

Landschaften aus dem Computer

Maturaarbeit von Roman Riesen
Betreuung durch Kai Rollé
10.10.2015

Inhaltsverzeichnis

1	Zielsetzung	1
2	Motivation	3
3	Konzepte und Begriffe in der Computergrafik	5
4	Übersicht über Blender	7
5	Eine kurze Einführung ins Programmieren	9
6	Die Landschaften	13
6.1	Terrain	13
6.2	Seen	16
6.3	Flüsse	18
6.4	Erosion	19
6.5	Wälder und Bäume	21
6.6	Material	23
6.7	Benutzeroberfläche	25
7	Fazit	27
8	Bilder	29
9	Danksagung	33

Abbildungsverzeichnis

1	Ein Modell einer einfachen Landschaft mit Wireframe-Modifier	i
3.1	Gelände auf Voxelbasis [1]	6
4.1	Zufällig eingefärbe Vertices einer Landschaft	8
6.1	Diamond-Square Funktionsweise	14
6.2	Darstellung der Wertung der Felder mit gausssscher Verteilung	15
6.3	Vergleich einer Landschaft ohne und mit zweifachem Smoothing	16
6.4	Wasser Simulation: Links 1000 Zyklen und rechts 10.	17
6.5	Wassersimulation mit unvorsichtig gewählten Parametern	18
6.6	Früher Prototyp einer Flusssimulation	19
6.7	Oben: Ohne Erosion Unten: Mit invertierter Erosion	20
6.8	„Tree Height“-Vertex Gruppe generiert durch mein Add-on	21
6.9	R-Pentomino Abfolge der ersten fünf Schritte im Game of Life	22
6.10	Bildschirmfoto vom Game of Life mit meinen Regeln	22
6.11	Bildschirmfoto von Node-Editor und Viewport	24
7.1	Oben: Landschaft auf Basis von realer Höhenkarte Unten: Landschaft aus meinem Add-on	27
8.1	Bild aus einem Turtle Programm, welches als Übung diente	29
8.2	Eine Landschaft mit 2 Array-Modifiern	30
8.3	Auch mit 2 Array-Modifiern, jedoch interessanter eingefärbt	30
8.4	Wüste mit Bergen im Hintergrund	31
8.5	Die Waldverteilung wurde durch das Add-on berechnet	31
8.6	Etwas polygonärmerer Ansatz	32
8.7	Herbstlicher Wald in kahler Berglandschaft	32

Kapitel 1

Zielsetzung

Ziel meiner Maturaarbeit ist ein Add-on für Blender¹, einer Open-Source 3D Software, zu erstellen. Es soll eine Möglichkeit bieten, zufällige Landschaften zu generieren - mit einer einfachen Wassersimulation für Flüsse und Seen. Ebenso soll es Erosion und die Verteilung von Pflanzen berechnen können. Dabei habe ich den Anspruch, dass sie nicht allzu unrealistisch aussehen sollen.

¹www.blender.org

Kapitel 2

Motivation

Während es vor 50 Jahren noch revolutionär war, dass ein Computer nicht nur Text ausgeben konnte, sondern Bilder, ist heute das Umgekehrte kaum mehr vorstellbar. Von der Lochkarte bis zum Internet - von der Kontrollleuchte bis zur Informationsrevolution. Die Computertechnik beeinflusst das Leben aller. Wie die Kunst dies bereits Jahrtausende vorher tat. Die 3D-Animation ist an der Spitze von beidem zugleich. Sie erforscht und erweitert unser Verständnis von Kunst und bringt Computer an ihre Grenzen.

Persönlich interessiert mich auch das Zusammenspiel von Technik und Kunst - von Foto über Film bis Spiel. Deshalb wollte ich eine Maturaarbeit in diesem Bereich schreiben. Bald kam ich auf die Idee, ein Add-on für Blender zu schreiben. Zuerst experimentierte ich etwas mit Mandelbulbs, eine Art dreidimensionale Fraktale, herum. Doch schon nach Kurzem war klar, dass dies in einem eigenen Programm besser aufgehoben ist als in einem Blender Add-on. Ich entschied mich die Mandelbulbs und nicht die Idee des Add-ons fallen zu lassen. Herr Rollé hatte bereits während eines Treffens erwähnt, dass die Hintergrundlandschaften in Filmen auch auf fraktalen Methoden beruhen. Nach etwas Recherche entschied ich mich ein Add-on zum Generieren von Landschaften zu machen.

Kapitel 3

Konzepte und Begriffe in der Computergrafik

Wie wir später sehen werden, nehme ich eine sogenannte Höhenkarte als Grundlage für das Terrain-Model. Diese enthält für jeden Punkt auf einer Fläche jeweils die Höhe. Als Bild dargestellt ist schwarz dann hoch und weiss tief. Da so nur eine Höhenangabe pro Punkt existiert und alles darunter als regelmässig gefüllt angenommen wird, können mit dieser Methode keine Höhlen oder Überhänge dargestellt werden.

Ausser es wird ein 3D-Modell erstellt, welches beliebig bearbeitet werden kann. Um dieses mithilfe des Computers darzustellen werden meistens *Polygone* verwendet, welche aus drei oder mehr Punkten im Raum bestehen (meist *Vertices* genannt, Einzahl: *Vertex*), die mit Kanten (*Edges* genannt) verbunden sind. Die Fläche, die durch die Vertices aufgespannt wird, wird *Face* genannt. Faces sind das eigentlich Wichtige für die Bildberechnung, oder auch *rendern*. Rendern ist der Vorgang, bei dem aus den Positionen und Ausrichtung von Polygonen, Kamera und Lichtern das Bild berechnet wird. Der Programmteil, der dafür verantwortlich ist, wird oft Render-Engine genannt. Ebenfalls sind die Normalen, Vektoren die 90° Grad auf den Faces stehen, wichtig damit das Render-Programm weiss, welche Seite innen und welche aussen ist. Eine sehr verbreitete Methode Polygon-Objekt-Daten zu organisieren und zu speichern ist das .obj-Format. Dabei werden die Vertices durch ihre kartesischen Koordinaten, die Faces durch die Indices der Vertices definiert. Ein einfaches Beispiel:

```
o simplesDreieck
# Koordinaten der Vertices:
v 0.0 0.0 0.0
v 0.0 1.0 0.0
v 1.0 0.0 0.0
# Angaben der Faces:
f 1 2 3
```

Nach dem o folgt der Name des Objektes. Alles nach dem # ist ein Kommentar, also eine Bemerkung, die keinen Einfluss auf die eigentlichen Daten hat und nur für das Verständnis von anderen Personen da ist. Das v am Beginn einer Zeile

markiert, dass die folgenden drei Zahlen die Position eines Vertex darstellen. Das f kennzeichnet, dass die folgenden Zahlen besagen, welche Indizes, wie miteinander zu einem Face verbunden werden sollen. Die Edges müssen nicht spezifisch angeben werden, da ein Face die Information über die dazugehörigen Edges ja bereits enthält. Dadurch, dass dies genau so in ein Textdokument geschrieben wird, ist es auch für Menschen verständlich, im Gegensatz zu binären Dateiformaten, die nur als Abfolge von 0 und 1 gespeichert werden.

Eine andere Methode 3D-Objekte darzustellen sind die sogenannten Voxel. Das sind meist kleine Würfel, die das Volumen des Objektes ausfüllen. Diese Variante wird dort eingesetzt, wo das Volumen der Objekte selbst wichtig ist, etwa in der Medizin (z.B. MRI-Daten) oder in der Darstellung von komplexen Geländeformen, etwa Überhängen wie in Abb. 3.1 Ein voxelbasierter Ansatz wäre sicher interessant gewesen. Jedoch würde das Resultat in Blender unhandlich sein, da Blender Voxel nicht direkt unterstützt. Somit würde die Landschaft - zumindest ohne weitere Verarbeitung - aus vielen Würfeln bestehen, welche selbst polygonal aufgebaut sind. Deshalb blieb ich bei den Höhenkarten.



Abbildung 3.1: Gelände auf Voxelbasis [1]

Kapitel 4

Übersicht über Blender

3D-Programme sind meist eine teure Angelegenheit. So kosten die Marktführer 3ds Max und Maya, beide von Autodesk, je 133\$ pro Lizenz und Monat. Hier kommt Blender ins Spiel, denn dies ist gratis und Open Source. Das heisst, dass alle den Quellcode einsehen und modifizieren dürfen. Auch wenn trotz schneller Weiterentwicklung hier und da einige Features fehlen, welche etwa in Maya vorhanden sind, ist es dennoch eine professionell einsetzbare Software. Nicht zuletzt wegen der Vielzahl an Add-ons die es dafür gibt - von der Generierung von Bäumen bis hin zu deutlichen Veränderungen in der Bedienung.

Blender findet, wie die meisten anderen 3D-Grafik-Programme ein breites Einsatzspektrum. Von digitalen Spezialeffekten für Filme, über die Gestaltung von Modellen für Spiele bis hin zu wissenschaftlichen Anwendungen. Für Letzteres ist Blender besonders geeignet, da es seit 2003 unter einer *Open Source Lizenz* veröffentlicht wird. Somit kann es nicht nur gratis heruntergeladen werden, sondern erlaubt auch die Modifikation des kompletten Quellcodes, also der Programmierung selbst, um Anpassungen zu machen.

Dies ist jedoch oft gar nicht nötig, da es auch eine sogenannte API¹ hat. Diese ist in der Programmiersprache Python geschrieben und ermöglicht es, jeden Befehl welcher auch von Hand ausgeführt werden könnte, in ein Programm zu schreiben, um Prozesse zu automatisieren. Ein Beispiel dafür wäre etwa das zufällige Einfärben aller Vertices, wie in Abb. 4.1.

