

MASTERARBEIT IM STUDIENGANG GAME ENGINEERING UND VISUAL COMPUTING

NOISE-BASIERTE GENERIERUNG VON 3D WELTEN FÜR EINE VR FLUGSIMULATION

Kurzbeschreibung

Diese Arbeit beschreibt die Implementation einer Flugsimulation, dessen Spielwelt auf Noise basierend komplett prozedural generiert ist. Die Spielwelt wird mittels einer Funktion definiert, über welche die Extraktion des Terrains in der Form einer Isofläche und der Hilfe des Dual Marching Cubes Algorithmus stattfindet. Des Weiteren wird die immersive Wahrnehmung und Interaktion mittels der Nutzung eines VR-Headsets und eines Icaros-Systems ermöglicht.

Verfasser:

Dennis Vidal

Aufgabensteller/Prüfer:

Prof. Dr. Christoph Bichlmeier

Arbeit vorgelegt am:

01.04.2022

Fakultät:

Fakultät für Informatik

Studiengang:

Game Engineering und Visual Computing

Matrikelnummer:

322958

VORWORT:

Ich möchte mich an dieser Stelle herzlich bei sowohl Prof. Dr. Christoph Bichlmeier für die Betreuung dieser Masterarbeit als auch bei allen Testern für die Teilnahme an der Evaluierung bedanken.

INHALTSVERZEICHNIS

INHALTSVERZEICHNIS	3
1 EINLEITUNG	5
1.1 Zielsetzung.....	6
1.2 Aufbau der Arbeit	7
2 NOISE.....	7
3 ISOFLÄCHENEXTRAKTION.....	10
3.1 Marching Cubes	11
3.2 Dual Contouring	14
3.3 Dual Marching Cubes von Schaefer und Warren	17
3.4 Dual Marching Cubes von Nielson	19
4 TERRAINDEFINIERUNG ÜBER EINE FUNKTION	20
5 TEXTURIERUNG DES TERRAINS.....	24
5.1 Triplanar Mapping.....	25
5.2 Texture Splatting	26
5.3 Texture Bombing	27
6 ANFORDERUNGEN	28
6.1 Virtual Reality.....	28
6.2 Icaros	30
7 GENERIERUNG DER SPIELWELT	31
7.1 Erster Extraktionsansatz	32
7.2 Zweiter Extraktionsansatz	36
7.3 Chunks	41
7.4 Level of Detail	42
7.5 Seams.....	46
7.6 Terrainfunktion	58
7.7 Texturierung.....	60

8	IMPLEMENTATION DER SPIELELEMENTE	63
8.1	Hauptmenü.....	63
8.2	Spawnen des Spielers	64
8.3	Drill-Ray	65
8.4	Items	66
8.5	Expeditionsbike	67
8.6	Sammeln von Punkten	69
8.7	Schwierigkeitsgrad	70
9	EVALUIERUNG DER SPIELBARKEIT.....	70
10	DISKUSSION DER ERKENNTNISSE.....	72
11	ABBILDUNGSVERZEICHNIS.....	75
12	LITERATURVERZEICHNIS	77
13	ANHANG	80
14	ERKLÄRUNGEN	83
14.1	Selbstständigkeitserklärung	83
14.2	Ermächtigung	83

1 EINLEITUNG

Dank der steigenden Hardwareleistung von Computern und Konsolen, neuen und besser optimierten Softwarealgorithmen, und vor allem auch den damit verbundenen steigenden Erwartungen der Spieler, werden Spiele heutzutage immer größer und komplexer. Das betrifft nicht nur Bereiche wie die Grafik oder die Spielmechaniken, mit denen der Spieler die virtuelle Welt wahrnimmt und mit dieser interagiert, sondern vor allem auch die virtuelle Welt selbst. Viele Spiele bieten daher eine umfangreiche und meist auch sehr detaillierte Spielwelt, in die der Spieler eintauchen kann. Der Großteil dieser Welten ist dabei von Hand erstellt. Das erlaubt es den Entwicklern unter anderem leicht die Landschaft und die Platzierung von Objekten und NPCs an die vorgesehenen Geschehnisse innerhalb des Spieles anzupassen und damit auch die Glaubhaftigkeit der Welt zu erhöhen. Dieser Ansatz bringt allerdings auch einige Nachteile mit sich. Einer dieser Nachteile ist die aufgrund des Erstellungsaufwands begrenzte Größe der virtuellen Welt. Während zum Beispiel eine Landschaft mit einigen Quadratkilometern noch mit relativ überschaubarem Aufwand erstellt und mit Objekten versehen werden kann, ist eine Landschaft mit tausenden Quadratkilometern aufwandstechnisch nahezu nicht manuell realisierbar. Die limitierte Größe führt auch dazu, dass der Spieler gestoppt werden muss über die Grenzen der Welt hinaus zu wandern. Der einfachste Weg dies zu erreichen ist die Nutzung von unsichtbaren Wänden in Form von Collidern. Unsichtbare Wände sind allerdings nicht gerade immersiv für das Spielerlebnis. Stattdessen werden die Grenzen oft immersiver mit Klippen, Bergen, Flüssen, einem Meer oder anderen Effekten versehen, um den Spieler vom Erkunden zu stoppen. In vielen Fällen wirken sich aber auch diese immersiveren Grenzen negativ auf das Spielerlebnis aus, da der Spieler in seiner Erkundungsfreiheit eingeschränkt ist. Ein weiterer Nachteil ist, dass das Layout der Landschaft immer gleich ist. So sind beispielsweise Gebäude immer an der gleichen Position und der Pfad zum Ziel ist auch immer derselbe. In vielen Spielen, die aufgrund des Settings, der Story oder aus anderen Gründen auf eine bestimmte Region begrenzt sind, ist dieser unveränderliche Teil der Spielwelt zwar meist erwünscht, er führt aber auch oft zu einer allgemein geringeren Wiederspielbarkeit des Spieles.

Eine Alternative zu dieser manuellen Erstellung ist die komplett oder zumindest teilweise prozedurale Generierung der virtuellen Welt. Anstatt das Terrain und Objekte von Hand zu modellieren und zu platzieren, werden bei der prozeduralen Generierung lediglich bestimmte Regeln, oft in der Form von mathematischen Formeln, zur Generation festgelegt. Verschiedene Algorithmen nutzen dann diese Regeln, um daraus die virtuelle Welt zu erstellen und diese mit Objekten zu versehen. Die prozedurale Generierung beseitigt einige Nachteile, die im Zusammenhang mit der Erstellung von Hand auftreten. Nachdem das Aussehen der Spielwelt über bestimmte Regeln definiert ist, gibt es keine direkte Größenlimitation des spielbaren Bereiches. Solange die Regeln dafür ausgelegt sind, kann der Spieler die Welt also in jede beliebige Richtung endlos frei erkunden. Zusätzlich kann über das Einbauen von zufälligen Parametern und dem Anpassen der Generationsregeln jede Spielsession und generierte Welt variieren und ein neues Erlebnis bieten. Diese Variation führt zu einer höheren Wiederspielbarkeit und in vielen Spielen auch zu einem intensiveren Spielerlebnis, da der Spieler auf nicht vorhersehbare Ereignisse reagieren und für diese eine Strategie formulieren muss.

Das Ausmaß des prozeduralen Generierungsanteils kann in diesem Ansatz jedoch von Spiel zu Spiel stark variieren. Während Spiele wie *Hades*¹ oder *Deep Rock Galactic*² vorgefertigte Bausteine nutzen und diese über bestimmte Regeln miteinander verbinden, um ein Level zu

¹ Hades - <https://www.supergiantgames.com/games/hades/>

² Deep Rock Galactic - <https://www.deeprockgalactic.com/>

erstellen, generieren Spiele wie *Minecraft*³ und *Terraria*⁴ ihre Spielwelt nahezu komplett prozedural. In allen dieser Fälle profitiert das Spielerlebnis und letztendlich auch die Spieler von der Existenz prozeduraler Elemente, nahezu unabhängig vom letztendlichen Grad, zu dem diese prozedural generiert sind.

Unabhängig von der prozeduralen Generierung hat über die vergangenen Jahre auch Virtual Reality (VR) mehr und mehr an Bedeutung gewonnen. Heutzutage gibt es eine Vielzahl an Spielen, die diese immersive Technologie nutzen, um das Interesse der Spieler zu wecken. Das Wachstum von VR zeigt sich dabei nicht nur in der Software, sondern auch hardwareseitig tragen stetig neue Produkte dazu bei die Immersivität des Erlebnisses zu verbessern. Dazu zählen vor allem neue VR-Headsets mit besseren Spezifikationen, wie einer höheren Displayauflösung, aber auch Accessoires, wie Treadmills, die weitere, meist immersivere, Interaktionsmöglichkeiten mit der virtuellen Welt bieten.



ABBILDUNG 1: AUSSCHNITT AUS DER LETZTENDLICHEN FLUGSIMULATION.

1.1 ZIELSETZUNG

Das grundlegende Ziel dieser Arbeit ist es eine immersive Flugsimulation zu entwickeln, in der das gesamte Terrain basierend auf Noise prozedural generiert ist. Für die immersive Wahrnehmung der virtuellen Welt macht sich die Simulation ein VR-Headset zu nutzen. Um die Immersion des Spielers noch weiter zu erhöhen, findet der Großteil der Interaktionen in der Flugsimulation über ein Icaros-System statt, über das der Spieler das virtuelle Flugzeug komplett mit den eigenen Körperbewegungen steuern kann.

Der Fokus der Arbeit liegt neben den prozeduralen Aspekten der Generierung vor allem auf der Spielbarkeit der erzeugten Welt und der Simulation als gesamtes. Vor allem für ersteres muss sichergestellt sein dass die virtuelle Welt jederzeit einen Pfad für den Spieler bereit hält oder es möglich ist einen solchen Pfad zu schaffen. Entsprechend sind verschiedene

³ Minecraft - <https://www.minecraft.net/>

⁴ Terraria - <https://terraria.org/>

Spielelemente vorhanden welche den Spaßfaktor der Simulation erhöhen und gleichzeitig die Nutzbarkeit gewährleisten.

Um die Spielbarkeit der Flugsimulation sicherzustellen, wird am Ende der Arbeit eine Evaluation durch eine Reihe an Testern durchgeführt. Neben der Sicherstellung der grundlegenden Spielbarkeit wird dadurch ebenfalls Feedback gesammelt, mit dem Aspekte dieser oder auch zukünftiger Arbeiten verbessert werden können, um den Grad der Spielbarkeit weiter zu erhöhen.

1.2 AUFBAU DER ARBEIT

Um die späteren Kapitel dieser Arbeit auf die Implementation der Flugsimulation fokussieren zu können, beschäftigen sich die nächsten vier Kapitel zunächst mit der Beschreibung der grundlegenden Methoden, welche für die Generation des Terrains in der Flugsimulation verwendet werden. Als erstes beschäftigt sich Kapitel 2 dafür mit dem Rauschen welches als Basis für das Terrain dient. Darauffolgend werden die in Frage kommenden Verfahren zur Isoflächenextraktion, über welche das Mesh des Terrains erzeugt wird, nähergebracht. In Kapitel 4 wird erklärt wie aus Rauschen in Kombination mit mathematischen Operationen eine Funktion definiert werden kann, mit der die Isofläche für die Extraktion des Terrainmeshes beschrieben wird. Als letztes der Grundlagenkapitel beschäftigt sich Kapitel 5 mit den Methoden, welche in der Texturierung des erzeugten Terrains Anwendung finden. Aufgrund der Steuerung der Flugsimulation mittels eines Icaros-Systems und der Wahrnehmung der Spielwelt über VR entstehen einige Anforderungen an die Flugsimulation, welche sichergestellt werden sollten, um die Spielbarkeit zu garantieren. Die wichtigsten dieser Anforderungen werden in Kapitel 6 behandelt. Die eigentliche Beschreibung der Implementation der Simulation beginnt mit Kapitel 7. In diesem werden die grundlegenden Komponenten für die Generation der Spielwelt erklärt. Getrennt davon befinden sich die Informationen zur Implementation der Spielelemente im darauffolgenden Kapitel 8. Die Ergebnisse der Spielbarkeitevaluierung können zusammen mit der Diskussion der Erkenntnisse der gesamten Arbeit jeweils am Ende in Kapitel 9 und 10 gefunden werden.

2 NOISE

Rauschen, auch allgemein mit dem englischen Begriff *Noise* bezeichnet, hat zahlreiche Anwendungsgebiete. Im Kontext von Videospielen liefern die durch Noise-Algorithmen generierbaren Zahlen nicht nur eine einfache Möglichkeit eine zufällige Variation in verschiedene Events und Elemente des Spieles einzubringen, sie können auch schon in der Modellierung und Texturierung von Modellen genutzt werden, um deren Aussehen zu variieren und prozedural Texturen oder sogar ganze Objekte für die virtuelle Welt zu erstellen.

Je nachdem welcher Noise-Algorithmus verwendet wird, besitzen die erzeugten Zahlen teilweise stark unterschiedliche Eigenschaften. Eine klassische Variante von Noise ist das *White Noise*, welches uniform verteilt komplett zufällige Zahlen liefert, die entsprechend keinerlei Korrelation zueinander aufweisen (Ebert et al., 2002, S. 67). Eventuell kontraintuitiv ist es bei Noise jedoch nicht immer erwünscht das die gelieferten Zahlen wirklich zufällig gewählt sind. In Spielen werden stattdessen vorwiegend Noise-Algorithmen verwendet welche sogenannte Pseudozufallszahlen generieren. Eine Eigenschaft dieser Pseudozufallszahlen ist, dass sie zwar mehr oder weniger zufällig gewählt erscheinen, in Wahrheit aber berechenbar sind und dementsprechend zuverlässig auf eine bestimmte Eingabe immer dieselbe Ausgabe liefern können. Diese Berechenbarkeit bietet im Rahmen der Spielentwicklung enorme Vorteile, da viele unterschiedliche Elemente, von Modellen und Texturen bis hin zu Charakterverhalten oder sogar gesamten Levels, über verschiedene

Eingabeparameter parametrisiert und entsprechend zuverlässig prozedural generiert werden können.

Im Kontext der prozeduralen Generierung von Terrain ist es für Noise ebenfalls oft nicht erwünscht, dass Zahlen an nahe beieinanderliegenden Positionen stark unterschiedliche Werte besitzen können, auch wenn diese berechenbar sind. Für die Erstellung von Terrain ist meist eine graduelle Änderung benachbarter Zahlen vorteilhafter, da das Ergebnis dadurch um ein Vielfaches natürlicher wirkt. Zwei Arten von Noise, die solche graduellen Übergänge besitzen sind beispielsweise *Value Noise* und *Gradient Noise* (Ebert et al., 2002, S. 72). Vor allem die letztere Kategorie eignet sich gut für die Generierung von natürlich aussehenden Texturen und Landschaften. Nicht jeder Noise-Algorithmus besitzt jedoch die gleiche Qualität und es gibt eine Reihe an wichtigen Eigenschaften, die ein idealer Algorithmus besitzen sollte. Green definiert die folgenden fünf Eigenschaften als wichtig (Green, 2005, Kapitel 26.1):

- Für eine bestimmte Eingabe wird immer derselbe pseudozufällige Wert ausgegeben
- Die Ausgabewerte besitzen einen bekannten Wertebereich
- Das Noise hat eine bandbegrenzte Ortsfrequenz
- Es weist keine Muster auf, welche sich offensichtlich wiederholen
- Die Ortsfrequenz ist translationsinvariant

Ein Noise-Algorithmus, welcher die obigen Eigenschaften besitzt, ist Perlin Noise (Green, 2005, Perlin, 1985). Perlin Noise ist eine Art von Gradient Noise. Wie der Name schon impliziert, bestehen die Ausgabewerte von Gradient Noise aus Werten, welche basierend auf Gradienten an bestimmten Gitterpunkten berechnet werden (Ebert et al., 2002, S. 72). Als Eingabe für dreidimensionales Perlin Noise dient eine beliebige Position im Raum. Von der Dimensionalität her ist Perlin Noise selbst allerdings nicht begrenzt und es können Varianten für beliebige Dimensionalitäten formuliert werden, welche entsprechend eine n-dimensionale Position als Eingabe erhalten. Für die Nutzung im Zusammenhang mit der prozeduralen Terraingenerierung werden in der Regel jedoch Varianten für drei oder weniger Dimensionen genutzt.

Die Funktionsweise von Perlin Noise ist recht simpel. Der Raum wird zunächst in eine Gitterstruktur unterteilt. Jeder Punkt dieses Gitters befindet sich dabei an einer Integer-Position. Basierend auf diesem Gitter kann der Ausgabewert vereinfacht gesehen in fünf Schritten berechnet werden:

1. Finde den Gitterabschnitt bzw. die Zelle des Gitters in dem sich die Eingabeposition befindet
2. Wähle Gradientenvektoren aufgrund von pseudozufälligen Integer-Werten an den Eckpunkten der Gitterzelle
3. Bilde Richtungsvektoren von den Eckpunkten zur lokalen Position des Eingabewertes innerhalb der Gitterzelle
4. Forme das Skalarprodukt zwischen den jeweiligen Gradienten- und Richtungsvektoren
5. Interpoliere zwischen den Skalarprodukten

Das Ergebnis des 5. Schrittes ist der gesuchte pseudozufällige Ausgabewert. Wie die einzelnen Schritte im Detail ablaufen, wird im Folgenden am Beispiel von dreidimensionalem Perlin Noise und dem verbesserten Algorithmus⁵, der von Perlin im Jahr 2002 veröffentlicht worden ist, erklärt.

Um die zur Eingabeposition gehörige Zelle der Gitterstruktur für den ersten Schritt zu finden, werden die Komponenten der Eingabeposition auf die nächstkleineren Integer-Werte heruntergerundet. Die daraus entstehende Integer-Position stellt die Startposition einer

⁵ Improved Noise reference implementation - <https://mrl.cs.nyu.edu/~perlin/noise/>

Gitterzelle dar. Die Gitterstruktur auf der Perlin Noise operiert ist limitiert und wiederholt sich periodisch alle 256 Schritte entlang jeder der drei Hauptachsen. Entsprechend muss die momentan vorhandene Zellposition noch in den richtigen Bereich gebracht werden. Die Komponenten der finalen Zellposition befinden sich also jeweils im Wertebereich [0, 255].

Zur Wahl von Gradientenvektoren im zweiten Schritt wird zunächst ein pseudozufälliger Wert an jedem der acht Eckpunkte benötigt. Hierzu dient einerseits die Zellposition aus dem vorherigen Schritt und andererseits eine Lookup-Tabelle, welche eine gleichverteilte Permutation aller Integer-Werte zwischen 0 und 255 einschließlich der Grenzen enthält. Die drei einzelnen Komponenten der Zellposition werden dabei als Indizes in der Lookup-Tabelle genutzt. Zuerst findet die X-Komponente der Zellposition Anwendung, um einen ersten Wert zu bestimmen. Auf diesen Wert wird die Y-Komponente addiert und der resultierende Wert genutzt, um wieder auf die Elemente der Lookup-Tabelle zuzugreifen. Auf den nun erhaltenen Wert wird als letztes die Z-Komponente addiert. Das Resultat wird anschließend ein letztes Mal verwendet, um noch einmal auf die Tabelle zuzugreifen. Das letztendliche Ergebnis ist ein einziger pseudozufälliger Integer-Wert, welcher sich wieder im Bereich [0, 255] befindet. Damit ist die Pseudozufallszahl an der Startposition der Zelle bekannt. Die beschriebenen Zugriffe auf die Lookup-Tabelle werden mit den restlichen Zelleckpunkten mit entsprechenden Offsets der Startposition wiederholt. Darauffolgend liegen insgesamt acht pseudozufällige Integer-Werte vor.

Wie eventuell auffällt, befindet sich der resultierende Wert nach der Addition der jeweiligen Komponenten nichtmehr im ursprünglichen Wertebereich, sondern im Bereich [0, 512]. Dies würde beim folgenden Tabellenzugriff zu Problemen führen. Entsprechend müsste nach jeder Addition eine Modulo-Rechnung durchgeführt werden, um das Ergebnis wieder in den gültigen Wertebereich zu verschieben. Nachdem dies allerdings nicht gerade effizient ist, wird bei einer effizienteren Implementierung in der Regel Rechenzeit gegen Speicherplatz eingetauscht. Dazu wird die Lookup-Tabelle um eine exakte Kopie der ersten Zahlenpermutation erweitert, welche am Ende der Tabelle angehängt wird. Die Tabelle enthält also in der Praxis 512 Elemente, wodurch der Zugriff nach der Addition der Komponenten immer gültig ist.

Die Wahl der Gradientenvektoren über die berechneten acht Eckpunktewerte, erfolgt in Perlins verbesserten Version über eine weitere Lookup-Tabelle. Diese Tabelle besteht aus zwölf Richtungsvektoren, die von der Mitte der Zelle zu deren Kanten zeigen und die letztendlichen Gradientenvektoren darstellen. Um die Effizienz zu verbessern, fügt Perlin vier weitere Richtungsvektoren in die Tabelle ein, wodurch eine Division durch zwölf verhindert werden kann (Perlin, 2002, S. 2). Diese vier Vektoren formen einen regulären Tetraeder. Das mehrfache Vorkommen der Vektoren in der Tabelle führt daher zu keinem visuellen Bias im Ergebnis (Perlin, 2002, S. 2). Die Tabelle enthält insgesamt also 16 Richtungsvektoren. Die Berechnung des Gradientenvektorindexes jedes Eckpunktes erfolgt am Ende dieses Schrittes über eine bitweise Verundung des entsprechenden pseudozufälligen Eckpunktewertes mit 15.

Das Durchführen des dritten Schrittes, dem bestimmen von Richtungsvektoren ausgehend von den acht Eckpunkten zur lokalen Position der Eingabe innerhalb der Zelle, benötigt logischerweise zuerst die lokale Position. Diese Position kann einfach berechnet werden. Dazu wird wieder die heruntergerundete Integer-Position der Eingabe verwendet. Diese wird von der originalen Eingabeposition subtrahiert. Da die Ergebnisvektoren dieses Schrittes Richtungen sind, kann, je nach Eckpunkt, 1 von einer oder mehreren Komponenten der lokalen Position abgezogen werden, um den Richtungsvektor eines Eckpunktes zu bilden.

Als Basis des vierten Schrittes dienen die Gradientenvektoren aus Schritt 2 und Richtungsvektoren aus Schritt 3. Für jeden Eckpunkt wird zwischen dem entsprechenden Gradienten- und Richtungsvektor lediglich das Skalarprodukt gebildet.

Im letzten Schritt müssen nur noch die acht Skalarprodukte der Eckpunkte miteinander trilinear interpoliert werden. Als Interpolationswerte dienen die Komponenten der lokalen Eingabeposition aus Schritt 3. In der Theorie könnten diese drei Werte direkt für die trilineare Interpolation genutzt werden. Um jedoch einen weicheren Übergang zu erzeugen, wendet Perlin vor der Interpolation eine Fade-Funktion auf die drei Komponenten der lokalen Position an. Das Ergebnis der trilinearen Interpolation ist der gesuchte Noise-Wert an der Eingabeposition. Je nach Anwendungsfall muss das Ergebnis in einen Wert zwischen 0 und 1 transformiert werden. Für die prozedurale Generierung von Terrain ist diese Transformation aber meistens nicht nötig.

