Entwicklung eines prozeduralen Landschaftsgenerators

Marcus Gagelmann
Hochschule Anhalt & Martin-Luther-Universität Halle-Wittenberg
marcus.gagelmann@student.uni-halle.de

ABSTRACT

Diese Projektarbeit beschäftigt sich mit der Entwicklung einer Software, die zur Generierung landschaftähnlicher Höhenkarten in Videospielen fähig ist. Ziel ist es hierbei, dem Nutzer eine breite Auswahl an Anpassungsmöglichkeiten zu bieten, um das erstellen vielfältiger Landschaften zu ermöglichen. Umgesetzt wird dieser Ansatz innerhalb der Unity Game Engine unter Verwendung von Perlin Noise.

KEYWORDS

Interaktive Medien Projekt 1, Sommer 2022, Unity, Perlin Noise, Prozedurale Landschaftsgenerierung

```
TerrainGenerator() {
    calculateVertexHeights() {
        calculateNoiseDensity();
        calculateOffset();
        calculateNoiseLayers();
        calculateTerrainHeight();
    }
    vertices = calculateVertexHeights();
    triangles = calculateTriangles();
    colors = calculateColors();
    Mesh = calculateMesh(vertices, triangles, colors);
}
```

Abbildung 1:
Überblick über das Konzept in Pseudocode

1 EINLEITUNG

Glaubwürdige weitläufige Landschaften finden in Videospielen schon seit den 70er Jahren mit der Entwicklung der ersten Open-World-Games, der Textadventures, Anwendung [4]. Was damals noch textuell für den Spieler beschrieben wurde, muss in modernen Spielen grafisch dargestellt werden, um mit den Standards der Spieleindustrie mitzuhalten. Sowohl kleine als auch große Spieleentwickler greifen in den letzten Jahren zur Umsetzung dieses Ziels immer häufiger auf das Mittel der prozeduralen Generierung zurück. Hiermit ist es möglich, automatisch digitale Inhalte mit gigantischen Ausmaßen in kurzer Zeit zu erstellen, was die Qualität der produzierten Spiele erhöhen und Unternehmensressourcen einsparen kann. Ein Beispiel hierfür ist das 2016 erschienene Spiel No Man's Sky, in welchem der Spieler mehr als 18 Trillionen Planeten frei erkunden kann [3].

Prozedurale Generierung bezeichnet die automatische Erstellung von virtuellen Welten, Texturen, Effekten und anderen Inhalten mithilfe deterministischer Algorithmen und vordefinierter Parameter [1]. Für die Umsetzung eines prozeduralen Generators werden in der Praxis häufig Rauschfunktionen verwendet. Diese Funktionen bilden eine unregelmäßig wellenförmige sowie pseudozufällige Grundlage für die angestrebte Synthese. Im Rahmen dieses Projekts wird eine Software innerhalb der Unity Game Engine entwickelt, welche die prozedurale Generierung realistischer 3D-Höhenkarten als Grundlage virtueller Landschaften mithilfe des sogenannten Perlin Noise [2] realisiert.

2 THEORETISCHE GRUNDLAGEN

Perlin Noise wurde aufgrund seiner vorteilhaften Zufallseigenschaften als Grundlage für die Terrainsynthese [2] im Rahmen dieser Arbeit ausgewählt. Das Design klassischer Zufallsgeneratoren ist darauf ausgerichtet, vollständig zufällige Zahlensequenzen zu erzeugen, welche keine Voraussagen über die vorangegangenen und folgenden Zahlenwerte zulassen. Natürliche Strukturen lassen sich jedoch nicht gut durch solche Werte beschreiben. Für diesen Fall ist die Verwendung von Perlin Noise angebrachter, welches ein Gradient-Rausch-Verfahren darstellt. Ein solcher Algorithmus definiert an festgelegten Gitterpunkten gleichen Abstands jeweils pseudozufällige Gradientwerte. Die gesuchte Rauschfunktion folgt diesen Gradientwerten in der Nähe der zugehörigen Gitterpunkte. Zwischen den Gitterpunkten wird interpoliert, um die Funktion zu glätten [2].

Hiermit generierte Werte liegen immer in einer ähnlichen Größenordnung, solange sie eine räumliche Nähe zueinander besitzen und folgen einem An- oder Abstieg zu einem lokalen Minimum oder Maximum. Die dadurch erzeugte wellenförmige Rauschfunktion kann für beliebig viele Dimensionen implementiert werden. Im Rahmen unseres Projekts reicht jedoch die von der Unity Game Engine vorgegebene Perlin Noise Funktion [5] in zwei Dimensionen aus. Dass Perlin Noise eine gute Grundlage für die Abstrahierung natürlicher Landschaften darstellt, wird durch die praktische Anwendung des Algorithmus in digitalen Spielen wie No Man's Sky [3] deutlich.

3 KONZEPT

Das Ziel des Projektes ist es, einem Nutzer die Erstellung beinahe beliebiger 3D-Höhenkarten durch prozedurale Generierung zu vereinfachen. Hierfür ist es nötig, mithilfe eines Noise-Algorithmus und unterschiedlicher Parameter Höhenwerte zu Koordinaten im zweidimensionalen Raum zu berechnen. Diese zweidimensionalen Koordinaten und ihre zugehörigen Höhenwerte können im Anschluss mithilfe des in Unity integrierten Mesh Renderers visualisiert werden. Zusätzlich soll die Visualisierung einer Schnee- sowie einer Wassergrenze innerhalb der Höhenkarte ermöglicht werden, um die Glaubwürdigkeit einer solchen Höhenkarte als Grundlage einer Landschaft zu unterstreichen. Zu diesem Zweck darf auch eine farblich anpassbare Gestaltung der dargestellten Objekte nicht ausbleiben. Für eine bessere User Experience durch einen responsiveren Workflow soll die Höhenkarte außerdem in Echtzeit an veränderte Parameter angepasst werden.

Der Pseudocode in Abbildung 1 zeigt die grundsätzliche Reihenfolge der Generatorberechnungen. Die einzelnen benötigten Funktionen werden hierbei stark abstrahiert dargestellt und dienen lediglich dazu, einen Überblick über das Gesamtkonzept zu vermitteln. Genauere Ausführungen zum Code sind im Abschnitt 4 zu finden.

Im Folgenden werden die benötigten, durch den Nutzer einstellbaren Parameter aufgeführt, die direkten Einfluss auf die generierte Höhenkarte haben.

