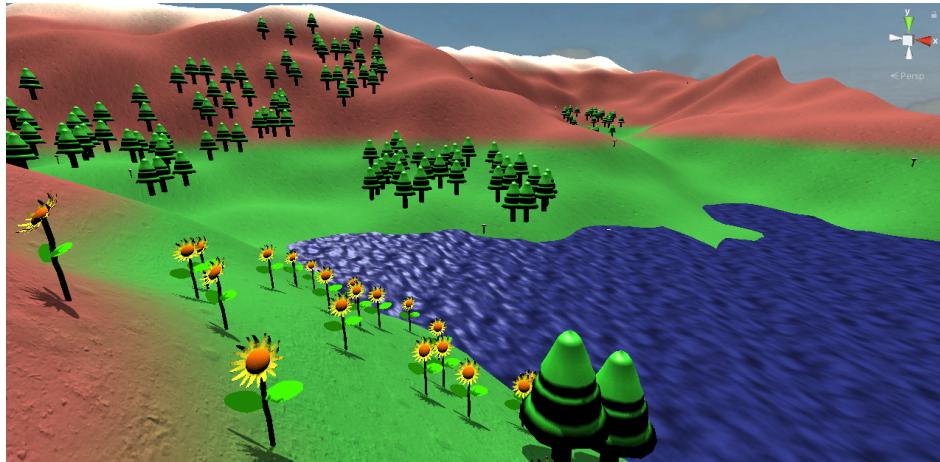


Abbildung 12: Landschaft Beispiel 1

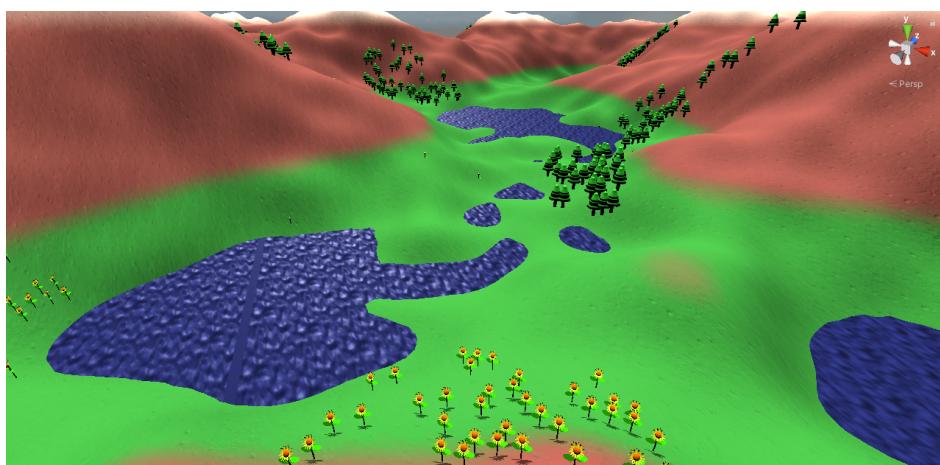


Abbildung 13: Landschaft Beispiel 2

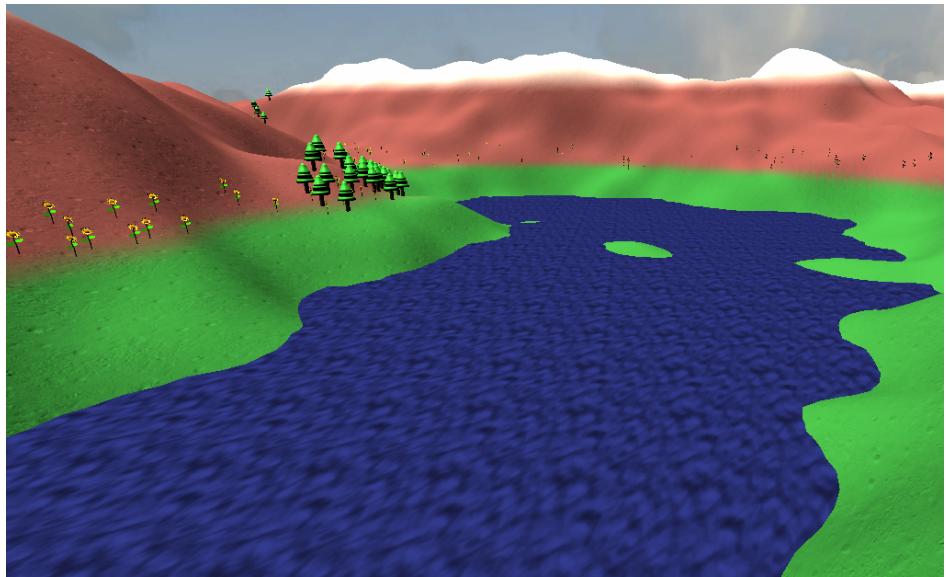


Abbildung 14: Landschaft Beispiel 3

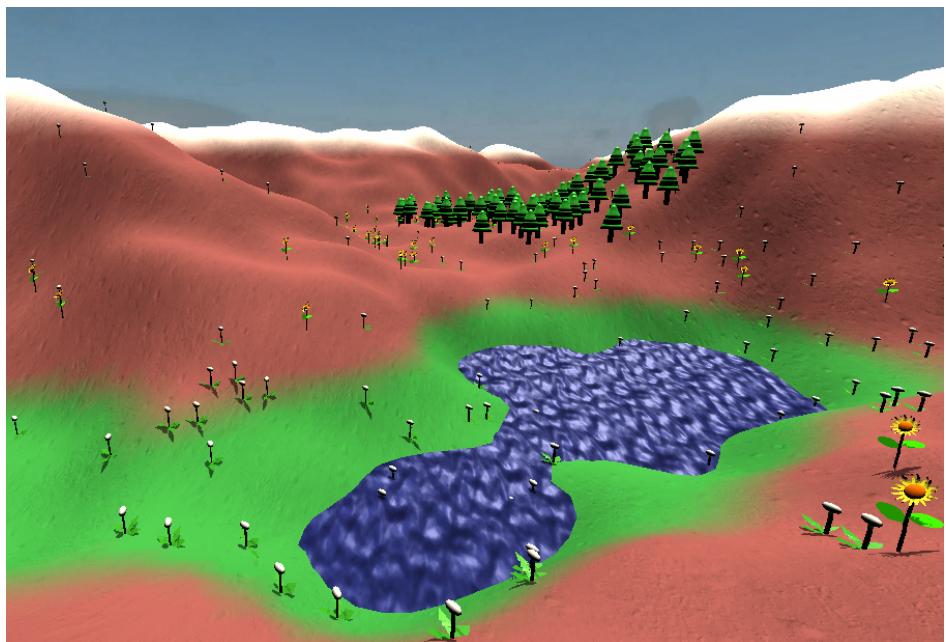


Abbildung 15: Landschaft Beispiel 4

9 Erfahrung

9.1 Der Weg war lang

Kein Projekt läuft von Anfang bis Ende reibungslos und auch hier mussten unterschiedlichste Hürden genommen werden bevor das Ziel erreicht werden konnte. Dieser Abschnitt enthält eine kurze Zusammenfassung unserer Erfahrungen sowie Beschreibungen und Bilder einiger Schwierigkeiten und Erfolge im Laufe des Projekts.

Nachdem Perlin Noise und Mesh Generator implementiert waren, mussten in vielen Versuchen sinnvolle Einstellungen für die Stellschrauben und Wertebereiche gefunden werden. Die Ergebnisse waren unterwegs entsprechend unbrauchbar. Bei den ersten Versuchen sich auf der Karte zu bewegen traten verschiedene Performanceprobleme auf. Der Arbeitsspeicher wurde überfüllt und ein Kern der CPU wurde komplett ausgelastet während die anderen Kerne sowie die GPU ungenutzt blieben. Erst der Einsatz des Compute Shaders konnte Abhilfe schaffen. Zusätzlich musste der Code verändert werden um den Arbeitsspeicher an geeigneter Stelle freizugeben. Das Programm lief nun dauerhaft stabil, jedoch konnte bei der Bewegung keine höhere Geschwindigkeit erreicht werden als 1 Schritt pro Sekunde. Um eine schnellere und vor allem flüssige Bewegung zu erhalten musste die Funktion grundsätzlich verändert werden (siehe Kapitel 3.4).

Nachdem die Pflanzenmodelle mit der Software 'Blender' erstellt wurden, mussten sie natürlich in das Unity-Projekt integriert und mit einem Material versehen werden. Ein zusätzlich zu beachtender Aspekt dabei war, dass die Objekte jeweils aus verschiedenen Elementen bestehen die alle ihre eigenes Material benötigen. Letztendlich nur eine Frage der korrekten Verwendung von Unity, aufgrund der fehlenden Vorkenntnisse mit der Software dennoch sehr zeit- und arbeitsintensiv. Damit die Pflanzen möglichst realistisch aussehen, spielen natürlich die Größe, Ausrichtung und Rotation eine Rolle.

Für die Preset-Funktion musste eine Sammlung von Daten in einer Liste gespeichert werden. Nun wurde eine Möglichkeit benötigt diese Liste sinnvoll in einer Datei zu speichern, so dass später die ausgelesenen Daten korrekt abgerufen werden können. Nachdem einige Versuche die Liste als JSON-Objekt zu speichern fehlschlugen wurde

die Methode geändert. Nun werden die Daten als langer String in einer Textdatei gespeichert. Da die Daten nicht vor äußerem Zugriff geschützt werden müssen ist dies kein Problem.

Das User Interface unterlag einer dauernden Anpassung. In jedem Arbeitsschritt wurden neue Parameter und Stellschrauben benötigt. Einige Funktionen wurden ebenso mit der Zeit obsolet und konnten wieder entfernt werden. Einige Funktionen dienen auch zusätzlicher Convenience.

Abbildung 16 zeigt die aller erste Landschaft die in diesem Projekt generiert wurde, noch ohne mehrere Perlin Noise Ebenen zu verrechnen. Diese sieht noch sehr gleichmäßig und 'wellig' aus.