Ein kleineres Problem, das bei der Verwendung von Perlin Noise für die prozedurale Terraingenerierung auftreten kann, entsteht wenn die Eingabeposition genau auf dem Zelleckpunkt liegt, die Eingabe also einer Integer-Position entspricht. Dadurch wird der Richtungsvektor des gleichpositionierten Eckpunktes in Schritt 3 ein Nullvektor. In Schritt 4 ist das Skalarprodukt dieses Nullvektors und des Gradientenvektors entsprechend 0. Der Noise-Wert ist für Integer-Positionen also immer derselbe Wert unabhängig von der Eingabeposition. In der Praxis ist dieses Problem jedoch leicht behebbar. Entweder kann die Verwendung von Integer-Positionen vermieden oder alternativ die Position um einen kleinen Offset verschoben werden, damit Perlin Noise auch in diesen Fällen brauchbare Werte liefert.

3 ISOFLÄCHENEXTRAKTION

Über einen Noise-Algorithmus wie Perlin Noise kann nun für eine beliebige Position im Raum ein pseudozufälliger Wert zwischen beispielsweise -1 und 1 ermittelt werden. Wie daraus aber ein Mesh aus Polygonen generiert werden kann ist bisher noch nicht geklärt worden. Für diesen Vorgang gibt es eine Menge an Ansätzen, die grundsätzlich verschieden funktionieren und dementsprechend auch unterschiedliche Ergebnisse liefern. Einige der Ansätze, die für diese Arbeit Bedeutung haben, werden im Folgenden beschrieben.

Ein simpler und oft genutzter Ansatz ist die Interpretation der Noise-Werte als eine Art Höhenwert. Der Wertebereich kann dafür optional in einen Bereich zwischen 0 und 1 transformiert werden. Zusätzlich wird eine maximale Terrainhöhe festgelegt. Denkbar einfach kann der Noise-Wert an einer zweidimensionalen Position dann mit dieser maximalen Terrainhöhe multipliziert werden, um den endgültigen Höhenwert zu erhalten. Daraus kann dann ein Mesh erstellt werden, dessen Vertices an einer kombinierten Position aus den gesampelten zweidimensionalen Positionen und der berechneten Höheninformation liegen. Für viele Spiele kann mit diesem Ansatz bereits eine ausreichende und hochqualitative Landschaft erzeugt werden. Diese Vorgehensweise hat allerdings ein potenziell gravierendes Problem, wodurch die erzeugten Landschaften unnatürlich wirken können. Pro gesampelter zweidimensionaler Position ist eben nur ein einziger Höhenwert vorhanden, entsprechend können durch dieses Verfahren keinerlei Überhänge oder Höhlen generiert werden. Für viele Anwendungen ist das eventuell kein allzu großes Problem, vor allem wenn das generierte Terrain nur das grundlegende Layout der Welt festlegt und weitere Objekte genutzt werden, um Überhänge zu erzeugen. Aufgrund dieser Limitation ist der Heightmap-Ansatz für diese Arbeit jedoch nicht gut genug.

Ein Ansatz, welcher meistens natürlichere Landschaften generieren kann und sowohl Überhänge als auch Höhlen ermöglicht, ist die Extraktion von Isoflächen aus einer Funktion. Die genaue Funktionsart, die benötigt wird, um solche Flächen zu extrahieren, ist eine Skalarfeld-Funktion ϕ , welche jedem Punkt in \mathbb{R}^n , spezifisch in diesem Fall \mathbb{R}^3 , ein Skalar zuordnet (Wenger, 2013, S. 1). Perlin Noise stellt beispielsweise eine solche Funktion dar. Mit der Hilfe einer weiteren Konstante σ kann nun eine Levelmenge $\{x : \phi(x) = \sigma\}$ definiert werden (Wenger, 2013, S. 1). Im dreidimensionalen Fall dieser Arbeit wird diese Levelmenge auch implizite Fläche oder Isofläche genannt (Wenger, 2013, S. 1). Die Konstante σ wird

weitergehend als Isolevel bezeichnet. Die durch das Isolevel definierte Isofläche separiert die von der Funktion generierten Werte in drei Bereiche: Werte die kleiner als, größer als oder gleich dem Isolevel sind, oder im Kontext der Terraingenerierung interpretiert, Werte, die innerhalb, außerhalb oder auf dem Terrain liegen.

Es gibt verschiedene Algorithmen, die eine solche Funktion nutzen, um ein trianguliertes Mesh aus der beschriebenen Isofläche zu approximieren. Jedes dieser Meshes approximiert, je nach dem gewähltem Extraktionsalgorithmus, Aspekte der Isofläche zu unterschiedlichen Graden und besitzt entsprechend auch bestimmte Eigenschaften dieser, wodurch sich das gewählte Verfahren mehr oder weniger gut für die prozedurale Generierung von Terrain eignet. Eine erste Kategorisierung der Algorithmen zur Isoflächenextraktion kann über die Einteilung in sogenannte Primal- und Dual-Methoden erfolgen (Schaefer und Warren, 2004, S. 1-2). Der Unterschied zwischen diesen beiden Kategorisierungen wird im Folgenden anhand der beiden Methoden Marching Cubes und Dual Contouring erklärt. Daraufhin folgt die Beschreibung der beiden zur Implementation gewählten gleichnamigen Algorithmen namens Dual Marching Cubes. Das erste Verfahren ist die Version von Schaefer und Warren und das zweite die Version von Nielson.

3.1 MARCHING CUBES

Marching Cubes, der wahrscheinlich bekannteste Algorithmus zur Isoflächenextraktion, ist 1987 in einer Arbeit von Lorensen und Cline veröffentlicht worden. Der eigentliche Verwendungszeck von Marching Cubes liegt in der Visualisierung von medizinischen Daten, wie sie beispielsweise aus Scans durch Computertomographie oder Magnetresonanztomographie hervorgehen. Auch heutzutage finden diese oder darauf basierende Methoden noch weitverbreitete Anwendung im medizinischen Bereich. Die Nutzung ist jedoch nichtmehr nur auf den medizinischen Kontext begrenzt. Dank der grundsätzlich simplen Natur, effizienten Implementierbarkeit und nicht zuletzt auch der leichten und hohen Parallelisierbarkeit des Verfahrens stößt Marching Cubes vor allem auch in der prozeduralen Terraingenerierung auf große Beliebtheit.

Die generelle Funktionsweise des Algorithmus wird im Folgenden anhand der originalen Arbeit von Lorensen und Cline (1987), und der von Bourke (1994) beschriebenen Implementation des Verfahrens erklärt. Die Funktionsweise des Algorithmus selbst kann nach Lorensen und Cline in zwei Hauptschritte eingeteilt werden (Lorensen & Cline, 1987, S. 2):

1. Berechne und trianguliere die durch die Isofläche beschriebenen Oberfläche.
2. Bestimme die Vertexnormalen aller Vertices des nun vorliegenden Meshes.

Der Name des Verfahrens stemmt von der grundlegenden Funktionsweise des ersten Verfahrensschrittes. Hierbei wird zunächst der Bereich, für den die Isofläche approximiert werden soll, in eine gleichmäßige Gitterstruktur unterteilt. Die daraus entstehenden Zellen des Gitters werden daraufhin separat betrachtet und nach und nach abgearbeitet. In diesem Divide-And-Conquer-Ansatz marschiert ein Würfel also so gesehen über den gesamten zu bearbeitenden Bereich. Für jede Gitterzelle wird bei der Betrachtung in diesem Schritt zunächst die zu approximierende Funktion an jedem der acht Zelleckpunkte ausgewertet. Jeder dieser acht Werte kann darauffolgend, mit der Hilfe des festgelegten Isolevels, als entweder außer- oder innerhalb der Oberfläche klassifiziert werden. Die klassifizierten Werte werden je nach Zuordnung entsprechend durch eine 0 oder 1 repräsentiert. Für jeden Eckpunkt ist also bekannt, ob er sich außer- oder innerhalb des Terrains befindet.

Mit diesem Wissen könnte bereits ein Algorithmus formuliert werden, der die vorhandenen Eckpunktdata nutzt, um dynamisch ein Mesh zu triangulieren. In der Praxis ist das ständige Neuberechnen des Zellmeshes aber weitaus weniger effizient als das Verwenden einer vorberechneten Konfiguration. Da jeder der Eckpunkte über einen Binärwert klassifizierbar ist, kann das Mesh einer Zelle über insgesamt 256 verschiedene

Konfigurationen dargestellt werden. Unter Einbezug verschiedener Symmetrien können diese Konfigurationen auf eine weitaus kleine Zahl an grundlegenden Fällen reduziert werden (Cline & Lorensen, 1987, S. 3). Marching Cubes macht sich in der Regel zwei Lookup-Tabellen zu nutzen, um die Konstruktionsdaten der vorberechneten Zellkonfigurationen zur Verfügung zu stellen und somit das wiederholte Berechnen zu vermeiden. Als Startindex für den Nachschlag der Konstruktionsinformationen werden die Binärklassifikationen der acht Eckpunktswerte genutzt. Jeder dieser Werte wird dazu als einzelnes Bit interpretiert und kombiniert, um daraus einen 8-Bit-Index zu erstellen.

Die Platzierung der Vertices erfolgt bei Marching Cubes immer entlang der Kanten von Gitterzellen. Je Zellkante kann dabei ein einziger Vertex entstehen. Wenn zwei benachbarte Eckpunkte, die durch eine solche Kante verbunden sind, jeweils unterschiedlich klassifizierte Werte besitzen, bedeutet dies, dass entlang der verbindenden Kante die Oberfläche der Isofläche geschnitten wird und diese Kante dementsprechend einen Vertex enthalten muss. Die erste der beiden Lookup-Tabellen enthält Informationen dazu, welche Zellkanten geschnitten werden, und wird daher folglich als Kanten-Tabelle bezeichnet. Die für eine bestimmte Zellkonfiguration relevanten Daten können aus dieser Kanten-Tabelle mithilfe des zuvor beschriebenen Indexes ausgelesen werden. Die Tabelle enthält entsprechend 256 Werte. Jeder dieser Werte stellt einen 12-Bit-Wert dar, bei dem jedes Bit beschreibt, ob eine Kante der Gitterzelle die Isofläche schneidet.

Die Wahl der Vertexpositionen entlang solcher isoflächenschneidender Kanten bestimmt das letztendliche Aussehen des Meshes und kann auf unterschiedliche Arten und Weisen erfolgen. Eine simple Möglichkeit ist es die Vertices einfach auf dem halben Weg zwischen den jeweiligen Kanteneckpunkten zu platzieren. Das daraus entstehende Mesh besitzt ein recht blockhaftes Aussehen und eignet sich daher eventuell nur für das Terrain von Spielen mit einem bestimmten Grafikstil. Ein allgemein glatteres Aussehen kann alternativ mittels der Bestimmung der Vertexpositionen durch eine lineare Interpolation erreicht werden. Dazu sind lediglich die ursprünglichen Werte der Kanteneckpunkte vor der Klassifizierung und das Isolevel nötig. Mit diesen drei Werten kann dann der Schnittpunkt der Isofläche mit der Kante interpoliert werden. Dieser Schnittpunkt ist die genaue Position, an der sich der Vertex entlang der Kante befinden sollte. Das resultierende Mesh approximiert die Isofläche besser, da die Vertices direkt auf oder zumindest nahe der Isofläche liegen. Im Vergleich zu der ersten Variante bringt diese Positionsinterpolation jedoch auch einen großen Nachteil mit sich. Sollten die Vertices wie in der ersten Variante immer entlang der Mitte der Zellkanten gesetzt werden, existiert auch immer ein bestimmter Mindestabstand zwischen jedem Vertex, was zu einem sehr „sauberen“ Mesh führt. Sobald die Vertexposition jedoch durch eine Interpolation berechnet wird, ist dieser Mindestabstand nichtmehr gegeben. In der Praxis führt die zweite Variante daher zu potenziell sehr kleinen und ungünstig geformten Dreiecken, welche ein typischer Bestandteil von vielen Implementierungen von Marching Cubes sind.

Nachdem die Positionen der Vertices nun bekannt sind, müssen diese nur noch zu Dreiecken verbunden werden. Dazu dient die zweite Lookup-Tabelle, welche die benötigten Verbindungsinformationen der Dreiecke enthält. Diese Tabelle wird daher als Dreiecks-Tabelle bezeichnet. Wie die Kanten-Tabelle enthält die Dreiecks-Tabelle Einträge für die 256 Zellkonfigurationen. Allerdings besteht jeder dieser Einträge nicht nur aus einem einzigen Wert, sondern aus mehreren Werten, wobei jeder dieser Werte der Index einer Kante ist und immer die Vertices von drei aufeinanderfolgenden Kanten ein Dreieck des Zellmeshes bilden. Pro Gitterzelle werden hierbei, abhängig von der Variante der Tabelle, maximal vier (Lorensen & Cline, 1987, S. 3) bzw. fünf (Bourke, 1994, S. 1) Dreiecke erzeugt.

Durch die bisher beschriebene Vorgehensweise wird das Mesh einer einzigen Gitterzelle erstellt. Durch die gleichartige Bearbeitung aller weiteren Zellen entsteht das Mesh des betrachteten Bereiches. Um das Mesh auch korrekt beleuchten zu können, fehlt allerdings noch der zweite Hauptschritt von Marching Cubes: das Berechnen der Vertexpnormalen aller Meshvertices. Dazu müsste in der Theorie die Normale an einem bestimmten Punkt der

Isofläche berechnet werden, was bei einer beliebigen Funktion als Grundlage nicht zwingend trivial wäre. Wie Lorensen und Cline beschreiben ist es jedoch glücklicherweise möglich und ausreichend die Richtung des Gradientenvektors als Normale zu interpretieren (Lorensen & Cline, 1987, S. 3). Hierzu muss zunächst der Gradientenvektor an den zu einem Vertex gehörenden Eckpunkten approximiert werden. Dies erfolgt über die separate Approximation der Komponenten des Gradientenvektors. Für jede Komponente wird der Wert der Isoflächenfunktion des vorherigen vom folgenden Eckpunkt entlang der entsprechenden Achse subtrahiert. Nach der Wiederholung dieses Vorgangs für alle drei Komponenten, kann die Normale an einem Eckpunkt über die Normalisierung des Gradientenvektors bestimmt werden. Um die letztendliche Normale eines Vertex zu erhalten, müssen die Normalen der zugehörigen Eckpunkte wieder entlang der verbindenden Kante interpoliert werden.

Wie bereits kurz erwähnt eignet sich Marching Cubes relativ gut für die prozedurale Generierung von Landschaften. Die vergleichsweise simple Funktionsweise in Kombination mit der Nutzung von vorberechneten Tabellen, um an den Großteil der für die Darstellung benötigten Daten zu gelangen, machen diesen Algorithmus im Vergleich zu anderen Extraktionsverfahren sehr effizient. Die separate Betrachtung der einzelnen Zellen und der damit verbundenen hohen Parallelisierbarkeit dieser Methode bieten zudem einen enormen Vorteil im Kontext von Spielen, in denen gute Performanz ein essenzielles Kriterium darstellt. Zusätzlich zu dieser Parallelisierbarkeit beitragend ist die Tatsache, dass keinerlei direkte Abhängigkeiten zwischen den einzelnen Zellen der Gitterstruktur bestehen. Folglich eignet sich Marching Cubes ausgezeichnet für Implementationen, welche sich beispielsweise Threads auf der CPU oder Shader auf der GPU zunutze machen, um die Bearbeitung der einzelnen Zellen parallelisiert vorzunehmen. Ein weiterer Vorteil von Marching Cubes ist zudem, dass die erzeugten Meshes immer mannigfaltig sind (Ju et al., 2007, S. 1).

Marching Cubes ist jedoch bei weitem nicht der perfekte Algorithmus zur Extraktion von Isoflächen. Je nach den Eigenschaften der zu extrahierenden Isofläche bringt dieses Verfahren auch einige mehr oder weniger problematischen Nachteile mit sich. Einer dieser Nachteile des originalen Marching Cubes, ist die Existenz von mehrdeutigen Zellkonfigurationen, welche zu Löchern im letztendlichen Mesh führen können. Dieses Problem wird allerdings von vielen Erweiterungen des Algorithmus behoben, wie zum Beispiel der erweiterten Methode von Nielson und Hamann (Hamann & Nielson, 1991, S. 84). Neben dem bereits erwähnten Problem der kleinen und ungünstig platzierten Dreiecke, führt auch die komplett separate Betrachtung der Gitterzellen zu einem potenziellen Problem. Da keine Abhängigkeiten zwischen benachbarten Zellen bestehen haben die Teilbereiche des generierten Meshes folglich auch keinerlei Abhängigkeit zueinander. Entsprechend sind die Vertices entlang der Zellkanten auch insgesamt bis zu vier Mal im finalen Mesh vorhanden. Für kleinere Landschaften stellen diese zusätzlichen Vertices normalerweise kein allzu großes Problem dar, für größere Welten hingegen kann diese erhöhte Anzahl an mehr oder weniger unnötigen Vertices jedoch schnell außer Kontrolle geraten. Mehrfache Vertices sind allerdings auch in der prozeduralen Generierung von Landschaften nicht grundsätzlich unerwünscht. Obwohl Terrain in den meisten Fällen eher graduelle und glatte Übergänge aufweist, gibt es auch Situationen in denen schärfere Kanten auftreten können. In solchen Situationen werden im Mesh ohnehin mehrere Vertices an derselben Position benötigt, da im Normalfall ein Vertex nur eine einzige Vertexnormal mit sich assoziieren kann, aber mindestens zwei benötigt werden, um eine scharfe Kante darzustellen. Das führt hingegen zu einem weiteren Nachteil von Marching Cubes. Scharfe Kanten in der zu extrahierenden Isofläche werden in vielen Fällen nur schlecht im finalen Mesh approximiert. Dieses Problem stammt von der Tatsache das Marching Cubes zu den primalen Methoden zählt. Primale Methoden extrahieren ein Mesh, indem sie die Schnittpunkte zwischen einer Gitterstruktur und einer Isofläche miteinander verbinden (Ju et al., 2002, Löffler & Schumann, 2012). Im Falle von Marching Cubes ist dies das Setzen der Vertices entlang der Kanten, welche die Verbindungen der Gitterpunkte darstellen.

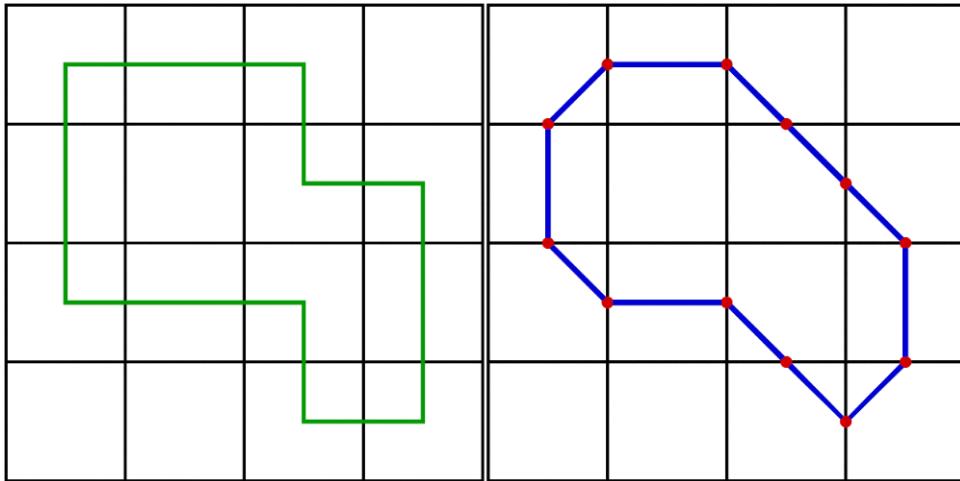


ABBILDUNG 2: ZWEIDIMENSIONALES BEISPIEL FÜR DIE SCHLECHTE EXTRAKTION EINES MESHES DURCH MARCHING CUBES (RECHTS) AUS EINEM OBJEKT MIT SCHARFEN KANTEN (LINKS). WIE ERSICHTLICH WIRD DAS MESH ZWAR GRUNDLEGEND APPROXIMIERT, DIE SCHARFEN KANTEN GEHEN ALLERDINGS VERLOREN.

3.2 DUAL CONTOURING

Seit der Publizierung vom Marching Cubes sind über die Jahre zahlreiche Erweiterungen und Alternativen zum ursprünglichen Verfahren veröffentlicht worden. Viele dieser Methoden und Varianten versuchen verschiedene Probleme des originalen Algorithmus zu beseitigen und allgemein ein Mesh als Endergebnis zu erhalten, welches die Isosfläche besser approximiert. Einer dieser alternativen Methoden ist ein Verfahren namens Dual Contouring von Ju et al. (2002). Im Vergleich zu Marching Cubes zählt Dual Contouring, wie der Name bereits vermuten lässt, zu den dualen Methoden. Die durch duale Methoden erzeugten Flächen sind topologisch gesehen dual zu den Flächen von primären Methoden (Ju et al., 2002, Löffler & Schumann, 2012). In der Praxis bedeutet dies, dass jede Kante einer Gitterzelle keine Vertices mehr enthält, sondern Faces. Anstatt die Vertices entlang der Kanten zu platzieren, befinden sich diese an einem beliebigen Punkt in der Gitterzelle selbst (Ju et al., 2002, S. 2).

Die grundlegende Funktionsweise von Dual Contouring basiert nach Ju et al. auf einer Mischung zweier Verfahren (Ju et al., 2002, S. 3): Extended Marching Cubes von Botsch et al. (2001) und Surface Nets von Gibson (1998). Sie beschreiben zwei Varianten ihrer Methode. Wie auch Marching Cubes nutzt die erste der beiden als Basis eine gleichmäßige Gitterstruktur deren Gitterpunkte gesampelt werden. Die allgemeine Vorgehensweise bei dieser Variante kann in zwei grobe Schritte aufgeteilt werden (Ju et al., 2002, S. 3):

1. Erstelle einen Vertex in jeder Gitterzelle in der die Isosfläche geschnitten wird.
2. Für jede noch nicht behandelte Zellkante an der die Isosfläche geschnitten wird, verbinde die Vertices der Zellen, die sich diese Kante teilen, zu einem Quad bzw. zwei Dreiecken.

Der erste Schritt besitzt zunächst die gleiche Vorgehensweise wie Marching Cubes. Alle Eckpunkte einer momentan bearbeiteten Gitterzelle werden anhand der Funktionswerte klassifiziert. Dadurch wird bestimmt, welche Gitterzellen überhaupt die Isosfläche schneiden und falls dies der Fall ist, welche genauen Kanten daran beteiligt sind. Entsprechend kann der erstellte Vertex mit solchen Kanten assoziiert werden.

Die wirkliche Komplexität von Dual Contouring stemmt von der Positionierung der Vertices innerhalb solcher schneidenden Gitterzellen. Eine simple Möglichkeit wäre es die Vertices

immer in der Mitte der Zellen zu setzen. Wie auch bei Marching Cubes führt ein solch simpler Ansatz jedoch ebenfalls zu einem sehr blockhaften Aussehen und einer nur groben Approximation der Isofläche. Für natürlich aussehendes Terrain eignet sich diese Platzierungsmethode also nicht. Eine eventuell bessere Alternative ist das trilineare Interpolieren der Vertexposition anhand der acht Eckpunktswerte einer Zelle. Das von Ju et al. vorgeschlagene Positionierungsschema ist hingegen um einiges komplexer, liefert aber auch eine bessere Approximation. Sie nutzen eine quadratische Fehlerfunktion zusammen mit Hermite-Daten an Schnittpunkten der Isofläche mit den Zellkanten, an denen eine Normale bekannt ist bzw. berechnet werden kann (Ju et al., 2002, S. 2). Über den Schnittpunkt und die entsprechende Normale kann für jede geschnittene Kante eine Ebene definiert werden. Der letztendliche Vertex innerhalb einer Gitterzelle liegt an der Position, die die quadratische Fehlerfunktion dieser Hermite-Daten minimiert (Ju et al. 2002, S. 2).