¹Application Programming Interface

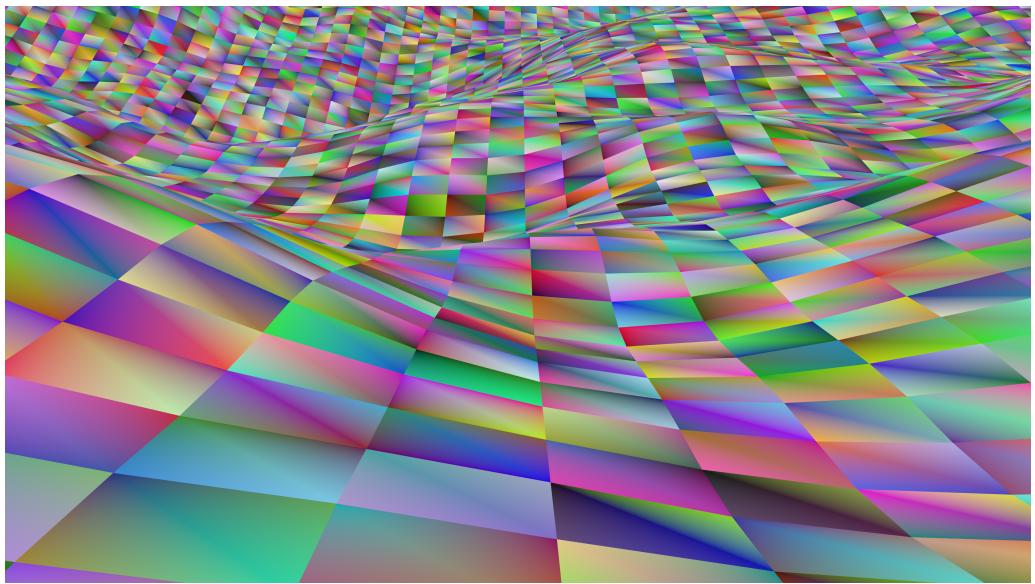


Abbildung 4.1: Zufällig eingefärbte Vertices einer Landschaft

Kapitel 5

Eine kurze Einführung ins Programmieren

Wer in Blender ein Add-on programmieren will hat nicht eine allzu hohe Einstiegs-hürde. Zum einen gibt es bereits ein Layout für die Arrangierung der Oberfläche namens „Scripting“. Zum anderen wird eine Info-Box angezeigt, in welcher - nebst Details zum Wert, den sie beeinflussen - auch der API-Befehl enthalten ist mit dem dieser in Python aufgerufen oder verändert werden kann, wenn der Mauszeiger eine Weile über einem Eingabefeld schwebt. Mit etwas Übung muss also kaum mehr die Dokumentation neben Blender offen sein um an einem Add-on zu arbeiten. Durch einen Rechtsklick auf solche Eingabefelder lässt sich, zumindest der letzte Teil des Pfades zum Wert, in Python kopieren. Persönlich habe ich jedoch einen Grossteil der Arbeit im Text-Editor „Atom“ geschrieben, und daneben die Dokumentation geöffnet gehabt.

Im folgenden Teil werden einige Programmierkonzepte beschrieben. Dies hat im Grunde nichts mit meiner Maturaarbeit zu tun und soll nur einen - hoffentlich nicht allzu steilen - Einstieg in das Programmieren ermöglichen, so dass die Code-Stückchen, die immer Mal wieder vorkommen werden, verstanden werden können. Variablen sind ein wichtiger Teil des Programmierens. Dies sind Namen für Zahlen, Buchstabenabfolgen oder anderer Objekte. Um eine Variable in Python zu definieren wird der Name der Variable, dann ein „=“ und zum Schluss der Wert geschrieben. Um einen Wert auszugeben werden „Print(Variable)“ benutzt. Einige Beispiele in einer Python Konsole:

```
1 >>> nummer1 = 6
2 >>> nummer2 = 7
3 >>> print(number1 * number2)
42
5 >>> a = "Apfel"
6 >>> b = "Banane"
7 >>> print(a + " & " + b)
Apfel & Banane
```

Ein Konsolenfenster ist in Python daran zu erkennen, dass es ein „>>>“ zu Beginn jeder Zeile hat, ausser es handelt sich um eine Ausgabe. Für komplexere

Programme, die in Blender geschrieben werden, ist es ratsam den Text-Editor zu benutzen.

```
1 def begruessung (Name):  
2     #Ein hilfreicher Kommentar  
3     print("Willkommen ",Name)  
4     for i in range (0,100,50):  
5         if i==50:  
6             print("Hallo nochmals")  
7     return('Fertig')
```

Die erste Zeile definiert eine Funktion mit dem Argument „Name“. Alles was eingerückt ist, ist ein Teil der Funktion. Ja - Begrüßung ist mit Absicht klein geschrieben. Denn Funktionen werden fast nie gross geschrieben. Ein Argument ist dabei eine Variable, die an eine Funktion übergeben wird. Alles nach einem `#` ist auch hier, wie im .obj Dateiformat ein Kommentar und nur zum Verständniss für den Leser da. Es gibt auch die Möglichkeit mit drei Anführungszeichen einen Kommentar über mehrere Zeilen zu markieren. Die dritte Zeile ist eine Ausgabe in die Konsole, welche „Willkommen“ und dann den Wert von Name ausgibt. Die vierte Zeile ist eine For-Schleife, die i zuerst auf Null setzt und jedes Mal um 50 erhöht, bis $i = 100$ ist. Die fünfte Zeile enthält eine Wenn-Bedingung. Alles was eingerückt ist, wird ausgeführt, wenn die Bedingung wahr ist. Nebst dem `==`, welches testet ob die beiden Werte genau gleich sind, existieren auch noch `>` (grösser als), `<` (kleiner als), `>=` (grösser gleich) und `<=` (kleiner gleich.). Return gibt dann den Wert, der danach genannt wird zurück. Die Klammern sind dabei nicht unbedingt nötig. Wenn dieses Programm ausgeführt wird passiert - nichts. Denn es wird zwar eine Funktion definiert, sie wird jedoch nicht aufgerufen. Wenn nun jedoch die Zeile

```
1 b = Begrueßung("Hans Mustermann")
```

hinzugefügt wird, wird die Funktion aufgerufen. Somit erscheint in dem Debugging Fenster, welches über „Window“ in Blender aktiviert werden muss:

```
1 Willkommen Hans Mustermann  
2 Hallo nochmals
```

und b hat nun den return-Wert, also "Fertig".

Eine Funktion ist praktisch, da damit das Programm strukturiert und Abschnitte, die häufig gebraucht werden, nicht jedes Mal neu geschrieben werden müssen. Das Argument ist dabei eine Variable, welche nur innerhalb der Funktion aufgerufen werden kann, welche für den übergebenen Wert steht.

Ein sehr wichtiges Element (fast) jeder Programmiersprache sind Listen. Dies sind Objekte, die andere Objekte in einer fixen Reihenfolge enthalten. Die Elemente werden über ihren Index angesprochen. Das erste Element hat dabei in (beinahe) jeder Sprache den Index 0. Es ist auch möglich, dass Listen Listen enthalten. Je nachdem wie viele Listen ineinander verschachtelt sind, wird von 1-, 2-

(oder mehr) hierarchischen, manchmal auch dimensionalen Listen gesprochen. Ein Beispiel für eine 2-dimensionale Liste:

```
|>>> meineListe = [[1,2,3], [4,5,6], [7,8,9]]
```

Die Werte einer solchen werden folgendermassen angesprochen:

```
1 |>>> print(meineListe[1])  
| [4,5,6]  
3 |>>> print(meineListe[2][1])  
| 8
```

Der Wert der ersten eckigen Klammer ist also der Index der Liste 1. Ordnung, was in diesem Fall eine Liste ist. Der Wert in der zweiten ist der Index der 2. Ordnung, in diesem Fall die Zahl 8. Eine solche Liste ist also besonders geeignet, um ein Feld darzustellen. Denn die Werte der beiden Klammern entsprechen den X und Y Koordinaten des Feldes.

Manchmal werden jedoch auch pseudozufällige Werte benötigt. Dies wird durch das Modul „random“ erreicht. Ein Modul ist eine Pythondatei, deren Funktionen in eine andere importiert werden kann. Um ein solches zu importieren wird die Zeile

```
|from random import *
```

an den Beginn des Programms oder in die Konsole geschrieben. Das * steht dafür, dass alles aus diesem Modul importiert wird. Es ist jedoch auch möglich nur einzelne Funktionen zu importieren. Dazu listet man anstatt das Sternchen die gewünschten Funktionen, mit Koma getrennt, auf. Ein Pseudo-Zufallsgenerator ist ein Programm, dass eine Zahl als Input hat und Werte ausgibt, die zufällig erscheinen. Das heisst, dass es keine periodischen Wiederholungen gibt. Sie sind Pseudo, da sie bei selbigem Input immer dieselbe Zahlenfolge ausgeben. Somit ist es nicht wirklich zufällig. Normalerweise wird als Anfangszahl die Systemzeit genommen. Dieser Wert kann jedoch auch mit der Funktion `Seed()` definiert werden. Dies ermöglicht es die Zahlenfolge später reproduzierbar zu machen.

```
1 |>>> from random import random, seed  
2 |>>> seed(5)  
3 |>>> random()  
| 0.6229016948897019  
5 |>>> seed(2)  
| 0.9560342718892494  
7 |>>> seed(5)  
|>>> random()  
| 0.6229016948897019
```

Das ist auch schon alles, was benötigt wird um mein Programm zu verstehen.

Kapitel 6

Die Landschaften

„Art challenges technology, but technology inspires the art.“

- John Lasseter (*Regisseur und Animator, u.a. Toy Story*)

Dieses Zitat fasst in meinen Augen den Zusammenhang zwischen Technologie und Kunst sehr schön zusammen:

Jede technische Neuerung inspiriert Menschen zu neuen Ausdrucksformen und eventuell zum Verlangen nach noch fortgeschrittenen technischen Möglichkeiten.