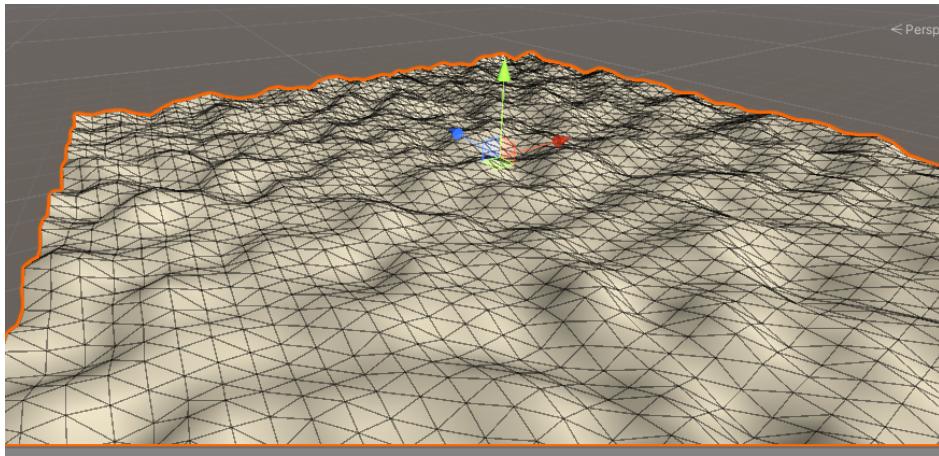


Abbildung 16: Erste generierte Landschaft

Um den teilweise auch durch eigene Fehler bedingten hohen Rechenaufwand zu vermeiden, wurde beim kombinieren der Heightmaps während der Entwicklung zunächst nur eine kleine Karte erstellt (Abbildung 17). In Abbildung 18 ist die erste Version der Bedienoberfläche zu sehen. Es beinhaltet die Möglichkeit, die Kartengröße anzugeben und die Skalierung für Perlin Noise einzustellen. Zusätzlich kann die Startposition auf der Karte mit 'shift' manuell verändert werden.

Es folgten Experimente mit der Grobkörnigkeit (coarse) der Karte. Abbildung 19 zeigt ein Beispiel einer Landschaft mit hoher Grobkörnigkeit. Die Beschaffenheit der Oberfläche könnte als hügeliges Feld wahrgenommen werden. Zu sehen sind auch die entsprechenden neuen Regler für 'coarse' und 'contrib' auf dem User Interface um

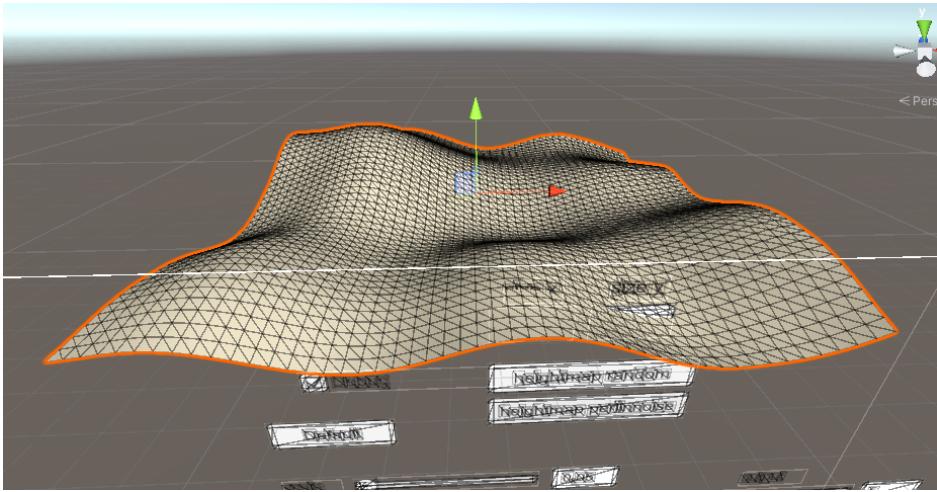


Abbildung 17: Erste kombinierte Heightmap

Grobkörnigkeit und Gewichtung der einzelnen Heightmap-Ebenen anpassen zu können. Abbildung 20 zeigt im Vergleich eine Karte mit niedrigerer Grobkörnigkeit. Die Oberflächenbeschaffenheit könnte z.B. als Wüste wahrgenommen werden. Um mehr Spielraum für Ergebnisse zu schaffen wurde der 'output'-Regler eingebaut. Damit ist es möglich, die Höhenwerte der kombinierten Heightmap nochmals zu skalieren. Das User Interface wurde um die Möglichkeit erweitert, die Anzahl der Heightmap-Ebenen zu ändern um zu testen wie sich die Ergebnisse mit einer anderen Ebenenzahl unterscheiden. Die Versuche zeigen, dass mehr als 5 Ebenen zu keinen besseren Resultaten mehr führen. Bei den ersten Versuchen sich auf der Karte zu bewegen wurde die Karte in jedem Schritt neu berechnet. Die Daten des Mesh Generators wurden jedoch nicht gelöscht was dazu führte, dass der Arbeitsspeicher rapide gefüllt wurde und das Programm früher oder später zum Absturz führt (Abbildung 21). Das Problem konnte durch das Freigeben des Speichers an der richtigen Stelle behoben werden.

Abbildung 22 zeigt die erste erstellte Landkarte in Farbe. Der Höhenwert bestimmt ob der Bereich Grün, Braun oder Weiß ist um den Eindruck von Gras, Erde und Schnee zu erzeugen.

Um die Beschaffenheit der Oberfläche der Realität anzupassen wird eine sogenannte Normalmap eingesetzt welche die Reflexion von Licht regelt (Abbildung 23).

Um das Gesamtergebnis noch realistischer werden zu lassen wird zusätzlich zur Lichtquelle welche als Sonne fungiert, eine Skybox verwendet (Abbildung 24) welche im Hintergrund einen Himmel mit Wolken darstellt.

Da die Performance trotz aller Verbesserungen sehr zu wünschen übrig lies, musste

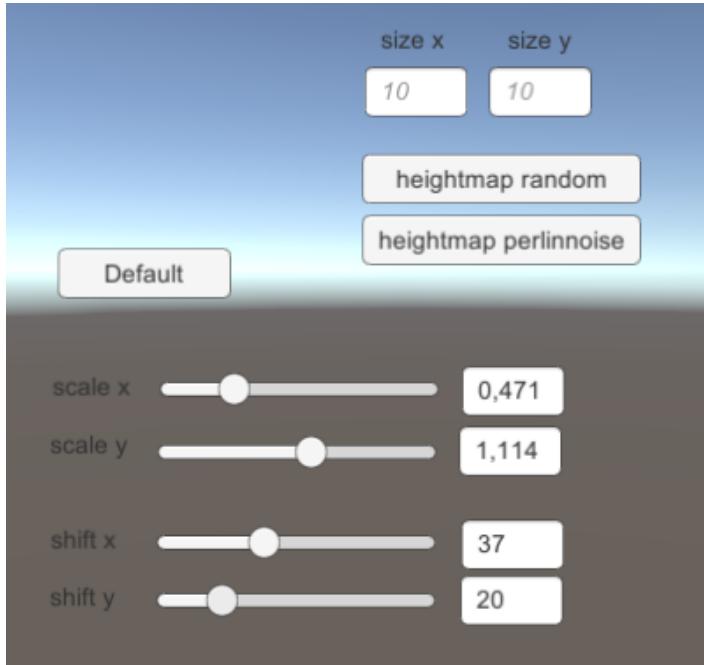


Abbildung 18: User Interface Version 1

schon frühzeitig die Verwendung eines Compute Shaders realisiert werden, damit die Berechnungen auf die GPU verlagert werden können. Die Implementierung erwies sich wie erwartet nicht als leicht und die ersten Ergebnisse waren wieder fern von jeglicher realischer Landschaft (Abbildung 25). Erst nachdem die Kommunikation zum Compute Shader fehlerfrei war konnten wieder die gewohnten Ergebnisse erzielt werden.

Das Verteilen auf mehrere Threads lies sich dank des integrierten Unity Job Systems vergleichsweise problemlos integrieren. Die Performance wurde dadurch deutlich besser, auf Grund der Funktionsweise der Bewegungsmethode wurde die CPU aber dennoch stark belastet (Abbildung 26). Die Lösung dazu war ein komplett neues Bewegungskonzept mit einer Bewegung der Kamera und der Arbeit mit Kartenabschnitten (siehe Kapitel 3.4).

Zur Verteilung von Objekten wurde zunächst einfach eine Kugelform ('sphere') verwendet. Nachdem Probleme mit der Höhe beseitigt wurden, konnten die Objekte korrekt auf der Karte platziert werden (Abbildung 27, Abbildung 28).

Nachdem die Funktionalität der Verteilung gewährleistet war, konnte mit dem Design eigener Pflanzen begonnen werden. Das erste Modell mit dessen Hilfe der Algorithmus entwickelt und entsprechende Stellschrauben ermittelt wurden, war eine vereinfachte Variante (Low Poly) einer Löwenzahnpflanze (Abbildungen 29-31).

Bei der Platzierung traten verschiedene Probleme bis hin zur 'Invasion der Pusteblumen'