Damit kann ein Vertex in jeder geschnittenen Gitterzelle erstellt werden. Im zweiten Schritt müssen all diese Vertices nur noch miteinander zu Dreiecken verbunden werden. Der genaue Ablauf dieses Schrittes kann von Implementation zu Implementation stark variieren. Allgemein gesehen kann jede von der Isofläche geschnittene Zellkante mit den bis zu vier Vertices der anliegenden Zellen assoziiert werden. Eine mögliche Vorgehensweise zum Verbinden der Vertices wäre es daher Referenzen auf Kanten im ersten Schritt zu nutzen. Im zweiten Schritt müssten dann nur alle Kanten durchlaufen und die jeweiligen Vertices zu Dreiecken verbunden werden. Anstelle der Nutzung von Referenzen könnte alternativ nach dem ersten Schritt eine weitere Iteration über alle Gitterzellen durchgeführt werden. Für jede Zelle müssten bei dieser Vorgehensweise alle noch nicht betrachteten Kanten behandelt und die Vertices der entsprechenden Nachbarzellen verbunden werden.

Die bisher beschriebene Version von Dual Contouring macht sich eine uniforme Gitterstruktur zur Triangulierung der Isofläche zu nutzen. Ein Hauptproblem dieser Variante und allgemein der Nutzung uniformer Gitterstrukturen zur Approximation ist, dass ein Großteil der Gitterzellen entweder komplett innerhalb oder außerhalb der Isofläche liegen (Ju et al. 2002, S. 5). Folglich wird ein nicht unerheblicher Anteil der Rechenzeit auf diese leeren Zellen verschwendet. Dazu kommt, dass die Auflösung des erstellten Meshes direkt vom Unterteilungsgrad der Gitterstruktur abhängt. Um also eine höhere Auflösung an sehr detailreichen Stellen der Isofläche, wie beispielsweise Kanten oder kleineren Elementen, zu erhalten, muss das gesamte Gitter feiner unterteilt werden. Damit steigt entsprechend auch die Anzahl an leeren Zellen und vor allem auch die Anzahl an unnötigen Vertices in Regionen mit weitaus weniger Detail, in denen eventuell schon eine Handvoll Vertices ausreichen würden.

Dieses Problem wird von Ju et al. mit der zweiten Variante von Dual Contouring umgangen. In dieser Version wird die uniforme Gitterstruktur mit einem adaptiven Octree ersetzt. Zur Bestimmung des Unterteilungsgrades der einzelnen Octreezellen findet wieder eine quadratische Fehlerfunktion Anwendung. Daraus wird der Fehler einer Octreezelle bestimmt. Mit der Hilfe eines bestimmten Toleranzwertes kann dann eingestuft werden, ob der Fehler groß genug ist, um eine weitere Unterteilung der Zelle zu benötigen (Ju et al., 2002, S. 5). Durch die Verwendung eines Octrees entsteht jedoch das Problem, dass die Nachbarn einer Zelle nichtmehr aus Zellen der gleichen Größe bestehen, sondern aus Zellen mit eventuell mehreren unterschiedlichen Größen. Das Verbinden mit einer Kante assoziierter Vertices ist entsprechend nichtmehr so trivial. Um aus den Blättern des Octrees dennoch ein geschlossenes Mesh zu erzeugen, setzen Ju et al. ein rekursives Verfahren ein, welches den Octree durchläuft. Dieses Verfahren besteht aus drei Funktionen, die in jeder Iteration Informationen für die Polygonisierung sammeln und letztendlich terminieren, sobald der komplette Octree abgearbeitet ist (Ju et al., 2002, S. 6). Die genaue Funktionsweise dieser rekursiven Funktionen hat für diese Arbeit keine direkte Bedeutung und wird in einer sehr ähnlichen Ausführung im Kontext der Methode Dual Marching Cubes von Schaefer und Warren näher beschrieben.

Der große Vorteil der Nutzung eines Octrees in der adaptiven Variante von Dual Contouring, ist die Verteilung der Meshvertices auf Bereiche mit einem höheren Detailgrad. Da der Octree an Stellen, an denen bei einer uniformen Gitterstruktur ein großer Fehler auftreten würde, einfach weiter unterteilt werden kann, sind feine Elemente darstellbar, ohne den gesamten Unterteilungsgrad erhöhen zu müssen. Zudem werden große Bereiche die komplett inner- oder außerhalb der Isofläche liegen im besten Fall durch eine einzige große Octreezelle dargestellt. Insgesamt kann also oft die Anzahl an Vertices im finalen Mesh reduziert oder die Verteilung der Vertices zumindest eher auf wichtige Bereiche fokussiert werden.

Ein allgemeiner Vorteil von Dual Contouring ist die „freie“ Positionierung der Vertices innerhalb einer Zelle. Wie in Abbildung 3 dargestellt führt dies unter anderem dazu, dass in bestimmten Fällen auch scharfe Kanten der Isofläche, welche bei der Nutzung von Marching Cubes schlecht extrahiert werden würden, gut approximiert werden können. Dazu kommt, dass das generierte Mesh im Vergleich zu Marching Cubes in den meisten Fällen besser geformte Dreiecke enthält. Auch mehrfach vorhandene Vertices können implementationsabhängig schon während der Erstellung vermieden werden, da sich benachbarte Zellen direkt keine Vertices teilen. Vor allem für die Terraingenerierung von größeren Welten ist dies ein riesiger Vorteil, da keine eventuell aufwändige Nachbearbeitung des Meshes erforderlich ist, um unnötige und mehrfach vorhandene Vertices zu entfernen.

Dual Contouring ist allerdings keine absolute Verbesserung über Marching Cubes. Zum einen kann das Auswerten der quadratischen Fehlerfunktion vergleichsweise aufwändig ausfallen. Zum anderen ist der Algorithmus selbst aufgrund der Abhängigkeit zwischen benachbarten Zellen nicht ganz so leicht parallelisierbar wie es bei Marching Cubes der Fall ist. Zusätzlich garantiert das von Dual Contouring erzeugte Mesh keine Mannigfaltigkeit (Ju et al., 2007, S. 1), was zu eventuellen Mesh- und Beleuchtungsproblemen der Landschaft führen kann. Neben der nicht garantierten Mannigfaltigkeit können in bestimmten Fällen auch Selbstüberschneidungen im letztendlichen Mesh auftreten (Ju & Udeshi, 2006, S. 1), was zu weiteren Problemen führen kann.

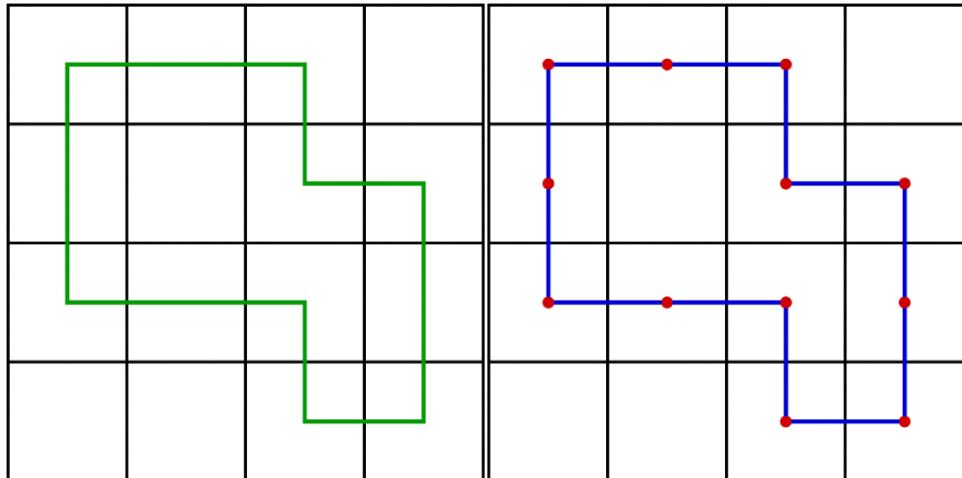


ABBILDUNG 3: ZWEIDIMENSIONALES BEISPIEL FÜR DIE EXTRAKTION EINES MESHES DURCH DUAL CONTOURING (RECHTS) AUS EINEM OBJEKT MIT SCHARFEN KANTEN (LINKS), IN WELCHEM DIE KANTEN DURCH DIE PLATZIERUNG DER VERTICES INNERHALB DER ZELLEN BESSER APPROXIMIERT WERDEN ALS ES BEI MARCHING CUBES DER FALL WÄRE.

3.3 DUAL MARCHING CUBES VON SCHAEFER UND WARREN

Im idealen Fall wäre eine Methode gewünscht, welche die positiven Eigenschaften von Marching Cubes und Dual Contouring kombiniert, um das Beste aus beiden Verfahren zu erhalten. Ein Verfahren, das versucht dies zu erreichen und in vielen Bereichen eine Verbesserung beider Methoden darstellt ist das Verfahren Dual Marching Cubes von Schaefer und Warren (2004). Wie der Name dieses Verfahrens schon vermuten lässt, wird der Marching Cubes Algorithmus genutzt, um das letztendliche Mesh zu generieren. Als Basis für diese Generierung dient allerdings nicht die normalerweise verwendete primale Gitterstruktur. Stattdessen macht sich dieses Verfahren eine Struktur zu nutzen, die topologisch gesehen dual zu dieser herkömmlichen Gitterstruktur ist (Schaefer & Warren, 2004, S. 2). Auch sind die Zellen dieser dualen Gitterstruktur nicht uniform verteilt. Wie bei der adaptiven Version von Dual Contouring findet auch in diesem Verfahren ein Octree mit Blättern unterschiedlicher Unterteilungsstufen Anwendung. Wie Schaefer und Warren erwähnen bringt diese Vorgehensweise einige Vorteile mit sich (Schaefer & Warren, 2004, S. 2): Zum einen ermöglicht die Nutzung einer dualen Gitterstruktur das Platzieren von Vertices an Featuren der zu approximierenden Funktion, wodurch es möglich ist bei der Verwendung von Marching Cubes auch scharfe Kanten der Isofläche darzustellen. Zum anderen ermöglicht die Octree-basierte Unterteilung des Gitters, genauso wie bei Dual Contouring, das fokussierte Platzieren der Meshvertices an Stellen der Isofläche die detailreicher sind und daher eine feinere Unterteilung benötigen. Die Nutzung von Marching Cubes, um die duale Gitterstruktur in ein Mesh zu konvertieren, sorgt außerdem dafür, dass das erzeugte Mesh immer mannigfaltig ist und keinerlei Löcher aufweist (Schaefer & Warren, 2004, S. 2).

Schaefer und Warren teilen das Erstellen des dualen Gitters in zwei Schritte ein (Schaefer & Warren, 2004, S. 2). Der erste Schritt ist *Feature Isolation*. In diesem werden die Vertices des dualen Gitters erstellt. Dazu wird als erstes ein Octree benötigt. Der Octree wird startend mit einer einzigen Zelle immer feiner unterteilt. Der Unterteilungsgrad der einzelnen Blätter wird, wie bei Dual Contouring, über eine quadratische Fehlerfunktion bestimmt. Für jede Octreezelle wird die zu approximierende Funktion an bestimmten Punkten gesampelt (zum Beispiel an den Eckpunkten der Zelle). Mit den gesampelten Punkten wird dann eine quadratische Fehlerfunktion aufgestellt und minimiert. Durch diese Fehlerfunktion kann nicht nur die Position des Vertex dieser Zelle bestimmt werden, sondern auch der Fehler, der im Zusammenhang mit diesem Vertex entsteht. Anhand dieses Fehlers und einem bestimmten Toleranzwert, der den maximal erlaubten Fehler darstellt, kann nun wie bei Dual Contouring ebenfalls entschieden werden, ob die Zelle eine weitere Unterteilung benötigt oder nicht. Nach dem Durchlauf dieses Verfahrens entsteht ein Octree dessen Blätter alle einen Fehler kleiner als der gegebene Toleranzwert aufweisen und jeweils einen Vertex des dualen Gitters enthalten.

Damit ist die Grundlage des dualen Gitters gegeben. Die nächste Herausforderung liegt darin die entstandenen Vertices miteinander zu einem dualen Gitter zu verbinden. Dazu dient der zweite Schritt des Verfahrens: *Topology Creation*. Für jeden Zelleckpunkt innerhalb des Octrees entsteht dabei eine Zelle in der dualen Gitterstruktur dessen Eckpunkte aus einigen der zuvor berechneten Vertices bestehen (Schaefer & Warren, 2004, S. 4). Um die Zellen der dualen Struktur zu bestimmen, durchläuft das Verfahren von Schaefer und Warren den Octree. Sie erweitern dazu den Traversal-Algorithmus, der bei Dual Contouring Anwendung findet und beschreiben das Ergebnis der einfachheitshalber im Kontext eines Quadtrees (Schaefer & Warren, 2004, S. 4). Um die Beschreibung der allgemeinen Funktionsweise simpel zu halten, wird auch hier ihre Beschreibung anhand eines Quadtree genutzt. Insgesamt werden für den Traversal-Algorithmus drei Funktionen genutzt, die den Quadtree über rekursive Aufrufe durchlaufen. Die drei Funktionen sind *faceProc*, *edgeProc* und *vertProc* (Schaefer & Warren, 2004, S. 4). Jede der drei Funktionen bekommt eine bis vier Zellen des Quadtree als Parameter übergeben, welche innerhalb der Funktionen unterschiedlich weiter behandelt werden. *faceProc* bekommt als Eingabe eine einzige Zelle

und ruft sich selbst rekursiv auf die vier Kinder dieser Zelle auf. Anschließend ruft *faceProc* zum einen *edgeProc* auf die vier Paare von Kinderzellen, die sich jeweils eine innere Kante der Eingabezelle teilen, und zum anderen auch noch einmal *vertProc* auf alle Kinderzellen auf, da sie sich den inneren Vertex teilen. *edgeProc* erhält jeweils zwei Zellen als Parameter. Da sich diese beiden Zellen eine Kante teilen, teilen sich auch die Kinder dieser Zellen an zwei Stellen eine Kante. *edgeProc* ruft auf diese beiden Paare von Kinderzellen wieder *edgeProc*. Zusätzlich wird *vertProc* auf die jeweils zwei Kinder der Eingabezellen gerufen, die sich insgesamt einen Vertex teilen. Die Eingabe für die *vertProc*-Funktion sind vier Zellen, die sich einen Vertex teilen. Die Kinder dieser Zellen enthalten daher ebenfalls vier Kinder, die sich diesen inneren Vertex teilen. Auf diese vier Kinder wird daher wieder rekursiv *vertProc* gerufen. Da der Baum in den meisten Fällen keine gleiche Unterteilungsstufe an allen Stellen besitzt, kann es vorkommen, dass eine der Eingabezellen für eine dieser Funktionen bereits ein Blatt ist, während der Rest noch weitere Kinderzellen enthält. In diesem Fall wird anstelle der nichtvorhandenen Kinder dieser Blattzelle die Zelle selbst an die rekursiven Aufrufe übergeben. Das Abbruchkriterium der rekursiven Funktionen ist erreicht, sobald alle Eingabezellen Blätter sind. Sobald dieses Kriterium für vier benachbarte Zellen erreicht ist, enthält der letzte Aufruf von *vertProc* auf diese vier Zellen alle Informationen, um daraus eine Zelle der dualen Gitterstruktur zu bilden.

Schaefer und Warren beschreiben zwei potenzielle Probleme, die bei diesem Verfahren auftreten können. Einerseits kann es vorkommen, dass *vertProc* keine vier unterschiedlichen Zellen übergeben bekommt, sondern nur drei, was in der Zelle des dualen Gitters letztendlich geometrisch gesehen zu einem Dreieck führt anstatt eines Vierecks (Schaefer & Warren, 2004, S. 4). Andererseits kommt es durch die Platzierung der dualen Gittervertices innerhalb der Octreezellen dazu, dass die Vertices des Gitterrandes mit hoher Wahrscheinlichkeit nicht den Rand des Baumes abdecken (Schaefer & Warren, 2004, S. 4). Während das letztere Problem von Schaefer und Warren gelöst wird, indem sie den Baum als Mittelpunkt eines erweiterten Gitters betrachten und über weitere Aufrufe von *edgeProc* und *vertProc* die Zellen des dualen Gitters entlang der Baumkanten generieren (Schaefer & Warren, 2004, S. 4), stellt ersteres für die letztendliche Triangulierung kein Problem dar. Die Verwendung der zuvor beschriebenen Marching Cubes Methode basiert eigentlich auf einer Reihe von Zellen bzw. Würfeln, die entlang der Achsen ausgerichtet sind. Die Achsenausrichtung der Zellen ist jedoch nicht zwingend erforderlich und wie Schaefer und Warren erkennen stellen die Zellen des dualen Gitters, trotz potenzieller Überlappung einiger Vertices, topologisch gesehen immer noch einen Würfel dar (Schaefer & Warren, 2004, S. 5). Dementsprechend kann der Marching Cubes Algorithmus ohne Probleme auf das vorhandene duale Gitter angewandt werden, um das finale Mesh zu erhalten.

Neben den bereits genannten Vorteilen dieses Verfahrens, existieren auch bei dieser Methode einige Nachteile. Einer der großen Nachteile ist die Realtime-Performanz dieses Algorithmus. Während das Erstellen des letztendlichen Meshes auf der Basis der dualen Gitterstruktur genauso performant ist wie auf Basis des herkömmlichen primalen Gitters, kommt der Großteil des Performanzverlustes von der Konstruktion der grundlegenden dualen Gitterstruktur innerhalb des Octrees (Schaefer & Warren, 2004, S. 6). Durch das freie Platzieren der Vertices des dualen Gitters innerhalb der Octreezellen kann es ebenfalls wie bei Dual Contouring zu Selbstüberschneidungen im finalen Mesh kommen (Denis et al., 2012, S. 2). Nachdem Marching Cubes für die Konstruktion des Meshes genutzt wird, wirken sich einige Nachteile dieses Verfahrens auch auf das letztendliche Mesh von Dual Marching Cubes aus. Dazu zählen vorwiegend die hohe Anzahl an mehrfach vorhandenen und oft unnötigen Vertices im finalen Mesh aber auch die Marching Cubes typischen oft schlecht geformten Dreiecke. Um das letztere Problem zu lösen, beschreiben Schaefer und Warren eine mögliche Vorgehensweise, um das Generieren eines Großteils dieser Dreiecke schon bei der Erstellung der dualen Gitterstruktur zu verhindern (Schaefer & Warren, 2004, S. 5). Diese Vorgehensweise findet in dieser Arbeit allerdings keine direkte Anwendung und wird daher nicht näher beschrieben.

3.4 DUAL MARCHING CUBES VON NIELSON

Das letzte Verfahren zur Extraktion von Isoflächen, welches im Zusammenhang mit dieser Arbeit eine direkte Relevanz hat und auch in der letztendlichen Implementation verwendet wird, ist ein Verfahren von Nielson (2004) welches ebenfalls als Dual Marching Cubes bezeichnet wird. Trotz des gleichen Namens nutzt das Verfahren von Nielson eine größtenteils andere Vorgehensweise, um das finale Mesh aus einer Isofläche zu extrahieren. Nielsons Methode nutzt weder einen Octree, um die zugrundeliegende Gitterstruktur zu erstellen, noch wird überhaupt direkt eine duale Gitterstruktur genutzt. Da kein Octree genutzt wird ist das Verfahren entsprechend nicht adaptiv bezüglich des benötigten Detailgrades. Nielson nutzt auch kein duales Gitter für die Extraktion, sondern ein primales Gitter wie im ursprünglichen Marching Cubes Algorithmus. Der Begriff Dual im Namen der Methode stammt bei Nielsons Verfahren von der Nutzung einer modifizierten Version von Marching Cubes. In dieser modifizierten Version ist das schlussendlich entstehende Mesh selbst dual zu dem normalerweise durch Marching Cubes entstehenden Mesh (Nielson, 2004, S. 1). Während die Methode von Schaefer und Warren also, vereinfacht gesehen, Marching Cubes auf eine duale Gitterstruktur anwendet, wendet Nielson in seiner Methode eine duale Version von Marching Cubes auf eine primale Gitterstruktur an. Die Funktionsweise von Nielsons Dual Marching Cubes ist dementsprechend nahezu gleich der Funktionsweise von Marching Cubes. Nielson gibt daher in seiner Arbeit eine eher theoretische Definition seiner Methode und der Flächen, die durch diese entstehen.

Zunächst betrachtet Nielson zusammenhängende Dreiecke der originalen Zellkonfiguration von Marching Cubes nicht als einzelne Dreiecke, sondern als ein einziges Polygon bzw. als einen Patch (Nielson, 2004, S. 1). Die Vertices eines solchen Patches liegen dabei weiterhin auf den Kanten der Zelle. Die Kanten des Patches liegen hingegen immer innerhalb der Zellwände. Das durch Nielsons Dual Marching Cubes erzeugte Mesh ist dual zu dieser Patch-Version des Meshes von Marching Cubes und besteht aus einer Reihe an Quads, welche entsprechend jeweils über vier Vertices definiert werden (Nielson, 2004, S. 1).

Um die Eigenschaften seiner Methode besser beschreiben zu können, bringt Nielson zusätzlich eine bereits erwähnte Eigenschaft des originalen Marching Cube Verfahrens näher: Jeder von Marching Cubes erzeugte Meshvertex liegt immer auf einer der Kanten einer Gitterzelle. Nachdem sich jeweils bis zu vier Zellen eine Kante teilen, folgert Nielson daraus die Erkenntnis, dass sich ein Vertex auch von bis zu vier Zellen entlang ihrer gemeinsamen Kante geteilt wird (Nielson, 2004, S. 2). Mit diesem Wissen beschreibt Nielson die Eigenschaften der dualen Oberfläche seiner Methode im Vergleich zu Marching Cubes (Nielson, 2004, S. 3):

1. Für jeden Patch einer Zellkonfiguration von Marching Cubes entsteht ein Vertex der dualen Oberfläche.
2. Für jeden Vertex entlang einer Kante in Marching Cubes, entsteht ein Quad-Patch der dualen Oberfläche. Die Vertices dieses Patches sind die aus 1. entstehenden Vertices, dessen primale Patches die zugehörige Kante gemeinsam haben.
3. Die Zellwände, die eine Kante eines primalen Patches enthalten, werden von einer der Kanten eines dualen Quad-Patches geschnitten. In anderen Worten, jede Zellseite, die eine Kante in Marching Cubes enthalten würde, wird von einer Kante eines Patches von Dual Marching Cubes geschnitten.

Ein erster Vorteil dieses Verfahrens ist, dass die Konstruktion der dualen Flächen nicht dynamisch berechnet werden muss (Nielson, 2004, S. 3). Wie im originalen Marching Cubes Algorithmus gibt es auch in dieser Methode 256 Konfigurationen die vorberechnet und in Tabellen gespeichert werden können. Das aus diesem Verfahren entstehende Mesh ist sehr ähnlich zu einem Mesh wie es durch Dual Contouring entstehen würde. Die Vertices werden in beiden Verfahren nichtmehr entlang der Kanten von Gitterzellen platziert, sondern „frei“ innerhalb der Zelle selbst. Nielsons Dual Marching Cubes beschränkt die Anzahl der Vertices innerhalb einer Zelle jedoch nicht auf einen einzigen Vertex. Stattdessen können bis zu vier

Vertices innerhalb einer Zelle generiert werden. Im Vergleich zu Dual Contouring wird dadurch sichergestellt, dass das entstehende Mesh immer mannigfaltig ist (Nielson, 2004, S. 7).