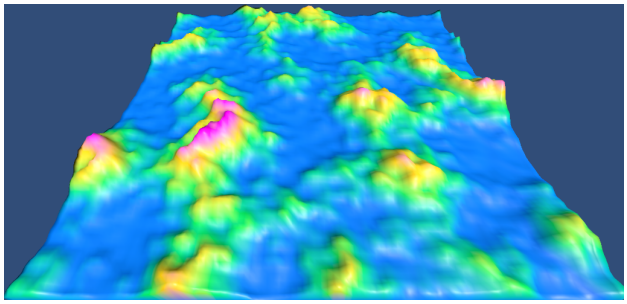

Abbildung 2:
Standard Höhenkarte

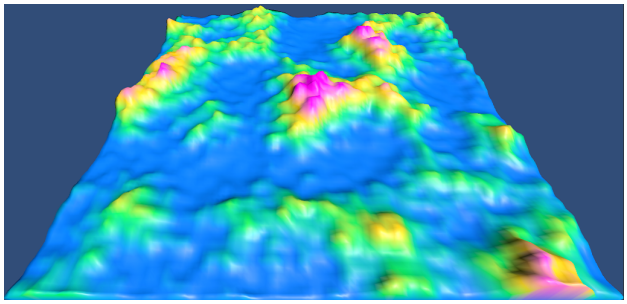

Abbildung 3:
höherer Offset

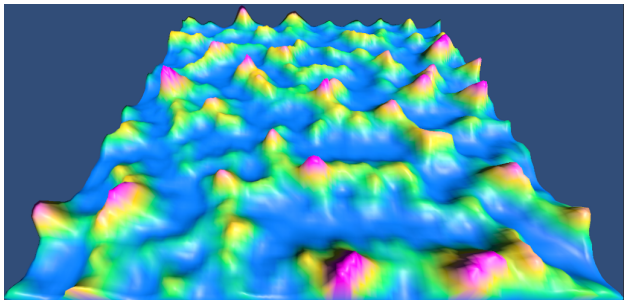

Abbildung 4:
höhere Height Density

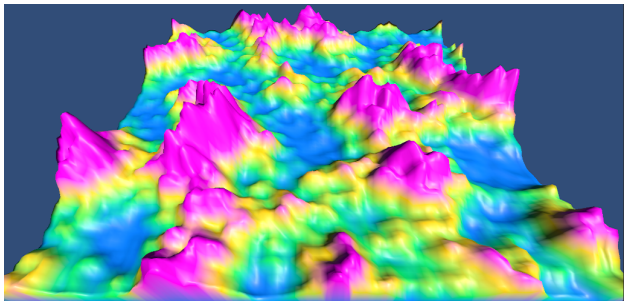

Abbildung 5:
höhere Height Variety

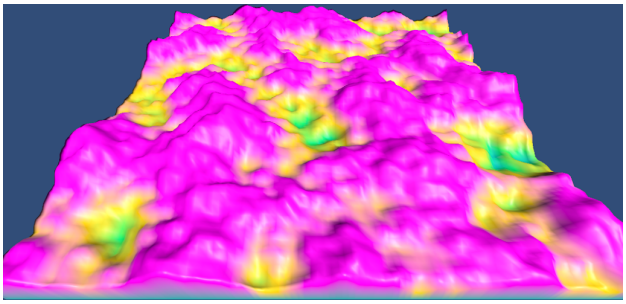

Abbildung 6:
niedrige Noise Power

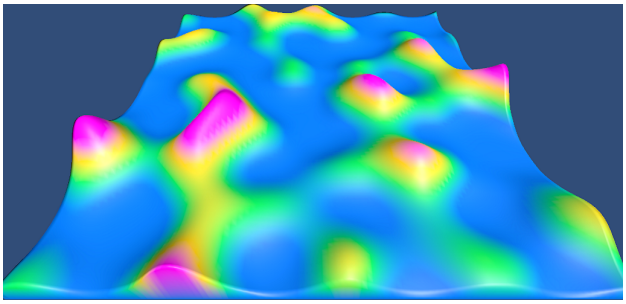

Abbildung 7:
keine Noise Layer

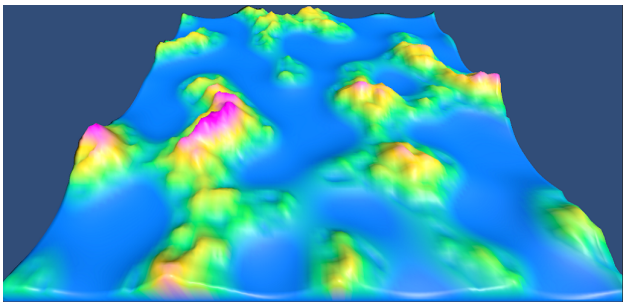

Abbildung 8:
keine Noise Layer in tieferen Regionen

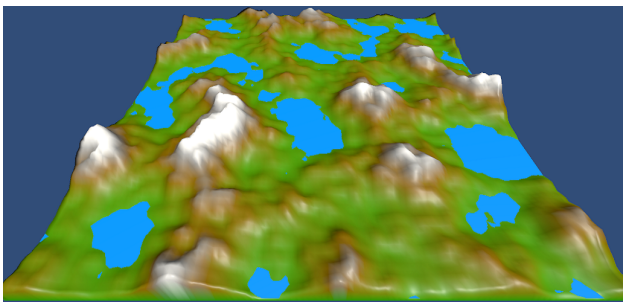

Abbildung 9:
Anpassung von Wasser und Terrain Gradient

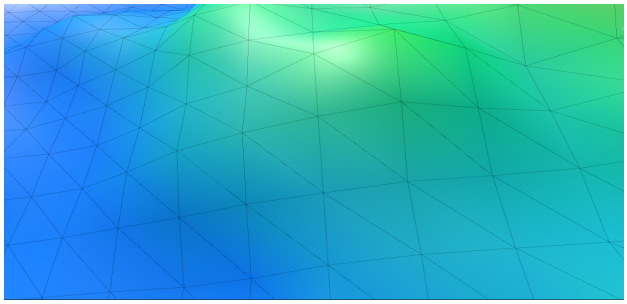

Abbildung 10:
Aufbau eines Meshs am Beispiel einer Höhenkarte mit Wireframe

3.1 Basisparameter

Die Grundlage der generierten Höhenkarte bilden folgende vier Basisparameter:

- Terrain Length
- Terrain Width
- x-Coordinate-Offset
- z-Coordinate-Offset

Das Mesh, das durch den Generator erstellt wird, besteht aus vielen miteinander verbundenen Quadraten, welche jeweils aus genau zwei verbundenen rechtwinkligen Dreiecken bestehen. Diese Dreiecke bestehen wiederum jeweils aus drei der kleinsten Teile des Meshs, den Knoten. In Abbildung 10 ist der Aufbau eines Meshs anhand eines vergrößerten Teils einer Höhenkarte mit aktiviertem Wireframe im Editor der Unity Game Engine erkennbar. Die Parameter *Terrain Length* und *Terrain Width* geben die Ausmaße der zu generierenden Höhenkarte im zweidimensionalen Raum in der Anzahl der Quadrate innerhalb des Meshs in jeder Dimension an. Diese Maße werden zu einem späteren Zeitpunkt bei der Generierung des Höhenkarten-Mesh verwendet. Die Offsets der jeweiligen Koordinaten geben an, welche Teilmenge des Wertebereichs der verwendeten zweidimensionalen Rauschfunktion vom Generator für die Berechnung der Höhenwerte ausgelesen wird. Durch die Randomisierung dieser beiden Parameter würden bei jedem Start des Generators Höhenkarten erstellt werden, die einem sehr ähnlichen Aufbau folgen. Die Berge und Täler solcher Offset-Höhenkarten können jedoch an gänzlich unterschiedlichen Stellen vermutet werden, wie im Vergleich zwischen Abbildung 2 und Abbildung 3 ersichtlich wird.