Es gibt bereits ein Add-on in Blender für die Generierung von Landschaften namens „ant landscape“. Es fühlt sich jedoch in vielen Bereichen etwas beschränkend an, auch wenn es viele Parameter zum Verändern der eigentlichen Landschaft enthält. Denn fehlt jegliche integrierte Möglichkeit zum Berechnen von Flussverläufen oder Seen. Auch müsste die Verteilung von Bäumen im „Weight Paint“-Modus von Hand gezeichnet werden. Diese Probleme versuchte ich durch mein Add-on zu lösen.

6.1 Terrain

Zu Beginn wird eine Möglichkeit zur Generierung einer Landschaft benötigt. Ein sehr oft verwendeter Algorithmus zur Erstellung von Terrain ist der sogenannte Perlin Noise. Erfunden wurde dieser 1982 durch Ken Perlin für einige der Spezialeffekte im Tron Film. Dieser ist jedoch schon als generierte Textur in Blender implementiert und steht im „ant landscape“ Add-on zur Auswahl.

Deshalb wurde der Diamond-Square Algorithmus benutzt, welcher auch sehr beliebt ist. Dieser ist 1982 von Fournier, Fussell und Carpenter an der „SIGGRAPH“¹ vorgestellt worden und ermöglicht die Erstellung von Höhenkarten, welche relativ gut aussehenden (Gebirgs-) Landschaften entsprechen.

Dieser funktioniert so:

- Die minimale Höhe der Landschaft definieren. H gleich die Hälfte davon setzen.
- Die Abnahme von H, r, definieren.

¹Special Interest Group on Graphics and Interactive Techniques

- Ein Quadrat mit zufällig hohen Eckpunkten nehmen. Das Quadrat in regelmässigen Abständen mit zufälligen Werten besetzen. Dies dient dazu interessantere Landschaften zu kreieren.
- Den Diamond-Schritt: Den Mittelpunkt des Quadrates, die Höhe der Durchschnittshöhe der umgebenden Punkte plus einen zufälligen Wert zwischen -H und H geben.
- Der Square-Schritt: Nun die Mittelpunkte der Seiten des ersten Quadrates berechnen, ebenfalls mit dem Durchschnitt der benachbarten Punkthöhen, zu welchem jeweils noch ein zufälliger Wert, auch zwischen -H und H, addiert wird.
- H durch r teilen.
- Nun die letzten drei Schritte für jedes so entstandene Quadrat wiederholen, sofern die gewünschte Auflösung noch nicht erreicht ist.

In Abb. 6.1 ist dieser Prozess auf einem Feld der Grösse 5 bildlich dargestellt.

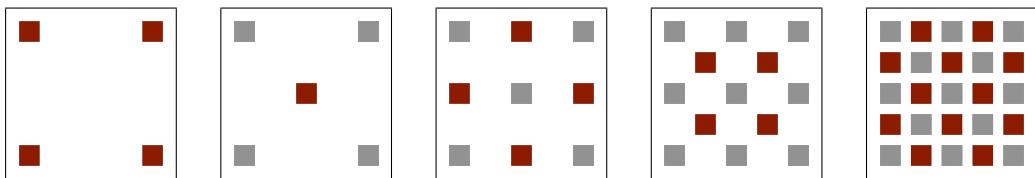


Abbildung 6.1: Diamond-Square Funktionsweise

Würden Linien zwischen den zu berechnenden Punkten gezogen, so ergäben sich im ersten Schritt Quadrate, im zweiten sogenannte Diamanten. Daher der Name Algorithmus.

Für die Implementierung, also das Umsetzen in einem Programm, wurde zuerst eine zweidimensionale Liste der gewünschten Grösse erstellt, mit zufälligen Werten für die vier Ecken.

Bei den Schritten, bei denen Quadrate entstehen (ausser beim Allerersten) ist jeweils einer der benötigten Werte nicht auf dem Feld. Die Lösung, die dazu in [4] verwendet wurde war jeweils den Wert auf der anderen Seite des Feldes zu nehmen. Dafür schrieb ich eine Funktion, die jeweils den Rest der Division der X- und Y-Koordinate durch die Anzahl Punkte in der Liste zurückgab. Somit konnte ich problemlos auch einen Wert für einen vorher zu grossen Index zurückbekommen. Es verhält sich so, als ob in jede Richtung unendlich viele Listen angehängt wären. In Python:

```

1 def punktHolen(x,y,liste):
2     groesse = len(liste)
3     x=x%groesse
4     y=y%groesse
5     return liste[x][y]

```

Die Funktion für das Verändern des Zufallsbereiches ist so definiert:

```

1 | def zufall(self):
2 |     r=uniform(-H,H)/(r**schritt)
3 |     return r

```

Die Funktion „uniform()“ gibt eine zufällige Zahl zwischen den beiden Argumenten zurück. Sie ist Teil des **random**-Modules. Zwei Sternchen bedeuten „hoch“. Ich wählte also eine exponentielle Abnahme der Zufallsverschiebung.

Leider hinterlässt der Diamond-Square-Algorithmus relativ unschöne Spitzen in der Landschaft. Vor allem bei einem hohen Wert für die Abnahme der maximalen Verschiebung von Schritt zu Schritt. Dies lässt sich mit einem relativ einfachen Trick beheben: Es wird jeder Punkt seinen Nachbaren angenähert.

Genauer: Ich bediene mich der gaußschen Unschärfe. Wie der Name vermuten lässt, wird diese oft für das nachträgliche Hinzufügen von Unschärfe in Bildern verwendet. Dazu werden alle acht Nachbarfelder entsprechend einer gaußschen Verteilung gewichtet.

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

Abbildung 6.2: Darstellung der Wertung der Felder mit gaußscher Verteilung

Nun werden die Höhen aller neun Punkte mit ihrer Gewichtung multipliziert und die Summen addiert. Diese Summe ist nun die neue Höhe für den mittleren Punkt. Der Benutzer kontrolliert die Stärke durch die Anzahl an Iterationen der Funktion. Also wie oft sie angewendet wird. Zuerst nahm ich die Wertungen aus Abb. 6.2, dies fand ich jedoch zu stark; der Benutzer hatte nicht genügend Feinkontrolle über die Rauheit des Terrains. So gewichtete ich den Mittelpunkt mehr, damit der Effekt schwächer wird. Das Verfahren lässt sich folgendermassen implementieren:

```

1 | def gaußscheUnschärfe(schritte):
2 |     for n in range(schritte):
3 |         for x in range(groesse):
4 |             for y in range(groesse):
5 |                 setVert(x,y,
6 |                         (vert(x-1,y-1)+2*vert(x-1,y)+vert(x-1,y+1) +
7 |                          vert(x,y-1)+12*getVert(x,y)+2*vert(x,y+1) +
|                           vert(x+1,y-1)+2*vert(x+1,y)+vert(x+1,y+1))/24)

```

Ich habe also die Punkte jeweils mit dem Zähler multipliziert und durch den Nenner geteilt.

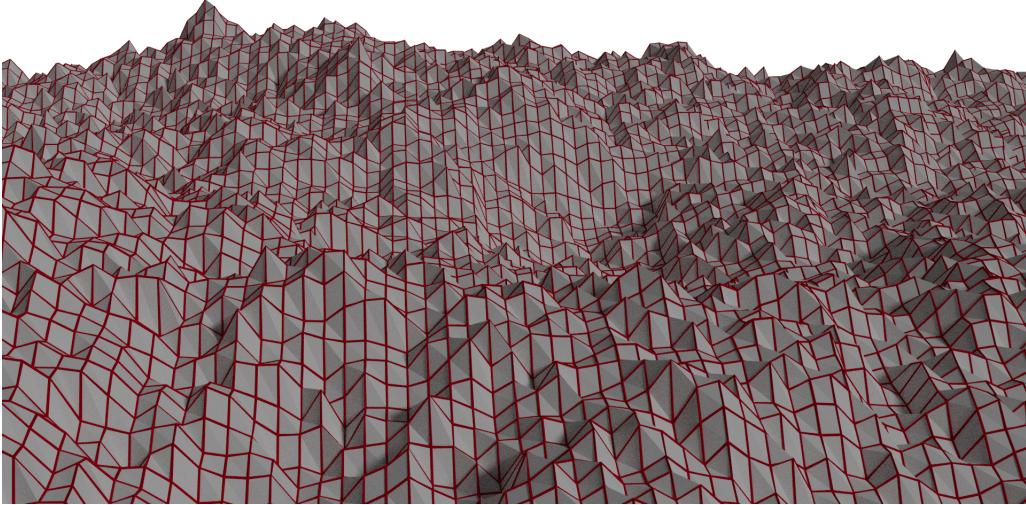


Abbildung 6.3: Vergleich einer Landschaft ohne und mit zweifachem Smoothing

Wie in Abb. 6.3 zu sehen ist, erfüllt diese Funktion ihren Zweck und lässt die Landschaften nicht ganz so zackig aussehen.

6.2 Seen

Das Wasser ist ein sehr wichtiger Aspekt jedes Landschaftsbildes. Ebenso dessen Auswirkung, die Erosion. Blender hat eine integrierte Wasser-Simulation, welche das Lattice-Boltzmann-Verfahren verwendet. Dieses basiert darauf, dass das Wasser in Partikel eingeteilt wird, welche Masse, Geschwindigkeit, eventuell thermodynamische Eigenschaften haben und kollidieren können. Es ist ein sehr gutes und ziemlich effizientes Modell für die Simulation auf relativ kleinen Räumen, wie etwa einer Turbine oder eines Tropfens.