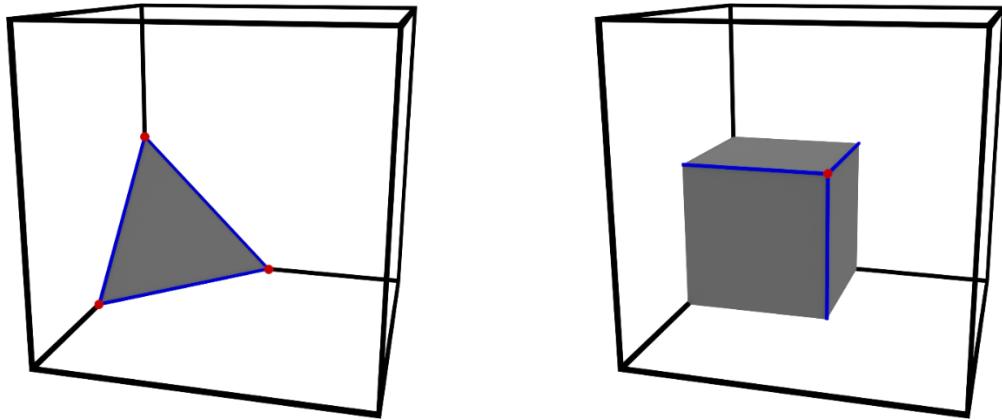


ABBILDUNG 4: BEISPIEL FÜR DEN UNTERSCHIED ZWISCHEN EINER ZELLKONFIGURATION VON MARCHING CUBES UND NIELSONS DUAL MARCHING CUBES.

Wie die Vertices innerhalb der Zellen platziert werden sollten, wird von Nielson nicht direkt festgelegt. Verschiedene Platzierungsmethoden führen hierbei natürlicherweise zu unterschiedlichen Ergebnissen. Für die Beispiele in seiner Arbeit verwendet Nielson den geometrischen Schwerpunkt eines zu einem dualen Vertex zugehörigen primalen Patches. Er beschreibt aber auch, dass die Nutzung von weiteren Daten, wie zum Beispiel die Hermite-Daten bei Dual Contouring, interessante Ergebnisse liefern könnten (Nielson, 2004, S. 8).

Wie erwähnt ist die Funktionsweise dieser Methode grundlegend sehr ähnlich zu der Funktionsweise von Marching Cubes. Zudem kommt, dass durch den theoretischen Fokus von Nielsons Arbeit, die genutzten Tabellen und deren Nutzung sehr implementationsabhängig sind. Daher wird die genauere Vorgehensweise für dieses Verfahren erst zusammen mit dessen Implementation in Kapitel 7.2 beschrieben.

4 TERRAINDEFINIERUNG ÜBER EINE FUNKTION

Über eine der im vorherigen Kapitel beschriebenen Methoden kann nun ein Mesh aus einer Funktion extrahiert werden, welches die durch die Funktion gegebene Isofläche approximiert. Das Aussehen und die Eigenschaften dieser Isoflächenfunktion müssen allerdings noch festgelegt werden. Dieses Unterkapitel beschäftigt sich mit genau diesen Aspekten. Der Fokus liegt hierbei vorwiegend auf theoretischen Grundlagen und Ideen, die in einer solchen Funktion genutzt werden können, um eine interessante Landschaft zu generieren. Die konkreten Details der in dieser Arbeit verwendeten Isoflächenfunktion, weitergehend passender Terrainfunktion genannt, werden erst in Kapitel 7.6 beschrieben.

Die Eingabeparameter einer Terrainfunktion können je nach den gewünschten Terraineigenschaften und Anwendungsfall variieren. Da jedoch in den erwähnten Algorithmen zur Isoflächenextraktion zumindest eine Position an jedem Punkt, an dem die Funktion evaluiert werden muss, zur Verfügung steht, liegt es nahe, diese Position als Basis zu nutzen. Natürlich können aber grundsätzlich auch andere oder weitere Parameter genutzt werden. Aufgrund dieser Eingabe muss die Terrainfunktion einen Wert liefern, der mithilfe eines Isolevels als innerhalb oder außerhalb des Terrains klassifiziert werden kann. Die Wahl des Isolevels selbst ist hierbei stark abhängig von den Gegebenheiten der Funktion. Da Noise-Funktionen ein grundlegendes Element der meisten Terrainfunktionen darstellen und

der Wertebereich von Noise oft zwischen -1 und 1 bzw. 0 und 1 liegt (und damit leicht auf [-1, 1] transformiert werden kann), eignet sich die Wahl von 0 als Isolevel. Damit können alle negativen Werte als außerhalb des Terrains klassifiziert werden und alle positiven Werte als innerhalb.

Für den Rest dieses Kapitels wird der Einfachheit halber Gradient-Noise, wie das zuvor beschriebene Perlin Noise, lediglich als Noise bezeichnet. In der Theorie kann in der Funktion zur Terraingenerierung aber auch ein beliebiger anderer Noise-Algorithmus Anwendung finden, solange dieser eine Form von graduellen Übergängen aufweist. Wie die Arbeit von Olsen (2004) zeigt, kann vor allem durch die Kombination mehrerer verschiedener Noise-Algorithmen eine sehr abwechslungsreiche und natürlich wirkende Landschaft generiert werden.

Da eine Position als Eingabe zur Verfügung steht, wäre ein eventueller erster Ansatz zur Definierung einer Isofläche das Nutzen des direkten Ergebnisses einer Noise-Funktion auf diese Eingabe. Dieser Ansatz funktioniert zwar bei Heightmap-basierten Generierungsvarianten, da es jedoch keinerlei direkte höhenabhängige Gewichtung der Werte gibt, kann Terrain in jeder Richtung generiert werden. Wie in Abbildung 6 dargestellt ist das Ergebnis dieser Vorgehensweise daher zwar interessant, in den meisten Fällen aber ungeeignet. Noise allein reicht normalerweise also nicht aus, um eine brauchbare und natürlich wirkende Landschaft zu formen.

Die Landschaft einer natürlich aussehenden Welt verfügt in der Regel über eine Oberfläche, die das Terrain in Erde und Luft separiert. Die Terrainfunktion muss also eine Oberfläche definieren, die diese Trennung beschreibt. Um die höhenunabhängige Bildung von Polygonen des ersten Ansatzes zu verhindern, muss es schwerer werden das Vorzeichen und damit die Klassifikation eines Wertes an einer Position zu ändern, umso weiter diese Position von der Oberfläche entfernt ist. Genau dieser Ansatz wird von Geiss beschrieben (Geiss, 2007, Kapitel 1.3.3). Als Basis dient in diesem Ansatz der Y-Wert der Position in der folgenden Form:

$$\text{Value} = -\text{Position.y} + \text{Offset}$$

Dadurch wird die gerade genannte Oberfläche beschrieben. Ist der Offset auf 0 gesetzt, werden alle Werte kleiner als 0 (dem Isolevel) positiv und damit Teil des Terrains und alle Werte größer als 0 negativ und liegen damit außerhalb des Terrains. Zusätzlich wachsen die Werte mit zunehmender Entfernung von der Oberfläche. Dementsprechend wird eine stärkere Änderung benötigt, um die Klassifizierung zu ändern. Über die Anpassung des Offset kann zudem die grundlegende Oberfläche verschoben werden, ohne das Isolevel selbst verändern zu müssen. Damit entsteht eine ebene Oberfläche. Um nun Berge und Täler entstehen zu lassen, kann Noise eingebbracht werden. Dazu muss zusätzlich lediglich das Noise an einer bestimmten Position evaluiert werden (Geiss, 2007, Kapitel 1.3.3):

$$\text{Value} += \text{Noise}(\text{Position} * \text{Frequency}) * \text{Amplitude}$$

Je nach Amplitude wird damit auf den bisherigen Terrainwert ein mehr oder weniger großer Noise-Wert addiert oder von diesem subtrahiert. Liegt der Wert zuvor also nahe genug am Isolevel kann sich dadurch dessen Klassifikation ändern. Über die Skalierung der Position mittels des Frequenzwertes kann zudem eingestellt werden wie stark sich die Noise-Werte in Abhängigkeit der Position ändern. In obiger Form führt eine hohe Frequenz zu einer starken Änderung der Noise-Werte und eine niedrige Frequenz zu einer leichten Änderung.

Das Hinzufügen einer einzigen Oktave von Noise führt bereits zu einer natürlicher aussehenden Landschaft. Allerdings ist das entstehende Terrain immer noch sehr glatt und enthält keinerlei Detail. Um mehr Detail einzubringen, kann Noise in Kombination mit *Fractional Brownian Motion* genutzt werden (Bakaoukas & Rose, 2016, Geiss, 2007). Dafür werden mehrere Oktaven an Noise miteinander summiert. Jede weitere Oktave besitzt dabei eine höhere Frequenz und niedrigere Amplitude als die vorherige. Jede Oktave hat also eine

stärkere positionsabhängige Änderung des Noise-Wertes, liefert insgesamt aber einen kleineren Beitrag am Gesamtwert. Umso mehr Oktaven genutzt werden, desto detaillericher wird das entstehende Noise. Das durch Fractional Brownian Motion entstehende Noise weist zusätzlich Selbstähnlichkeit auf und ist damit ähnlich zu Fraktalen (Bakaoukas & Rose, 2016, S. 1). Mittels Oktaven kann zunächst also das generelle Layout einer natürlichen Landschaft, in der Form von groben Bergen und Tälern, über eine erste Oktave mit einer sehr niedrigen Frequenz und relativ hohen Amplitude generiert werden. Über weitere Oktaven kann dann progressiv mehr Detail und Varianz, in Form von beispielsweise Ausbuchtungen und anderen Unebenheiten, hinzugefügt werden.

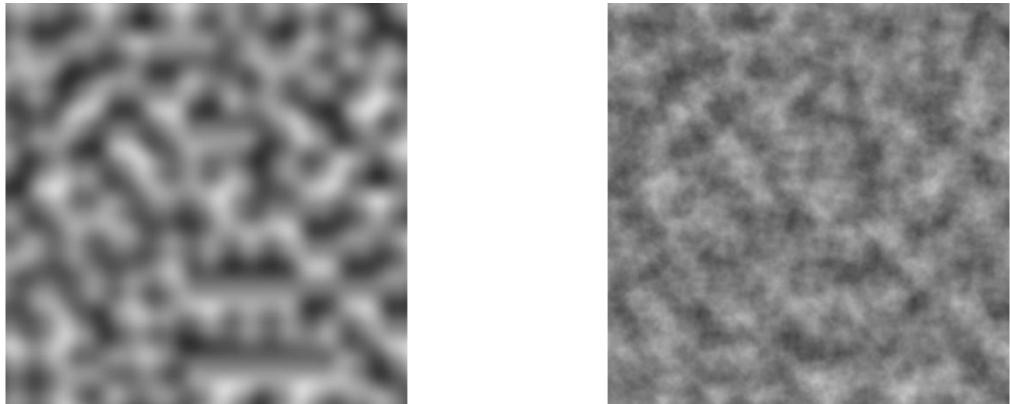


ABBILDUNG 5: VERGLEICH ZWISCHEN DEN WERTEN EINER OKTAVE (LINKS) UND VIER OKTAVEN (RECHTS) AN PERLIN NOISE.

Sowohl die Wahl der initialen Frequenz- und Amplitudenwerte als auch deren Änderung mit jeder Oktave ist abhängig von den gewünschten Eigenschaften des entstehenden Terrains. Ein erster Ansatz, der in den meisten Fällen gute Ergebnisse liefert, ist das Verdoppeln der Frequenz und Halbieren der Amplitude mit jeder weiteren Oktave. Wie Geiss erwähnt ist jedoch ein Problem, welches hierbei auftreten kann, dass sich die Landschaft bei einer genauen Verdopplung der Frequenzen periodisch wiederholt. Dieses Problem entsteht aus der periodischen Wiederholung der Werte von beispielsweise Perlin Noise. Um dieses Problem zu umgehen, sollten die Frequenzen in jedem Schritt nur grob verdoppelt werden (Geiss, 2007, Kapitel 1.3.3). Dadurch überschneidet sich die periodische Wiederholung der einzelnen Oktaven nicht mehr (oder zumindest weniger häufig und mit weniger Oktaven auf einmal). Um weitere Variationen einzubringen, kann die Position vor der Nutzung in den einzelnen Oktaven angepasst werden. Geiss beschreibt zum Beispiel eine leichte Rotation der Position für die niedrigsten Oktaven, um die Wiederholung der dominantesten Features der Landschaft zu verhindern (Geiss, 2007, Kapitel 1.3.3).

Da die Anzahl an genutzten Oktaven ausschlaggebend für den Detailgrad des Terrains ist, wäre es in der Theorie von Vorteil viele Oktaven zu nutzen, um eine möglichst detaillierte Landschaft zu erstellen. Praktisch gesehen ist dies allerdings mit einigen Problemen verbunden. Eines der Hauptprobleme ist die Performanz. Die Noise-Funktion muss für jede Oktave erneut ausgewertet werden. Entsprechend steigt auch der Evaluierungsaufwand der Terrainfunktion. Da die Terrainfunktion an einer potenziell extrem großen Menge an Positionen ausgewertet werden muss, sollte die Anzahl der Oktaven also nicht zu hoch gewählt werden, um das Auswerten der Funktion nicht zu aufwändig zu gestalten. Zudem kommt dazu einerseits, dass der Beitrag, den eine Oktave liefert, aufgrund der sinkenden Amplitude, mit jeder weiteren Oktave abnimmt und andererseits, dass je nach Wahl des Extraktionsverfahrens sehr detaillierte Stellen, aufgrund des Abstandes zwischen Samplepunkten, erst gar nicht extrahiert werden können und entsprechend spätere Oktaven, wenn überhaupt, nur wenig zum Endergebnis beitragen.

In der Praxis kann bereits über eine Handvoll Oktaven eine sehr detaillierte Landschaft erzeugt werden. Weiterhin wird über die Anpassung der initialen Werte der Frequenz und Amplitude bereits eine Reihe an Effekten ermöglicht. Um beispielsweise eine flachere Landschaft zu erzeugen kann die initiale Amplitude verringert werden. Die ersten Oktaven erzeugen dann keine Berge, sondern nur noch leichte Hügel. Zusätzlich sind durch eine Variation der Frequenz die Steigung und die Ausmaße dieser Hügel anpassbar.

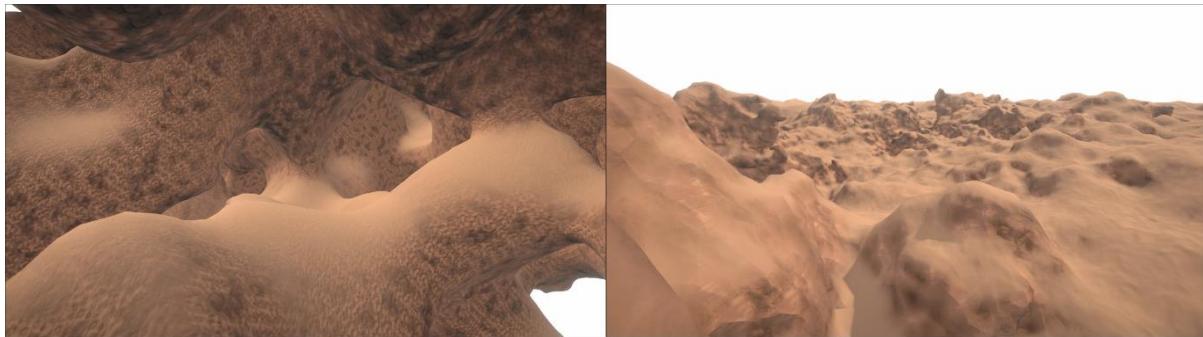


ABBILDUNG 6: VERGLEICH ZWISCHEN EINEM TERRAIN, WELCHES AUF NUR EINER OKTAVE AN NOISE BASIERT (LINKS), UND EINEM TERRAIN, WELCHES HÖHENABHÄNGIG MEHRERE OKTAVEN AN NOISE KOMBINIERT (RECHTS).

Mittels der bisher beschriebenen Vorgehensweisen kann eine Terrainfunktion zwar generell natürlich wirkende Landschaften erzeugen, manche Terraineigenschaften sind damit jedoch schwer und nur selten zuverlässig erzeugbar. Höhlensysteme sind ein gutes Beispiel dafür. Durch die Oktaven entstehen gelegentlich Einbuchtungen in die Erde, wirkliche Höhlensysteme können aber allein aufgrund einer höheren Distanz zur grundlegenden Oberfläche nicht direkt generiert werden. In Spielen ist es zudem zusätzlich oft nötig Terrain mit sehr spezifischen und vor allem garantierten Eigenschaften zu erzeugen. Am Beispiel der Höhlen könnte sich dies in der Form eines Pfades im Terrain zwischen zwei festgelegten Punkten äußern, um sicherzustellen das der Eingang einer unterirdischen Struktur garantiert mit einer Höhle verbunden ist. Ein Ansatz, um Noise-basierend solche Höhlensysteme zu generieren, könnte eine Methode wie Perlin Worms⁶ nutzen. Eine Generation über L-Systeme wie in der Arbeit von Berechet et al. (2015) ist ebenfalls denkbar. Die Generation der meisten solcher anwendungfallabhängigen Eigenschaften ist sehr spezifisch. Entsprechend ist die Art und Weise wie diese Eigenschaften am besten erzielt werden abhängig von den Anforderungen der letztendlichen Anwendung und werden daher in dieser Arbeit im Allgemeinen nicht näher beschrieben.

Die bisherige Terrainfunktion kann mittels weiterer mathematischer Operationen um eine Vielzahl an Effekten erweitert werden. Eine Reihe solcher Effekte werden ebenfalls von Geiss beschrieben (Geiss, 2007, Kapitel 1.3.3). Dazu zählt unter anderem ein canyonähnlicher Terrasseneffekt und das Verzerren der Position über weitere Noise-Werte, um eine surreale Landschaft zu erzeugen. Zwei Effekte die Geiss beschreibt, welche insbesondere für Spiele sehr interessant sein könnten, sind das Generieren eines Planeten anstatt einer endlosen „flachen“ Welt und das Einbringen manueller Einflüsse. Für den erstenen Effekt wird die grundlegende Oberfläche durch die folgende Formel ersetzt:

$$\text{Value} = \text{Radius} - \text{Length}(\text{Position} - \text{Center})$$

Center ist hierbei die Position des Mittelpunktes des Planeten und *Radius* der Planetenradius. Auch hier wird es schwerer die Klassifikation mit einer zunehmenden Abweichung vom Radius zu ändern, aber eben sphärisch. Wird der Radius relativ groß

⁶ libnoise: Perlin worms - <http://libnoise.sourceforge.net/examples/worms/>

gewählt könnten damit beispielsweise nicht endlose, aber dennoch sehr umfangreiche Welten oder eventuell Planetensysteme in einem Sci-Fi-Spiel generiert werden. Die Platzierung der Planeten selbst könnte wiederum über weiteres Noise geschehen. Manuelle Einflüsse könnten sich in der Form einer von Hand erstellten Textur oder Funktion äußern, welche das Aussehen einer bestimmten Landmarke in der Welt darstellt und dynamisch in der prozedural generierten Landschaft platziert wird (Geiss, 2007, Kapitel 1.3.4). Über eine solche Textur könnten auch manuell festgelegte Biome innerhalb der Landschaft generiert werden. Jedes Biom kann dabei ein eigenes Set an Regeln besitzen, wodurch unterschiedliche Terraineigenschaften der entsprechenden Bereiche angepasst werden könnten.

Wie eventuell auffällt, sind die Möglichkeiten ein Terrain über eine Funktion darzustellen nahezu endlos. Solange eine Eigenschaft über eine mathematische Formel oder auf anderem Wege dargestellt werden kann, ist es wahrscheinlich möglich diese in die Landschaftsgenerierung einer Welt zu integrieren. Die hier erwähnten Methoden beschreiben gerade einmal die Spitze des Eisbergs und die Generation einer Spielwelt bietet bei weitem genug Material, um darüber eine eigene Arbeit zu verfassen. Dieses Unterkapitel sollte jedoch eine gute Idee darüber geben wie eine Landschaft über die Formulierung einer Funktion definiert werden kann.

5 TEXTURIERUNG DES TERRAINS

Die Texturierung der Landschaft stellt im Falle eines prozedural generierten Terrainmeshes ebenfalls eine nicht unerhebliche Herausforderung dar. Das Texturieren von Modellen erfolgt normalerweise über eine oder mehrere Texturen, welche jeweils verschiedene Informationen enthalten und über die UV-Koordinaten der Meshvertices auf das letztendliche Modell gemappt werden. Eine Textur ist dabei meistens spezifisch für ein Modell erstellt. Dieser Ansatz funktioniert aufgrund der Ausmaße für Terrain allgemein eher selten und im Zusammenhang mit prozedural generiertem Terrain erst recht nicht, da das Mesh erst zur Laufzeit erzeugt wird und entsprechend das genaue Aussehen des Meshes zuvor meistens nicht bekannt ist. Texturen können daher nicht im traditionellen Sinne erstellt und auf das Mesh angewendet werden. Zudem treten einige weitere Probleme auf, die das Texturieren eines prozeduralen Terrains erschweren. Dieses Unterkapitel beschäftigt sich mit diesen Problemen und beschreibt einige Methoden, die in dieser Arbeit Anwendung finden, um die mit der prozeduralen Generierung verbundenen Texturierungsprobleme zu beheben.

Eine erste Komplikation entsteht, wenn versucht wird eine herkömmliche Textur auf das erzeugte Mesh anzuwenden. Zum einen enthält das Mesh eines prozeduralen Terrains keinerlei direkte UV-Informationen. Das Mapping der Textur auf das Mesh kann also nicht einfach ohne weiteres erfolgen. Zum anderen werden bei einer Textur, welche für ein spezifisches Modell erstellt ist, nicht zwingend alle Bereiche der Textur vollständig genutzt. Die Anwendung einer solchen Textur auf ein prozedurales Terrain würde also zu sichtbaren Artefakten führen. Prozedurales Terrain kann sich zudem theoretisch endlos und mit nahezu unendlichen Variationsmöglichkeiten in jede Richtung erstrecken. Das Festlegen von UV-Koordinaten für bestimmte Bereiche der Textur würde sich entsprechend als eher mühsam und sehr aufwändig gestalten.

Das Beheben beider dieser Probleme ist aber glücklicherweise relativ simpel. Als Ersatz für zuvor festgelegte UV-Koordinaten an den Meshvertices kann grundlegend die globale Position der Vertices genutzt werden. Texturkoordinaten befinden sich normalweise in einem Bereich zwischen 0 und 1. Bei einer Unter- oder Überschreitung dieses Bereiches wird die Textur je nach Setting wiederholt. Durch die Nutzung der globalen Koordinaten wiederholt sich die Textur also ständig über das gesamte Terrain. Über einen Skalierungsfaktor kann hierbei eingestellt werden, wie oft diese Wiederholung auftritt.

Um zusätzlich die erwähnten Artefakte zu verhindern, muss die Terraintextur als erstes nicht für ein spezifisches Modell entworfen werden. Stattdessen sollte die Textur nur ein generelles Material, wie Stein, Sand, Erde oder Grass, darstellen welches sich über den gesamten Texturbereich erstreckt. Damit sind aber nicht alle Artefakte gelöst. Da die Textur schrittweise wiederholt wird, wären an den Schnittstellen dieser Texturen weiterhin sichtbare Wechsel erkennbar. Die Ränder der Texturen müssen also nahtlos Übergänge zu den gegenüberliegenden Rändern aufweisen. Dafür gibt es verschiedene Verfahren, die in Abhängigkeit der Textur unterschiedlich gute Ergebnisse liefern.