3.2 Höhenparameter

Die Höhenparameter legen fest, welche Höhen sich für die Mesh-Knoten aus den ausgelesenen Werten der Rauschfunktion ergeben. Sie sind maßgeblich für den Aufbau der Höhenkarte und sollten mit Bedacht gewählt werden. Die Umsetzung der folgenden Parameter ist notwendig, um dem Nutzer die Kontrolle über die Dichte, die Höhe und das Gefälle von Bergen innerhalb der Landschaft zu gewähren.

- Height Density
- Height Variety
- Noise Power

Der *Height Density* Parameter hat direkten Einfluss auf die Frequenz der Maximal- und Minimalwerte innerhalb der Höhenkarte. Erhöht man diesen Wert, so nimmt auch die Dichte der Berge und Täler zu, wie im Vergleich von Abbildung 2 und Abbildung 4 zu erkennen ist. Die Parameter *Height Variety* und *Noise Power* beeinflussen hingegen die Höhenwerte der einzelnen Mesh-Knoten. Die beiden Variablen unterscheiden sich in der Art ihrer Verrechnung zu den entgültigen Höhenwerten. Im Vergleich zur *Height Variety*, die lediglich mit den Noise-Werten multipliziert wird, werden die Noise-Werte mit dem *Noise Power* Parameter potenziert. Dieser asymptotische Unterschied in der Verrechnung führt in der Praxis zu erkennbar unterschiedlichen Ergebnissen, wie im Vergleich von Abbildung 2, Abbildung 5 und Abbildung 6 auffällt.

Eine erhöhte *Height Variety* bewirkt einen Anstieg jedes Punktes im Terrain. Höher gelegene Stellen der Karte verzeichnen hierbei ein höheres Wachstum als niedrige Punkte. Der Parameter beeinflusst also die Ausprägung der Höhenunterschiede im Terrain. Das Erhöhen der *Noise Power* hingegen führt zum Absenken eines jeden Punktes der Karte. Hohe Stellen senken sich dabei weniger stark ab als niedrige, was die Entstehung von spitzer zulaufenden Bergen und flacheren Tälern bewirkt. Durch das verringern der Noise Power werden im Umkehrschluss bestehende Täler von Hügeln geprägt und bestehende Berge abgeflacht. Das Gesamtterrain verzeichnet einen Anstieg. Dieser Parameter wirkt sich somit primär darauf aus, wie Steil die Abhänge der Berge im Terrain sind.

3.3 Noise-Layer Parameter

Durch die Verwendung einer einzigen einfachen Rauschfunktion können Meshes generiert werden, welche im Ganzen betrachtet bereits einem realen Landschaftsbild folgen. An diesem Punkt ergibt sich jedoch das Problem, dass die Oberflächen von einzelnen Bergen und Tälern im Detail sehr abgerundet erscheinen. Um dieses Problem zu lösen, werden mithilfe der Rauschfunktion in höherer Frequenz weitere Schichten (s. g. Noise Layers) auf das bestehende Terrain projiziert. Hierdurch ergibt sich auf den größeren Strukturen der Höhenkarte eine regelmäßige raue Oberfläche, die im Vergleich von Abbildung 2 und Abbildung 7 ersichtlich wird. Zu den Parametern, die im Generatorskript für die Kontrolle dieser Noise Layers verantwortlich sind, zählen:

- Lower Noise Layer Height Variety
- Lower Noise Layer Density
- Upper Noise Layer Height Variety
- Upper Noise Layer Density
- Upper Height Percent

Die *Noise Layer Height Variety* und *Noise Layer Density* funktionieren ähnlich, wie die gleichnamigen Höhenparameter aus Abschnitt 3.2. Das Erhöhen der Height Variety bewirkt extremere Höhenunterschiede und eine Erhöhung der Density vergrößert die Frequenz der Noise Layers, welche auf die Höhenkarte projiziert werden.

Eine Mechanik, die sich als nützlich bei der Generierung von Landschaftsähnlichen Höhenkarten erweisen könnte, ist die Unterteilung der Karte in höhere und niedrigere Regionen mithilfe des *Upper Height Percent* Parameters. Dieser gibt an, wie viel Prozent der Karte, von oben beginnend, entlang der y-Achse (Höhenachse), zu den höher gelegenen Regionen gezählt werden. Mithilfe dieser Funktion wird es dem Nutzer ermöglicht, die Beschaffenheit der Oberflächen in verschiedenen Höhenlagen unterschiedlich ausfallen zu lassen. Ein Beispiel hierfür ist in Abbildung 8 erkennbar. Hier wurden die Noise Layer in den unteren 50% der Höhenkarte komplett entfernt. Eine Ausweitung dieser

Mechanik auf mehr als zwei verschiedene Höhenlagen wäre bei Bedarf denkbar.

3.4 Ästhetische Parameter

Abschließend ist noch die Umsetzung einiger Parameter nötig, die dem Nutzer Möglichkeiten zur optischen Aufwertung der Höhenkarte bieten. Diese Parameter sollten mindestens dazu in der Lage sein, Wasser- und Schneegrenzen anzupassen sowie das Terrain unterschiedlich einzufärben, um einen ersten Schritt in Richtung glaubwürdiger Landschaft zu setzen. Für diese Aufgabe werden folgende Parameter benötigt:

- Relative Water Height
- Terrain Gradient

Relative Water Height gibt an, bei wie viel Prozent der maximalen Terrainhöhe innerhalb der Karte der globale Wasserspiegel liegt. Mithilfe des *Terrain Gradient* Parameters ist es möglich, die Karte abhängig von den jeweiligen lokalen Höhenlagen einzufärben. Im Vergleich von Abbildung 2 und 9 ist ein Beispiel dafür zu sehen, was eine Anpassung dieser Parameter bewirken kann.

4 IMPLEMENTIERUNG

Im folgenden Abschnitt werden die Implementierungsdetails des Ansatzes näher besprochen. Zuerst wird hierbei der Code zur Mesh-Generierung erläutert. Im Anschluss wird dann die praktische Umsetzung sämtlicher Parameter zur Generierung von Höhenkarten besprochen.

4.1 Mesh-Generierung

Zur Generierung von 3D-Meshes in Unity muss eine neue Mesh-Komponente erzeugt werden mithilfe eines Vertex-Arrays und eines Triangles-Arrays. Diese beiden Arrays beinhalten die wichtigsten Daten, aus denen ein solches Mesh berechnet werden kann.