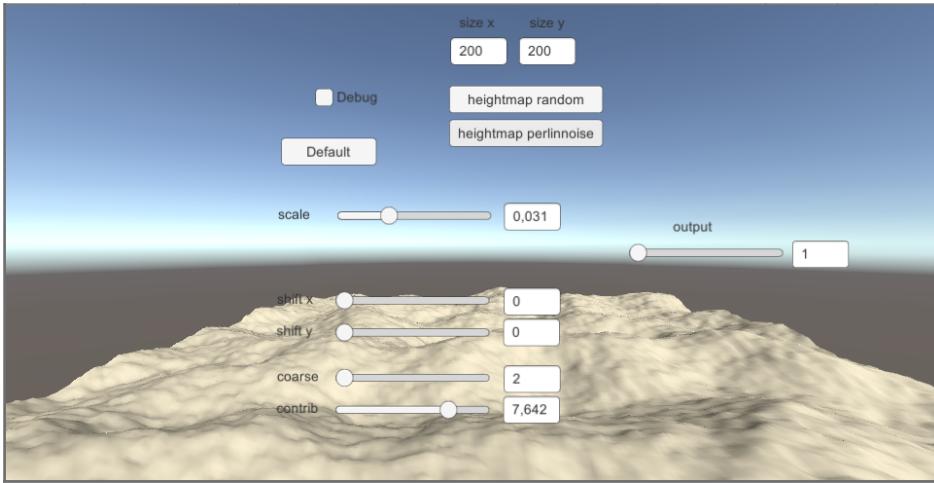


Abbildung 19: Versuche mit Grobkörnigkeit 1

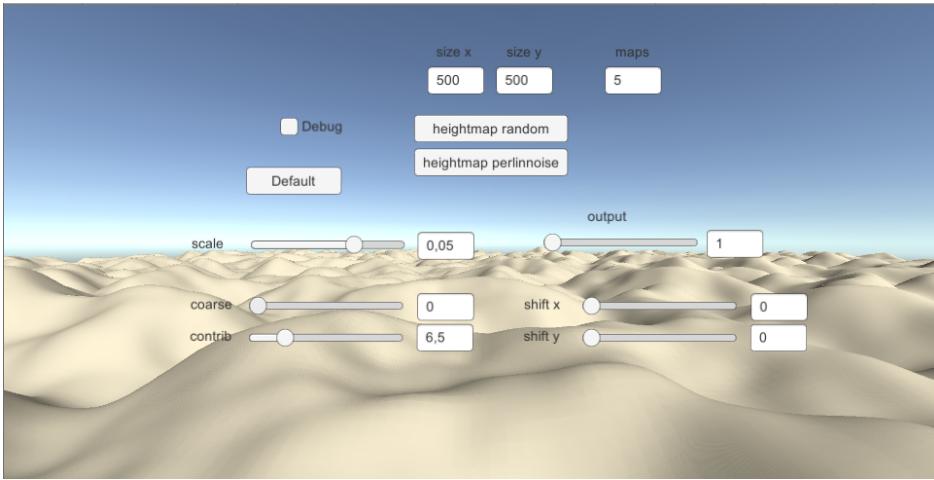


Abbildung 20: Versuche mit Grobkörnigkeit 2

auf bevor eine sinnvolle Verteilung erreicht wurde (Abbildung 32, Abbildung 33). Auf dem Weg wurden verschiedene Alternativen ausprobiert, Pflanzen zu platzieren, unter anderem die Simulation eines Samenflugs. Nach einigen Fehlschlägen (Abbildung 34, Abbildung 35) konnten damit auch brauchbare Ergebnisse erzielt werden (Abbildung 36). Letztendlich können aber mit Hilfe von Perlin Noise solch vielseitige Landschaften generiert werden, dass sämtliche anderen Methoden mit der Zeit obsolet wurden. Das User Interface wurde in der Zwischenzeit um die nötigen Parameter für die Pflanzenverteilung sowie Optionen für Threading und Compute Shader erweitert (Abbildung 37). Eine besondere Herausforderung bot die Realisierung der Karte in Abschnitten. An geeigneter Stelle musste jeweils der neue Kartenabschnitt berechnet und platziert werden. Damit die Kartenabschnitte nahtlos ineinander übergehen, müssen die Position und die

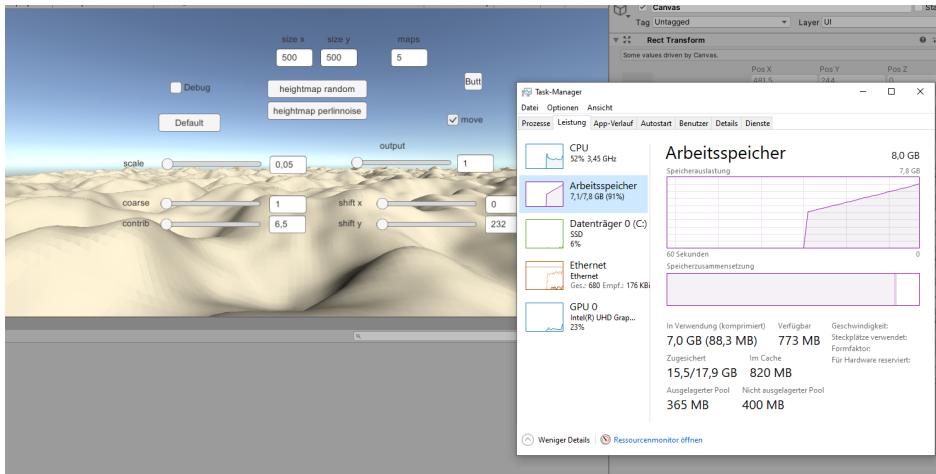


Abbildung 21: RAM-Überflutung bei Bewegung über die Karte

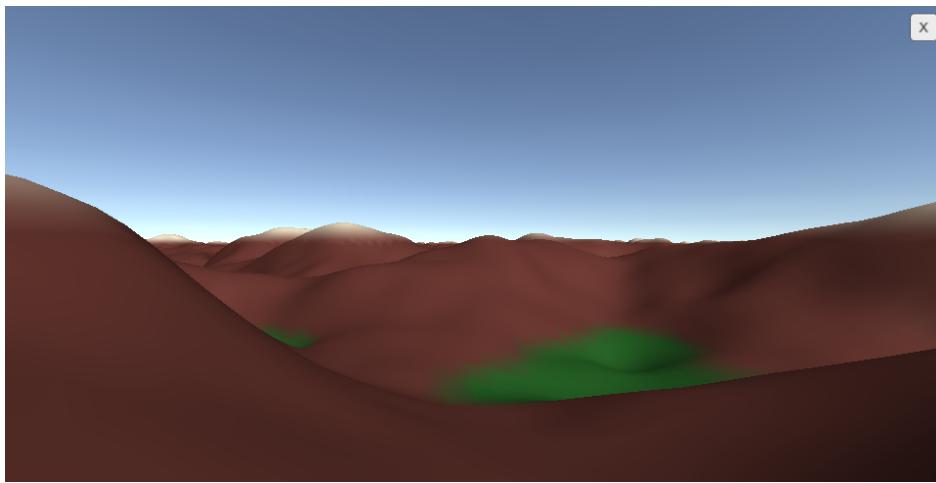


Abbildung 22: Landschaft in Farbe Version 1

Startwerte für Perlin Noise unter Berücksichtigung der Skalierung korrekt berechnet werden. Bis dies der Fall war, hatte man mit überlappenden und falsch positionierten Karten zu kämpfen (Abbildung 38, Abbildung 39).

Mittlerweile kann mit den vorhandenen Möglichkeiten eine ansehnliche Landschaft generiert werden auf der sich frei bewegen kann (Abbildung 40).

Parallel wurde an einem neuen, mehr detaillierten (High Poly) Pflanzenmodell gearbeitet, einer Sonnenblume. Dies sollte erst nur ein Test sein wie sehr die Unity Engine ausgelastet werden kann, doch stellte sich heraus, dass dieses Level an Detail, selbst bei hunderten von Instanzen, kein Problem für die Unity Engine darstellt. Zuerst wurde mit der Blüte und deren Blättern begonnen. Zusammen mit dem Stiel und Stielblättern entstand die fertige Sonnenblume (Abbildung 41-44).

Während der Versuche, die Sonnenblumen zu platzieren trat plötzlich ein Effekt auf, der

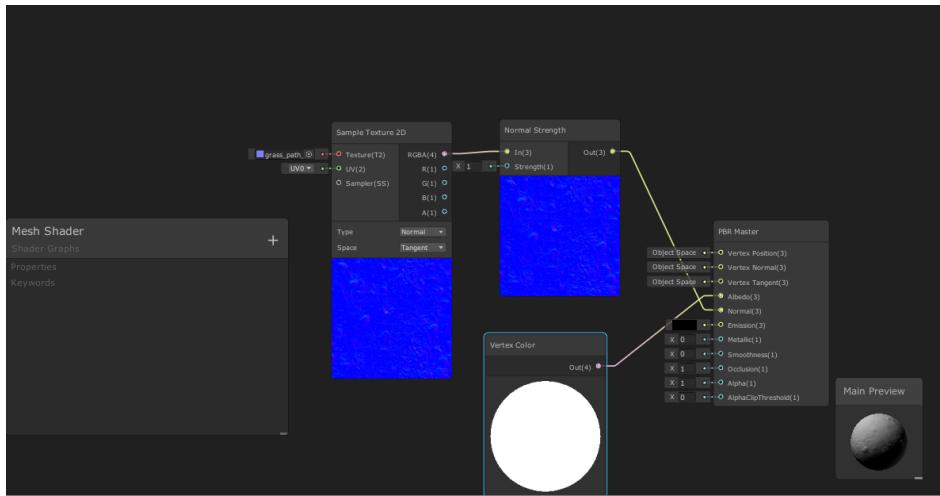


Abbildung 23: Mesh Shader Normalmap

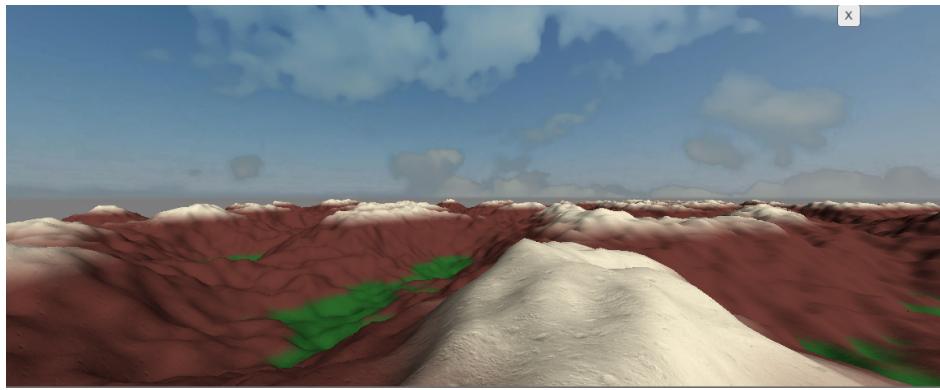


Abbildung 24: Landschaft mit Skybox und Normalmap

die Landschaften stufig aussehen lies (Abbildung 45). Der Effekt konnte nicht erklärt werden und es lies sich kein Fehler feststellen, der dafür verantwortlich sein könnte. Erst das Zurücksetzen zu einer vorigen Version (git reset) konnte Abhilfe schaffen.

Es folgten weitere Experimente mit den Parametern der Pflanzenverteilung um Blumenfelder zu erzeugen. Dabei spielt nicht nur das Aufkommen der Pflanze eine Rolle sondern ebenfalls ihre Ausrichtung - ein komplett homogenes Feld mangelt an Realismus (Abbildung 46). Versuche mit der Rotation der Objekte zu experimentieren führten unter anderem zu chaotischen Zuständen (Abbildung 47) auf Grund der Wahl der falschen Achse. Nach einigen Versuchen und einer Anpassung der Lichtverhältnisse konnte sich das Ergebnis endlich sehen lassen (Abbildungen 48-51).