Damit kann das Terrain in der Theorie texturiert werden. Bisher ist allerdings nicht erwähnt welches Paar an Komponenten der globalen Vertexkoordinaten als Ersatz für die UV-Koordinaten verwendet werden sollte. Die Wahl dieses Paares ist ausschlaggebend für die Richtung, die genutzt wird, um die Textur auf das Mesh zu projizieren. Abbildung 7 zeigt beispielsweise das Ergebnis der Nutzung des XY- und des XZ-Koordinatenpaars für die planare Projektion. Wie erkennbar werden Stellen des Meshes, dessen Normalen mit der Projektionsebene übereinstimmen, gut texturiert. Umso stärker allerdings die Facen normale von der Normale der Ebene des Koordinatenpaars abweichen, desto stärkere Verzerrungen treten auf. Da prozedurales Terrain selten in nur einer dieser Ebenen liegt, treten diese Verzerrungen, unabhängig von der Wahl des Koordinatenpaars, immer mehr oder weniger stark ausgeprägt auf. Zur Vermeidung dieser Verzerrungen ist ein einziges Koordinatenpaar also nicht ausreichend.

5.1 TRIPLANAR MAPPING

Ein Verfahren, das dieses Problem grundlegend löst, ist die Texturierung mittels *Triplanar Mapping* bzw. *Triplanar Texturing*. Triplanar Mapping nutzt kurzgefasst eine Mischung aus mehreren planaren Projektionen, um den letztendlichen Farbwert zu bestimmen (Geiss, 2007, Kapitel 1.5). In der Regel werden dafür insgesamt drei Projektionen genutzt, eine für jede Achse. Als Ersatz für die UV-Koordinaten dient dabei entsprechend das ZY-, XY- und XZ-Koordinatenpaar. Während die Komponenten der globalen Vertexposition genutzt werden, um die Texturkoordinaten zu bestimmen an denen die Texturen gesampelt werden müssen, finden die Komponenten der Vertexnormalen Anwendung in der Bestimmung des Beitrages der individuellen planaren Projektionen. Jede dieser Komponenten kann sich zwischen -1 und 1 befinden. Umso stärker die Vertexnormale in Richtung einer der drei Hauptachsen zeigt, desto näher befindet sich die entsprechende Komponente bei einer dieser beiden Grenzen, während die restlichen beiden Komponenten entsprechend kleiner ausfallen, um den Richtungsvektor weiterhin normiert zu halten. Eine simple Möglichkeit dies zu nutzen, wird unter anderem von Palko beschrieben (Palko, 2004, S. 1). Er nutzt die absoluten Werte der Normalenkomponenten und normiert diese durch eine anschließende Teilung durch die Summennorm der Normalen. Dadurch entsteht ein Mischvektor dessen Komponenten den Beitrag der einzelnen Achsen am Gesamtergebnis darstellen.

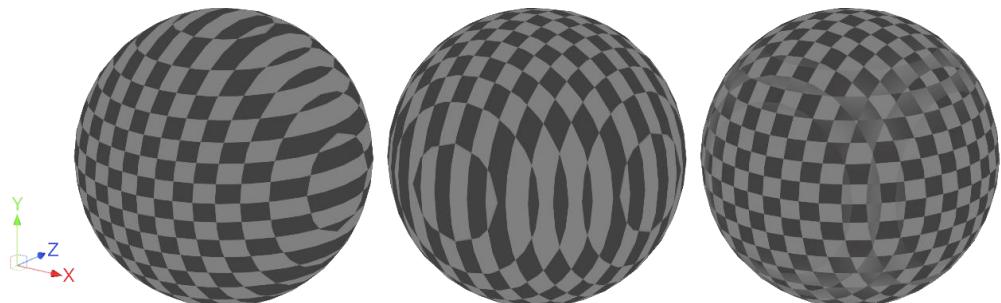


ABBILDUNG 7: VERGLEICH DER TEXTURPROJEKTIONEN IN DER XY-EBENE (LINKS) UND XZ-EBENE (MITTE) MIT DEM ERGEBNIS DES TRIPLANAR MAPPINGS MIT EINEM ERHÖHTEN SCHÄRFEFAKTOREN (RECHTS).

Um daraus einen Farbwert zu bestimmen kann nun zuerst mit Hilfe der drei Koordinatenpaare eine oder mehrere Texturen gesampelt werden. Daraus entstehen die Farbwerte entlang der entsprechenden Achsen. Anschließend werden die drei Farbwerte gewichtet, indem sie mit den Mischfaktoren multipliziert werden. Schlussendlich werden die gewichteten Farbwerte miteinander addiert, um den finalen Farbwert zu erhalten.

Die beschriebene Vorgehensweise ist ein nur relativ grundlegender Ansatz, liefert aber bereits vergleichsweise gute Ergebnisse. Eine Verbesserung des Endergebnisses kann über verschiedene Erweiterungen erzielt werden. Über das Exponenzieren der Beträge der einzelnen Normalenkomponenten, vor der Normalisierung, mit einem weiteren Faktor kann beispielsweise die Schärfe der Vermischung angepasst werden (Palko, 2014, S. 1).

Weitergehend kann alternativ zum Einbezug aller gesampelten Texturen, über einen Schwellwert bestimmt werden, wann eine Textur einbezogen wird (Geiss, 2007, Kapitel 1.5). Stimmt die Normale stark genug mit einer der Achsenrichtungen überein, trägt zum Großteil nur eine der Texturen zum Gesamtergebnis bei, während die anderen beiden, wenn überhaupt, nur leichte Farbänderungen bewirken.

Obwohl Triplanar Mapping das Problem der Verzerrungen zum größten Teil löst, ist dieses Verfahren auch aufwändiger als die herkömmliche Texturierung. In dem beschriebenen Ansatz müssen Texturen nicht nur einmal sondern mindestens drei Mal gesampelt werden. Zusätzlich findet die Texturierung des Terrains (und allgemein Objekten) in den meisten Fällen nicht über nur eine einzige Textur statt. Stattdessen werden nahezu immer Informationen mehrerer Texturarten kombiniert, um gutaussehende Objekte zu erzeugen. Jede dieser Texturen beschreibt dabei eine Objekteigenschaft, die zum Gesamtbild beiträgt. Wird Triplanar Mapping genutzt muss jede dieser Texturen auch mindestens drei Mal gesampelt werden. Diese Aufwandserhöhung wird noch problematischer, sobald weitere Texturierungsverfahren genutzt werden, um das Endergebnis zu verbessern. Abgesehen von der Aufwandserhöhung entstehen auch weitere Probleme bei der Mischung bestimmter Texturarten. Normal-Maps sind eine Art dieser problematischen Texturen und können nicht direkt miteinander vermischt werden, ohne ein falsches Ergebnis zu erzielen (Golus, 2017, Geiss, 2007). Um das Problem mit solchen Texturarten zu vermeiden, werden weitere Anpassungen benötigt, welche wieder zu etwas mehr Aufwand führen. Ein weiteres Problem entsteht durch die Texturwiederholung entlang der gesamten Landschaft. Basierend auf der Wiederholungsrate der Textur fällt diese Wiederholung ziemlich deutlich auf. Durch die Texturmischung durch Triplanar Mapping wird dieses Problem etwas gemildert. Über größere Distanzen und in Fällen, in denen die Terrainoberfläche eine ähnliche Ausrichtung besitzt, erweist sich diese Wiederholung allerdings weiterhin als problematisch.

5.2 TEXTURE SPLATTING

Ein Ansatz, um die verbleibenden erkennbaren Wiederholungen weniger auffällig zu gestalten und gleichzeitig etwas mehr Variation in das Aussehen des Terrains zu bringen, ist das Nutzen einer Vorgehensweise namens Texture Splatting, wie sie unter anderem von Bloom beschrieben wird (Bloom, 2000, S. 1). Diese Methode nutzt grundsätzlich einen Mischfaktor in der Form eines Alphawertes, um, je nach Position, Texturen miteinander zu mischen. Das Verhältnis, zu dem die Texturen gemischt werden, ist hierbei natürlich ausschlaggebend für das Aussehen des Terrains. Entsprechend gibt es eine Vielzahl an Methoden diesen Mischfaktor zu bestimmen.

Angenommen es wird nur zwischen zwei Texturen interpoliert, kann je nach den gewählten Texturen ein simpler Ansatz, wie das Mischen über eine manuell erstellte Textur, bereits ausreichen. Diese Textur legt über einen Alphawert fest zu welchem Anteil die Farbe an einer Texturposition zum Ergebnis beiträgt. Die zweite Textur nutzt entsprechend den ungenutzten Teil des Alphawertes, um den Farbbeitrag an derselben Texturposition zu ermitteln. In vielen Fällen kann jedoch eine komplexere Interpolation zu einem allgemein besseren und natürlicher aussehenden Ergebnis führen. Wie von Mishkinis beschrieben,

kann beispielsweise zusätzlich eine Heightmap mit Höheninformationen einbezogen werden, um einen natürlicheren Übergang zu erzeugen (Mishkinis, 2017, S. 1).

Die Interpolation zwischen nur zwei Texturen ist nahezu trivial. In der Praxis kann es aber von Vorteil sein auch drei oder mehr Texturen miteinander zu vermischen. Die Mischfunktion kann dabei beliebig komplex erweitert werden und es muss im Endeffekt je nach Anwendungsfall abgewägt werden, wann das Ergebnis gut genug ist und sich ein höherer Berechnungsaufwand zur Mischung nicht mehr lohnt.

5.3 TEXTURE BOMBING

Durch Texture Splatting kann das Problem der auffallenden Texturwiederholungen weiterhin etwas verringert werden. Über längere Distanzen sind die Wiederholungen jedoch meistens trotzdem noch auffällig genug, um ein Problem darzustellen. Eine Möglichkeit diese Wiederholungen auch in diesen Fällen zu eliminieren ist das Nutzen einer weiteren Methode namens Texture Bombing. Die grundlegende Idee hinter Texture Bombing ist es den UV-Raum mit einer Gitterstruktur in eine Reihe von Zellen zu unterteilen und eine oder mehrere Texturen an zufälligen Positionen innerhalb dieser Gitterzellen zu platzieren (Glanville, 2004, Kapitel 20.1). Die genaue Vorgehensweise wird in einer Arbeit von Glanville (2004) beschrieben. Über eine Mischung der Texturübergänge zwischen den einzelnen Zellen kann in der Theorie die sichtbare Wiederholung der Textur verringert werden. Über weitere Rotationen und Skalierungen der Texturen in den einzelnen Zellen kann die Wiederholung weitergehend nahezu komplett vermieden werden (Glanville, 2004, Kapitel 20.3.1).

In der Praxis ist dieses Verfahren aber relativ aufwändig (Glanville, 2004, Kapitel 20.2.1). In Kombination mit weiteren Texturierungsverfahren ist es also eventuell besser eine weniger aufwändige Methode zu nutzen, um Wiederholungen zu vermeiden. Eine solche alternative Methode wird beispielsweise von Quilez (2015) beschrieben. Hierbei werden mehrere virtuelle Varianten des durch die Wiederholung entstehenden Texturmusters verwendet und gemischt. Jede Variante bekommt einen unterschiedlichen aber konstanten Offset zugewiesen. Zur Erstellung dieser Varianten werden zuerst die eigentlichen Texturkoordinaten genutzt, um eine Variationstextur zu sampeln. Diese Textur steht beispielsweise über eine niedrigfrequente Noise-Textur zur Verfügung. Über den nun vorhandenen Wert der Variationstextur kann über die Multiplizierung mit der Anzahl der Variationen sowohl ein Variationsindex als auch ein Interpolationsfaktor in Form des Dezimalteiles bestimmt werden. Für die Interpolation werden Textursamples zweier Variationen benötigt. Zur Bestimmung der Offsets dieser beiden Variationen findet der Index als Eingabe in eine Hashfunktion Anwendung. Durch die Kombination der eigentlichen Texturkoordinaten und dem jeweiligen berechneten Offset können nun zwei Variationen der Objekttextur gesampelt und mittels des Interpolationsfaktors miteinander vermischt werden.

Ein Ansatz wie der von Quilez benötigt lediglich drei Textursamples (einen für die Variationstextur und zwei für die Variationen der Objekttextur) und ist damit in diesem Bezug effizienter als einige Beispiele von Glanvilles Texture Bombing, welche teilweise bis zu acht Textursamples benötigen (Glanville, 2004, Kapitel 20.2.1). Isoliert betrachtet ist dieser Unterschied nicht zwingend gravierend, werden aber zusätzlich sowohl Triplanar Mapping als auch Textur Splatting einbezogen, wird diese Methode mindestens sechs Mal genutzt, drei Mal für die planaren Projektionen entlang der Achsen und in jedem dieser Fälle zwei Mal für die zu interpolierenden Texturen. Dieser Aufwand steigt weiterhin, wenn weitere Projektionen genutzt, mehr als zwei Texturen interpoliert, oder weitere Verfahren eingesetzt werden.

6 ANFORDERUNGEN

Sowohl durch die beschriebenen Verfahren zur prozeduralen Terraingenerierung als auch durch die verwendete Hardware entstehen einige Anforderungen an die Flugsimulation damit die letztendliche Spielbarkeit und das allgemeine Nutzererlebnis gewährleistet sind. Vor allem durch die auf der Hardwareseite verwendeten Komponenten entstehen Anforderungen, welche Designentscheidungen in der Anwendung beeinflussen. Im Vordergrund stehen dabei vorwiegend die Hardwarekomponenten, über die der Spieler die virtuelle Welt wahrnimmt und mit den Elementen der Flugsimulation interagiert, namentlich das verwendete VR-Headset und das Icaros-System, welches für den größten Anteil der Eingaben und Interaktionen genutzt wird. Dieses Kapitel beschreibt die wichtigsten dieser entstehenden Anforderungen und behandelt zugleich einige der spezifischen Aspekte der verwendeten Hardware.

6.1 VIRTUAL REALITY

Um dem Spieler eine immersive Wahrnehmung der virtuellen Landschaft zu ermöglichen, finden im Zusammenhang mit der Flugsimulation Virtual Reality Anwendung. Die Verwendung von VR selbst bringt bereits einige sowohl positive als auch negative Aspekte mit sich und erfordert damit verbunden einige Anpassungen in der entstehenden Flugsimulation, um für diese Aspekte zu kompensieren.

Ein erster im Zusammenhang mit VR in vielen Nutzern auftretender negativer Aspekt, der eines der wahrscheinlich größten Probleme mit dieser Technologie darstellt, ist Motion Sickness. Mit Motion Sickness wird ein Phänomen bezeichnet, bei dem der Körper einer Person das Gesehene bzw. Wahrgenommene nicht mit den tatsächlichen physischen Geschehnissen in der Realität in Einklang bringen kann. Die Symptome äußern sich in der Regel in der Form von Übelkeit und Gleichgewichtsproblemen in unterschiedlichen Stärken und variieren von Person zu Person. In Bezug auf VR entsteht Motion Sickness meist durch die Diskrepanz zwischen den virtuellen und realen Bewegungen des Spielers. Motion Sickness kann personenabhängig aber auch bei einer kompletten Übereinstimmung dieser Bewegungen auftreten. Unabhängig von der Art der Anwendung ist es daher bei der Nutzung von VR zunächst wichtig sicherzustellen, dass die virtuellen Bewegungen bestmöglich mit den realen Bewegungen des Spielers übereinstimmen, um diese Diskrepanz weitestgehend zu vermeiden. Weiterreichend bedeutet das Vermeiden dieser Bewegungsdiskrepanz folglich auch das einige Bewegungen und damit verbundene Effekte innerhalb der Flugsimulation vermieden werden sollten. Stürzt der Spieler in der virtuellen Welt beispielsweise nach einer Kollision ab oder wird er durch einen Sturm durch die Luft geschleudert, stellt das für traditionelle Spiele allgemein kein Problem dar, in Kombination mit VR hingegen schon, da die meisten solcher Effekte logischerweise nicht ohne weiteres in der realen Welt repliziert werden können und die Bewegungsdiskrepanz entsprechend hoch wäre. Schlussfolgend bedeutet dies für die Flugsimulation, dass Bewegungen wie das Rollen des Flugzeuges, um Objekten auszuweichen, und Effekte wie das Abstürzen und entsprechendes Trudeln bei einer Kollision vermieden werden sollten.

Durch die Nutzung von VR zur Wahrnehmung entstehen auch allgemeine Probleme mit traditionellen UI-Elementen wie sie in vielen Spielen Anwendung finden. Der herkömmliche Weg dem Spieler verschiedene Informationen der virtuellen Welt über ein UI zu präsentieren, macht sich zweidimensionale Elemente zu nutzen, welche lediglich über das gerenderte Bild „gelegt“ werden. Damit der Spielfluss nicht gestört wird, werden diese UI-Elemente meistens entlang des Bildrandes platziert. Ein solcher Ansatz funktioniert bei der Verwendung von VR nur extrem selten. Ein Grund dafür ist das weite Sichtfeld, welches in Kombination mit VR genutzt wird, um die Immersion des Spielers zu erhöhen. UI-Elemente, die am Rand des Bildes platziert sind, würden gleichermaßen am Sichtfeldrand

erscheinen. Abgesehen von eventuell auftretenden Verzerrungen müsste der Spieler jedes Mal mit den Augen relativ stark in Richtung des Bildrandes blicken, um Informationen in einem solchen UI sehen zu können. Dies würde bereits nach einer kurzen Spieldauer schnell unangenehm werden. Dazu kommt, dass bei der stereoskopischen Wahrnehmung des gerenderten Bildes ein darübergelegtes UI keinerlei Tiefe besitzt. Entsprechend würde das UI als eher störend wahrgenommen werden und aufgrund der Nähe wahrscheinlich erst gar nicht vom Spieler interpretierbar sein. Die Erkennbarkeit ist jedoch nicht das einzige Problem von traditionellen UI-Elementen im Zusammenhang mit VR. Auch das grundlegende Level an Immersion dieser Elemente kann sich negativ auf das Nutzererlebnis auswirken. Während VR eine inhärent immersive Technologie ist, sind herkömmliche UI-Elemente meistens das Gegenteil und sind entsprechend in vielen Fällen schädlich für die Immersion des Spielers. Ein Hauptgrund dafür ist die Präsentation dieser UIs. Während der Stil der einzelnen Elemente nahezu immer an die Anwendung angepasst ist, sind die Elemente selbst grundsätzlich von der virtuellen Welt getrennt und stellen dem Spieler Informationen zur Verfügung, deren Präsentation unnatürlich wirken kann. Eine Alternative zu diesem traditionellen UI ist das Platzieren von UI-Elementen als dreidimensionale Objekte in der virtuellen Welt. Eine immersivere Variante dieses Ansatzes, welche in immer mehr VR-Spielen genutzt wird, ist das Integrieren dieser dreidimensionalen UI-Elemente in verschiedene Objekte der virtuellen Umgebung. Beispielsweise kann der Spieler anstelle eines UI-Elements für einen Kompass diesen in Form eines Items besitzen, um ihm bei der Orientierung zu helfen, und das verbleibende Leben kann als Schwarz-Weiß-Filter oder über Wunden anstelle eines Lebensbalken vermittelt werden. Diese Integration der UI-Elemente in Objekte verbindet die Vermittlung der Informationen mit der virtuellen Welt auf eine natürlich wirkende Art und Weise und ermöglicht entsprechend ein weitaus höheres Level an Immersion als es ein traditionelles UI bieten kann. Für die Flugsimulation dieser Arbeit bedeuten diese Aspekte der UI-Nutzung, dass das gesamte UI in die virtuelle Welt integriert werden sollte, um das allgemeine Erlebnis des Spielers nicht negativ zu beeinflussen und gleichzeitig ein möglichst hohes Immersionslevel zu ermöglichen.

Nicht nur die Verwendung von VR beeinflusst einige Aspekte der Flugsimulation, sondern auch die Eigenschaften der spezifischen VR-Hardware haben einen Einfluss auf die Simulation. Das genaue VR-Headset, das in Kombination mit der Flugsimulation und dem Icaros-System Anwendung findet, ist das *Vision 8K X* von *Pimax*⁷. Mit einer nativen Bildschirmauflösung von 3840 x 2160 Pixeln pro Auge liegt die Auflösung dieses VR-Headsets am oberen Ende des Spektrums. Die Bildwiederholrate ist variabel und kann zwischen 60 Hz, 75 Hz, den nativen 90 Hz oder durch Upscaling 114 Hz gewählt werden. Zudem beträgt das Field of View (FOV) insgesamt 200°. Es bietet damit einen ebenfalls höheren Sichtbereich als die meisten VR-Headsets. Dank dieser vergleichsweise hohen Spezifikationen muss die letztendliche Flugsimulation performant laufen und eine entsprechend hohe Bildwiederholungsrate aufweisen, um Probleme die in Verbindung mit VR und einer niedrigen oder inkonsistenten Bildwiederholungsrate auftreten, wie zum Beispiel die bereits erwähnte Motion Sickness, zu vermeiden. Insbesondere das hohe FOV bedeutet, dass ein großer Teil des Terrains auf einmal sichtbar sein kann und entsprechend auch vorhanden sein muss. Der Algorithmus für die Terraingenerierung muss folglich auch mit dieser großen Menge an Terrain zureckkommen können.

⁷ Pimax Vision 8K X - <https://pimax.com/product/vision-8k-x/>



ABBILDUNG 8: DIE PIMAX VISION 8K X, DAS ICAROS-SYSTEM UND DER ZUGEHÖRIGE CONTROLLER, WELCHE IN KOMBINATION MIT DER FLUGSIMULATION VERWENDET WERDEN.

6.2 ICAROS

Wie bereits erwähnt findet nahezu die gesamte Interaktion mit der Flugsimulation über ein Icaros-System, folglich abgekürzt bezeichnet mit Icaros, statt. Das genaue Modell, das hierbei im Zusammenhang mit der Flugsimulation Anwendung findet, ist das *Icaros Pro*⁸. Das Icaros stellt eine Möglichkeit bereit den Spielspaß von Videospielen und die immersive Natur von VR mit dem Training verschiedener Muskelbereiche und allgemein dem Fitnesstraining des Spielers zu verbinden. Dieser positive Effekt auf die Fitness der Nutzer ist zu unterschiedlichen Graden durch verschiedene Studien nachgewiesen (Dębska et al., 2019, Feodoroff et al., 2019). Bei der Verwendung des Icaros befindet sich der Nutzer in einer nahezu liegenden Position in diesem. Über die Verlagerung des Schwerpunktes ist sowohl die Neigung nach vorne und hinten als auch die seitliche Rotation nach links und rechts steuerbar. Die Empfindlichkeit der Rotationen kann zu einem gewissen Grad manuell eingestellt und falls nötig weiter über die Position der Handgriffe angepasst werden. Die Bewegung des Icaros ist dabei komplett mechanisch. Das Tracking der Rotationen und Eingaben für die Verwendung in der Software erfolgt über den mitgelieferten Controller, welcher an einem der Handgriffe befestigt ist.