Jedoch ist es ungeeignet, um ein ganzes Fluss- und Seensystem zu simulieren, da es viel zu viel Speicher benötigt. Eine gute Lösung wäre gewesen, eine „shallow

water“-Simulation zu implementieren, ein Ansatz, bei dem nur die Oberflächenveränderung berechnet wird. Allerdings ist das eher eine Maturaarbeit für sich. Deshalb habe ich selbst eine simple Wassersimulation geschrieben. Sie basiert auf der Tatsache, dass Wasser vom Himmel fällt, in Form von Niederschlag, herunterfliesst so lange nichts im Wege ist und ein Teil des Wassers dann verschwindet (in der Realität durch Verdunstung oder Versickerung). Jeder Schritt der Simulation besteht aus drei Unterschritten:

- Dem Regnen: Alle Punkte auf der Wasserkarte werden um einen bestimmten Wert erhöht.
- Dem Fliessen: Hier wird nun für jeden Punkt geprüft, ob die Summe der Wasserkarte plus der Wert der Geländehöhe an diesem Punkt, λ , grösser ist als die selbe Summe ihrer Nachbarn. Wenn dies der Fall ist, wird ermittelt wie viele Nachbaren eine tiefere Summe haben. Daraus wird berechnet, wie viel Wasser jeder benachbarter Punkt erhält, ω .
 - Für jeden der vier Nachbarn wird nun getestet:
 - Ist die Summe aus dem Wasser des Nachbars plus ω Geländehöhe dieses Punktes tiefer als λ .
 - Nun wird nochmals für alle vier Nachbaren getestet:
 - Wenn die Summe aus dem Wasser des Nachbars plus ω plus Geländehöhe dieses Punktes höher ist als λ , wird der Wasserstand des entsprechenden Punktes um $\frac{1}{2}\omega$ erhöht, und das Wasser des Ausgangspunktes um diesen Wert abgesenkt.
- Am Ende jedes Schrittes wird das Wasser auf jedem Feld noch um einen bestimmten Wert erhöht.

Dieser ziemlich einfache Ansatz führt bei wohl gewählten Anfangsbedingungen zu sehr akzeptablen Resultaten, wie in Abb. 6.4 zu sehen ist.

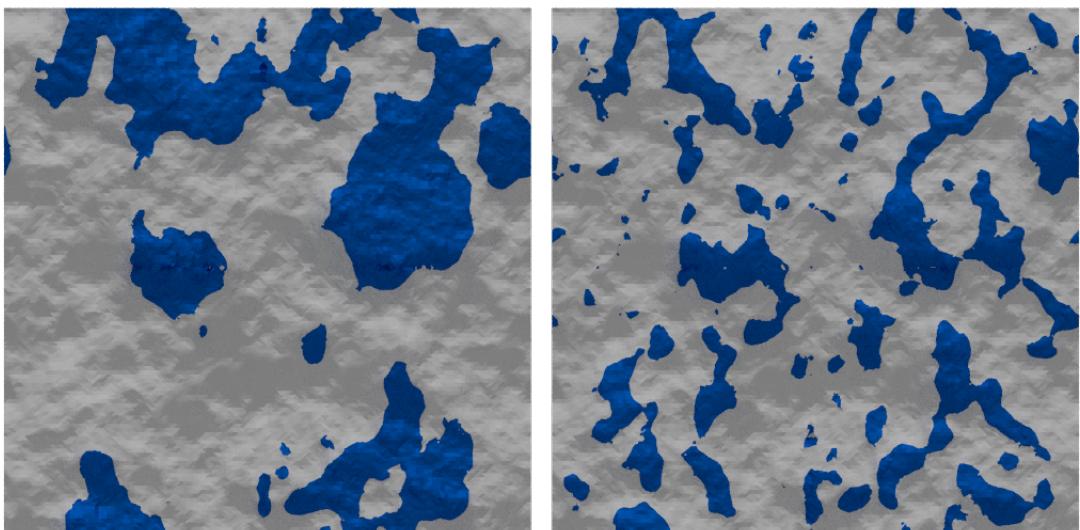


Abbildung 6.4: Wasser Simulation: Links 1000 Zyklen und rechts 10.

Jedoch muss nicht viel an den Parametern verändert werden, um weniger realistische Resultate zu bekommen. In Abb. 6.5 wurde etwa die Wasserzunahme deutlich höher als die Abnahme gewählt.



Abbildung 6.5: Wassersimulation mit unvorsichtig gewählten Parametern

Eine minimale Optimierung, die ich vorgenommen habe ist, dass ich anstatt in jedem Schritt Wasser hinzugebe und wegnehme, dies ganz zu Beginn und ganz am Ende erledige. Ich brachte es jedoch auch bis zum Ende nicht zustande, dass die Seen wirklich wie Seen und nicht wie Gletscher aussahen. Gerade auf Landschaften mit grossen Höhendifferenzen ist dies ein Problem.

6.3 Flüsse

Zu Beginn überlegte ich mir, dass für die Generierung der Flüsse an ein paar zufälligen Punkten mehr Wasser entstehen könnte. Jedoch wären dies sehr breite Flüsse gewesen und die Simulation hätte wieder über einige hundert oder gar tausend Zyklen laufen müssen, um vernünftige Resultate zu bekommen, was in Python zu lange dauerte, für mich. Deshalb werden die Flüsse nach den Seen hinzugefügt. Sie beginnen an einem zufälligen Ort, in einem einstellbaren Höhenbereich und wählen von dort den Weg mit dem leichtesten Widerstand. Sie gehen also jeweils zum tiefsten benachbarten Punkt. Dies so lange, bis sie auf einen Punkt treffen auf dem ein See ist. Alternativ hätte auch eine Karte des gesamten Durchflusses des Wassers auf jedem Punkt erstellt werden können, um dann jeweils zum Punkt mit höherem Gesamtdurchfluss zu gehen, dies war Herrn Rollés Vorschlag. Auch eine höhere Wahrscheinlichkeit für das Erscheinen von Flussquellen am Rande von Seen, welche genug hoch sind, wäre denkbar gewesen.

Ihr Verlauf wird dabei auf einer Karte gespeichert, wobei der Wert eines Punk-

tes auf dieser Jedesmal erhöht wird, wenn der Pfad eines Flusses dort vorbei kommt. Diese wird dann benutzt, um alle Vertices abzusenken, die der Fluss überquert. Mithilfe des „Proportional“-Editings, welches umliegende Punkte in einem bestimmten Radius „mitverschiebt“, entsteht dadurch ein Flussbett. Ebenfalls wird der Flussverlauf sogenannter „Path“ in Blender gespeichert. Dieser erlaubt es dem Benutzer mithilfe „Curve“-Modifiers ein Objekt der Path-Form anzuschmiegen.

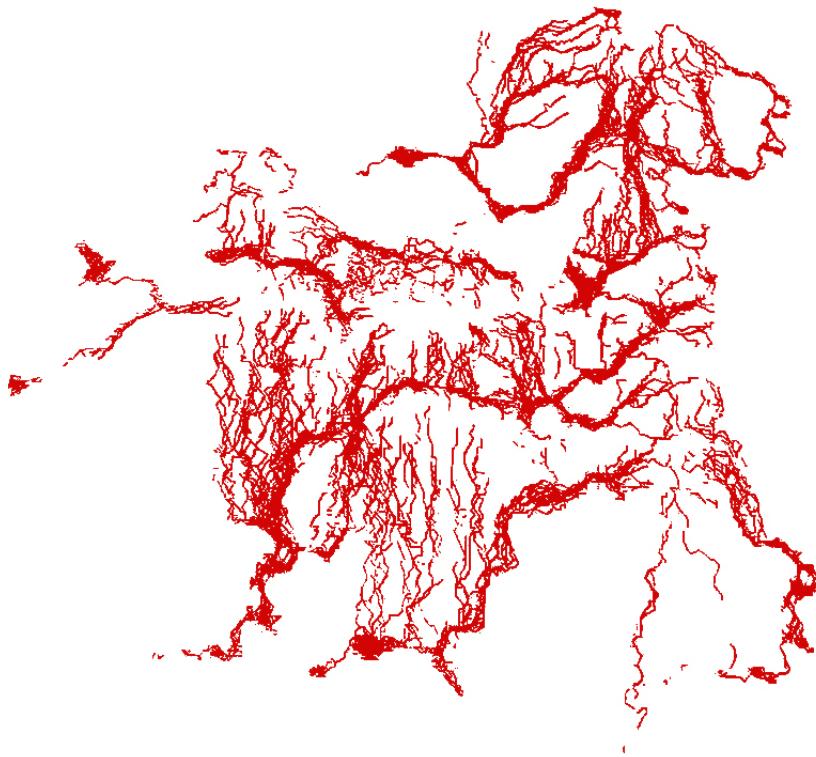


Abbildung 6.6: Früher Prototyp einer Flusssimulation

Zumindest war dies meine Idee. Die Zeit reichte jedoch nicht aus, um die Flüsse vollständig und fehlerfrei zu implementieren. 6.6 ist ein Bild eines grossen Flussystems, bestehend aus etwa 250 Quellen, aus einer relativ frühen Version.

6.4 Erosion

Die Landschaften haben bisher zwar nach Hochgebirge ausgesehen, um eine überzeugende Hügellandschaft zu generieren, werden jedoch steilere Berge und vor allem flachere Täler benötigt. Gletscher hatten in den Eiszeiten Jahrtausende Zeit dies zu erreichen. Ich will jedoch nicht so lange vor dem Computer sitzen und auf Gletscher hoffen. Eine Möglichkeit zur Erosion muss her. Ursprünglich koppelte ich die Erosion mit der Wassersimulation. Das verwendete Prinzip ist folgendes gewesen: Das Wasser hat eine maximale Sättigung erhalten, welche bestimmt wie viel Material pro Höheneinheit Wasser gelöst werden kann. Eine Sedimentkarte dient als Speicher dafür, wie viel Material wo gelöst ist. Jeder Punkt auf der Sedimentkarte hat Material entsprechend der Wasserflussrichtung an seine Nachbaren

abgeben. In jedem Schritt der Wassersimulation wurde an jedem Punkt geprüft ob noch Material gelöst werden kann. Ist dies der Fall, wird das Terrain um einen Wert proportional zur Differenz zwischen Wasserstand und Wasserstand mal Sättigung gesenkt, und die Sedimentkarte an dieser Stelle um diesen Wert erhöht. Ist jedoch der Wert der Sedimentkarte höher als Wasser mal Sättigung, dann wird der Überschuss dem Terrain hinzugerechnet und der Sedimentkarte abgezogen. Das Resultat war jedoch ernüchternd. Es benötigte viele hundert Zyklen der Berechnung, damit überhaupt wirkliche Unterschiede zu sehen waren. Diese waren auch nicht gerade von erhoffter Qualität. Zwar wurden die Täler etwas breiter und die Bergwände steiler, aber es gab auch immer wieder Spitzen in der Landschaft. Also suchte ich eine andere Möglichkeit der Erosion. Diese fand ich in der „thermalen“-Erosion, so wurde diese Variante zumindest in [6] genannt. Die Idee ist, dass der höchste Winkel der Geraden durch einen Punkt und dessen Nachbarn zur Z-Achse, berechnet wird, θ . Ist dieser höher als ein durch den Benutzer bestimmter Winkel, so wird der obere Punkt abgesenkt und der untere erhöht.