Das Vertex-Array speichert Einträge vom Typ Vector3, welche die Positionen sämtlicher, im Mesh verwendeter Knoten im dreidimensionalen Raum angeben. Diese Vektoren bestehen immer aus den, im zweidimensionalen Raum in gleichmäßigen Abständen gitterförmig verteilten x- und z-Koordinaten sowie der Höhenlage des jeweiligen Punktes als y-Koordinate. An dieser Stelle findet die Berechnung der Höhenlage jedes Knotens statt, die in den Abschnitten 4.3 und 4.4 genauer erläutert wird.

Mithilfe der Daten des Triangles-Arrays wird wiederum bestimmt, wie die einzelnen Knoten zu einer zusammenhängenden Oberfläche verknüpft werden. Zu diesem Zweck speichert dieses Array Knoten-IDs (Position im Vertex-Array) als Int-Wert. Die Reihenfolge, in der die IDs der Knoten im Triangles-Array abgespeichert werden, gibt die Verbindungsreihenfolge der Knoten zu Dreiecken vor, die anschließend als Teilflächen des Meshs verwendet werden können. Jeweils drei aufeinanderfolgende IDs stellen hierbei immer genau ein Dreieck dar.

In dieser Arbeit wird bei der Knotenerstellung im Vertex-Array immer zuerst eine gesamte Knotenreihe entlang der x-Achse erstellt, bevor anschließend die aktuell zu verwendende z-Koordinate um 1 erhöht wird und damit die nächste Knotenreihe in x-Richtung abgehandelt werden kann. Durch diese Anordnung wird eine Knotenreihenfolge vorgegeben, die bei der Erstellung der Mesh-Dreiecke beachtet werden muss. Ein Mesh-Quadrat besteht in der Anwendung immer aus zwei Dreiecken. In Abbildung 11 ist dies schematisch dargestellt. Bei der Reihenfolge des Hinzufügens von Knoten-IDs zum Triangles-Array muss beachtet werden, dass die von Unity gezeichneten Dreiecke aus

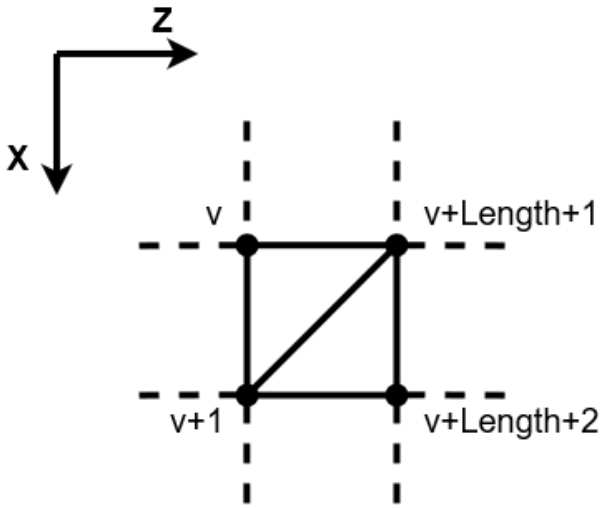

Abbildung 11:
Relative Knoten-IDs eines Mesh-Quadrats im zweidimensionalen Raum

Performance-Gründen einseitig transparent umgesetzt werden. Wird das Dreieck mit einer Knotenreihenfolge im Uhrzeigersinn erstellt, so zeigt die intransparente Seite des Dreiecks zum Betrachter hin.

Die beschriebenen Sachverhalte führen zu einer Methode zur Erstellung von Dreiecken, welche im Codebeispiel 12 und schematisch in der Abbildung 11 zu erkennen ist. Die Variable *v* aus Abbildung 11 ist gleichbedeutend mit der Variable *currentVertex* aus dem Codebeispiel 12. *Length* ist äquivalent zu *meshLength* und bestimmt die Länge des fertigen Meshs auf der x-Achse in Quadraten.

Es gibt einige Möglichkeiten, um mithilfe der in Abbildung 11 erkennbaren relativen Knoten-IDs Dreiecke im Uhrzeigersinn zu erstellen. Für diese Anwendung wurde die im Codebeispiel 12 erkennbare Reihenfolge gewählt. Für das erste Dreieck wird zuerst der aktuell betrachtete Knoten hinzugefügt (*v*), gefolgt vom rechts daneben liegenden Knoten. Hierbei muss beachtet werden, dass *Length* lediglich die Länge des Meshs in Quadraten angibt. Zu diesem Wert muss, wie in Abschnitt 4.2 genauer beschrieben, 1 hinzuaddiert werden, um die ID des gesuchten Knotens (*v+Length+1*) eindeutig zu bestimmen. Der letzte Knoten, der zum ersten Dreieck hinzugefügt wird, liegt direkt unterhalb des aktuell betrachteten Knotens. Um dessen ID zu erfassen, reicht es, die ID des aktuell betrachteten Knotens um 1 zu erhöhen (*v+1*), da dieser der nächstfolgende Knoten in x-Richtung ist. Die Knoten-IDs der zweiten Dreiecke eines jeden Quadrats folgen einer ähnlichen Logik. Hier werden die Knoten *v+1*, *v+Length+1* und *v+Length+2* in dieser Reihenfolge verwendet.

4.2 Umsetzung der Basisparameter

Addiert man zu den Parametern *Terrain Length* und *Terrain Width*, welche die Größe des Meshs in Quadraten angeben, jeweils den Wert 1 hinzu, so geben diese direkt die Gesamtknotenanzahl innerhalb des Meshs vor. Dies hat den Hintergrund, dass Quadrate innerhalb des Meshs immer nebeneinander liegen, jeweils zwei Knoten lang und zwei Knoten breit sind und sich angrenzende Knoten teilen. Die genannten Parameter werden aus diesem Grund bei der Erstellung des Vertex-Arrays, des Triangles-Arrays

```
for (int z = 0; z < meshWidth; z++)
{
    for (int x = 0; x < meshLength; x++)
    {
        terrainTriangles[0 + currentTriangles] =
            currentVertex;

        terrainTriangles[1 + currentTriangles] =
            currentVertex + meshLength + 1;

        terrainTriangles[2 + currentTriangles] =
            currentVertex + 1;

        terrainTriangles[3 + currentTriangles] =
            currentVertex + 1;

        terrainTriangles[4 + currentTriangles] =
            currentVertex + meshLength + 1;

        terrainTriangles[5 + currentTriangles] =
            currentVertex + meshLength + 2;

        currentVertex++;
        currentTriangles += 6;
    }
    currentVertex++;
}
```

Abbildung 12:
Schleifen zum Erstellen von Dreiecken für das Mesh

```
float noiseXCoord =
    x * heightDensity + noiseOffsetX;

float noiseYCoord =
    z * heightDensity + noiseOffsetZ;

float vertexHeight =
    Mathf.PerlinNoise(noiseXCoord, noiseYCoord);
```

Abbildung 13:
Berechnung der Basis-Rauschwerte zur späteren Höhenberechnung

sowie auch bei der Berechnung der Höhenlagen zur Erfassung der Mesh-Ausmaße als Schleifenbegrenzung zur Iteration über allen Knoten eingesetzt, wie im Codeausschnitt 12 in Zeile 1 und Zeile 3 an einem Beispiel erkennbar ist. Innerhalb des Codes werden die Parameter mit den Variablennamen *meshLength* und *meshWidth* bezeichnet.