Der Nutzer steuert bei der Verwendung des Icaros die virtuellen Bewegungen des Flugzeuges komplett mit seinen eigenen physischen Bewegungen. Da ein VR-Headset für die Wahrnehmung der Flugsimulation verwendet wird, muss auch hier darauf geachtet werden, dass die virtuellen mit den realen Bewegungen übereinstimmen, um die Wahrscheinlichkeit des Auftretens von Motion Sickness zu minimieren. Die Abhängigkeit der Bewegungen innerhalb der virtuellen Welt von den Bewegungen des Spielers auf dem Icaros sorgt außerdem dafür, dass sichergestellt werden muss, dass die nötigen virtuellen Bewegungen

⁸ Icaros Pro - <https://www.icaros.com/de/produkte/icaros-pro/>

auch mit physischen Bewegungen für den Spieler möglich sind, ohne diesen komplett zu überfordern. Wäre beispielsweise eine starke Linksdrehung im Spiel erforderlich, um einem entgegenkommenden Objekt auszuweichen, ist das an sich kein zwingendes Problem, folgt nach diesem Objekt allerdings direkt ein weiteres Hindernis, bei dem der Spieler in die komplett entgegengesetzte Richtung ausweichen muss, kann sich dies schon eher als problematisch erweisen. Einerseits sind solche starken Richtungsänderungen auf die Dauer extrem anstrengend für den Spieler, und entsprechend hemmend für die Spielbarkeit, da sich dieser jedes Mal in die entsprechende Richtung „werfen“ muss. Andererseits sind solche Bewegungsexrema eventuell gar nicht möglich, da der Spieler bei solchen Richtungsänderungen mit dem entstehenden Momentum zu kämpfen hat. Daraus folgt auch, dass solche Situationen eine nicht unerhebliche Verletzungsgefahr darstellen. Der Spieler liegt lediglich im Icaros und ist nicht direkt an diesem befestigt. Unterschätzt der Spieler die entstehenden Kräfte solcher extremen Bewegungen ist es durchaus möglich das er den Halt verliert und aus dem Icaros geworfen wird. Da der Spieler ein VR-Headset bei der Nutzung des Icaros trägt und dessen Wahrnehmung der realen Welt damit verbunden stark eingeschränkt ist, stellen diese Situationen ein weiteres Risiko für die Beschädigung der Hardware und natürlich wichtiger auch eine höhere Verletzungsgefahr für den Spieler selbst dar. Umso mehr Erfahrung ein Spieler mit dem Icaros hat, desto besser kann dieser die Kräfte einschätzen und desto geringer wird diese Verletzungsgefahr folglich. Damit die Flugsimulation aber auch für Personen mit weniger Icaroserfahrung kein unnötiges Verletzungsrisiko darstellt, sollten solche extremen Bewegungen entsprechend, wenn überhaupt, nur sehr selten nötig sein oder am besten komplett vermieden werden.

7 GENERIERUNG DER SPIELWELT

Die Implementation der Flugsimulation erfolgt in der Unity Engine⁹. Dadurch wird die Nutzung des Icaros-SDKs ermöglicht, welches sich um die komplette Kommunikation zwischen der Engine und dem Icaros kümmert. Die Integration von VR findet mittels des *SteamVR Plugins* aus dem Unity Asset Store¹⁰ statt. Bevor auf die Designentscheidungen und den Code der Simulation eingegangen wird, ist es vorteilhaft kurz auf das finale Produkt einzugehen und sowohl das Setting als auch den generellen Gameplay-Loop zu beschreiben, um ein grobes Gesamtbild der Flugsimulation zu vermitteln.

Das allgemeine Setting der Flugsimulation ist futuristisch gehalten und das gesamte Spiel findet auf einem größtenteils unerkundeten Wüstenplaneten statt. Es wird vermutet, dass der Wüstenplanet eine potenziell enorme Menge an seltenen und entsprechend begehrten Mineralien enthält. Der Spieler ist Teil einer wissenschaftlichen Expedition zu diesem Planeten, mit dem Ziel, möglichst viele Proben dieser Mineralien zu sammeln, um größere Abbauoperationen rechtfertigen. Nach dem Sammeln von mehr als genug Mineralproben befindet sich das Expeditionsteam auf dem Weg den Planeten zu verlassen, wenn das Raumschiff durch einen abrupten und extrem starken Sandsturm außer Kontrolle gerät und zum Abstürzen gebracht wird. Der Charakter des Spielers wacht nach diesem Absturz in einem scheinbar endlosen Labyrinth an Schluchten auf und findet ein altes Expeditionsbike, mit dem er um Hilfe rufen kann. Das grundlegende Gameplayziel für den Spielers ist es den Flug durch diese Schluchten zu überleben bis das Rettungsteam ankommt und während dieser Zeit einerseits möglichst viele der im Absturz verlorenen Mineralproben zu finden und andererseits auf dem Weg zu diesen Mineralproben neue Stichproben zu sammeln, um einen möglichst hohen Score zu erhalten. Der Spieler wird dazu anfangs an einer zufälligen Position innerhalb der generierten Schluchten der Spielwelt gespawnt und hat dann, je nach der ausgewählten Missionsdauer, eine gewisse Zeit die Spielwelt zu erkunden und Punkte zu sammeln.

⁹ Unity - <https://unity.com/>

¹⁰ SteamVR Plugin - <https://assetstore.unity.com/packages/tools/integration/steamvr-plugin-32647>

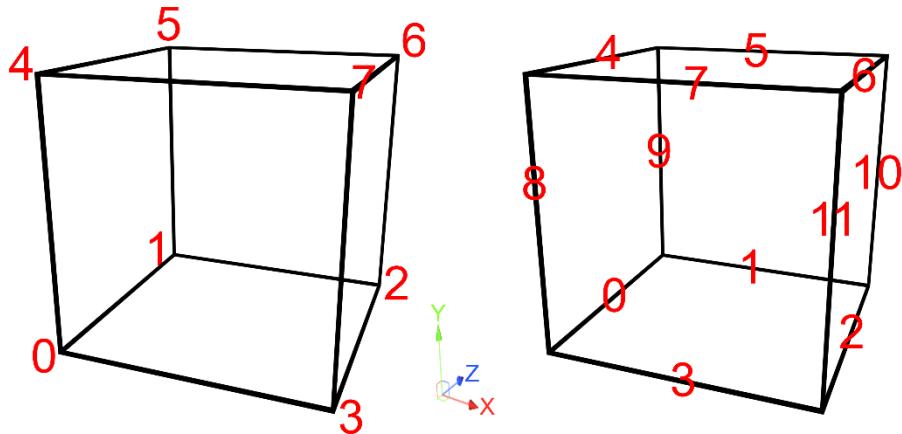


ABBILDUNG 9: DIE GEWÄHLTE ECKPUNKT- (LINKS) UND KANTENREIHENFOLGE (RECHTS).

7.1 ERSTER EXTRAKTIONSANSATZ

Die erste und wichtigste Entscheidung bei der Implementation ist die Wahl des Verfahrens zur Isoflächenextraktion. Nach einer Einarbeitung in die Materie ist die Auswahl der Extraktionsmethoden auf die vier in Kapitel 3 beschriebenen möglichen Kandidaten begrenzt. Zwischen diesen vier Algorithmen bieten theoretisch vor allem die beiden komplexeren Dual Marching Cubes Verfahren potenziell einige positive Eigenschaften für das Mesh des prozeduralen Terrains.

Die erste Wahl fiel auf die Version des Dual Marching Cubes Algorithmus von Schaefer und Warren. Der Grund dafür ist hauptsächlich die adaptive Natur dieses Verfahrens. Während die mit Marching Cubes verbundenen ungünstigen Dreiecke nicht optimal für das Ergebnis sind, stellen diese für das finale Mesh kein zwingendes Problem dar. Auch die mehrfach vorhandenen Vertices könnten am Ende, falls nötig, auch nachträglich über ein weiteres Verfahren entfernt werden. Die adaptive Anpassung des Meshes an die Gegebenheiten des Terrains stellt hingegen einen enormen Vorteil für die prozedurale Generierung dar, welcher viele der Nachteile dieses Verfahrens übertrumpft. Zum einen kann dadurch ein sehr detailliertes Terrain in der Nähe des Spielers erzeugt werden, ohne die Anzahl der Vertices uniform erhöhen zu müssen. Zum anderen kann aber auch weiter entferntes Terrain durch die Anpassung des Fehlertoleranzwertes mit insgesamt weniger aber dennoch genug Detail generiert werden, damit die Eigenschaften der Isofläche bzw. des Terrains in diesen Regionen trotzdem noch einen ausreichenden Detailgrad besitzen, um erkennbar zu bleiben. Potenziell ist es über die progressive Anpassung des Toleranzwertes zudem möglich harte Übergänge zwischen diesen Detailstufen weitestgehend zu vermeiden.

Für die Implementation des Verfahrens wird zunächst eine klassische Octreestruktur benötigt. Zur Bestimmung des Unterteilungsgrades dieses Octrees muss eine Fehlerfunktion festgelegt werden, mit welcher der mit einer Zelle verbundene Fehler bestimmt werden kann. Die von Schaefer und Warren beschriebene Fehlerfunktion ist wie erwähnt jedoch vergleichsweise komplex und aufwändig. Da diese Fehlerfunktion häufig verwendet wird, wäre es daher von Vorteil eine günstigere Alternative mit vergleichbaren Ergebnissen zu finden. Glücklicherweise findet Dual Marching Cubes auch in der OGRE Engine¹¹ Anwendung. Als Teil einer Spieleengine werden dabei entsprechende Optimierungen vorgenommen, die der Performanz helfen. Eine dieser Optimierungen ist das Nutzen einer effizienteren Fehlerfunktion für die Unterteilung des Octrees. Lehmann-Böhm (2013 (a))

¹¹ OGRE Webseite - <https://www.ogre3d.org/>

beschreibt diese Optimierung und die allgemeine Funktionsweise der Enginekomponente, die dieses Verfahren enthält. Wie er beschreibt, wird ein Verfahren von Bajaj et al. (Bajaj et al., 2004, S. 15-16) genutzt. Als Grundlage für diese alternative Fehlerfunktion dient die Abweichung von interpolierten Werten an bestimmten Punkten der Octreezelle zu den tatsächlichen Funktionswerten an diesen Positionen. Dazu werden zunächst die acht Eckpunkte der Zelle gesampelt, für die momentan die Nötigkeit einer weiteren Unterteilung geprüft wird. Mit diesen acht Werten können darauffolgend alle Eckpunktswerte der acht theoretisch möglichen Kinderzellen durch eine trilineare Interpolation approximiert werden, mit Ausnahme der acht zuvor gesampelten Eckpunktpositionen. Werden zusätzlich die tatsächlichen Funktionswerte an all diesen möglichen Eckpunkten berechnet, kann der Fehler der betrachteten Octreezelle wie folgt gebildet werden (Bajaj et al., 2004, S. 16):

$$Error = \sum \frac{|f^{i+1} - f^i|}{|\nabla f^i|}$$

Wobei $|f^{i+1} - f^i|$ den Betrag der Differenz zwischen dem interpolierten und tatsächlichen Wert an einem der möglichen Eckpunkte darstellt und $|\nabla f^i|$ die Länge des Gradientenvektors an demselben Eckpunkt ist. Der letztendliche Fehler einer Octreezelle ist also die Summe aller Fehler an den Eckpunkten von potenziellen Kinderzellen, die nicht den Eckpunkten der originalen Zelle oder bereits betrachteten Eckpunktpositionen entsprechen. Genaueres zu dieser Fehlerfunktion kann in der Beschreibung von Lehmann-Böhm (2013 (a)) und der Arbeit von Bajaj et al. (2004) nachgelesen werden. Diese Vorgehensweise zum Bestimmen des Fehlers ist ebenfalls nicht komplett kostenlos, allerdings können einige Optimierungen vorgenommen werden, die der Performanz helfen. Sollte die momentan betrachtete Zelle eine Unterteilung benötigen, entsprechen die Werte der Terrainfunktion, die zur Fehlerbestimmung an jedem Eckpunkt der möglichen Kinderzellen berechnet werden, den benötigten Funktionswerten für die weiteren Unterteilungsentscheidungen der entstehenden Kinderzellen. Als solche müssen diese Werte nicht mehrfach erneut in den Kinderzellen berechnet werden, sondern können zwischengespeichert und bei einer Unterteilung an die entsprechenden Zellen weitergegeben werden. Weitere Optimierungen werden ebenfalls von Lehmann-Böhm beschrieben (Lehmann-Böhm, 2013 (a), S. 1). Falls nicht bereits Teil der Octreestruktur, sollte natürlich eine minimale Zellgröße festgelegt werden, damit die Unterteilung beim Erreichen dieser gestoppt wird, ohne dass der Fehler berechnet werden muss. Der Zeitpunkt des Prüfens, ob der Zellfehler den tolerierten Fehler überschritten hat, stellt eine weitere Optimierungsmöglichkeit dar. Da der gesamte Zellfehler über die Summe der einzelnen Fehler an gesampelten Punkten gebildet wird und diese Summe dementsprechend mit jedem weiteren Punkt nie kleiner werden kann, ist es in vielen Fällen effizienter das Überschreiten des Toleranzwertes direkt nach der Addition jedes Teilfehlers zu prüfen, anstatt erst nach der Bildung des Gesamtfehler. Sobald die Grenze erreicht ist, kann dann das weitere Berechnen der Teilfehler abgebrochen und die betrachtete Zelle direkt unterteilt werden. Da die Werte der Terrainfunktion für die Verwendung in den Kinderzellen zwischengespeichert werden, müssen diese im Falle eines vorzeitigen Abbruches noch nachberechnet werden. Nachdem diese Werte aber immer in den Kinderzellen erforderlich sind, entsteht dadurch kein wirklicher Mehraufwand. Mit der Hilfe dieser Fehlerfunktion entsteht ein Octree der als Basis für die duale Gitterstruktur dient. Um die Gitterstruktur zu generieren, muss zunächst eine Position eines Gittervertex pro Octreezelle bestimmt werden. Der Einfachheit halber und, wie Lehmann-Böhm anmerkt, um mögliche Selbstüberschneidungen im letztendlichen Mesh zu verhindern (Lehmann-Böhm, 2013 (b), S. 1), werden die Mittelpunkte der Blattzellen für diese Positionen gewählt.

Da nun auch die Position der Gittervertices festgelegt ist, ist der nächste Schritt das Verbinden dieser Vertices zum eigentlichen dualen Gitter. Schaefer und Warren beschreiben in ihrer Arbeit jedoch nur die zweidimensionale Version ihres Traversal-Algorithmus. Holmlid beschreibt eine dreidimensionale Variante dieses Algorithmus, die grundsätzlich gleich funktioniert aber eine Handvoll an Änderungen enthält, um in Kombination mit einem Octree genutzt werden zu können (Holmlid, 2010, S. 21-25). Die drei rekursiven

Funktionen *FaceProc*, *EdgeProc* und *VertProc* existieren weiterhin in einer angepassten Form. Da in einem Octree zusätzlich dreidimensionale Zellen existieren wird zusätzlich eine vierte Funktion namens *NodeProc* benötigt. Jede dieser vier Funktionen bekommt entweder 1, 2, 4 oder 8 Octreezellen als Parameter übergeben. Die Wahl und Reihenfolge dieser Zellen kommt auf die festgelegte Zellreihenfolge des Octrees an. Der letztendliche Code in der Implementation basiert zum einen auf dem Pseudocode von Holmlid und ist zum anderen, unter Anpassung der Octreezellen, auf welche die Funktionen angewendet werden, nahezu gleich dem Pseudocode von Lehmann-Böhm. Die Implementation dieser Methoden wird daher nicht als Teil dieser Arbeit behandelt und Details können in den entsprechenden Quellen nachgelesen werden (Holmlid, 2010, Lehmann-Böhm, 2013 (b)).

Damit kann das duale Gitter in der Theorie erstellt werden. In der Praxis besteht das Terrain jedoch nahezu nie aus einem einzigen großen Mesh. Stattdessen ist es in der Regel in einzelne Bereiche unterteilt, um zu verhindern, dass das gesamte Terrainmesh bei einer Änderung aktualisiert werden muss. Nachdem der Rand des Octrees nicht komplett von dem bisherigen dualen Gitter abgedeckt ist, würden Lücken mit variierender Größe zwischen benachbarten Bereichen entstehen. Um dies zu verhindern, muss das duale Gitter auf den Rand des Octrees erweitert werden. Die Methode von Schaefer und Warren benötigt dafür einen weiteren Durchlauf des Octrees (Lehmann-Böhm, 2013 (b), S. 1). Lehmann-Böhm beschreibt deshalb eine alternative Vorgehensweise, mit der diese Erweiterung bereits während der Erstellung des eigentlichen Gitters stattfinden kann. Jedes Mal, wenn eine Zelle des dualen Gitters erstellt wird, wird dazu zusätzlich geprüft, ob eine weitere Zelle für den Rand generiert werden muss (Lehmann-Böhm, 2013 (b), S. 1).

Allgemein muss, um diese Prüfung durchführen zu können, bekannt sein, ob eine Octreezelle ein Bestandteil des Randes ist. In der Implementation dieser Arbeit wird dafür ein Integer bzw. Enum in Kombination mit bitweisen Operationen verwendet. Die ersten sechs Bits dieses Wertes stellen je eine der sechs möglichen Zellseiten dar. Durch die Kombination dieser Bits kann bestimmt werden entlang welcher Seiten sich eine Zelle am Rand des Octrees befindet. Jede Zelle des Octrees enthält einen solchen Integer. Da sich jede der acht möglichen Kinderzellen immer drei Seiten mit der Elternzelle teilen, kann durch eine bitweise Verundung mit dem entsprechenden Bitcode festgelegt werden, welche Kinderzellen weiterhin am Rand des Octrees liegen. Nachdem die erste Zelle eines Octrees immer ein Teil aller Ränderseiten ist, wird der Octree anfangs mit allen sechs Bits gesetzt initialisiert. Bei der ersten Unterteilung werden dann die Bitcodes der Kinderzellen mit den Bitcodes der zugehörigen Seiten verundet. Bei jeder weiteren Unterteilung bleiben dadurch nur die Bits der Seiten erhalten, welche sich auch weiterhin von den entsprechenden Kinderzellen geteilt werden.

Da nun ermittelt werden kann, ob sich eine Zelle am Rand des Octrees befindet, kann die von Lehmann-Böhm beschriebene Funktion zur Erstellung der Randzellen umgesetzt werden. Diese zusätzliche Funktion prüft bei der initialen Erzeugung einer dualen Zelle mittels einer Reihe an If-Anweisungen, ob einige der acht Eingabezellen entlang einer oder mehrerer Octreeränder liegen und dementsprechend die Anforderung zum Erstellen einer Randzelle erfüllen (Lehmann-Böhm, 2013 (b), S. 1). Jede dieser If-Anweisungen prüft, ob die durch die acht Octreezellen definierte duale Zelle eine äußere Seite, Kante, oder einen der Eckpunkte berührt. Tritt einer oder mehrere dieser Fälle ein, müssen eine oder mehrere weitere duale Zellen in den entsprechenden Richtungen generiert werden, damit die Lücken zwischen der Gitterstruktur und den tatsächlichen Grenzen des Octrees geschlossen werden. Als Vertices der neuen Zellen dienen dabei die eigentlichen Mittelpunkte der Octreezellen und die Mittelpunkte der Seiten der Octreezellen welche am Rand des Octrees liegen (Lehmann-Böhm, 2013 (b), S. 1). Einerseits ist das Vorgehen in dieser Funktion relativ monoton und andererseits ist der letztendliche Code in der Implementation ebenfalls wieder sehr ähnlich zu dem von Lehmann-Böhm beschriebenen Pseudocode. Auf die Beschreibung dieses Codes wird daher verzichtet und genauere Details können in der Beschreibung von Lehmann-Böhm nachgelesen werden (Lehmann-Böhm, 2013 (b), S. 1).

Nach dem Erstellen aller solcher zusätzlichen Zellen deckt das duale Gitter den kompletten Octree ab. Der letzte Schritt ist nun die eigentliche Meshherstellung mittels Marching Cubes. Bei der Erstellung einer dualen Gitterzelle werden Daten in zwei Listen eingefügt. Die Einträge der ersten Liste bestehen jeweils aus den acht Positionen der Eckpunkte einer dualen Zelle und die Einträge der zweiten List aus den zugehörigen Gradientenvektoren und Funktionswerten dieser Eckpunkte. Die Eingabe für den Marching Cubes Algorithmus besteht aus diesen beiden Listen. Da kein uniformes Gitter verwendet wird, iteriert Marching Cubes nicht in bestimmten Schritten über einen Bereich, sondern direkt über alle dualen Zellen der Eingabelisten. Standardmäßig für Marching Cubes wird dabei zuerst die Zellkonfiguration anhand der Funktionswerte der Zelleckpunkte berechnet. Ist die Zelle nicht komplett innerhalb oder außerhalb des Terrains, werden die benötigten Vertexpositionen der Konfiguration entlang der Zellkanten interpoliert. Darauffolgend werden die Einträge der Dreieckstabelle genutzt, um die Vertices zu Dreiecken zu verbinden. In diesem Schritt werden die Normalen der Vertices zunächst über das Kreuzprodukt pro Dreieck berechnet. Aus den Listen der Vertices, Dreiecke und Normalen wird am Ende ein Mesh erstellt und zur Darstellung an Unitys MeshFilter-Komponente übergeben.

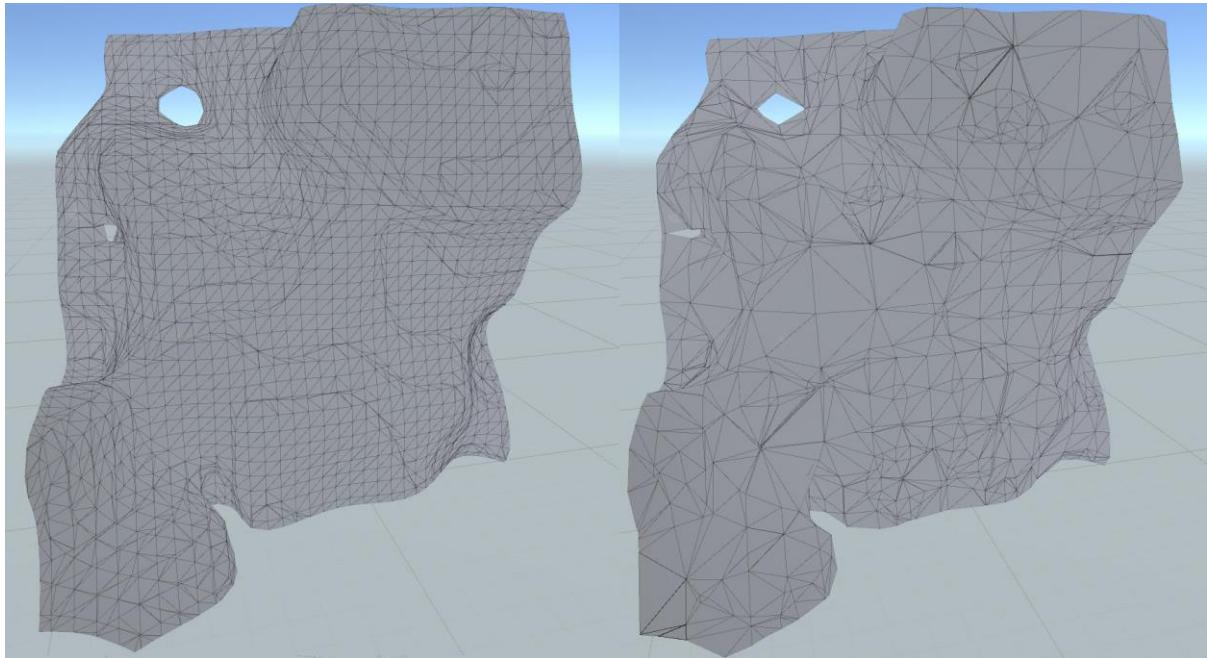


ABBILDUNG 10: VERGLEICH ZWISCHEN DEM DURCH MARCHING CUBES (LINKS) UND SCHAEFER UND WARRENS DUAL MARCHING CUBES (RECHTS) ERZEUGTEN MESH DES GLEICHEN TERRAINAUSSCHNITTS.