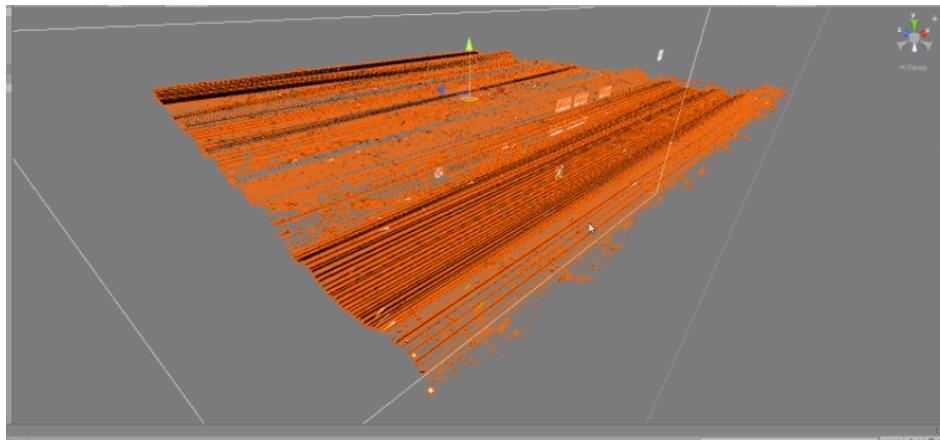


Abbildung 25: Compute Shader Fail

Ein Bild ist nicht in der Lage auszudrücken wie viel Zeit und Arbeit aufgebracht werden musste um beim Löschen eines Kartenabschnittes alle dazu gehörigen Pflanzen zu eliminieren (Abbildung 52). Die Struktur der für die Daten verwendeten Listen und Variablen musste mehrfach verändert werden und es hat einige Versuche gebraucht bis die Objekte, von denen es zu dem Zeitpunkt bereits mehrere Tausend gab, letztendlich korrekt gelöscht und der entsprechende Speicher freigegeben wurde. Da durch bestimmte Fehler in jedem Frame tausende Mesh Generatoren erzeugt wurden hatte man in dieser Phase mit häufigen Abstürzen der Software zu kämpfen.

Das User Interface wurde nun mit einem 'clear' Button ausgestattet der es ermöglicht auf Knopfdruck alle auf dem aktuellen Kartenabschnitt befindliche Pflanzen zu entfernen (Abbildung 53).