Die Offset-Parameter dienen der Bestimmung der Stelle der zweidimensionalen Rauschfunktion, an der das Sample für die Berechnung der Höhenkarte entnommen wird. Diese Offsets fließen direkt in die Koordinatenberechnungen für die verwendete Perlin-Noise-Funktion ein, die im Codeabschnitt 13 zu sehen sind. Die grundsätzliche Koordinatenberechnung erfolgt hier mithilfe der aktuellen Knotenposition sowie der gewählten *Height Density*, die im Abschnitt 4.3 näher erläutert wird. Zu diesen Noise-Koordinaten wird der Offset addiert, was eine Verschiebung der Rauschfunktion um eben diesen Wert hervorruft.

4.3 Umsetzung der Höhenparameter

Die *Height Density* stellt eine Möglichkeit dar, die Frequenz des Wechsels zwischen Bergen und Tälern innerhalb der Höhenkarte anzupassen. Um diese Funktionalität umzusetzen, muss die Frequenz der Rauschfunktion mithilfe des Parameters manipuliert

```
vertexHeight = Mathf.Pow(vertexHeight, noisePower);
vertexHeight *= heightVariety;
```

Abbildung 14:
Berechnungen mit der *Height Variety* und *Noise Power*

werden können, da diese den Wechsel von Bergen und Tälern vorgibt. Zu diesem Zweck wird, wie im Codeabschnitt 13 ersichtlich, die *Height Density* zur Berechnung der Noise-Koordinaten mit den Koordinaten des aktuellen Knotens multipliziert. Eine erhöhte *Height Density* führt somit zu einer größeren Teilfläche der zweidimensionalen Rauschfunktion, die auf die Höhenkarte mit gleichbleibender Fläche projiziert wird und damit eine Frequenzerhöhung hervorruft.

Die Perlin-Noise-Funktion, die im Code des Generators verwendet wird, liefert einen Ausgabewert zwischen 0 und 1. Um diese Ausgaben in angemessene Höhenwerte innerhalb der Unity-Umgebung umzuwandeln, bedarf es der Parameter *Height Variety* und *Noise Power*. Der Ausgabewert der Rauschfunktion wird mit *Noise Power* potenziert und anschließend mit Heigth Variety multipliziert, was im Codeabschnitt 14 erkennbar ist.

Da die Basis-Rauschwerte, welche die Perlin-Noise-Funktion erzeugt, im Bereich zwischen 0 und 1 liegen, sorgt das Potenzieren mit *Noise Power* zuallererst für eine verringerung der Rauschwerte. Aufgrund der exponentiellen Natur der *Noise Power* Variable bleiben die größten Rauschwerte relativ groß, wohingegen mittlere bis kleine Rauschwerte einen drastischen Abfall verzeichnen müssen. Aufgrund dieser Wirkung werden in der späteren Landschaft, bei größerer *Noise Power* steilere Anstiege bei Bergen zu erkennen sein. *Height Variety* führt im Anschluss zu einer multiplikativen Erhöhung der Rauschwerte, wodurch diese in verwendbare Unity-Koordinaten zur Verwendung als Höhenwerte in Mesh-Knoten überführt werden.

4.4 Umsetzung der Noise-Layer Parameter

Zur Umsetzung der in Abschnitt 3.3 beschriebenen Noise-Layer, bedarf es einiger hochfrequenzierter Rauschfunktionen, die auf die bestehende Oberfläche der Höhenkarte projiziert werden. Zu diesem Zweck werden im Code 15 den in 13 berechneten Basis-Höhenwerten zwei Noise-Layer-Werte hinzuaddiert. Diese Noise-Layer-Werte berechnen sich jeweils aus dem Produkt der *Noise Layer Height Variety* und einem weiteren Perlin-Noise-Wert, in dessen Koordinatenberechnung die *Noise Layer Density* verwendet wird.

Berechnet werden die Perlin-Noise-Werte für die Noise Layer (Code 15), Zeile 7 und 13), wie auch die Rauschwerte des Basisterrains, durch Multiplizieren der zugehörigen x- und z-Koordinaten des aktuell betrachteten Mesh-Knotens mit der *Noise Layer Density*. Im Anschluss wird noch der reguläre Offset der Rauschwerte des Basisterrains sowie einige möglichst unabhängige Summanden zu den Layer-Rauschkoordinaten hinzuaddiert, um die Korrelation zwischen den Basis-Rauschwerten und den Layer-Rauschwerten weitestgehend aufzuheben. Würde dieser Schritt ausgelassen werden, so könnten in den fertigen Landschaften unter Umständen Höhenmuster im Basisterrain beobachtet werden, welche ebenfalls im kleineren Stil als Terrainoberfläche vorkommen.

Bei der Umsetzung der Noise Layer wurde mithilfe einiger Tests entschieden, dass sich zwei Layer in unterschiedlichen Frequenz- und Höhenwertbereichen gut für die Berechnung einer natürlichen rauen Oberfläche eignen. Eine höhere Anzahl an

Layern würde ein Beobachter kaum bemerken. Weniger Layer schränken wiederum den Benutzer in seinen Möglichkeiten der Oberflächengestaltung ein. Im Falle der Anwendung dieser Arbeit wird einer der Layer immer in der doppelten Frequenz (Code 15, Zeilen 14-15) und mit lediglich einem halbierten Höhenfaktor (Code 15, Zeile 12) berechnet, um Abstufungen in der Rauheit der Oberflächen hervorzurufen.

Bei der Festlegung der *Noise Layer Density* ist darauf zu achten, dass diese um ein Vielfaches größer gewählt werden muss, als die *Noise Density* der Basis-Höhenwerte. Der Hintergrund ist hierbei, dass durch die Noise Density die Frequenz der Rauschfunktion auf der Höhenkarte festgelegt wird. Um den Effekt einer rauen Oberfläche auf dem Basisterrain herbeizuführen, sind Rauschfunktionswerte in einer höheren Frequenz erforderlich, als diejenigen, die für die Berechnung des Basisterrains genutzt wurden. Wird diese Bedingung nicht eingehalten, so kann kaum eine Veränderung an der Oberfläche des Terrains wahrgenommen werden. Die Abstände zwischen den Erhebungen der Noise Layer wären zu groß und die Höhenwerte dieser Erhebungen (bei einer sinnvoll gewählten *Noise Layer Height Variety*) zu niedrig, um das Erscheinungsbild des Terrains maßgeblich zu beeinflussen.