Die bisher beschriebene Implementation von Schaefer und Warrens Dual Marching Cubes ist bei weitem nicht ausreichend für eine letztendliche Nutzung in der Flugsimulation und könnte auch an vielen Stellen verbessert werden. Sie ist jedoch ausreichend, um zu beurteilen, ob dieser Algorithmus überhaupt für die Ziele dieser Arbeit geeignet ist. Wie in Abbildung 10 zu sehen ist, ist das Ergebnis dieser Methode grundsätzlich nicht schlecht. Die gewählte Fehlertoleranz spielt hierbei aber natürlich eine große Rolle. Ein weitaus günstiger unterteiltes Mesh könnte also durch die bessere Wahl dieser erzeugt werden. Auch die initiale Performanz lässt zu wünschen übrig. Vor allem das ständige Sampeln der Terrainfunktion, um an die Funktionswerte und Gradienten bestimmter Stellen zu kommen, stellt einen erheblichen Teil dieses Aufwandes dar. Zu diesen Problemen kommt, dass teilweise eine höhere Anzahl an Vertices und Dreiecken erzeugt werden als bei einem gleichen durch Marching Cubes erzeugten Mesh. Diese Erhöhung ist zwar erwartet, da sich die Vertices auf detailreiche Stellen konzentrieren, führt aber dennoch dazu, dass das Mesh

für die Darstellung einer größeren Landschaft eventuell weiterverarbeitet werden müsste, um die Gesamtzahl an Vertices zu reduzieren. Zusätzlich dazu werden die Normalen des Meshes momentan pro Dreieck und nicht pro Vertex berechnet. Um ein glatteres Shading zu erreichen, müssten also noch die Normalen von Vertices mit derselben Position zu den eigentlichen Vertexnormalen kombiniert werden, was jedoch wieder einen weiteren Aufwand mit sich bringt.

7.2 ZWEITER EXTRAKTIONSANSATZ

Aufgrund der im vorherigen Unterkapitel genannten Probleme ist zum Vergleich zunächst das ebenfalls vielversprechende Dual Marching Cubes Verfahren von Nielson implementiert worden. Die erste Implementation dieses Verfahrens ist ebenfalls komplett auf der CPU erfolgt. Abbildung 11 zeigt das Ergebnis dieses Algorithmus. Wie erkennbar ist das Mesh zwar nicht adaptiv unterteilt, liefert aber dennoch eine sehr gute Repräsentation des Terrains. Vor allem die Meshstruktur ist weitaus besser als bei einem von Marching Cubes oder von Schaefer und Warrens Methode erzeugten Mesh. Weitergehend von Vorteil sind für diese Arbeit auch die bereits kombinierten Vertices dieses Verfahrens und die durch die Abhängigkeit von benachbarten Zellen entstehende Leichtigkeit mit der die Vertexnormalen generiert werden können.

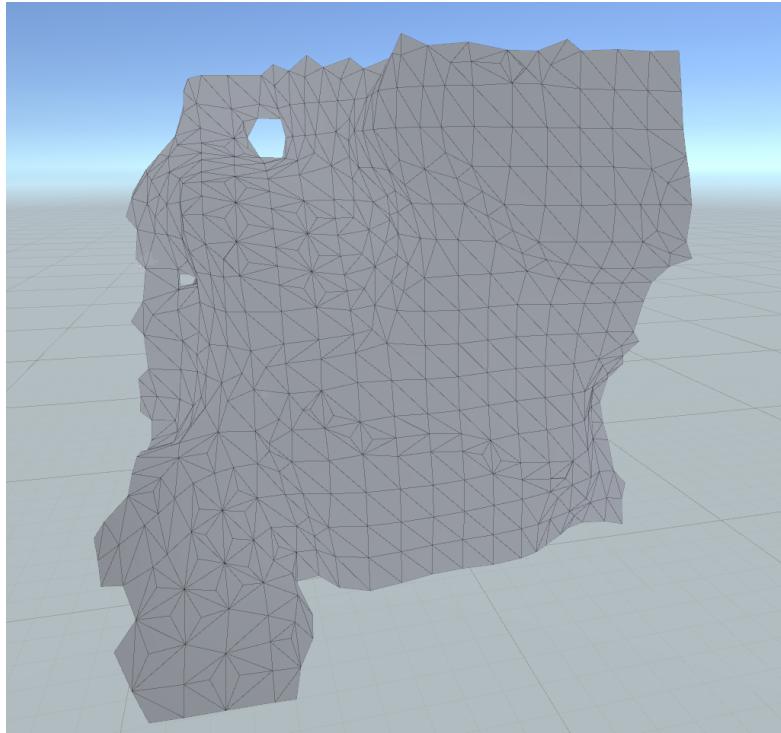


ABBILDUNG 11: AUSSCHNITT EINES TERRAINMESHES WIE ES DURCH NIELSONS DUAL MARCHING CUBES ERZEUGT WIRD.

Da dieses Verfahren im Endeffekt sehr ähnlich zu Marching Cubes ist und keinerlei Octree aufgebaut werden muss, ist die letztendliche Performanz ebenfalls um ein Vielfaches besser als bei Schaefer und Warrens Version. Obwohl auch hier nicht das letztendliche Erzeugen des Meshes am problematischsten für die Performanz ist, sondern auch wieder das Evaluieren der Terrainfunktion an den Gitterpunkten. Falls beispielsweise ein Mesh für einen Bereich von 32 Zellen entlang jeder Hauptachse erstellt werden sollte, werden im besten Fall 33^3 oder 35 937 Samples benötigt. Werden die Zellen komplett separat betrachtet und entsprechend die Samples an Zelleckpunkten öfters in benachbarten Zellen benötigt,

oder sollten weitere Samples für die Berechnung der Normalen benötigt werden, steigt diese Zahl gegebenenfalls noch weiter. Es gibt verschiedene Wege das Problem dieser Sampleanzahl zu lösen. Die Vorgehensweise in dieser Arbeit ist es die Rechenkraft einer GPU für die letztendliche Implementation von Nielsons Verfahren zu nutzen. Spezifischer werden die in Unity verwendbaren Compute Shader verwendet, um das gesamte Sampling der Terrainfunktion und die Berechnungen der Meshinformationen auf die GPU zu verlagern. Die Beschreibung der ersten Referenzimplementation auf der CPU wäre deswegen weitestgehend redundant und findet daher nicht als Teil dieser Arbeit statt.

Compute Shader erlauben es allgemein Code, getrennt von der eigentlichen Renderpipeline, auf der GPU auszuführen und die Ergebnisse bei Bedarf auf der CPU auszulesen. Dank der parallelen Natur der GPU-Architektur eignen sich vor allem hoch parallelisierbare Algorithmen für die Nutzung von Compute Shadern. Nielsons Dual Marching Cubes fällt durch die Ähnlichkeit zu Marching Cubes in diese Kategorie von Algorithmen und ist entsprechend gut für eine Implementation mittels Compute Shadern geeignet. Der Compute Shader dieser Arbeit behandelt Bereiche, weitergehend als Chunks bezeichnet, mit 32 Samplen entlang jeder der drei Hauptachsen. Die Abstände zwischen diesen Samplepunkten variieren mit der Größe des behandelten Chunks. Die Chunkgrößen entsprechen dabei immer einer Zweierpotenz. Ein Chunk der Größe 1 hat beispielsweise einen Abstand von einer Einheit zwischen Samplepunkten und deckt damit einen Bereich von 32^3 Einheiten ab. Ein Chunk der Größe 4 besitzt hingegen einen Abstand von vier Einheiten zwischen Samplepunkten und deckt einen Bereich von insgesamt 128^3 Einheiten ab.

Als Eingabe für den Compute Shader dient im Grunde nur die Position und Größe des Chunks. Für das zwischenzeitige Speichern von berechneten Informationen stehen zwei Compute Buffer zur Verfügung. Im ersten dieser beiden Buffer werden die Samples der Terrainfunktion und die Gradientenvektoren an den entsprechenden Stellen gespeichert. Folglich enthält dieser Buffer Platz für insgesamt 33^3 float4-Werte. Der zweite dieser Compute Buffer speichert die berechneten Informationen der Zellkanten. Da sich mehrere benachbarte Zellen einige dieser Kanten teilen und um weitestgehenden unnötige Speicherplatzverschwendungen zu vermeiden, werden pro Zelle nicht die Informationen aller zwölf Zellkanten gespeichert, sondern nur zu drei dieser Kanten. Diese Kantengruppe besteht aus der ersten, vierten und neunten Kante einer Zelle (Indices 0, 3 und 8 in Abbildung 9). Durch diese Aufteilung kommt es lediglich an einigen Rändern des Chunks zur unnötigen Speicherung von Kanteninformationen. Da weiterhin jede Kante mit maximal vier Vertices assoziiert werden kann, besteht die eigentlich gespeicherte Kanteninformation aus den bis zu vier Indices der erstellten Vertices. Der Buffer für diese Kantengruppen enthält folglich also Platz für 32^3 int3x4-Werte. Um die erzeugten Meshinformationen auslesen zu können werden vier weitere Computer Buffer benötigt. Drei Buffer in die die Vertexpositionen, Vertexnormalen und Vertexindices für die Dreiecke des Meshes geschrieben werden und ein weiterer Buffer, mit dem ausgelesen werden kann wie viele Vertices (und Normalen) und Dreiecke für das Mesh erzeugt werden. Der eigentliche Compute Shader berechnet das letztendliche Chunkmesh in drei aufeinanderfolgenden Pässen:

- 1. Sample:** Sammeln und Speichern der benötigten Informationen an allen Gitterpunkten des Chunks.
- 2. Create:** Berechnen und Speichern der Vertices und zugehörigen Normalen jeder Gitterzelle und Assoziieren der Vertices mit den entsprechenden Zellkanten.
- 3. Connect:** Verbinden der erzeugten Vertices zu den Dreiecken des Chunkmeshes.

Ein Flussdiagramm, welches die groben Abläufe innerhalb der drei Pässe des Compute Shaders zusammenfasst, ist in Abbildung 33 im Anhang dieser Arbeit enthalten.

Sample

Das Samplen der Gitterpunkte innerhalb des Chunks geschieht im ersten der drei Pässe separat für jeden einzelnen Gitterpunkt. Um zu berechnen, an welcher Stelle ein Sample-Thread die Funktion auswerten sollte, wird diese Position mittels des an den Thread übergebenen id-Parameters berechnet. Aus diesem Parameter kann der Index des Threads ausgelesen werden. Über den Index kann dann wiederum ein lokaler dreidimensionaler Offset berechnet werden. Dazu werden aus dem Index die einzelnen Komponenten des Offsets abgeleitet. Der Ablauf dieser Berechnung ist im Code der oberen Funktion in Abbildung 12 erkennbar. Die Sampleposition kann nach der Berechnung des lokalen Offsets über die Multiplikation dieses Offsets mit der Größe des Chunks und darauffolgender Addition des entstehenden Vektors auf die Chunkposition erfolgen.

Das eigentliche Sampling in diesem Schritt erfolgt anschließend in einer weiteren Funktion. Als Eingabeparameter erhält diese Funktion den Threadindex und die berechnete Sampleposition. Da es sich bisher bei letzterem um eine Integer-Position handelt, wird diese als erstes um einen kleinen Offset verschoben, damit in der Terrainfunktion keine Probleme im Zusammenhang mit Perlin Noise auftreten. Um den gesuchten Funktionswert zu erhalten kann die Terrainfunktion direkt an der abgeänderten Sampleposition evaluiert werden. Für die Berechnung der Normalen wird ebenfalls der Gradient an dieser Position benötigt. Um diesen zu erhalten, wird dieselbe Methode, die auch in Marching Cubes Anwendung findet, verwendet. Es werden also die Funktionswerte bestimmter benachbarter Samplepunkte berechnet und voneinander subtrahiert. Der dadurch entstehende Gradient wird daraufhin normalisiert und anschließend zusammen mit dem eigentlichen Funktionswert im Buffer für die Samples gespeichert.

```
int3 GetSamplePosition(uint idx)
{
    //SAMPLE_SIZE = 33, SAMPLE_SIZE_SQUARED = 1089
    uint3 localSamplePosition;
    localSamplePosition.z = idx / SAMPLE_SIZE_SQUARED;
    uint idxNoZ = idx - (localSamplePosition.z * SAMPLE_SIZE_SQUARED);
    localSamplePosition.y = idxNoZ / SAMPLE_SIZE;
    localSamplePosition.x = idxNoZ % SAMPLE_SIZE;
    return chunkPosition + localSamplePosition * chunkSize;
}

void AddTerrainSample(uint idx, int3 position)
{
    //SAMPLE_OFFSET = float3(0.0143f, 0.0143f, 0.0143f)
    float3 samplePosition = position + SAMPLE_OFFSET;
    float3 offsetX = float3(chunkSize, 0.0f, 0.0f);
    float3 offsetY = float3(0.0f, chunkSize, 0.0f);
    float3 offsetZ = float3(0.0f, 0.0f, chunkSize);

    float value = GetTerrainValue(samplePosition);

    float3 gradient;
    gradient.x = GetTerrainValue(samplePosition - offsetX) - GetTerrainValue(samplePosition + offsetX);
    gradient.y = GetTerrainValue(samplePosition - offsetY) - GetTerrainValue(samplePosition + offsetY);
    gradient.z = GetTerrainValue(samplePosition - offsetZ) - GetTerrainValue(samplePosition + offsetZ);
    gradient = normalize(gradient);

    sampleBuffer[idx] = float4(gradient, value);
}
```

ABBILDUNG 12: DER CODE DER BEIDEN FUNKTIONEN ZUM BERECHNEN DER SAMPLEPOSITION UND DEM EIGENTLICHEN SAMPLEN AN DIESER POSITION.

Create

Im zweiten Pass werden die Meshvertices und zugehörigen Normalen berechnet. Als erstes werden in diesem Pass die im vorherigen Pass berechneten acht Samples der Eckpunkte aus dem Samplebuffer gelesen. Der id-Parameter des Threads ist diesmal nicht wie im vorherigen Pass eindimensional, sondern dreidimensional. Folglich können die Positionskomponenten der Daten im Samplebuffer über die komponentenweise Multiplikation des Threadindexes mit einem entsprechenden Offsetvektor berechnet werden. Der eigentliche Index der Eckpunktdata entsteht aus der Summierung der entsprechenden Positionskomponenten. Mit der Hilfe dieser Samples kann daraufhin, über die Marching Cubes typische Veroderung der klassifizierten Eckpunktwerthe, der Bitcode für die Konfiguration der Zelle bestimmt werden. Über diesen Bitcode findet nun die Ermittlung der benötigten Vertices statt. Die Tabellen, die in diesem Zusammenhang verwendet werden, basieren auf der Tabelle von Bostanzo¹². Die originale Kantentabelle von Bostanzo ist mit dem Wert -1 aufgefüllt, um immer Einträge der Länge 16 zu erhalten. Damit diese Tabelle weniger Platz benötigt wird sie in dieser Arbeit in zwei Tabellen aufgeteilt. Die erste dieser Tabellen enthält für jede Zellkonfiguration den Start- und Endoffset der Informationen für die Vertexkonstruktion einer bestimmten Konfiguration. Die zweite Tabelle enthält diese Konstruktionsinformationen in der Form der Indices der an einem Vertex beteiligten Kanten. Da eine Konfiguration mehrere Vertices enthalten kann, werden die Informationen der jeweiligen Vertices innerhalb dieser Tabelle mit dem ungültigen Kantenindex 12 getrennt. Wie auch bei Marching Cubes entsprechen die Zellkonfigurationen 0 und 255 in Nielsons Dual Marching Cubes einer Zelle, die sich komplett außer- oder innerhalb des Terrains befindet und entsprechend leer ist.

Um die Vertices einer Zelle zu berechnen, werden zunächst die Start- und Endoffsets mit dem Bitcode der Konfiguration aus der Offsettabelle gelesen. Im Anschluss findet eine Iteration über den zwischen diesen beiden Offsets liegenden Bereich statt. In jedem Iterationsschritt wird der Index der als nächstes zu behandelnden Kante durch den iterierten Offset und die Kantentabelle bestimmt. Ist der Index kleiner als 12, also der Index einer wirklichen Kante, wird die Vertexposition und die Vertexpnormale entlang dieser Kante aus den Kantenendpunkten und den Samples an diesen Punkten interpoliert. Sowohl die Position als auch die Normale werden anschließend auf die entsprechenden Vektoren vorheriger Iterationen addiert. Neben der Addition des Kantenbeitrags muss die Kante ebenfalls mit dem entsprechenden Vertex assoziiert werden. Da jede Zelle so gesehen nur drei Kanten enthält muss zunächst die Information der Kantengruppe, in der die behandelte Kante enthalten ist, gefunden werden. Dazu zählt einerseits die Bufferposition der Kantengruppe und andererseits die Indices der entsprechenden Kante und des Vertex innerhalb dieser Kantengruppe. In der Praxis finden dazu zwei weitere Tabellen Anwendung. Die erste dieser Tabellen enthält einen Offset, der auf die Position der aktuellen Zelle addiert werden muss, um für eine bestimmte Kante die Position der benötigten Kantengruppe im zugehörigen Buffer zu erhalten. Entspricht der Index der Kante 0, 3 oder 8 ist der Offset ein Nullvektor, da diese Kante in der Kantengruppe der momentanen Zelle enthalten ist. Sowohl der lokale Index der Kante innerhalb der Kantengruppe als auch die Position, an der der Vertex in dieser Kantengruppe gesetzt werden muss, müssen nicht dynamisch berechnet werden und sind daher statisch in der zweiten Tabelle abgebildet.

Da nun die genaue Zugehörigkeit des Vertex innerhalb einer Kantengruppe bekannt ist, kann versucht werden den Vertex mit der entsprechenden Kante zu assoziieren. Dies kann jedoch nur stattfinden, wenn die Assoziation des Vertex mit der Kante gültig ist. Es gibt zwei Gründe, warum dies nicht der Fall sein kann. Der erste Grund ist die fehlende Existenz der Zelle, in der die Kantengruppe enthalten wäre. Kantengruppen bestehen immer aus den drei Kanten, welche sozusagen aus dem Ursprung einer Gitterzelle ausgehen. Alle anderen

¹² Dual Marching Cubes Tabelle - <https://stackoverflow.com/questions/16638711/dual-marching-cubes-table>

Zellkanten sind Teil der Kantengruppen benachbarter Zellen. Falls beispielsweise eine Zelle am Rand des Chunks in der positiven X-Richtung bearbeitet wird, kann es daher dazu kommen, dass ein Vertex mit Kanten die den Index 2, 6, 10 oder 11 besitzen assoziiert werden sollte. Die entsprechende Kante wäre also ein Teil einer Kantengruppe außerhalb des eigentlichen Chunks und würde folglich nicht im Buffer existieren. Der zweite Grund für die Ungültigkeit einer Assozierung entsteht an Chunkseiten entlang der negativen Achsenrichtungen. Die Kantengruppen sind an diesen Seiten zwar immer vorhanden, tragen aber nicht immer zum letztendlichen Mesh bei. Wird zum Beispiel die Zelle im Ursprung des Chunks bearbeitet, existieren alle Zellkanten auch in einer Kantengruppe im Buffer und können daher theoretisch mit einem Vertex assoziiert werden, aber nur Kanten, welche nicht entlang des Chunkrandes liegen, können überhaupt Dreiecke im letztendliche Mesh erstellen. Vertices die ausschließlich auf den im Chunkrand enthaltenen Kanten liegen, würden in diesen Fällen zwar generiert werden und folglich im erzeugten Mesh enthalten sein, aber nicht zu dessen Dreiecken beitragen. Ein Vertex kann also nur mit einer Kante assoziiert werden, wenn keiner dieser Fälle auftritt. Dies führt jedoch zu einem weiteren Problem: Da Vertices an den Chunkrändern nicht zwangsweise generiert werden, kann der Vertex nicht direkt erstellt werden, sobald er in einer Kante enthalten ist. Um dieses Problem zu umgehen, werden die Vertices einer Zelle erst am Ende ihrer Bearbeitung erstellt. Damit deshalb keine weitere Iteration über die beteiligten Zellkanten nötig ist, um sie mit dem Index des erzeugten Vertex zu assoziieren, wird ein Index im Vertexbuffer reserviert, sobald eine gültige Kantenassozierung existiert. Alle späteren an einem Vertex beteiligten Kanten werden mit diesem reservierten Vertex verbunden. Um später erkennen zu können, wie viele der maximal vier Vertices mit einer Kante verbunden sind, wird der Vertexindex beim Assoziieren inkrementiert. Für nicht gesetzte Vertices steht in den Daten der Kantengruppe entsprechend also eine 0. Bei der Generierung der Dreiecke wird der Vertexindex wieder dekrementiert, um den tatsächlichen Index zu erhalten.

```
[numthreads(4, 4, 4)]
void Create(uint3 id : SV_DispatchThreadID)
{
    float4 samples[8];
    GetSamples(id, samples);
    uint cellCase = GetCellCase(samples);
    if (cellCase == 0 || cellCase == 255) { return; }

    float4 vertexPosition = 0.0f;
    float3 vertexNormal = 0.0f;
    int vertexIndex = -1;
    uint2 edgeTableOffset = DMC_EDGE_TABLE_OFFSETS[cellCase];
    for (uint offset = edgeTableOffset.x; offset < edgeTableOffset.y; offset++)
    {
        uint currentEdge = DMC_EDGE_TABLE[offset];
        if (currentEdge < 12)
        {
            float3 interpolatedPosition;
            float3 interpolatedNormal;
            InterpolatePositionAndNormal(DMC_EDGE_CORNERS[currentEdge], samples, interpolatedPosition, interpolatedNormal);
            vertexPosition += float4(interpolatedPosition, 1.0f);
            vertexNormal += interpolatedNormal;

            uint2 edgeAndVertexIndex = DMC_EDGE_VERTEX_INDEX_MAPPING[currentEdge];
            uint3 edgeGroupPosition = id + DMC_EDGE_OFFSETS[currentEdge];
            vertexIndex = TryAddVertexToEdge(edgeGroupPosition, edgeAndVertexIndex.x, edgeAndVertexIndex.y, vertexIndex);
        }
        else
        {
            TryAddVertexAndNormal(vertexIndex, (id + (vertexPosition.xyz / vertexPosition.w)) * chunkSize, normalize(vertexNormal));

            vertexPosition = 0.0f;
            vertexNormal = 0.0f;
            vertexIndex = -1;
        }
    }
    TryAddVertexAndNormal(vertexIndex, (id + (vertexPosition.xyz / vertexPosition.w)) * chunkSize, normalize(vertexNormal));
}
```

ABBILDUNG 13: DER CODE DES ZWEITEN PASSES ZUM ERSTELLEN DER GRUNDELGENEN MESHINFORMATIONEN.