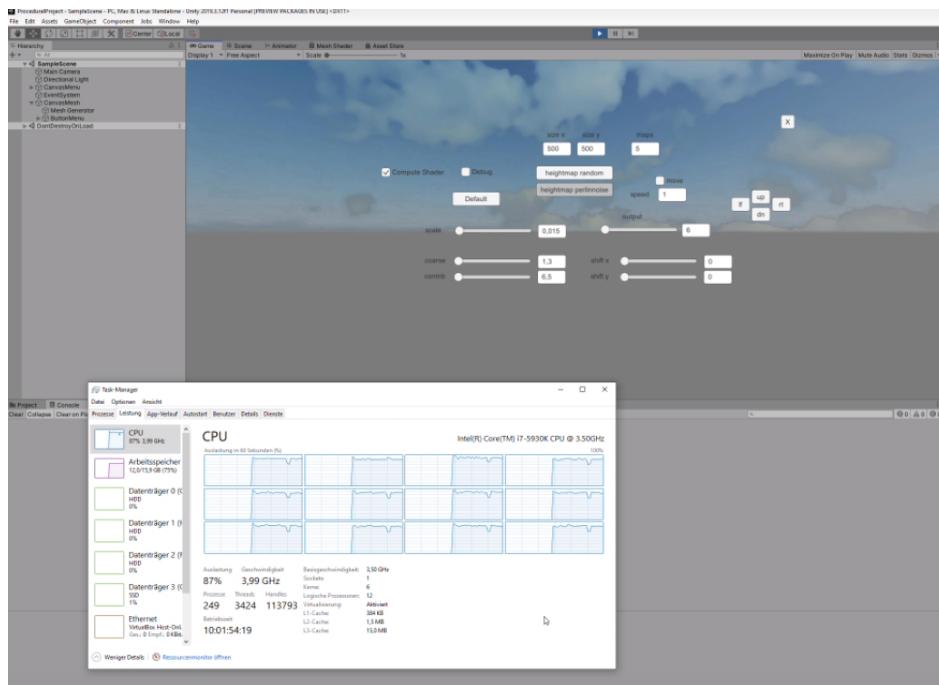


Abbildung 26: Threading funktioniert, aber hohe Auslastung

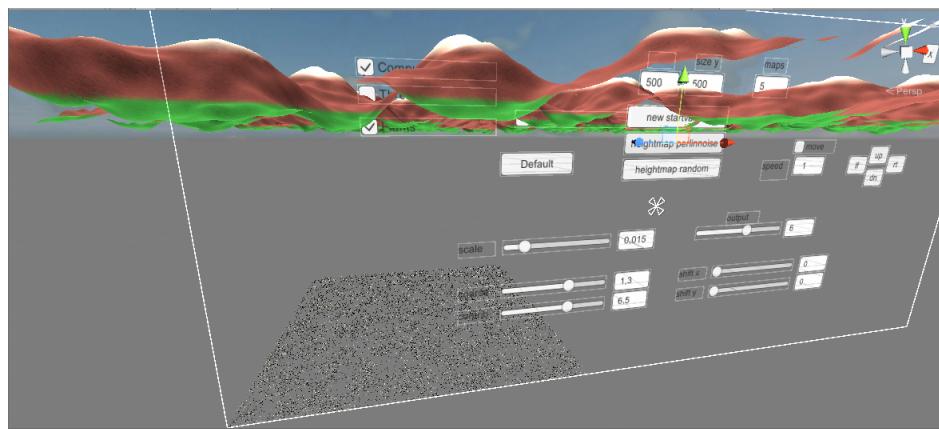


Abbildung 27: Objekte werden in falscher Höhe platziert

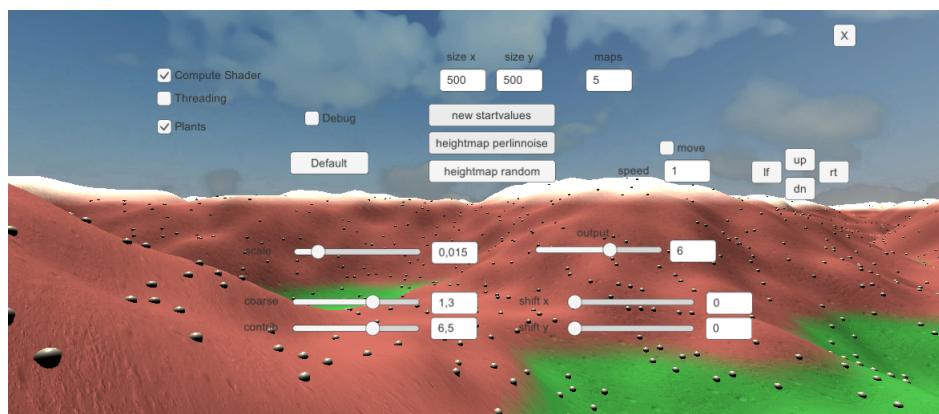


Abbildung 28: Objekte werden in korrekter Höhe platziert



Abbildung 29: Erstes Modell einer Pusteblume

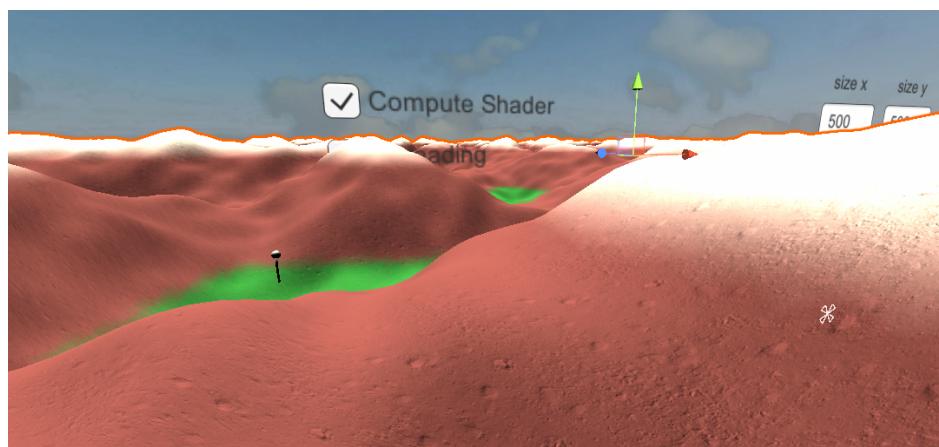


Abbildung 30: Erfolgreich die erste Pflanze platziert

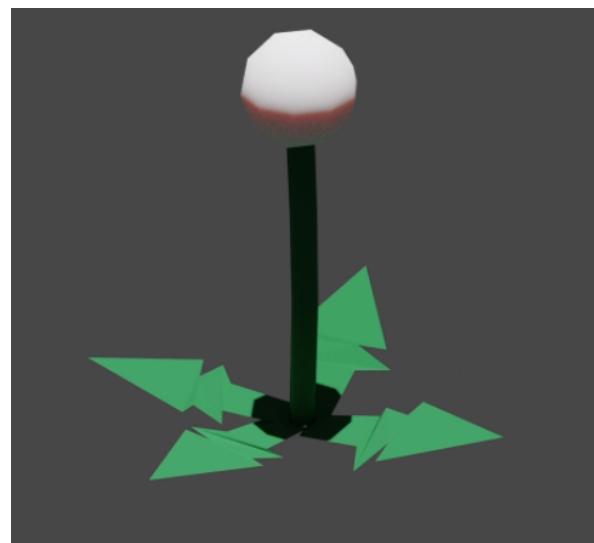


Abbildung 31: Pusteblume Version 2

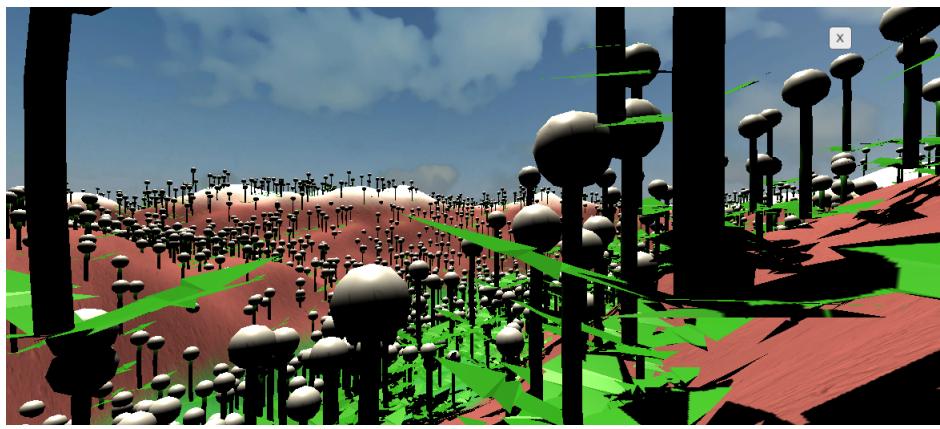


Abbildung 32: Pustebilume Invasion

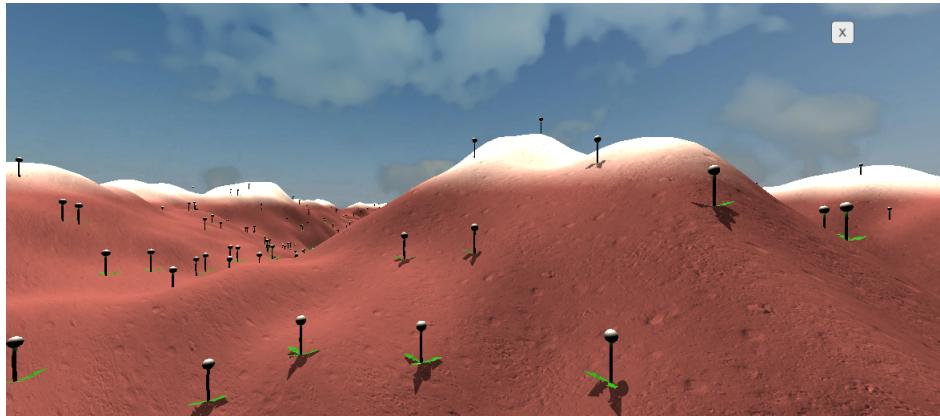


Abbildung 33: Erste sinnvolle Verteilung

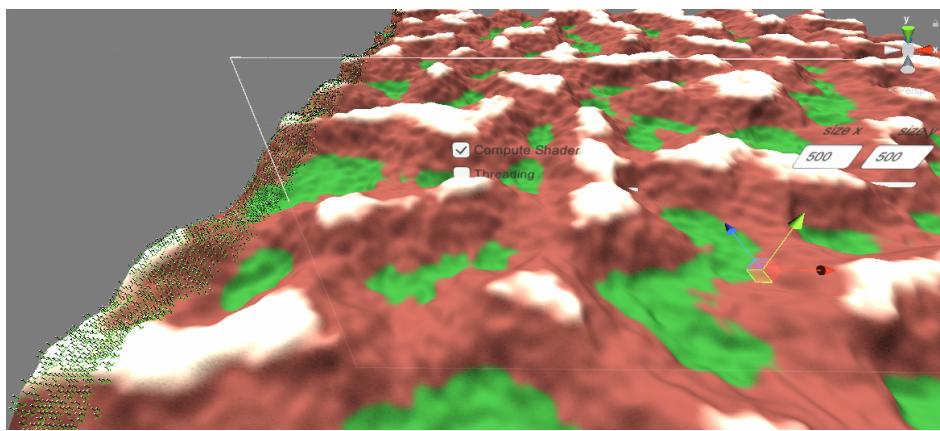


Abbildung 34: Erste Versuche einer alternativen Verteilung



Abbildung 35: Fliegende Pflanzen

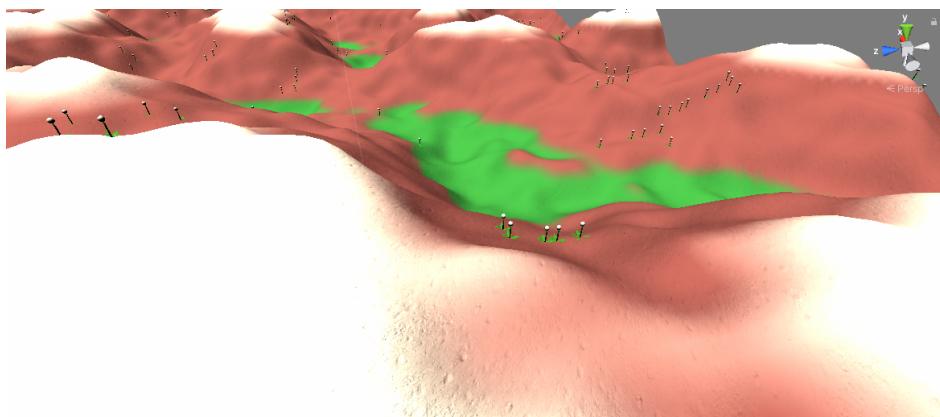


Abbildung 36: Alternative Verteilung

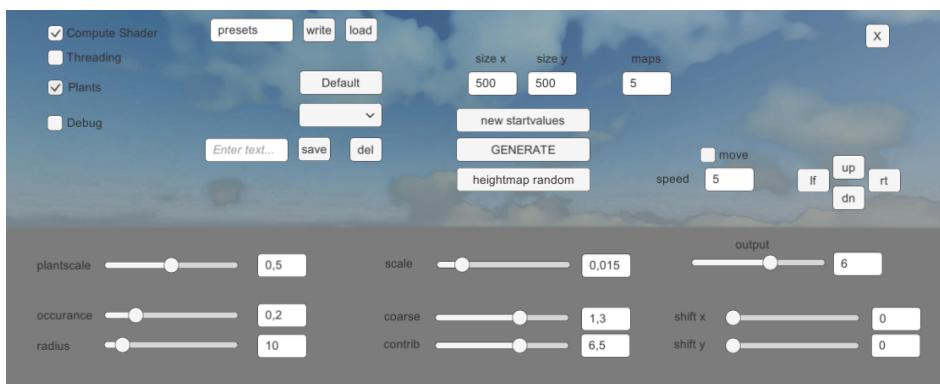


Abbildung 37: User Interface Version 4

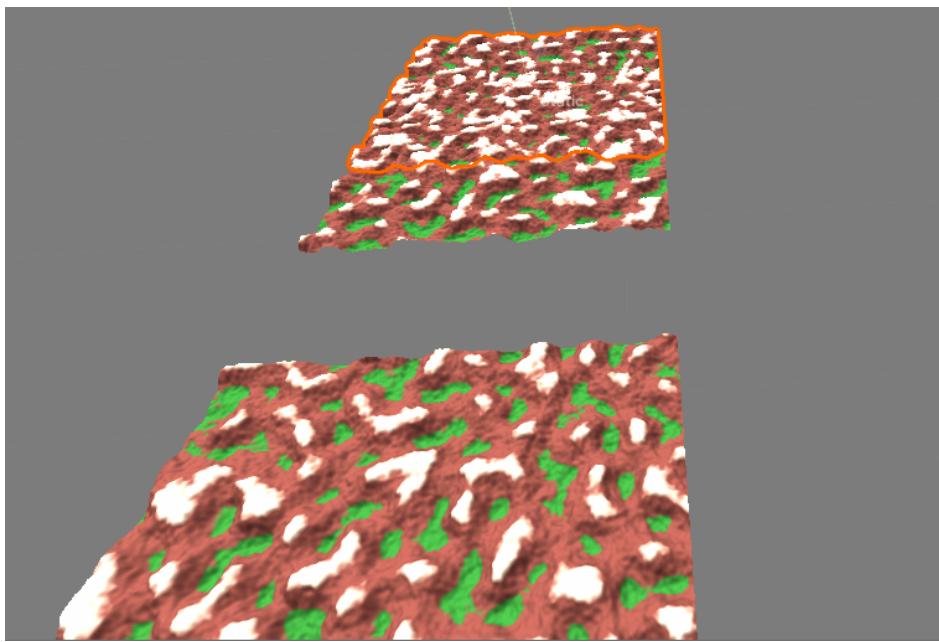


Abbildung 38: Überlappende Kartenabschnitte

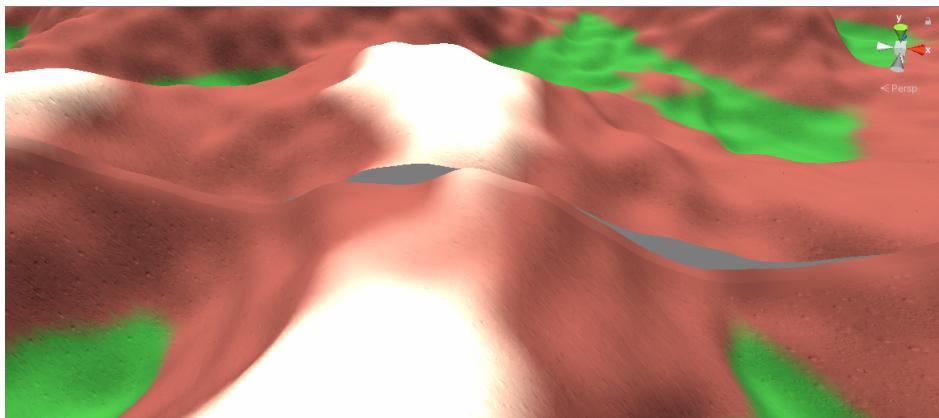


Abbildung 39: Fehlerhafter Übergang der Kartenabschnitte

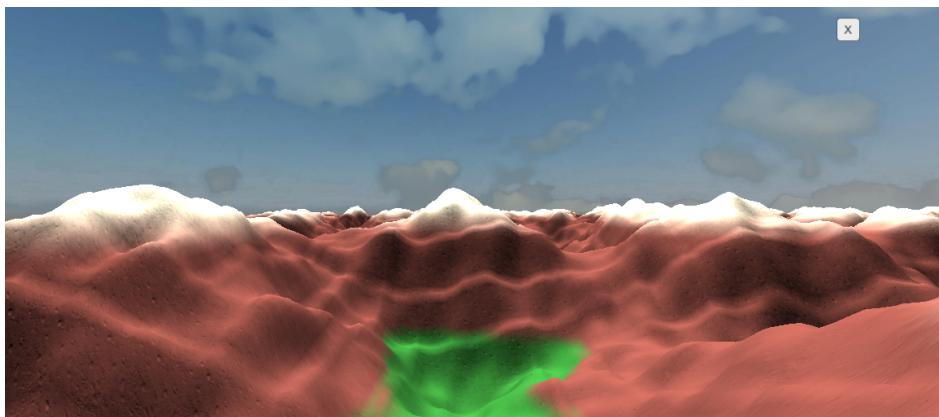


Abbildung 40: Ein schönes Zwischenergebnis

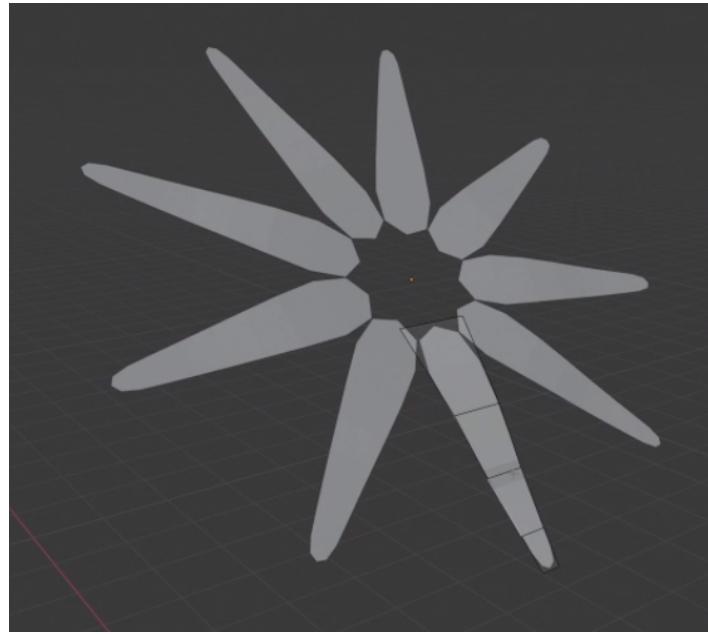


Abbildung 41: Prototyp der Blüte für die Sonnenblume Version 0

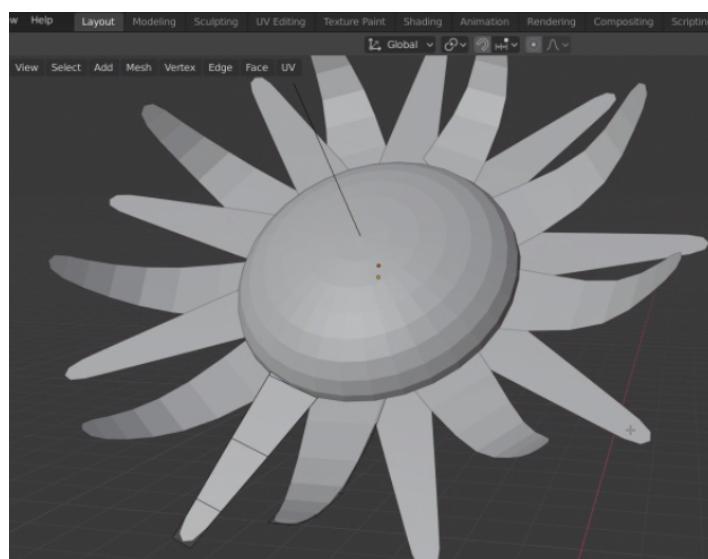


Abbildung 42: Blüte der Sonnenblume Version 1

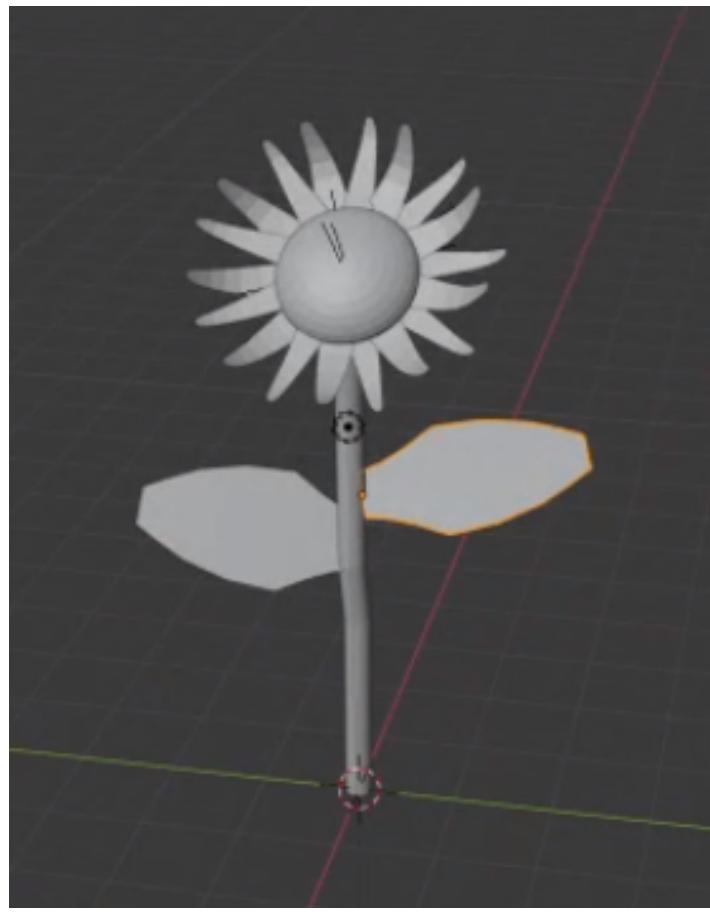


Abbildung 43: 3D-Modell der fertigen Sonnenblume



Abbildung 44: Sonnenblume zum ersten Mal in der Natur

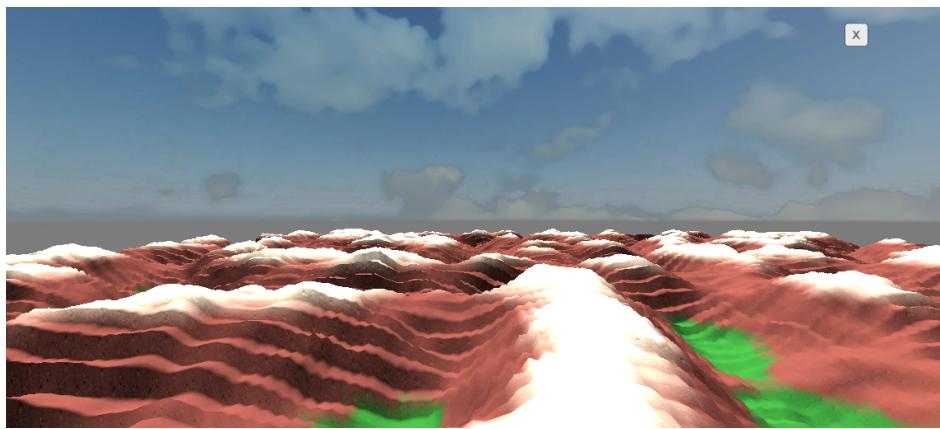


Abbildung 45: God bless the Backup

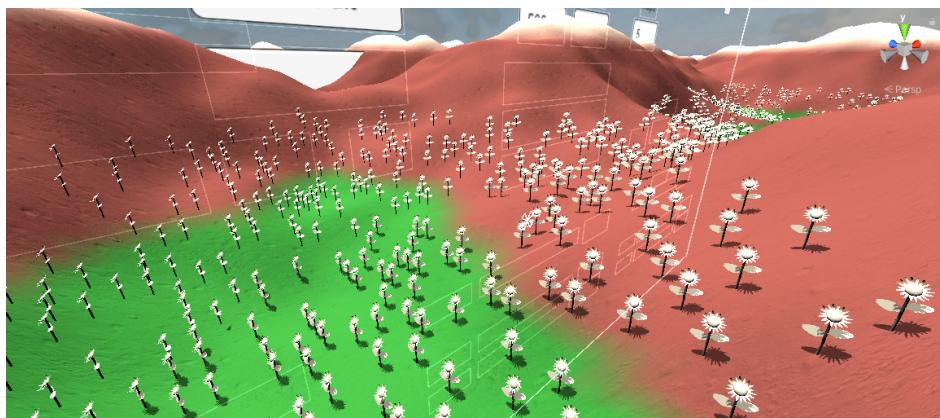


Abbildung 46: Homogenes Blumenfeld

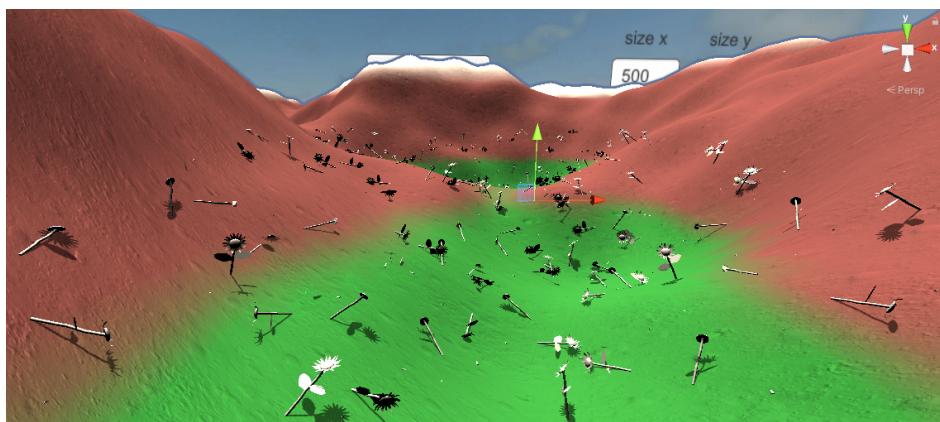


Abbildung 47: Blumen Schlachtfeld

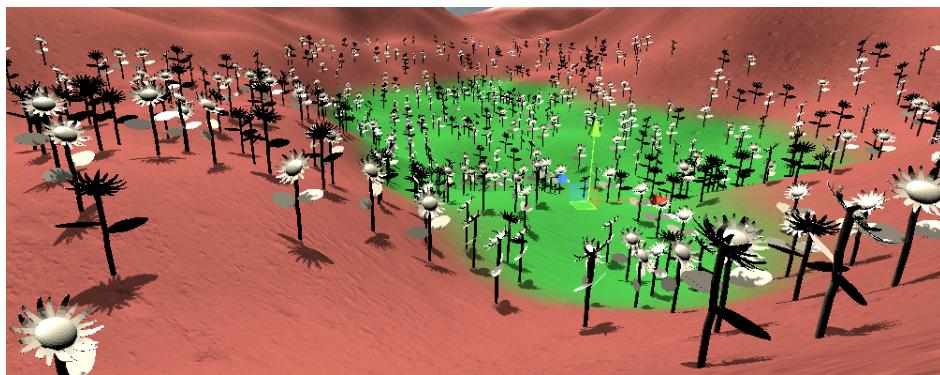


Abbildung 48: Rotationsversuche



Abbildung 49: Blumenfeld nach Licht-Anpassungen

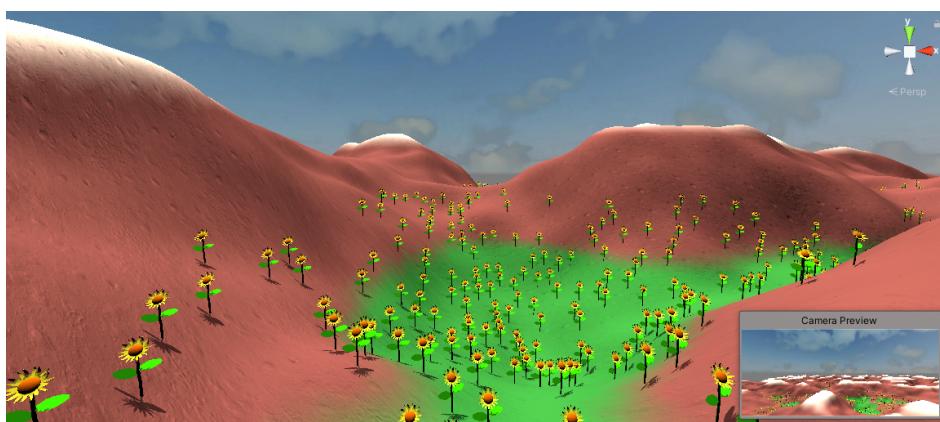


Abbildung 50: Blumenfeld Ergebnis

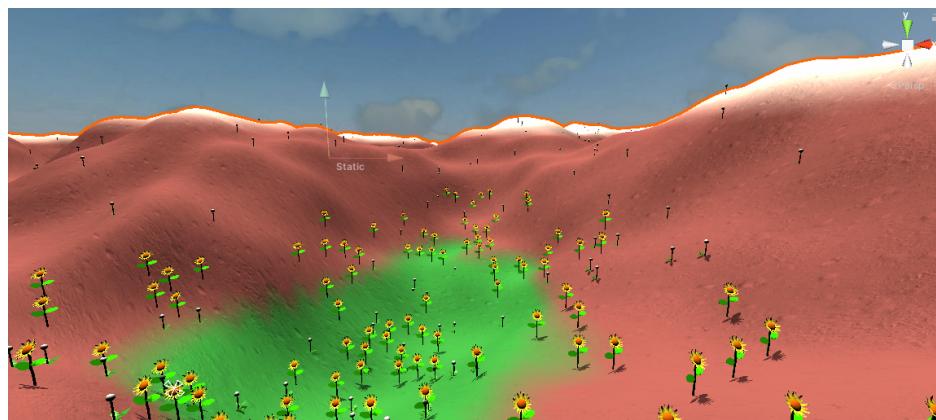


Abbildung 51: Einsatz verschiedener Pflanzen

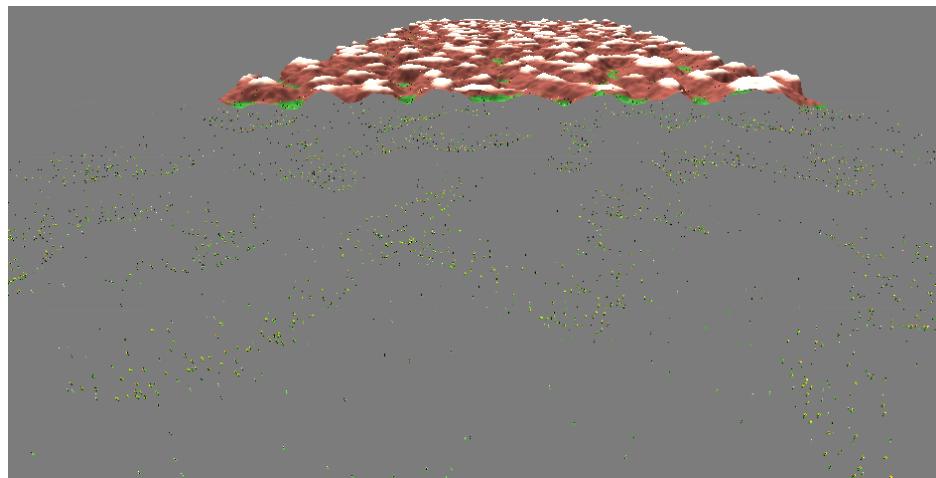


Abbildung 52: Pflanzen werden nicht gelöscht



Abbildung 53: User Interface Version 5

9.2 Debugging

Zur Fehlerbehebung und Analyse werden an verschiedenen Stellen Debug-Funktionen eingebaut. Beim Erstellen, Speichern oder Laden von Presets werden die entsprechenden String-Arrays auf der Konsole ausgegeben. Bei einer zufällig generierten Heightmap werden die eingestellten Optionen Ausgegeben. Es wird der Zeitpunkt ausgegeben an dem Objekte platziert oder gelöscht werden. Eine Ausgabe der Vertices des Mesh Generators muss zusätzlich im Code aktiviert werden, da es sich um sehr viele Daten handelt.

9.3 Nice to Have