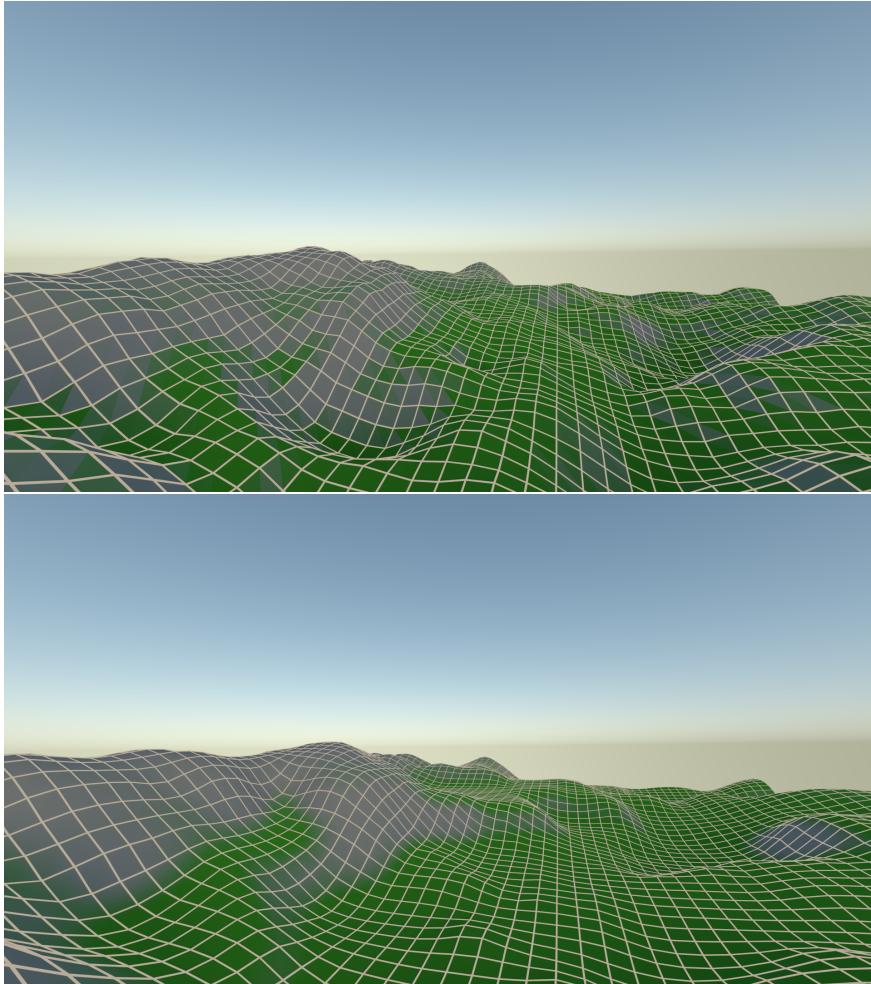


Abbildung 6.7: Oben: Ohne Erosion Unten: Mit invertierter Erosion

In [6] wird ebenfalls noch die „verbesserte“ thermale-Erosion genannt. Bei dieser wird der untere Punkt abgesenkt, wenn der Winkel θ niedriger ist. Dies führt zu einer canyonartigen Abstufung im Gelände. Ich nannte diese Variante jedoch

invertierte Erosion in meinem Programm, da der Unterschied nur ist, dass Material verschoben wird, wenn der Winkel im Terrain niedriger ist. Also das Gegenteil der Normalen.

Durch das nachträgliche Hinzufügen von gaussischer Unschärfe, wird eine seichtere Abstufung erreicht. Es entstehen steilere Flanken und flachere Täler. Also das, was auch bei der an die Wassersimulation gebundene Erosion das Ziel war - nur um den Faktor 100 schneller. Das Resultat ist in Abb. 6.7 zu sehen.

Beide Versionen sind im Add-on implementiert. Das Umschalten funktioniert über ein Häkchen.

6.5 Wälder und Bäume

Es war schon zu Beginn klar, dass ich den Wad als sogenannte Vertex Gruppe ausgeben wollte. Dies ist im Prinzip eine Maske für Vertices. Diese erhalten alle eine Gewichtung zwischen 0 und 1. In Blender wird das farblich dargestellt wenn sich ein Objekt im „Weight Paint“-Modus befindet. 0 ist dabei blau und 1 rot, Werte dazwischen entsprechen dem Farbkreis, im Uhrzeigersinn gekennzeichnet.

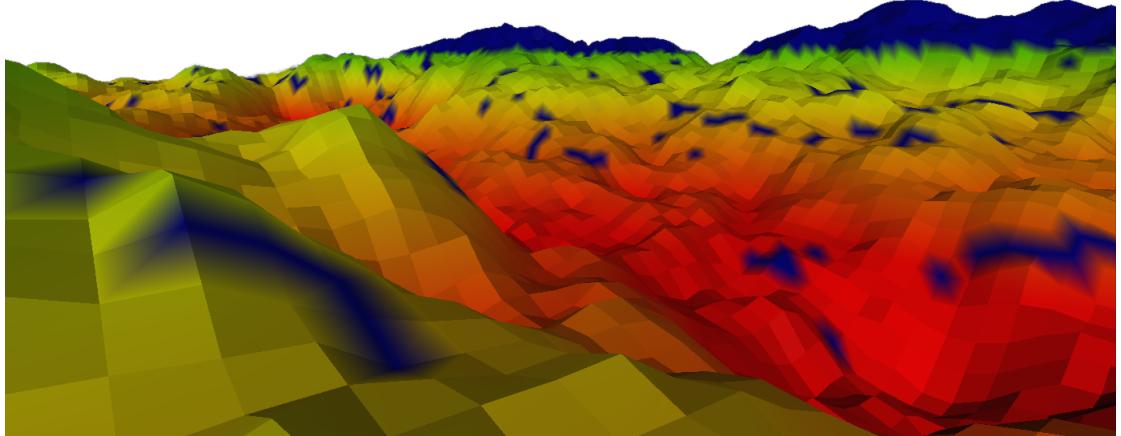


Abbildung 6.8: „Tree Height“-Vertex Gruppe generiert durch mein Add-on

Für die Verteilung der Wälder hat mir Herr Rollé den Tipp gegeben eine veränderte Version des „Game of Life“ zu verwenden. Dies ist ein vom Mathematiker Conway entworfenes System. Es basiert darauf, dass in einem regelmässigen Gitter die Quadrate jeweils mit einem Wert von **lebendig** oder **tot** versehen werden. Nun werden diese Werte durch die Anzahl Nachbaren verändert.

Hat ein lebendiges Quadrat zu viele oder zu wenige Nachbaren, so wird dessen Wert auf **tot** gesetzt. Wenn ein totes Quadrat genau drei Nachbaren hat, wird sie **lebendig**. Bei 2 oder 3 Nachbaren überlebt eine Zelle.

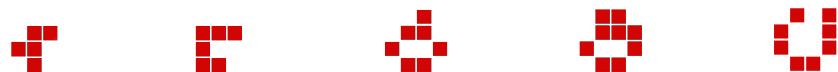


Abbildung 6.9: R-Pentomino Abfolge der ersten fünf Schritte im Game of Life

Die Figur aus Abb. 6.9 ist eine Abfolge die aus der Anwendung dieser Regeln entstand. Erst nach 1103 Schritten würde diese eine oszillierende Struktur bilden, also eine sich periodisch wiederholende Struktur.

Das Game of Life bildet zwar interessante Strukturen, jedoch sieht es nicht wie eine typische Verteilung eines Waldes aus. Die empfohlene Anpassung ist eigentlich eine Vereinfachung: Die Anzahl lebendiger Nachbarn wird angeschaut und der Zustand entsprechend gesetzt, ohne Berücksichtigung des vorherigen Zustandes. Da ich im Rahmen des Informatikunterrichts bei Herrn Rollé bereits das „originale“ Game of Life programmiert hatte, konnte ich die Regeln sehr schnell ausprobieren und nach meinem Gefallen anpassen. Nach etwas Experimentieren entschied ich mich für folgende Regeln:

Wenn das Quadrat weniger als 4 Nachbaren hat, wird es auf `tot` gesetzt, bei genau 4 wird der Zustand jeweils umgeschaltet, und bei mehr wird es auf `lebendig` gesetzt.