Bei der Berechnung der Noise Layer muss zusätzlich beachtet werden, dass der Basis-Rauschwert, welcher vor der Noise-Layer-Berechnung laut Dokumentation der Perlin-Noise-Funktion standardmäßig im Bereich zwischen 0 und 1 liegt, durch die Addition der Noise-Layer-Rauschwerte über den Wertebereich der Perlin-Noise-Funktion hinaus wachsen kann. Um in den folgenden Berechnungen weiterhin mit Rauschwerten im vorgegebenen Wertebereich zwischen 0 und 1 rechnen zu können, ist an dieser Stelle eine Problembehandlung erforderlich. Die Noise-Werte, die von der Perlin-Noise-Funktion im Code 15 in Zeile 7 und 13 berechnet werden, liegen per definition immer im Bereich zwischen 0 und 1. Hieraus folgt, dass das Produkt von Noise-Wert und Height Variety bzw halbierter Height Variety in Zeile 6 und 12 niemals den Wert der Height Variety bzw der halbierten Height Variety übersteigen kann. Teilen wir also am Ende der Methode die Summe der tatsächlichen Rauschwerte durch die Summe aller Obergrenzen der berechneten Rauschwerte (Code 15, Zeile 18-22), so ist das Ergebnis wieder auf den Wertebereich zwischen 0 und 1 begrenzt und enthält immernoch alle vorangegangenen Ergebnisse der Noise-Layer-Berechnungen.

Im Abschnitt 3.3 wurde eine Zweiteilung des Terrains in höhere und tiefere Regionen besprochen, für die im Anschluss die Möglichkeit besteht, mit unterschiedlichen Noise-Layern kombiniert zu werden. Zu diesem Zweck wurde die anpassbare *Upper Height Percent* Variable implementiert, die angibt, wie viel Prozent des Terrains, von oben beginnend entlang der Höhenachse, zur oberen Region gezählt werden. Hierfür ist es vorerst nötig, die Maximalhöhe sowie die Minimalhöhe der Höhenkarte während dessen Generierung zu erfassen. Diesem Zweck dient der Code im Beispiel 16, der nach der Berechnung jedes einzelnen Höhenwertes ausgeführt wird.

Diese Grenzwerte werden anschließend im Codebeispiel 17 zusammen mit dem *Upper Height Percent* Parameter verwendet, um zu entscheiden, ob auf den aktuellen Höhenwert die Noise-Layer der niedrigeren oder der höheren Regionen projiziert werden sollen. Zu diesem Zweck wird in den Zeilen 1-10 der Starthöhe der oberen Regionen in Unity-Koordinaten mithilfe der genannten Variablen ermittelt. Im Anschluss wird durch einen Vergleich des aktuellen Höhenwertes mit dem Startwert der oberen Region in den Zeilen 12-19 entschieden, ob in den Zeilen 19-33 die Noise Layer der unteren oder der oberen Region auf den Höhenwert

```
1  float AddNoiseLayers(
2      float height,
3      float noiseLayerHeightVariety,
4      float noiseLayerDensity
5  ) {
6      height += noiseLayerHeightVariety *
7      Mathf.PerlinNoise(
8          x * noiseLayerDensity + noiseOffsetX + 5.3f,
9          z * noiseLayerDensity + noiseOffsetZ + 9.1f
10     );
11
12     height += noiseLayerHeightVariety / 2 *
13     Mathf.PerlinNoise(
14         x * noiseLayerDensity*2 + noiseOffsetX + 17.8f,
15         z * noiseLayerDensity*2 + noiseOffsetZ + 23.5f
16     );
17
18     height /=
19         (   1
20             + noiseLayerHeightVariety
21             + noiseLayerHeightVariety/2
22         );
23     return height;
24 }
```

Abbildung 15:
Berechnung der Noise Layer

```
if (z==0 && x==0) {
    minTerrainHeight = vertexHeight;
    maxTerrainHeight = vertexHeight;
} else {
    if (vertexHeight < minTerrainHeight)
        minTerrainHeight = vertexHeight;
    if (vertexHeight > maxTerrainHeight)
        maxTerrainHeight = vertexHeight;
}
```

Abbildung 16:
Erfassung der Maximal- und Minimalhöhen während der Höhenberechnungen

angewendet werden sollen. Bei dem genannten Vergleich muss beachtet werden, dass in diesem Abschnitt des Game-Loops noch keine Berechnung der Basis-Höhenwerte erfolgen konnte (zuerst müssen die Noise Layer in den Rauschwert eingerechnet werden), und somit die vertexHeight-Variable lediglich einen Perlin-Noise-Rauschwert im Wertebereich zwischen 0 und 1 enthält. Um trotzdem Zugriff auf einen Höhenwert zu haben, der möglichst nahe an der Realität liegt, ohne alle Höhenwerte des vorangegangenen Game-Loops zu speichern, wird der Vergleich in Zeile 12-19 mithilfe einer linearen Interpolartion mit dem aktuellen Rauschwert zwischen den Höhen-Genzwerten des letzten Game-Loops realisiert.

4.5 Umsetzung der ästhetischen Parameter

Für sowohl das Wasser, als auch die farbliche Gestaltung, die innerhalb des Terrains erkennbar sein soll, reicht innerhalb dieses Projektes eine einfache Darstellung aus, da der eigentliche Fokus auf der generierten Höhenkarte liegen soll. Aus diesem Grund wird der Wasserspiegel lediglich mit Hilfe einer zweidimensionalen eingefärbten Plane und die Farbgestaltung mithilfe eines Höhengradienten umgesetzt.

Um bestimmbar zu machen, in welcher Höhe die Wasser-Plane in das Terrain eingesetzt werden soll, musste zunächst eine neue Float-Variable (*Relative Water Height*) mit einer Begrenzung auf

```
float heightDifference =
    maxTerrainHeight - minTerrainHeight;

float upperHeightRange =
    heightDifference * upperHeightPercent;

float upperHeightBegin =
    minTerrainHeight
    + heightDifference
    - upperHeightRange;

if
(   Mathf.Lerp(
        minTerrainHeight,
        maxTerrainHeight,
        vertexHeight
    )
    <= upperHeightBegin
) {
    vertexHeight =
        AddNoiseLayers(
            vertexHeight,
            lowerNoiseLayerHeightVariety,
            lowerNoiseLayerDensity
        );
} else {
    vertexHeight =
        AddNoiseLayers(
            vertexHeight,
            upperNoiseLayerHeightVariety,
            upperNoiseLayerDensity
        );
}
```

Abbildung 17:
Berechnung der Starthöhe der oberen Noise-Layer-Region und Anwendung passender Noise Layer

```
[Range(0,1)]
public float relativeWaterHeight = 0f;
```

Abbildung 18:
Begrenzung des Wasserhöhe-Parameters mithilfe eines Reglers

den Wertebereich zwischen 0 und 1 angelegt werden, um dem Benutzer die Anpassung der Höhe des flachen Wasserspiegels in Prozent der Gesamthöhe des Terrains zu ermöglichen. Eine solche Begrenzung ist in Unity innerhalb des Inspektors in Form eines Reglers möglich, der ein von Unity vorgegebenes UI-Element ist. Der Code für die Implementierung eines solchen Reglers zu einer Variable ist in Abbildung 18 zu sehen.