Ideen zur weiteren Entwicklung der Landschaft wären eine 'echte' Sonne sowie weitere Pflanzen und andere Objekte wie Häuser oder Schneemänner. Weiter denkbar wären ebenso Fische, Vogelschwärme und andere Lebewesen. Die kreativen Möglichkeiten sind sehr vielseitig. Auch die Bewegung könnte weiter verbessert werden indem man die Möglichkeit bietet sich auf dem Boden in alle Richtungen fortzubewegen. Dazu müsste die Kameraposition während der Bewegung an die Höhe des Bodens angepasst werden. Auch sollte man die Kamera drehen können um den Blickwinkel zu verändern. Dazu sollte im besten Fall ein 'Player Controller' konzipiert werden durch den die einzelnen Bewegungen ferngesteuert werden können. So entsteht das Gefühl, sich auf der Landschaft fortzubewegen, was die Simulation sinnvoll erweitert.

10 Code

In diesem Abschnitt sind einige wesentliche Code-Beispiele des Programms in den Abbildungen 54 bis 65 aufgeführt.

```
public static float[,] createHeightMapPerlinNoise(
    int x, int y, float scale, float startx, float starty, float shiftx, float shifty)
{
    float[,] heightmap = new float[x, y];

    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < y; j++)
        {
            heightmap[i, j] = Mathf.PerlinNoise(startx+(shiftx+i)*scale, starty+(shifty+j) * scale);
        }
    }

    return heightmap;
}
```

Abbildung 54: Funktion zur Erstellung einer mit Perlin Noise generierten Heightmap.

```

public float[,] createHeightMapPerlinNoiseCS(
    int x, int y, float scale, float startx, float starty, float shiftx, float shifty)
{
    if (GameObject.FindGameObjectWithTag("debugtoggle").GetComponent<UnityEngine.UI.Toggle>().isOn)
        { print("START COMPUTE SHADER"); }

    PerlinInfo[] data = new PerlinInfo[x * y];
    float[] output = new float[x * y];

    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < y; j++)
        {
            data[i * y + j] = new PerlinInfo ( x, y, scale, startx, starty, shiftx+i, shifty+j);
        }
    }

    ComputeBuffer buffer = new ComputeBuffer(data.Length, 28);
    buffer.SetData(data);
    ComputeBuffer outputBuffer = new ComputeBuffer(output.Length, 4);
    outputBuffer.SetData(output);

    int kernel = shader.FindKernel("computeHeightMap");
    shader.SetBuffer(kernel, "dataBuffer", buffer);
    shader.SetBuffer(kernel, "output", outputBuffer);
    shader.Dispatch(kernel, data.Length/32, 1, 1);
    outputBuffer.GetData(output);

    buffer.Dispose();
    outputBuffer.Dispose();

    float[,] heightmap = new float[x, y];

    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < y; j++)
        {
            heightmap[i, j] = output[i * y + j];
        }
    }

    if (GameObject.FindGameObjectWithTag("debugtoggle").GetComponent<UnityEngine.UI.Toggle>().isOn)
        { print("END COMPUTE SHADER"); }

    return heightmap;
}

```

Abbildung 55: Hier wurde die Funktion um die Verwendung des Compute Shaders erweitert. Es ist eine Vorbereitung der Daten notwendig damit sie über einen Daten-Puffer an den Compute Shader übergeben werden können.

```

public static float[,] createHeightMapPerlinNoiseJobs(
    int x, int y, float scale, float startx, float starty, float shiftx, float shifty)
{
    var perlinInfoArray = new NativeArray<PerlinInfo>(x * y, Allocator.Persistent);
    var output = new NativeArray<float>(x * y, Allocator.Persistent);