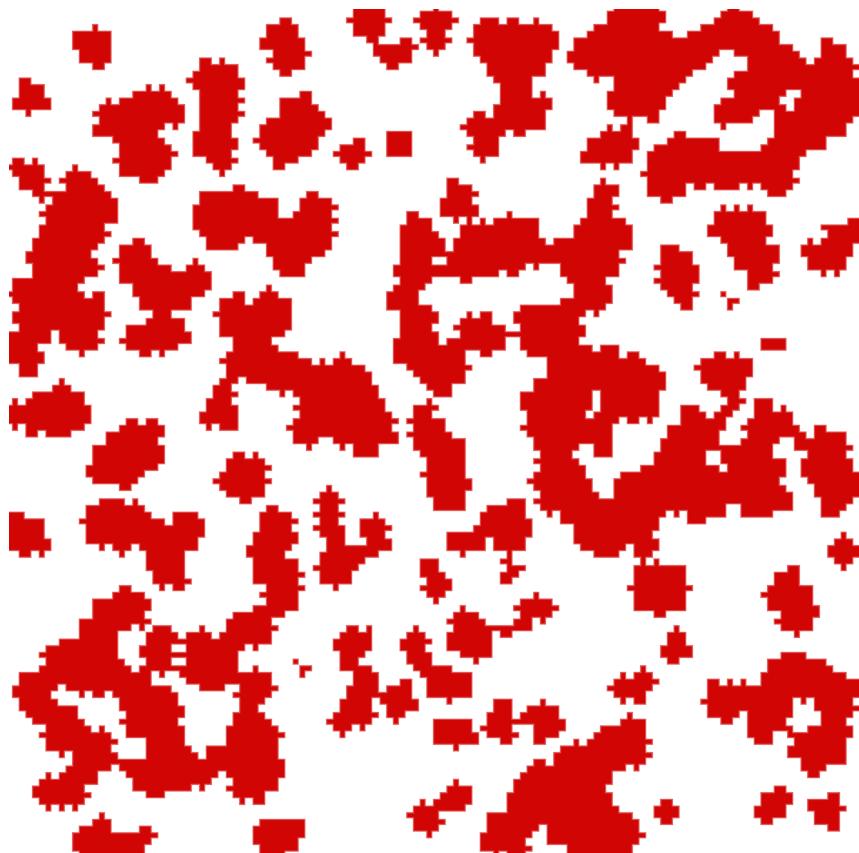


Abbildung 6.10: Bildschirmfoto vom Game of Life mit meinen Regeln

Abb. 6.10 ist ein Bildschirmfoto aus dem Game of Life Programm aus dem Informatikunterricht mit den alternativen Regeln. Zwar gibt es hier und da Formen,

die etwas gar rechteckig sind, Wälder, aber insgesamt ist es akzeptabel.

Das Erstellen der Waldverteilung in meinem Programm lässt das Game of Life mit einstellbarem Prozentsatz der am Anfang lebendigen Quadrate laufen. Dann erhält jeder Punkt, der **lebendig** ist den Wert 1 auf der Baumverteilungs-Vertex-Gruppe, wenn das Terrain an dieser Stelle nicht steil ist, es im Bereich ist wo Bäume wachsen sollen und sich dort weder ein Fluss noch ein See befindet. Die Gewichtung für die Vertex-Gruppe, welche dazu dient, die Abnahme der Grösse von Bäumen zur Waldgrenze hin zu Simulieren habe ich folgendermassen berechnet: $\delta H + s$. Wobei das δH für den Abstand des Punktes zur oberen und und das s die Minimalhöhe der Bäume darstellt.

Für das Nutzen der Weight Paint Texturen als Verteilungs- und Grösseninformation für Bäume muss von Hand ein Partikelsystem angelegt werden. Die Erstellung der Bäume selbst wird dem User überlassen, da es bereits ein sehr umfangreiches Add-on genau zu diesem Zweck in Blender gibt.

6.6 Material

Um das beste Erlebnis mit Blender zu haben, sollte die Render-Engine auf „Cycles“ gewechselt werden - vor allem bei Szenen, in denen die Beleuchtung oder die Oberfläche der Objekte wichtig ist. Denn nur so ist der Zugriff auf den sehr mächtigen Node-Editor gewährleistet. Für diesen generiert mein Add-on eine Textur für Landschaft und Wasser. Dies soll dem Benutzer die Arbeit abnehmen, jedes Mal ein Material zu kreieren. Dazu wurde zuerst ein Solches von Hand erstellt. Danach wurde es Stück für Stück in ein Programm geschrieben. Dazu wird zuerst ein „Node“. definiert. Dies sind Kästchen, welche einige einstellbare Werte haben. Ebenfalls haben sie Ein- und Ausgänge, über die sie verbunden werden. Welchen Typ sie annehmen ist dabei farbig gekennzeichnet. Ein RGB Wert - also ein Farbwert bestehend aus rot, grün und blau, wird gelb repräsentiert. Ein Zahlenwert grau, Vektoren violett. Wenn jedoch ein Farb-Output in einen Zahlen-Input führt, so wird schwarz als 0 und weiss als 1 interpretiert. Generell sind die wichtigsten Nodes die Shader-Nodes und der Material-Output-Node. Erstere sind für die Oberflächenbeschaffenheit und Farbe eines Objektes zuständig. Sie können auch mithilfe des „Mix-Shader-Nodes“ gemischt werden. Der Material-Output-Node hat nur Inputs. Dies liegt daran, dass er das Ende darstellt. Ein wichtiger Node für das generierte Material in meinem Add-on, ist der Geometry Node. Vor allem dessen „Normal“-Output. Dieser gibt einen RGB Wert aus, wobei der rote Kanal für den Winkel der Normale zur X-, der grüne für den Winkel zur Y- und der blaue für den Winkel zur Z-Achse steht. Ebenfalls wichtig ist der „Color-Ramp-Node“. Dieser nimmt einen Wert zwischen 0 und 1 als Input und gibt den entsprechenden Wert auf einem benutzerdefinierten Farbverlauf aus. Wird nun also der blaue Kanal eines Outputs eines Geometry-Nodes, welcher durch einen Color-Ramp-Node verarbeitet wird, verwendet, um zwei Shader-Nodes zu mischen, kann mithilfe der Color-Ramp der Winkel zur Z-Achse bestimmt werden, bis zu welchem welcher Shader angezeigt wird.

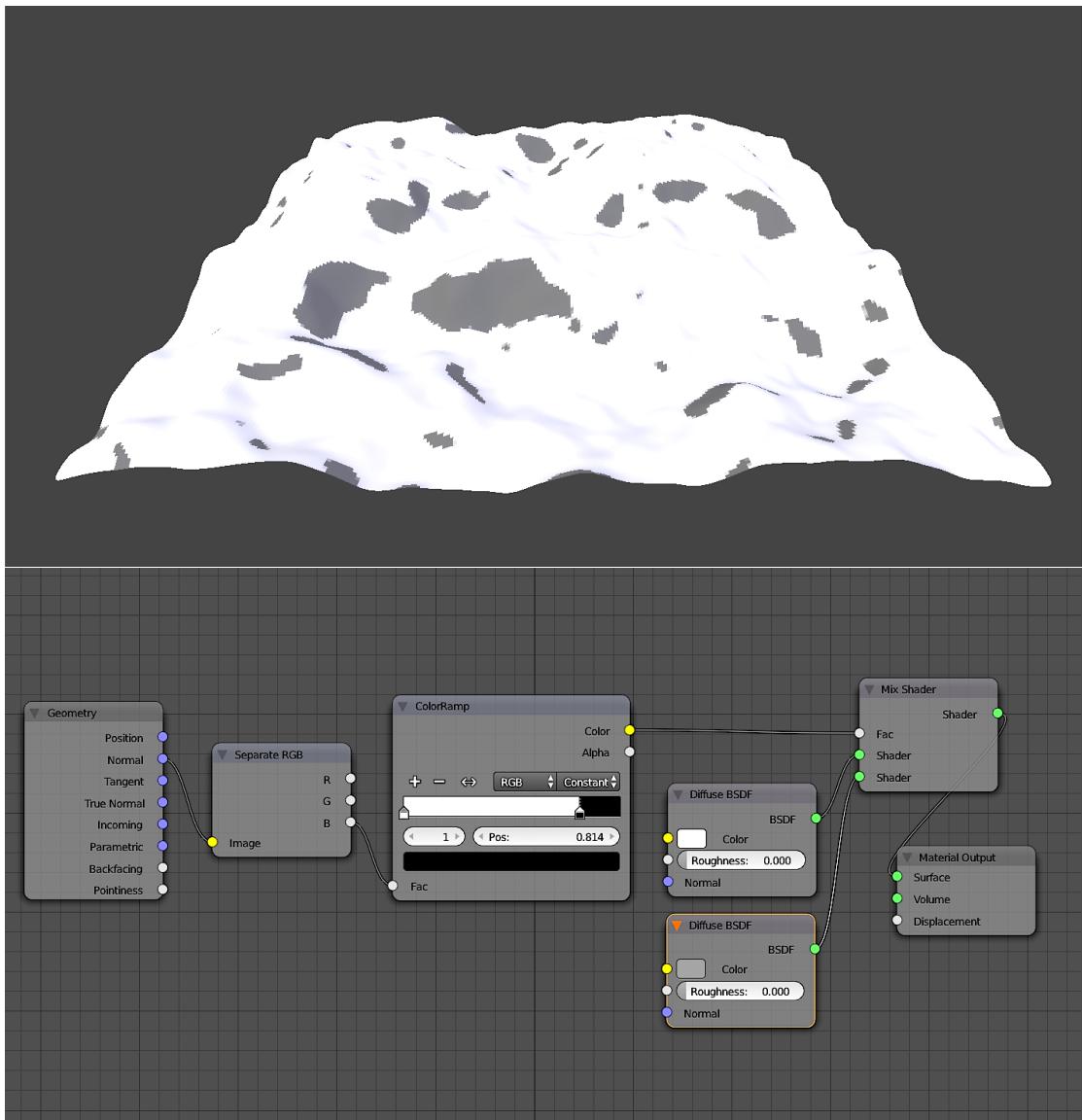


Abbildung 6.11: Bildschirmfoto von Node-Editor und Viewport

Das generierte Material für das Terrain enthält zwei nicht verwendete Nodes. Ein Inverse-Node, welche Farben invertieren, also zur komplementär Farbe machen kann und ein unverbundener Color-Ramp-Node. Wenn der „Pointiness“-Output des Geometry-Nodes mit dem Inverse-Node-Input, diesen mit dem Color-Ramp-Input und dessen Output mit dem „Displacement“-Output verbunden wird, so ist die Möglichkeit vorhanden dem Gelände etwas mehr Details zu geben. Jedoch ist es etwas schwierig genau die richtige Einstellung zu finden, damit es gut aussieht.