Diese relative Wasserhöhe wird anschließend in einem gesonderten Game Object, dem sogenannten WaterGenerator, weiter verarbeitet, da für diesen Abschnitt ein gesondertes Mesh generiert werden soll. Zwischen der minimalen und der maximalen Terrainhöhe in Unity-Koordinaten, die im Code 16 bestimmt wurden, kann nun über den *Relative Water Height* Parameter linear interpoliert werden, um die benötigte Höhe der Wasserplane in Unity-Koordinaten zu ermitteln. Anschließend erfolgt mithilfe dieses Wertes die Generierung der Wasserplane innerhalb des Terrains analog zur Mesh-Generierung in Abschnitt 4.1. Der Zugehörige Code ist in Abbildung 19 erkennbar.

Die Farbe, sowohl des Terrains als auch des Wassers, wird dem jeweiligen Mesh über ein Array vom Typ *Color* übermittelt und visualisiert. Für das Wasser wird in dieser Anwendung eine

```
if (relativeWaterHeight != 0) {
    float absoluteWaterHeight = Mathf.Lerp(
     waterMinHeight, waterMaxHeight,
     relativeWaterHeight);
    waterVertices[0] = new Vector3(1,
     absoluteWaterHeight, 1);
    waterVertices[1] = new Vector3(meshLength+1,
     absoluteWaterHeight, 1);
    waterVertices[2] = new Vector3(1,
     absoluteWaterHeight, meshWidth+1);
    waterVertices[3] = new Vector3(meshLength+1,
     absoluteWaterHeight ,meshWidth+1);

    int[] waterTriangles = new int[6];
    waterTriangles[0] = 2;
    waterTriangles[1] = 1;
    waterTriangles[2] = 0;
    waterTriangles[3] = 3;
    waterTriangles[4] = 1;
    waterTriangles[5] = 2;

    Color[] waterColors = new Color[4];
    for (int i = 0; i < waterColors.Length; i++)
        waterColors[i] = waterPlaneColor;

    waterPlaneMesh.Clear();
    waterPlaneMesh.vertices = waterVertices;
    waterPlaneMesh.triangles = waterTriangles;
    waterPlaneMesh.colors = waterColors;
    waterPlaneMesh.RecalculateNormals();
} else {
    waterPlaneMesh.Clear();
}
```

Abbildung 19:
Generieren und Einfärben der Wasserplane

```
for (int i = 0, z = 0; z <= meshWidth; z++)
{
    for (int x = 0; x <= meshLength; x++)
    {
        float gradientValue = Mathf.InverseLerp(
     gradientMinValue, gradientMaxValue,
     terrainVertices[i].y);
        terrainColors[i] = usedGradient.Evaluate(
     gradientValue);
        i++;
    }
}
```

Abbildung 20:
Einfärben des Terrains mithilfe eines Gradienten

einzige Farbe in dem Parameter *WaterPlaneColor* vom Nutzer gespeichert, die auf alle Knoten des Wasser-Meshs angewendet wird (Code 19, Zeile 28 und 33). Der Prozess zum Einfärben des Terrains (Code 20) verwendet wiederum einen Gradienten. Dieser kann im Unity-Inspektor frei und dynamisch an das bestehende Terrain angepasst werden. Im Anschluss wird mithilfe einer InverseLerp-Berechnung der aktuellen Terrainhöhe zwischen der minimalen und der maximalen Terrainhöhe ein Prozentwert berechnet, der zur Bestimmung eines einzelnen Farbwertes aus dem Gradienten verwendet wird.

Zuletzt ergab sich noch ein Problem innerhalb der Anwendung aus dem Zusammenspiel von sowohl den relativen Gradient- und Wasserhöhen als auch dem dynamisch vergrößerbaren Terrain. Erstellt der Benutzer ein spezifisches Terrain nach seinen Wünschen und vergrößert dieses anschließend beliebig, so existiert

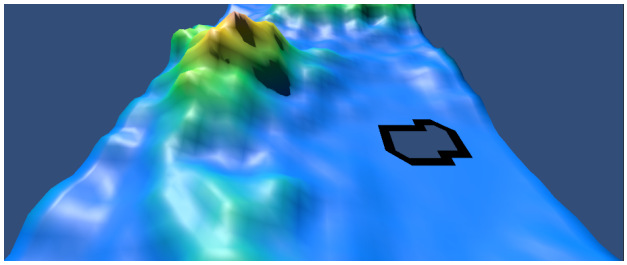

Abbildung 21:
Ein Loch an einem globalen Tiefpunkt im Mesh

eine große Chance, dass neue globale Hoch- oder Tiefpunkte auf der Höhenkarte generiert werden. Dieses Auftreten neuer Maxima und Minima führt im Anschluss zwangsweise zu einer Anpassung der absoluten Wasserhöhe sowie der Terraineinfärbung, da diese auf den relativen Höhenangaben des Nutzers innerhalb der aktuellen Grenzwerte beruhen. Tritt dieses Problem auf, so verändert sich während der Anpassung der Terrainausmaße das Aussehen des Terrainabschnitts, an dem der Nutzer ursprünglich den Höhenkartengenerator konfiguriert hat, was die Benutzerfreundlichkeit der Anwendung stört. Um dieses Problem zu beheben, wurden Buttons in die Anwendung integriert, die das Feststellen sowohl der Wasser- als auch der Gradienthöhen auf den aktuellen absoluten Höhenwert in Unity-Koordinaten ermöglichen. Auf diese Weise bewegen sich die relativen Wasser- und Gradienthöhen immer zwischen genau den globalen Höhengrenzwerten, die beim letzten Betätigen des jeweiligen Buttons aktuell waren.

4.6 Probleme mit imaginären Höhenwerten

Nach der vollständigen Implementierung der Anwendung wurde diese intensiv getestet, um die Funktionssicherheit im normalen Gebrauch festzustellen. Hierbei ist aufgefallen, dass sehr selten (etwa alle 300.000 Quadrate) Löcher an globalen Tiefpunkten im Mesh auftreten. Ein Beispiel für ein solches Loch ist in Abbildung 21 erkennbar.

Durch einiges Ausprobieren stellte sich heraus, dass im Schritt der Noise-Power-Berechnung (Code 14, Zeile 2) innerhalb solcher Anomalien zur Ermittlung des Höhenwertes ein negativer Rauschwert mit einem positiven *Noise Power* Parameter potenziert wird. Eine solche Berechnung ergibt zwangsweise imaginäre Zahlen, die nicht in Float-Variablen von Unity gespeichert werden können. Somit existierte innerhalb der Anomalien keine y-Koordinate zur Generierung von Knoten für das Mesh, wodurch Löcher entstanden.