    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < y; j++)
        {
            perlinInfoArray[i * y + j] =
                new PerlinInfo(x, y, scale, startx, starty, shiftx + i, shifty + j);
        }
    }

    var job = new generatePerlinJob
    {
        perlinInfoArray = perlinInfoArray,
        perlinOutput = output
    };

    var jobHandle = job.Schedule(x * y, 1);
    jobHandle.Complete();

    float[,] heightmap = new float[x, y];

    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < y; j++)
        {
            heightmap[i, j] = output[i * y + j];
        }
    }

    perlinInfoArray.Dispose();
    output.Dispose();
}

return heightmap;
}

```

Abbildung 56: Um Threading zu realisieren wird das Unity Job System genutzt. Die Daten werden als Array vorbereitet und an den Job übergeben.

```

float[,] heightmapCombined = new float[x, y];

for (int i=0; i<mapcount; i++)
{
    for (int j=0; j<x; j++)
    {
        for (int k=0; k<y; k++)
        {
            heightmapCombined[j, k] += (heightmaps[i][j, k]/((i*contribution+1)));
        }
    }
}

float highest = 0;
float lowest = 10000;

for (int i = 0; i < x; i++)
{
    for (int j = 0; j < y; j++)
    {
        if (heightmapCombined[i, j] > highest) highest = heightmapCombined[i, j];
        if (heightmapCombined[i, j] < lowest) lowest = heightmapCombined[i, j];
    }
}

for (int i = 0; i < x; i++)
{
    for (int j = 0; j < y; j++)
    {
        heightmapCombined[i, j] = Mathf.InverseLerp(lowest, highest, heightmapCombined[i, j]);
    }
}

```

Abbildung 57: Methode zur Kombination mehrerer Heightmaps. Die einzelnen Ebenen werden algorithmisch verrechnet und anschließend normalisiert.

```

#pragma kernel computeHeightMap
#include "noiseSimplex.cginc"

struct PerlinInfo
{
    int x;
    int y;
    float scale;
    float startx;
    float starty;
    float shiftx;
    float shifty;
};

RWStructuredBuffer<PerlinInfo> dataBuffer;
RWStructuredBuffer<float> output;

[numthreads(32,1,1)]
void computeHeightMap (uint3 id : SV_DispatchThreadID)
{
    output[id.x] = snoise(float2(dataBuffer[id.x].startx + (dataBuffer[id.x].shiftx) *
                                dataBuffer[id.x].scale, dataBuffer[id.x].starty + (dataBuffer[id.x].shifty) *
                                dataBuffer[id.x].scale));
}

```

Abbildung 58: Inhalt des Perlin Noise Compute Shaders inklusive Struct zur Datenübertragung mit Puffer.

```

#pragma kernel generateMesh

struct SquareInfo
{
    int vert;
    int x;
};

RWStructuredBuffer<SquareInfo> dataBuffer;
RWStructuredBuffer<int> output;

[numthreads(32, 1, 1)]
void generateMesh(uint3 id : SV_DispatchThreadID)
{
    output[id.x * 6] = dataBuffer[id.x].vert;
    output[id.x * 6 + 1] = dataBuffer[id.x].vert + dataBuffer[id.x].x + 1;
    output[id.x * 6 + 2] = dataBuffer[id.x].vert + 1;
    output[id.x * 6 + 3] = dataBuffer[id.x].vert + 1;
    output[id.x * 6 + 4] = dataBuffer[id.x].vert + dataBuffer[id.x].x + 1;
    output[id.x * 6 + 5] = dataBuffer[id.x].vert + dataBuffer[id.x].x + 2;
}