6.7 Benutzeroberfläche

Die Benutzeroberfläche ist der wichtigste Aspekt der Benutzer-Erfahrung, denn es ist der primäre Weg, über den der User mit dem Programm interagiert. Etwas Planung in das Interface zu investieren lohnt sich also, denn (fast) niemand würde heute noch ein Betriebssystem oder ein Programm per Befehlszeileneingabe ansteuern wollen. Es bedarf einer so genannten GUI, eines Graphical User Interfaces.

Blender hat ein eigenes, auf OpenGL basiertes GUI-System. Es bietet viele Features die über Python aufrufbar sind. Ich gehe hier nicht allzu tief darauf ein, denn wer sich interessiert findet viele Code-Beispiele und Tutorials zu diesem Thema auf dem Internet. Hier werden eher einige Gedanken erklärt, die ich mir gemacht hatte.

Zuerst etwas zur Aufteilung der vielen Werte. Diese wurden in die drei Kategorien Terrain & Erosion, Wasser und Wald unterteilt. Die Modi, welche über ein Drop-Down-Menu ausgewählt werden, zeigen jeweils andere Werte zum Einstellen. Somit wird der Benutzer nicht zu sehr überschwemmt mit Zahlenfeldern. Ebenfalls hat er überall die Möglichkeit auf die Häckchen zu klicken, um zu definieren, was geupdated wird. Somit kann die Berechnung von Erosion, Wasser und Wald einzeln an- und abgestellt werden und es muss nicht mehr jedes Mal alles berechnet werden, wenn ein Wert verändert wird.

Da meine Implementation des Diamond-Square-Algorithmus nur Felder der Grösse $2^n + 1$ erlaubt, gibt der Benutzer n an. Die entstehende Anzahl Vertices pro Seite der Landschaft und insgesamt wird rechts neben dem entsprechenden Eingabefeld angezeigt.

Kapitel 7

Fazit

Im Folgenden vergleiche ich ein Bild auf Basis einer generierten - mit meinen Seen - und eines auf Basis einer realen Höhenkarte - mit einer Ebene als See. Die Höhenkarte wurde von [8] heruntergeladen. In beiden Bildern wurde das selbe Material und die gleich positionierte Belichtung verwendet.

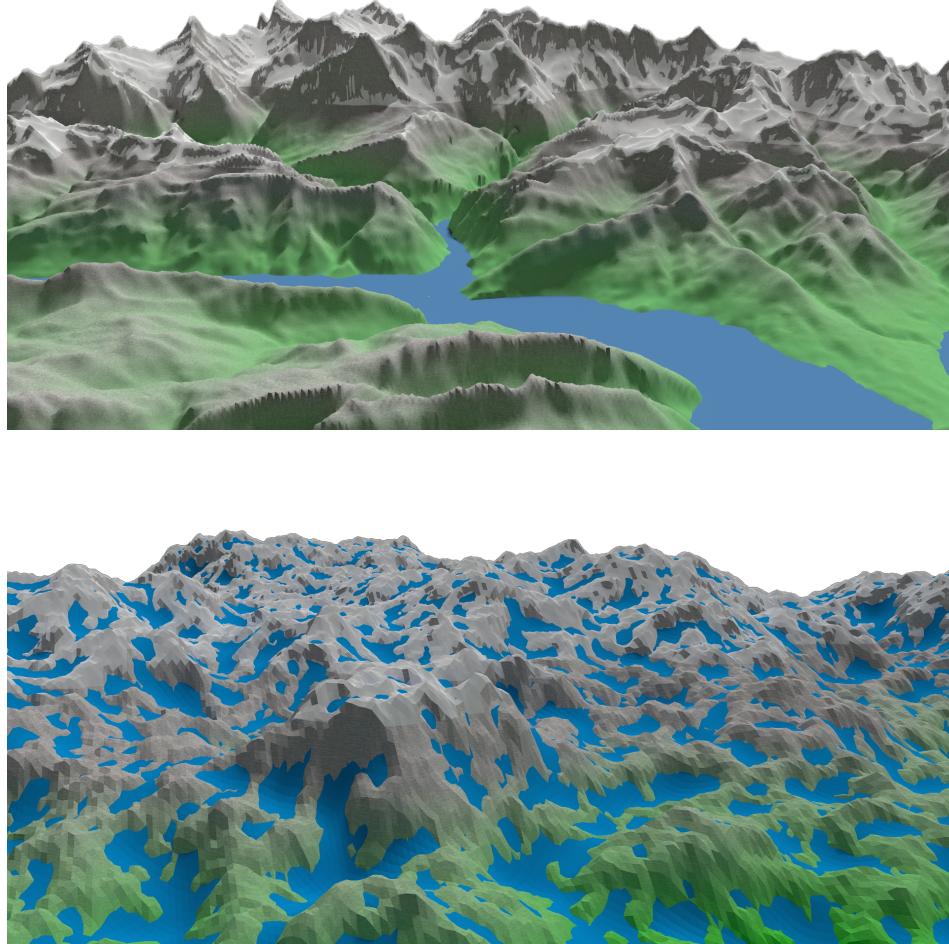


Abbildung 7.1: Oben: Landschaft auf Basis von realer Höhenkarte Unten: Landschaft aus meinem Add-on

Wie zu sehen ist sind die Seen in meiner Landschaft nicht gerade gelungen. Sie sind zu klein und sind oft sehr steil. Das Terrain selbst ist jedoch realistisch.

Im Grunde bin ich einigermassen zufrieden mit der Arbeit.

Ein immer wiederkehrendes Problem bei der Programmierung war jedoch, dass ich für einige Dinge deutlich länger hatte als gedacht. Einer der Hauptgründe dafür, war das Suchen nach API-Befehlen. Etwa der Befehl zum Setzen des Gewichtes eines Vertex erscheint zwar in der Dokumentation, jedoch nicht was für Variablen er entgegen nimmt.

Wenn ich noch mehr Zeit zur Verfügung gehabt hätte, würde ich zuerst die Flüsse implementieren. Danach gäbe es noch einige Verbesserungen an den Seen zu machen. Etwa jeder Punkt eines Sees auf den tiefsten Punkt seines Randes setzen. Dadurch wären die Seen flach und erinnerten nicht mehr so sehr an Gletscher. Auch die Möglichkeit, den Rand eines Flusses oder Sees als Vertex Gruppe zu speichern wäre nett. Interessant wäre die Höheninformationen auf Kugeln zu projizieren, wie dies das „Ant-Landscape“-Output auch kann, um Planeten zu generieren. Der Segen und der Fluch dieses Themas ist, dass alles immer mehr und noch besser gemacht werden könnte - wäre bloss die Zeit dazu.

Wer übrigens das Add-on oder diesen Text als PDF downloaden und benutzen will, kann dies unter https://github.com/RomanRiesen/Blender_Landscape_AddOn tun. Dort werde ich auch noch allfällige weitere Versionen - als solche markiert - speichern.

Kapitel 8

Bilder

In diesem Kapitel befinden sich einige Bilder, welche im Rahmen meiner Maturaarbeit entstanden sind.