Eine Erklärung dafür, weshalb in sehr seltenen Fällen negative Rauschwerte nach der Verrechnung der Höhenparameter übrig geblieben sind, bietet die Unity-Dokumentation [5] der Perlin-Noise-Funktion: „It is possible for the return value to be slightly less than 0.0f or slightly exceed 1.0f. You may need to clamp the return value if the 0.0 to 1.0 range is important to you.“ Folgt man dem auf der Website vorgeschlagenen Lösungsansatz und begrenzt die Rauschwerte vor der Verrechnung mit dem *Noise Power* Parameter nach unten mit dem Wert 0, so gehen innerhalb der Anwendung alle globalen und Noise-Layer-Höhendaten im negativen Rauschwertbereich verloren. Da diese Daten jedoch nur einen Bruchteil der eigentlichen Höhendaten ausmachen und bei der Generierung kaum ins Gewicht fallen, wurde diese Lösung

```
vertexHeight = Math.Max(0,vertexHeight);
```

Abbildung 22:
Begrenzung der Rauschwerte vor dem Potenzieren

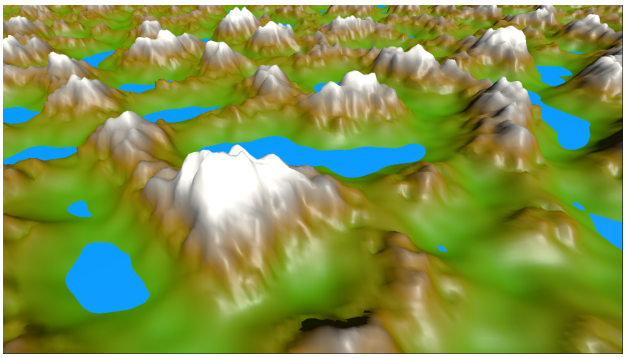

Abbildung 23:
Generierte Berglandschaft

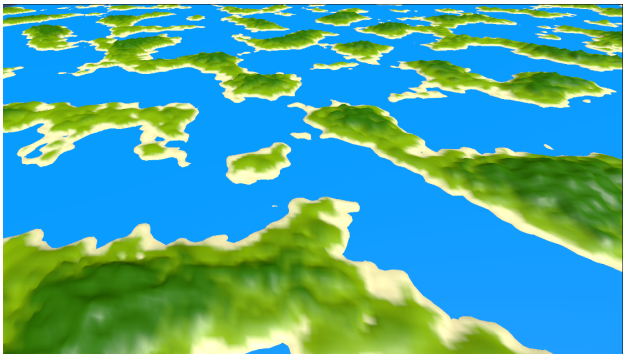

Abbildung 24:
Generierte Insellandschaft

letztendlich als ausreichende Annäherung in der Anwendung verwendet. Der zugehörige Code ist in der Abbildung 22 zu sehen.

5 EVALUATION UND FAZIT

Zusammenfassend kann die Implementierung eines prozeduralen Landschaftsgenerators im Rahmen dieser Arbeit als Erfolg angesehen werden. Zum einen wurde eine gut funktionierende Möglichkeit gefunden, dreidimensionale Terrain-Meshes zu generieren. Außerdem konnten die in Abschnitt 3 konzipierten Parameter zur Anpassung der Mesh-Eigenschaften vollständig in der Anwendung umgesetzt werden. Einige Beispiele dafür, welchen Umfang die entwickelte Anwendung bietet, sind in den Abbildungen 23 und 24 zu erkennen.

Trotz des Erreichens der Ziele dieser Arbeit werden potenzielle Nutzer des Generators in dieser Version noch mit einigen Hindernissen konfrontiert. Unter anderem wird es schwerfallen, mithilfe der 14 unterschiedlichen Parameter, die größtenteils eine sehr präzise Feinabstimmung benötigen, eigene mentale Modelle von Landschaften umzusetzen, ohne dass eine ausreichende Einarbeitungszeit besteht. Hierbei ist es in einigen Fällen sogar denkbar, dass ein mentales Modell mit dem Generator in seiner aktuellen Version gar nicht umgesetzt werden kann. Die

Generierung eines Canyons mit vielen senkrechten Abhängen ist beispielsweise aktuell nicht möglich.

Des Weiteren muss angemerkt werden, dass der Generator zwar beliebig große Landschaften generieren kann, diese aber ab einer bestimmten Größe einen zu hohen Berechnungsaufwand erfordern, wodurch die Framerate sinkt und die interaktive Ausgabe beeinträchtigt wird. Diese kritische Terraingröße ist abhängig von der Systemleistung und beträgt auf dem verwendeten Testsystem etwa 100.000 Mesh-Quadrate. Unter realen Bedingungen kann dieses Problem jedoch mithilfe von Techniken wie Culling-Verfahren und beschränkten Renderdistanzen relativ einfach behoben werden.

6 AUSBLICK

Es gibt ein breites Feld an Einsatz- und Erweiterungsmöglichkeiten für die Anwendung, die im Rahmen dieses Projektes entwickelt wurde. Um sich der Thematik digitaler Spiele weiter anzunähern, könnte als Nächstes an der Generierung von Biomen innerhalb der Höhenkarten gearbeitet werden. Hierbei wären sowohl die Konzeption eines Feuchtigkeits- bzw. Wettersystems als auch die Generierung glaubwürdiger Biom- und Terrainübergänge notwendig. Des Weiteren sollte das Terrain zur Schaffung einer interessanteren Spielwelt mit zusätzlichen Objekten wie Vegetation, Tieren, Siedlungen und NPCs ausgestattet werden.

Die Auswahl an Objekten und Systemen, die für ein solches Spiel umsetzbar und sinnvoll wäre, ist sehr groß. Diese Auswahl einzugrenzen, erfordert eine Präzisierung des Zielgenres. Eine Möglichkeit wäre es beispielsweise, den Terraingenerator für Spiele des Natur-Survival-Genres in der tropischen Wildnis zu erweitern.

LITERATUR

[1] 2022. Prozedurale Generierung. (Oktober 2022). https://de-academic.com/dic.nsf/dewiki/1136526

[2] Ken Perlin. 1985. An image synthesizer. In *Proceedings of ACM SIGGRAPH 85*. vol 19, pp. 287–296. https://dl.acm.org/doi/pdf/10.1145/325165.325247

[3] Ravi Hiranand. 2015. 18 quintillion planets: The video game that imagines an entire galaxy. (Juni 2015). http://edition.cnn.com/2015/06/18/tech/no-mans-sky-sean-murray/

[4] Richard Moss. 2017. Roam free: A history of open-world gaming. (März 2017). https://arstechnica.com/gaming/2017/03/youre-now-free-to-move-about-vice-city-a-history-of-open-world-gaming/

[5] Unity Technologies. 2022. Unity Documentation. (Oktober 2022). https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html