```

Abbildung 59: Inhalt des Compute Shaders für den Mesh Generator. Die Daten werden an dieser Stelle nicht verändert, sondern nur in die richtige Reihenfolge gebracht.

```

public void moveUp(float speed)
{
    Vector3 pos = camera.transform.position;

    pos.z += speed*Time.deltaTime;
    camera.transform.position = pos;

    if (pos.z > threshold)
    {

        for (int i = 0; i < GenerationFunctions.startwerte.Length; i++)
        {
            GenerationFunctions.startwerte[i].y +=
                (sizey - 1)*scale * Mathf.Pow(i + 1, coarse);
        }

        MeshGenerator newMeshGenerator = Instantiate(meshgen);
        print("MESH GENERATOR INSTANCIATED");

        float localScale = meshgen.transform.localScale.z;
        Vector3 newPosition = genPos;

        newPosition.z += (sizey - 2) * localScale;
        newMeshGenerator.transform.position = newPosition;

        foreach(DecorationFunctions deco in
            newMeshGenerator.GetComponents<DecorationFunctions>())
        {
            deco.inistializePlants();
        }

        generations.GeneratePerlinNoise(newMeshGenerator);
        print("PERLINNOISE GENERATED");

        meshGenerators.Add(newMeshGenerator);
        threshold += (sizey - 2) * localScale;
        genPos.z += (sizey - 2) * localScale;

        sectionCount++;
    }
}

```

Abbildung 60: Hier dargestellt die Methode zur Vorwärtsbewegung. Für jeden Mesh Generator muss eine jeweilige Pflanzenliste initialisiert werden wenn der Threshold überschritten wird.

```
if (sectionCount == 4)
{
    //print("count: "+sectionCount);
    print(meshGenerators.Count);

    print("CALL DESTROY ALL PLANTS");
    meshGenerators[0].destroyAllPlants();

    Destroy(meshGenerators[0].gameObject);
    meshGenerators.RemoveAt(0);

    sectionCount--;
}
```

Abbildung 61: Wenn die Anzahl der Mesh Generators 4 beträgt kann einer von ihnen gelöscht werden, weil er sich nicht mehr auf dem Bildschirm befindet. Die Pflanzen die auf diesem Kartenabschnitt lagen werden mit gelöscht.

```

public void placePlants (float[,] heightmap)
{
    int x = heightmap.GetLength(0);
    int y = heightmap.GetLength(1);

    float startx = Random.Range(0, 1000);
    float starty = Random.Range(0, 1000);

    float[,] probabilityDistribution = GameObject.FindGameObjectWithTag("heightmapbutton").
        GetComponent<GenerationFunctions>().createHeightMapPerlinNoiseCS(x, y, scale, startx, starty, 0, 0);

    float max = 0f;
    float min = 1000f;

    for (int i=0; i<x; i++)
    {
        for (int j=0; j<y; j++)
        {
            if (probabilityDistribution[i, j]>max) max=probabilityDistribution[i,j];
            if (probabilityDistribution[i, j]<min) min=probabilityDistribution[i,j];
        }
    }

    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < y; j++)
        {
            probabilityDistribution[i, j] = Mathf.InverseLerp(min,max,probabilityDistribution[i,j]);
        }
    }

    for (int i=0; i<x; i++)
    {
        for (int j=0; j<y; j++)
        {
            if (heightmap[i,j] > plantHeightMin && heightmap[i,j] < plantHeightMax &&
                probabilityDistribution[i,j] > probalilityThreshold)
            {
                if (Random.value < (1 - Mathf.Pow(heightmap[i, j], 0.25f)) *
                    Mathf.Pow(probabilityDistribution[i, j], 2) * occurance)
                { putPlant(i, j, heightmap[i, j], GetComponent<MeshGenerator>()); }
            }
        }
    }
}

```

Abbildung 62: Verteilung von Pflanzen: Die 'placePlants()' Methode erstellt eine Wahrscheinlichkeitsverteilung und bestimmt anhand dessen ob eine Pflanze gesetzt wird.

```

void putPlant(int x, int y, float height, MeshGenerator meshgen)
{
    string outputStr = GameObject.FindGameObjectWithTag("output").
        GetComponent<UnityEngine.UI.InputField>().text;

    float output = float.Parse(outputStr);

    float offsetX = meshgen.transform.position.x;
    float offsetHeight = meshgen.transform.position.y;
    float offsetY = meshgen.transform.position.z;

    float scaleX = meshgen.transform.localScale.x;
    float scaleHeight = meshgen.transform.localScale.y;
    float scaleY = meshgen.transform.localScale.z;

    toPlace.transform.localScale = new Vector3(0.5f*plantSize, 0.5f*plantSize, 0.5f);

    this.plantList.Add(Instantiate(toPlace, new Vector3
        (x * scaleX + offsetX, height * scaleHeight * output + offsetHeight, y * scaleY + offsetY),
        Quaternion.Euler(0f, -90f+Random.Range(-20f, 20f), 0f)));
}

```

Abbildung 63: 'putplant()' erzeugt ein Objekt an der durch die Variablen übergebenen Werte bestimmten Stelle.

```

void createShape()
{
    int x = heightMap.GetLength(0)-1;
    int y = heightMap.GetLength(1)-1;
    vertices = new Vector3[(x+1) * (y+1)];

    string outputInput = GameObject.FindGameObjectWithTag("output").
        GetComponent<UnityEngine.UI.InputField>().text;

    float output = float.Parse(outputInput);

    for (int k = 0, i = 0; i < y + 1; i++)
    {
        for (int j = 0; j < x + 1; j++)
        {
            vertices[k] = new Vector3(j, heightMap[j, i], i);
            k++;
        }
    }

    colors = new Color[vertices.Length];

    for (int i = 0; i < vertices.Length; i++)
    {
        colors[i] = gradient.Evaluate(vertices[i].y);
    }

    for (int i = 0; i < vertices.Length; i++)
    {
        vertices[i].y = vertices[i].y * output;
    }

    triangles = new int[x * y * 6];
    int vert = 0;

    SquareInfo[] data = new SquareInfo[x * y];

```

Abbildung 64: Vorbereitung der Daten um sie an den Compute Shader zu übergeben, der die Reihenfolge der Vertices festlegt.

```

for (int i = 0; i < y; i++)
{
    for (int j = 0; j < x; j++)
    {

        data[i*x+j] = new SquareInfo(vert, x);

        vert++;
    }

    vert++;
}

ComputeBuffer buffer = new ComputeBuffer(data.Length, 8);
buffer.SetData(data);
ComputeBuffer outputBuffer = new ComputeBuffer(triangles.Length, 4);
outputBuffer.SetData(triangles);

int kernel = shader.FindKernel("generateMesh");
shader.SetBuffer(kernel, "dataBuffer", buffer);
shader.SetBuffer(kernel, "output", outputBuffer);
shader.Dispatch(kernel, data.Length / 32, 1, 1);
outputBuffer.GetData(triangles);

buffer.Dispose();
outputBuffer.Dispose();

uvs = new Vector2[vertices.Length];

for (int i = 0, k = 0; i < y; i++)
{
    for (int j = 0; j < x; j++)
    {
        uvs[k] = new Vector2((float)j/(x*0.01f), (float)i/(y*0.01f));
        k++;
    }
}

if (GameObject.FindGameObjectWithTag("debugtoggle").
GetComponent<UnityEngine.UI.Toggle>().isOn) print("vert: " + vert);

```

Abbildung 65: Übergeben der Vorbereiteten Daten an den Compute Shader mit einem Compute Buffer.

11 Zusammenfassung und Ausblick

Mit Hilfe mathematischer Funktionen ist es gelungen, eine realistisch anmutende Landschaft zu gestalten auf der sich frei bewegen kann. Durch die unendliche Fortführbarkeit der Perlin Noise Funktion wird ein Rand niemals erreicht. Und dennoch können durch die Verwendung der selben Werte wiederholbare und damit nicht zufällige Umgebungen erzeugt werden. Die flüssige Bewegung wird einmal pro Kartenabschnitt kurz unterbrochen, auf Grund dessen, dass ein kompletter Kartenabschnitt immer in einem einzigen Frame berechnet wird. Dies wird durch Optimierung des Threadings lösbar sein, so dass die Daten schon während der Bewegung auf den nächsten Threshhold zu dem neue Kartenabschnitt im Hintergrund berechnet werden können. Ebenso konnten verschiedene Objekte in der Landschaft platziert werden. Dadurch, dass die Programmierung in diesem Abschnitt besonders dynamisch geregelt ist, kann mit geringst möglichem Aufwand direkt in Unity ein neues Deko-Objekt hinzugefügt werden, ohne einen Kompromiss in der Kontrolle der Verteilung eingehen zu müssen. Das Projekt wird mit dem Schwerpunkt der Parallelisierung fortgeführt. Ziel wird es sein die Berechnungen des Programms über die Rechnergrenze hinaus in einem 'Docker Swarm' zu verteilen. Dazu muss die Software entsprechend vorbereitet- und möglicherweise die Architektur grundsätzlich überdacht werden.

12 Literaturverzeichnis

- [1] KURI, DAVID: *GPU Raytracing in Unity*. 2018.
- [2] LAGUE, SEBASTIAN: *Coding Adventures*. https://www.youtube.com/playlist?list=PLFt_AvWsXl0ehjAfLFsp1PGaatzAwo0uK, 2019. [Online; accessed 15-April-2021].
- [3] SHORT, TANYA X. & ADAMS, TARN: *Procedural Generation in Game Design*. CRC Press & Taylor and Francis Group, 2017.

<https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html>

<https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html>

<https://docs.unity3d.com/Manual/Example-CreatingaBillboardPlane.html>

<https://docs.unity3d.com/Manual/GeneratingMeshGeometryProcedurally.html>

<https://docs.unity3d.com/Manual/JobSystem.html>

<https://docs.unity3d.com/Manual/JobSystemMultithreading.html>

<https://forum.unity.com/threads/c-job-system-example.518616/>

<https://learn.unity.com/tutorial/introduction-to-shader-graph#5f500900edbc2a0022843fb6>

<https://forum.unity-community.de/topic/13765-onclick-button-mit-script-erstellen/>

https://www.is.tu-darmstadt.de/media/bwl5s/is_lehre/richtlinienzur_fertigungswissarbeiten.de.pdf

A Anhang

A.1 Programmcode

Der vollständige Programmcode befindet sich in einem verfügbaren Git Repository:

<https://github.com/DieCoolenVonDerSchule/ProceduralProject>

Der Ordner muss als Projekt in Unity geöffnet werden.