Abbildung 8.1: Bild aus einem Turtle Programm, welches als Übung diente

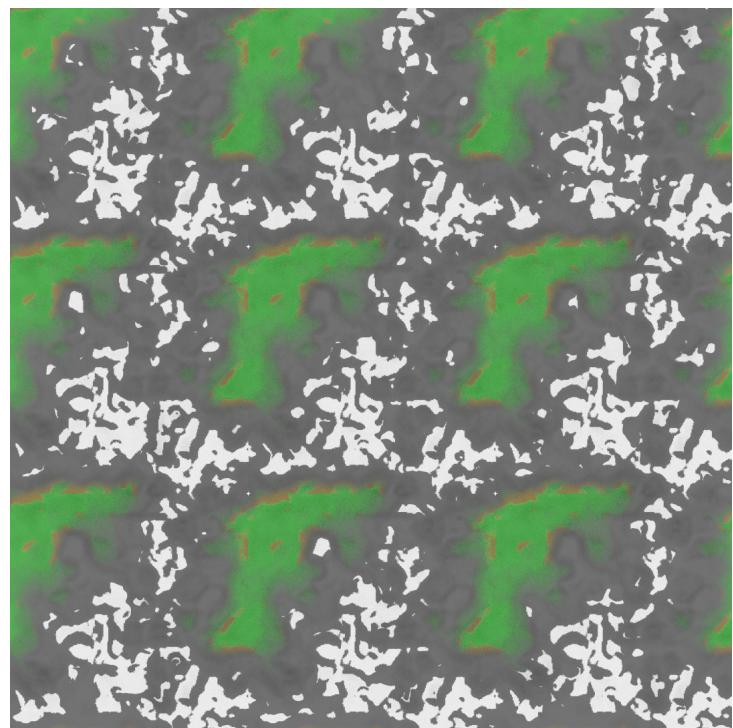


Abbildung 8.2: Eine Landschaft mit 2 Array-Modifiern

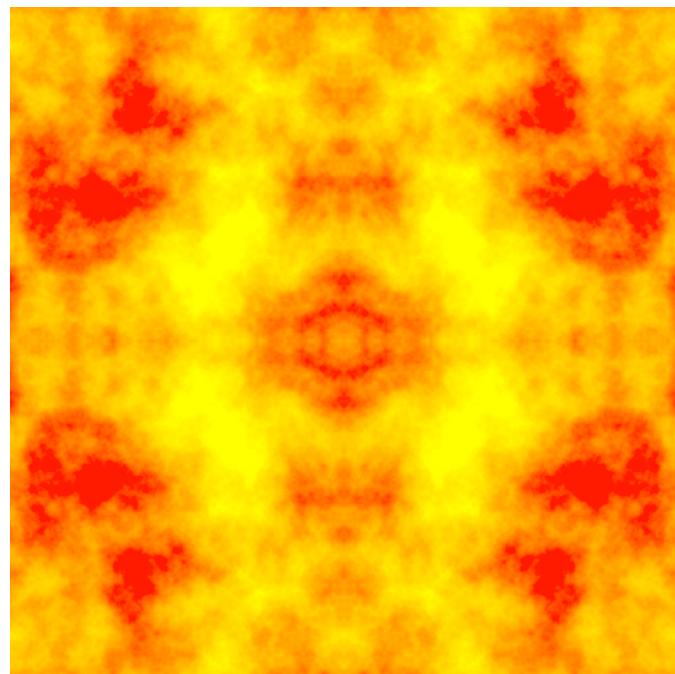


Abbildung 8.3: Auch mit 2 Array-Modifiern, jedoch interessanter eingefärbt

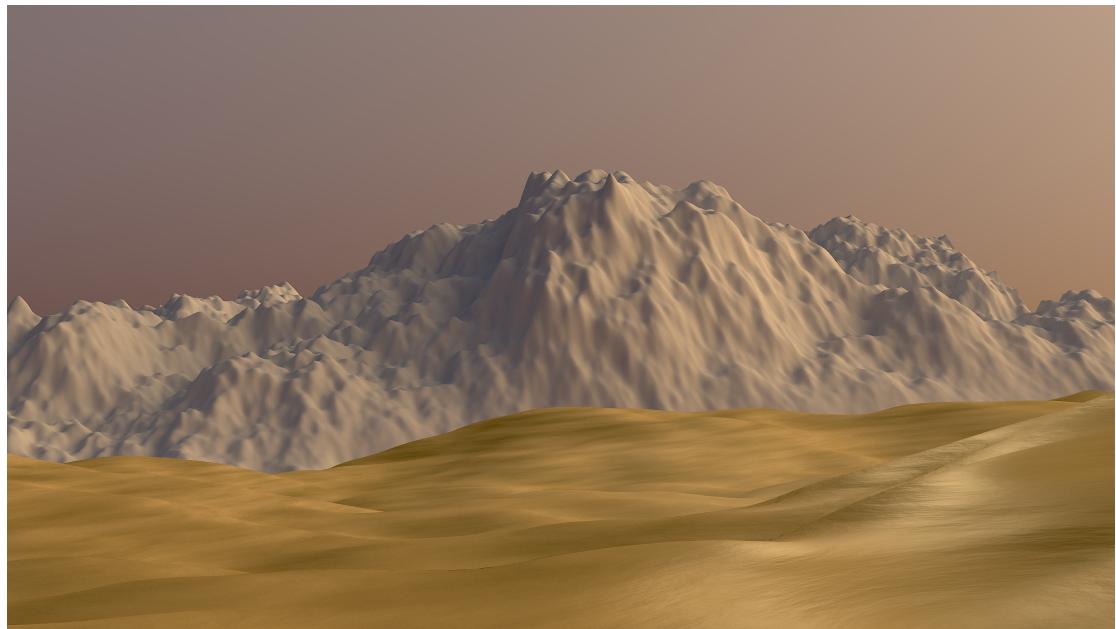


Abbildung 8.4: Wüste mit Bergen im Hintergrund



Abbildung 8.5: Die Waldverteilung wurde durch das Add-on berechnet

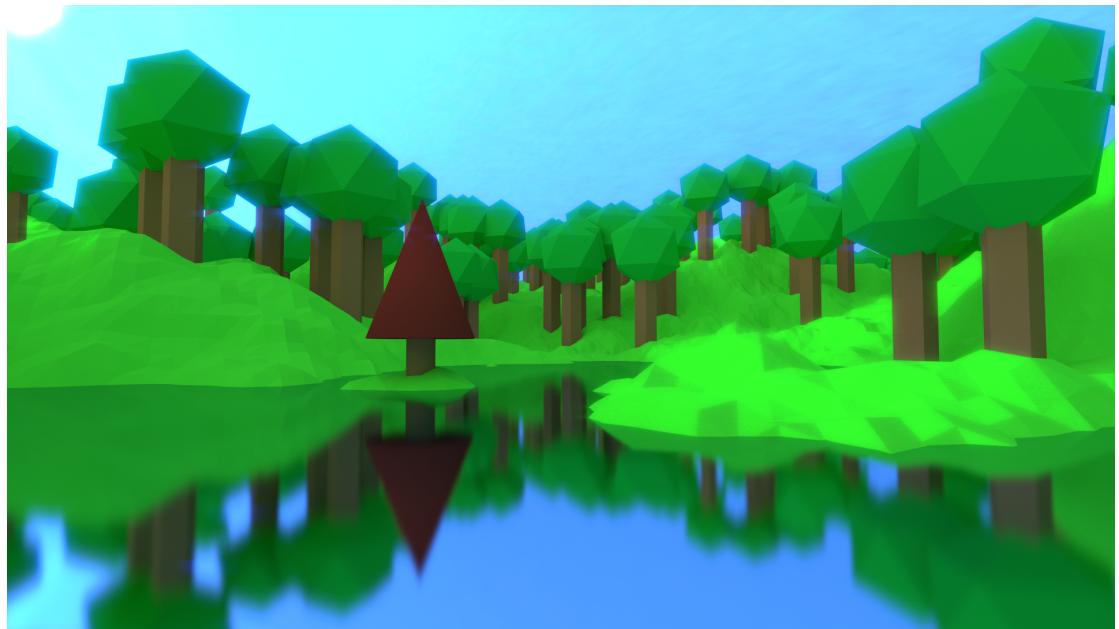


Abbildung 8.6: Etwas polygonärmerer Ansatz

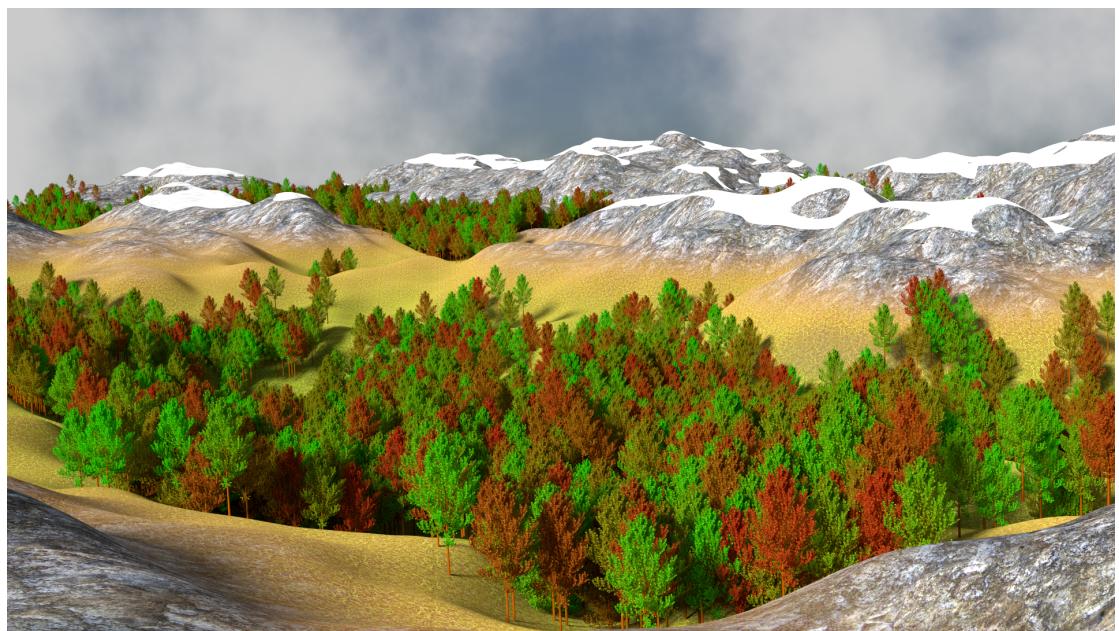


Abbildung 8.7: Herbstlicher Wald in kahler Berglandschaft

Kapitel 9

Danksagung

Ich möchte mich zuerst bei Herrn Rollé bedanken für die Unterstützung und die Tipps, die er mir gegeben hat. Natürlich auch bei allen, die mich sonstig unterstützt haben. Ob mit Ideen oder Rückmeldungen. Speziell meiner Mutter für die orthografische Aufmerksamkeit. Zum Schluss noch bei den Personen der Blender Foundation, dass sie so eine mächtige Software fleissig erweitern und verbessern.

Literaturverzeichnis

- [1] Voxellandschaft:
[http://www.blitzbasic.com/
Community/posts.php?topic=93982](http://www.blitzbasic.com/Community/posts.php?topic=93982)
3.10.2015
- [2] Perlin Noise:
[https://de.wikipedia.org/
wiki/Perlin-Noise](https://de.wikipedia.org/wiki/Perlin-Noise)
2.9.2015
- [3] Diamond Square:
[http://www.paulboxley.
com/blog/2011/03/
terrain-generation-mark-one](http://www.paulboxley.com/blog/2011/03/terrain-generation-mark-one)
4.5.2015
- [4] Diamond Square:
[http://www.bluh.org/
code-the-diamond-square-algorithm/](http://www.bluh.org/code-the-diamond-square-algorithm/)
8.10.2015
- [5] Lattice-Boltzmann-Methode:
[https://de.
wikipedia.org/wiki/
Lattice-Boltzmann-Methode](https://de.wikipedia.org/wiki/Lattice-Boltzmann-Methode)
8.9.2015
- [6] Erosion:
[http://web.mit.edu/cesium/
Public/terrain.pdf](http://web.mit.edu/cesium/Public/terrain.pdf)
30.5.2015
- [7] Game of Life:
[https://de.wikipedia.org/
wiki/Conways_Spiel_des_
Lebens](https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens)
5.9.2015
- [8] Höhenkarten der ganzen Welt:
<http://terrain.party/>
8.10.2015