Sobald alle zu einem Vertex beitragenden Kanten abgearbeitet sind, aber noch weitere Vertices folgen, wird bei der Iteration über die Kantentabellenoffsets ein ungültiger Kantenindex in der Form von 12 gelesen. Falls ein Vertex erzeugt worden ist, kann dessen Position und Normale nun in den Vertex- und Normalenbuffer eingefügt werden. Die Vertexposition berechnet sich aus der Division der gebildeten Summe der interpolierten Kantenpositionen aus vorherigen Iterationen durch die Anzahl an beitragenden Kanten. Die Vertexpnormale kann einfach über die Normalisierung der summierten Normalen erzeugt werden. Nachdem die Iteration den Endoffset erreicht hat, folgt kein ungültiger Kantenindex mehr. Damit aber auch die Informationen des letzten potenziellen Vertex gespeichert werden, wird am Ende ebenfalls getestet, ob ein Vertex erzeugt worden ist und entsprechend dessen Informationen gespeichert werden müssen.

Connect

Mit der Berechnung der Vertexinformationen ist der aufwändigste Pass erledigt. Im letzten Pass müssen die nun vorhandenen Information nur noch genutzt werden, um die erzeugten Vertices zu Dreiecken zu verbinden. Jede Kante, die mit genau vier Vertices assoziiert ist, erzeugt hierbei ein Quad bzw. zwei Dreiecke. In diesem Pass werden dazu alle Kantengruppen betrachtet. Jede der drei Kanten in einer solchen Gruppe wird separat behandelt. Sollten alle vier mit einer Kante assoziierten Vertexindices ungleich 0 sein bedeutet dies, dass alle vier Vertices gesetzt sind und zwei Dreiecke erzeugt werden können. Um die Umlaufrichtung der beiden Dreiecke zu bestimmen wird der Funktionswert im Zellursprung benötigt. Liegt dieser Funktionswert unterhalb des Isolvels, muss die Reihenfolge der Vertices bei der Dreieckserzeugung umgedreht werden. Aufgrund der Reihenfolge, in der die Vertexindices innerhalb der Kantengruppe gespeichert werden, ist die Reihenfolge, in der sie in den Dreiecksbuffer eingefügt werden müssen, dieselbe für alle drei Kanten. Entsprechend kann diese statisch in der Funktion festgelegt werden. Damit verhindert wird, dass die Vertexindices aufgrund der parallelen Bearbeitung der Kantengruppen, in der falschen Folge in den Buffer geschrieben werden, werden vor dem Einfügen immer sechs aufeinanderfolgende Plätze im Buffer reserviert.

7.3 CHUNKS

Mit dem beschriebenen Compute Shader kann nun die Meshinformation für einen Chunk erstellt werden. Sowohl das eigentliche Aufrufen des Compute Shaders als auch die Darstellung der erzeugten Informationen müssen aber noch in Unity stattfinden. In der Flugsimulation dienen dazu Chunk-Objekte in der Spielwelt. Jedes dieser Objekte verwaltet die Informationen und Funktionalitäten eines Chunks. Wenn die Aktualisierung des Chunkmeshes angefordert wird, erstellt eine Komponente des Chunk-Objekts die sechs Buffer des Compute Shaders und setzt diese entsprechend für die drei Pässe. Die Parameter, die in diesem Zusammenhang beschrieben werden sollten, sind die Anzahlen an Threadgruppen, die beim Absenden der Pässe festgelegt werden. Wie bei der vorherigen Beschreibung der einzelnen Pässe erwähnt, kann der id-Parameter den die Pässe als Eingabe erhalten, ein- bis dreidimensional sein. Der Wertebereich dieses Parameters wird je Dimension über die Anzahl der Threadgruppen multipliziert mit der Anzahl der Threads festgelegt. Während die Anzahl an Threads im Compute Shader über das numthreads-Attribut gesetzt wird, beschreibt die Anzahl der Threadgruppen beim Absenden des Passes, wie viele Gruppen mit dieser Threadzahl verwendet werden sollen. Da im ersten Pass 33^3 Positionen gesampelt werden ist es hierbei von Vorteil nur eine Dimension dieser Threads zu nutzen. Insgesamt gibt es dabei 1123 Gruppen mit je 32 Threads. Damit die letzte Position in dieser Konfiguration ebenfalls gesampelt wird, wird der Pass für diese am Ende des ersten Threads durchgeführt. Das ist nicht optimal, aber es wird verhindert das eine weitere Threadgruppe verwendet werden muss, bei der 31 Threads nicht direkt arbeiten. Für den zweiten Pass werden acht Gruppen und auf der GPU je vier Threads pro Dimension angefordert. Damit wird der gesamte Bereich der Zellen bzw. der Kantengruppen des Chunks abgedeckt. Dieselbe Konfiguration wird für den letzten Pass verwendet.

Nachdem die drei Pässe durchlaufen sind, müssen die Meshinformationen des Chunks auf der CPU ausgelesen werden. Das Hauptproblem hierbei ist, dass die GPU eine zwar kurze, aber dennoch vorhandene Zeit benötigt, um die Meshdaten zu berechnen. In der Praxis wäre es schlecht, wenn die CPU während dieser Zeit auf die Ergebnisse warten würde. Um dieses Problem zu lösen, werden die Daten asynchron zurückgelesen. Dazu wird zuerst der Buffer, welcher die Anzahl an Vertices und Dreiecken des Meshes enthält, angefordert. Sobald dieser zur Verfügung steht, ist bekannt wie viele Werte aus den weiteren Buffern gelesen werden müssen. Entsprechend können diese darauffolgend ebenfalls ausgelesen werden. Wenn die eigentlichen Meshinformationen dieser Buffer zur Verfügung stehen, werden diese als entsprechende Arrays im Mesh des Chunks gesetzt.

Für die Flugsimulation ist es ebenfalls wichtig, dass der Spieler mit den erstellten Chunkmeshen kollidieren kann. Da für die Kollisionen Meshcollider verwendet werden, welche nicht die effizientesten Collider darstellen, und da, dank des im Anschluss beschriebenen LOD-Systems, die Chunks in der Nähe des Spielers immer höher Samplingauflösungen besitzen, ist es nur für die kleinsten Chunks nötig einen Collider zu besitzen. Folglich wird, nachdem die Aktualisierung des Meshes abgeschlossen ist, die Kollision nur bei Chunks mit der Größe 1 und 2 aktiviert.

7.4 LEVEL OF DETAIL

Die Landschaft könnte nun durch eine Reihe an Chunks dargestellt werden. In der letztendlichen Flugsimulation wird ein Bereich von jeweils 1536 Einheiten entlang der X- und Z-Achsen und bis zu 512 Einheiten entlang der Y-Achse dargestellt. Sollte dieser Bereich komplett mit Chunks der höchsten Auflösung (32 Einheiten und Zellen je Achse) dargestellt werden, wären 36 864 dieser Chunks nötig. Je nach der verwendeten Hardware ist eine solche Anzahl eventuell noch stemmbar. Sobald sich der Spieler jedoch bewegen sollte, müssen entsprechend viele weitere Chunk generiert und alte Chunks freigegeben werden. In der Praxis funktioniert dieser Ansatz also nicht wirklich. Was stattdessen benötigt wird ist ein Level-Of-Detail-Mechanismus (LOD), welcher die Größen der Chunks in Abhängigkeit des Spielers dynamisch anpasst. Beispielsweise müssen weit vom Spieler entfernte Chunks nicht die höchste Auflösung besitzen, da das Detail der Landschaft aufgrund der Entfernung keinen zwingenden Mehrwert für den Spieler bietet. Das Gleiche gilt für Chunks, die außerhalb des Sichtfeldes des Spielers liegen. Diese werden ohnehin nicht gerendert und würden im besten Fall erst gar nicht generiert werden müssen. Das LOD-System dieser Arbeit verwendet einen Octree, bzw. eine Reihe an Octrees, als Basis für die Umsetzung dieses Mechanismus. Der Bereich um den Ursprung der Spielwelt wird dazu in neun Teilbereiche aufgeteilt, mit dem Ursprung in der Mitte. Jeder dieser Teilbereiche deckt einen Bereich von 512 Einheiten entlang jeder der drei Hauptachsen ab. Des Weiteren wird für jeden Teilbereich ein einzelner Octree definiert. Insgesamt wird der Spielbereich also über neun Octrees dargestellt, welche sich entlang der X- und Z-Achse erstrecken. Die Blätter dieser Octrees enthalten die Informationen für die letztendliche Erzeugung der einzelnen Chunks, die zu einem bestimmten Zeitpunkt für die Spielwelt benötigt werden. Die Aufteilung des Spielbereiches in mehrere Teilbereiche löst einige Probleme, welche bei der Darstellung des Bereiches über einen einzigen Octree auftreten würden. Da der Spieler die Schluchten der Flugsimulation endlos erkunden können sollte, müsste der Octree dem Spieler nach und nach folgen. Dies könnte theoretisch in Schritten der minimalen Chunkgröße erfolgen. Dadurch würden die Blätter des Octrees weiterhin den Chunks der Spielwelt entsprechen. Dies würde allerdings auch bedeuten, dass bei einem solchen Verschiebungsschritt im schlechtesten Fall alle Chunks der Spielwelt neu berechnet und im besten Fall der Octree neu unterteilt und alle Chunks neu zugeordnet werden müssten.

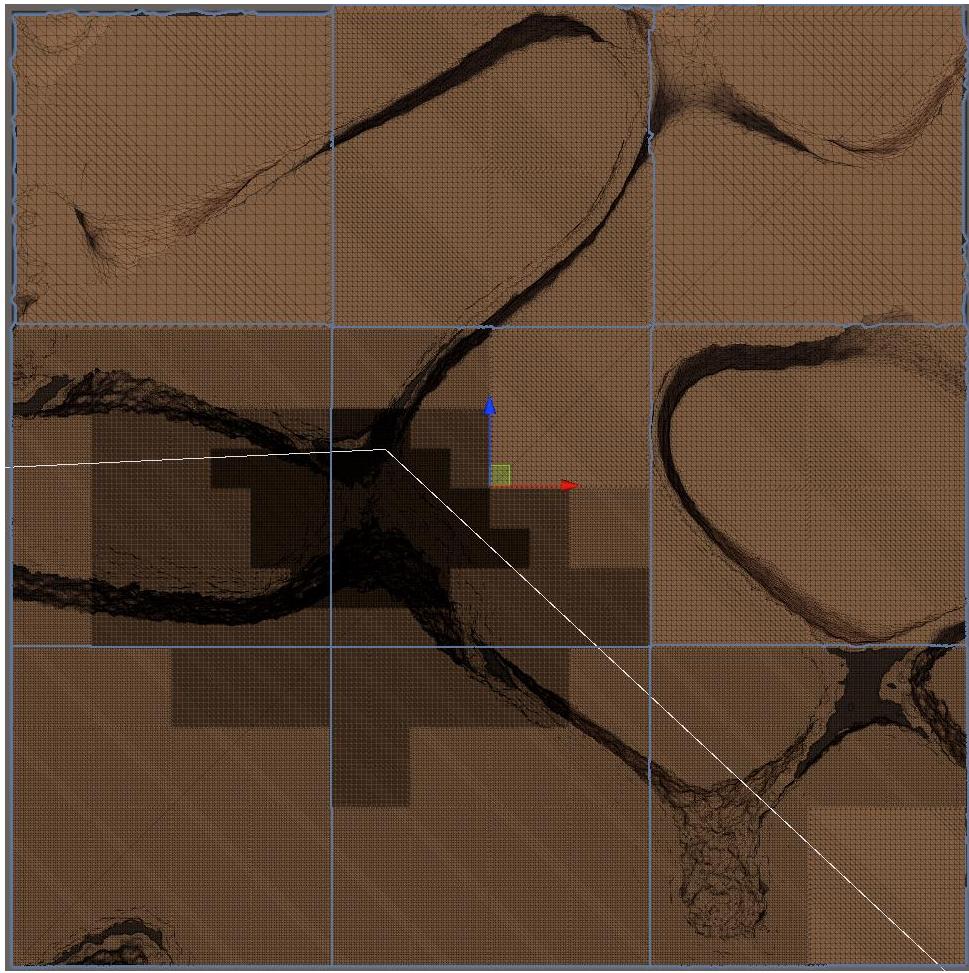


ABBILDUNG 14: DIE UNTERTEILUNG DER SPIELWELT IN NEUN TEILBEREICHE INKLUSIVE DER UNTERSCHIEDLICHEN AUFLÖSUNGEN VON CHUNKS JE NACH DER BLICKRICHTUNG DES SPIELERS.

Ein erster Implementationsansatz der Teilbereiche wäre es, den Spieler immer in einem mittleren Octree umschlossen von den restlichen acht zu halten. Theoretisch müssen dadurch bei der Bewegung des Spielers immer nur diejenigen Octrees verschoben werden, welche sich nun zu weit entfernt befinden. Würde sich der Spieler beispielsweise entlang der positiven X-Richtung bewegen und den mittleren Octree verlassen, könnte der Octree der nun betreten wird der neue mittlere Octree werden. Alle Octrees, die sich nicht in der direkten Nachbarschaft dieses neuen Mittelpunktes befinden, könnten freigegeben und im Anschluss wiederverwendet werden, um die nun fehlenden Octreepositionen aufzufüllen. Theoretisch könnte sich der Spieler bei einer solchen Implementation also frei durch die Spielwelt bewegen und die Octrees (und damit auch der Spielbereich) würden diesem ohne Probleme folgen.

In der Praxis funktioniert dieser Ansatz auch relativ gut, bis eine bestimmte Distanz zum Ursprung der Welt erreicht ist. Ab diesem Punkt macht sich ein weiteres Problem bemerkbar. Ab einer gewissen Distanz treten mit zunehmender Entfernung vom Ursprung der Welt merkbare und immer stärker werdende Verzerrungen in Kombination mit zunehmenden Jittereffekten in den gerenderten Objekten auf. Der Grund hierfür ist Unitys Verwendung von 32-Bit-Floats für die Speicherung der Objektkoordinaten. Die Präzision von Float-Werten nimmt mit dem steigenden Betrag der dargestellten Zahlen immer weiter ab. In vielen Spielen, in denen der Spieler zu Fuß unterwegs ist, stellt diese limitierte Präzision kein Problem dar, da die Welten klein genug sind, um nicht von dieser betroffen zu

sein. In prozedural generierten Welten hingegen, und vor allem auch durch das Fliegen in dieser Flugsimulation, kann der Spieler in einer Spielsession viel größere Distanzen zurücklegen und damit auch relativ leicht den Schwellwert erreichen, ab dem sich die verringerte Präzision bemerkbar macht.

Eine Möglichkeit dieses Präzisionsproblems zu lösen, nennt sich *Floating Origin*. Es gibt verschiedene Arten und Weisen wie diese Methode umgesetzt werden kann. Einige Ansätze werden beispielsweise von Thorne (2005) beschrieben. Die Basis dieser Methode ist es jedoch immer, die Position von Objekten auf eine bestimmte Art und Weise anzupassen damit sich das Präzisionslimit nicht auf die Darstellung auswirkt. In dieser Arbeit wird diese Anpassung mittels eines Offsets für den tatsächlichen Ursprung realisiert. Die grundlegende Idee hinter diesem Offset ist es, dass es für alle Objekte der Flugsimulation im Endeffekt egal ist an welcher Position sie sich letztendlich befinden, solange ihre theoretisch tatsächliche Position für bestimmte Berechnungen ermittelt werden kann. Der Offsetvektor selbst wird dabei als Integer-Position dargestellt und gespeichert. In diesem Ansatz wird der Spieler ebenfalls immer im Bereich des mittleren Octrees gehalten. Anstatt aber die Octrees und den Spielbereich beim Überschreiten der Bereichsgrenzen zu bewegen, werden alle anderen Objekte um die Größe eines Octrees in die entgegengesetzte Richtung verschoben. Dadurch spielt sich die komplette Flugsimulation immer in der Nähe des Weltursprungs ab und es wird nie der Punkt erreicht, ab dem die Präzision der Objektpositionen Probleme verursachen würde.

Diese Vorgehensweise wirkt sich ebenfalls auf die Konstruktion der Octrees aus. Eigentlich müssten die einzelnen Octrees bei jeder Terrainaktualisierung anhand der Spielerposition gegebenenfalls verschoben und größtenteils erneut aufgebaut werden. Vor allem durch letzteres entsteht Aufwand für die Berechnung der Zellinformation innerhalb des jeweiligen Octrees. Bei der Verwendung eines Ursprungsoffsets müssen die Octrees hingegen nicht bewegt werden. Die eigentlichen Informationen der Zellen ändern sich folglich ebenfalls nicht. Das Erzeugen von Octreezellen bei der Aktualisierung würde also immer wieder dieselben Informationen berechnen. Anstelle dessen werden die Octrees der Flugsimulation am Anfang komplett unterteilt. Dadurch wird zwar insgesamt mehr Speicherplatz benötigt, aber die wiederholte Berechnung der meisten Zellinformationen wird verhindert. Da folglich durch die Existenz von Kinderzellen nichtmehr hervorgeht, ob es sich bei einer Zelle um ein Blatt handelt, wird dies je Zelle zusätzlich gespeichert.

In der Implementation der Flugsimulation bildet sich für die Erstellung und Aktualisierung des Terrains daraus der folgende grobe Ablauf:

- 1.** Aktualisiere den Weltursprung
- 2.** Aktualisiere die Octrees aller Teilbereiche
- 3.** Gib die Chunks aller alten und nun nichtmehr existierenden Blätter der Octrees zur Löschung frei
- 4.** Aktualisiere die Chunks aller Octreeblätter die weiterhin existieren und erstelle Chunks für alle neuen Blätter

Das Aktualisieren des Weltursprungs basiert auf der Position des Spielers bzw. der Kamera. Da sich die Flugsimulation größtenteils in der XZ-Ebene abspielt wird in diesem Schritt zuerst berechnet, ob sich der Spieler entlang der X- oder Z-Achse außerhalb des mittleren Octrees befindet. Dies kann durch die Division der Spielerposition mit der Größe des Octrees geschehen. Die entstehenden Werte sagen zunächst aus, wie viele Octrees sich der Spieler entfernt vom Ursprung befindet. Da der Ursprung in der Mitte des Octrees liegt muss der Offset angepasst werden, sobald der Spieler die halbe Octreegröße überschreitet. Mittels einer Rundung auf den nächsten Integer-Wert entstehen die letztendlichen Faktoren mit der die Octreegröße multipliziert werden muss, um die korrekte Verschiebung des Ursprungsoffsets zu erhalten. Nach der Berechnung wird ein Event ausgelöst, mit dem alle Objekte die Verschiebung vornehmen können.

Während die Octrees selbst nicht bewegt werden, sind die generierten Chunks hingegen schon vom Ursprungsoffset betroffen. Folglich müssten die vorhandenen Chunks nach der Aktualisierung des Offsets ebenfalls verschoben werden. Damit aber nicht jeder Chunk einzeln behandelt werden muss, existieren sogenannte Chunk Nodes, mithilfe denen Chunks gruppiert werden. Für jeden der neun Octrees existiert dabei eine entsprechende Chunk Node, welche die zu den Blättern gehörenden Chunks enthält. Sobald der neue Offset berechnet ist, müssen also nicht hunderte Chunks verschoben werden, sondern immer nur die neun Chunk Nodes. Die Chunks einer Chunk Node werden direkt gelöscht, sobald sich die Node durch die Verschiebung außerhalb des 3x3-Bereiches um den Spieler befinden würde. Die Chunks aller anderen Chunk Nodes werden in eine Liste von potenziell zu löschen Chunks eingefügt. Der Grund dafür wird in Kürze behandelt.

Im Anschluss zur Berechnung des neuen Ursprungsoffsets werden die neun Octrees aktualisiert. Als Ausgabe entstehen zwei Listen mit Octreezellen. Diese enthalten jeweils die nach der Aktualisierung vorhandenen und gelöschten Blattzellen. Als Eingabe dient die Kamera und die Ebenen des Kamerafrustums. Für jede Zelle der Octrees wird mit diesen Daten rekursiv entschieden, ob sie weiter unterteilt werden sollte. Im Falle, dass sich eine Zelle weiter unterteilen sollte, werden die Kinder der Zelle aktualisiert. Davor wird aber zunächst geprüft, ob es sich bei der aktuellen Zelle bereits um ein Blatt handelt. Ist dies der Fall wird sie in die Liste der gelöschten Blätter eingefügt. Falls sich die Zelle nicht weiter unterteilen sollte, wird sie in die Liste der vorhandenen Blätter eingefügt. Handelt es sich bei dieser Zelle bereits um ein Blatt ist das Ende der Rekursion erreicht. Ist die aktuelle Zelle allerdings kein Blatt sind Blätter in den Kinderzellen enthalten. Diese müssen entsprechend behandelt werden. Der rekursive Durchlauf dieser Zellen endet, sobald die Blätter erreicht sind und in die Liste der gelöschten Blätter eingefügt sind. Für alle Zellen, die auf dem Weg zu diesen Blättern durchlaufen werden, wird zusätzlich ein weiterer Boolean für die spätere Nutzung gesetzt, welcher angibt, dass sich das Blatt dieses Teilbaumes in einer vorherigen Zelle des Octrees befindet.

Die eigentliche Unterteilungsentscheidung einer Zelle, wird in mehreren Schritten getroffen. Hat die aktuelle Zelle die minimale Zellgröße erreicht, darf diese nicht weiter unterteilt werden. Dazu kommt, dass die Schluchten des Terrains der Flugsimulation maximal etwas über 100 Einheiten hoch sind. Entsprechend kann die Unterteilung der Octreezellen ebenfalls gestoppt werden, wenn sich die Zelle komplett ober- oder unterhalb des Terrains befindet und daher nie zu diesem beitragen kann. Hat die Zelle noch nicht die minimale Größe erreicht und befindet sich ebenfalls im Generierungsbereich des Terrains bestimmt sich der Unterteilungsgrad basierend auf der Distanz zwischen dem Mittelpunkt der Zelle und der Position der Kamera. Ist diese Distanz kleiner als ein bestimmter Wert wird die Zelle weiter unterteilt. Dieser Vergleichswert setzt sich aus der Größe der Octreezelle multipliziert mit einem Gewichtungsfaktor zusammen. Befindet sich die Octreezelle komplett oder zumindest teilweise im Sichtfeld des Spielers entspricht der Gewichtungsfaktor 4, in allen anderen Fällen 1. Die Sichtbarkeit der Octreezellen wird mithilfe der übergebenen Frustumsebenen und der Ausmaße der Octreezelle geprüft.

Mit den beiden aus der Octreeunterteilung entstehenden Listen können nun die Terrainchunks aktualisiert werden. Zunächst werden die zu entfernten Blättern gehörenden Chunks zur Löschung freigegeben. Als nächstes werden die weiterhin vorhandenen und neuen Blätter durchlaufen, um die benötigten Chunks zu aktualisieren. Alle bereits vorhandenen Chunk, für die weiterhin ein Blatt existiert, können aus der Liste der potenziell zu löschen Chunks entfernt werden, da sie weiterhin existieren. Sollte kein passender Chunk vorhanden sein, muss ein neuer Chunk mit den Daten der zugehörigen Zelle angefordert werden. Aufgrund der Unterteilung kann es zusätzlich dazu kommen, dass ein Chunk an einer bestimmten Position vorhanden ist, er aber eine unterschiedliche Größe zu der zugehörigen Zelle besitzt. Die Auflösung des Chunks stimmt also nicht mehr und er deckt einen zu großen oder zu kleinen Bereich ab. Entsprechend muss in diesem Fall der alte

Chunk ebenfalls zur Löschung freigegeben und ein neuer mit den richtigen Daten angefordert werden.

Nach der Abarbeitung der beiden Blätterlisten kann es vorkommen, dass einige Chunks existieren, welche eigentlich nichtmehr vorhanden sein sollten. Der Grund dafür entsteht bei der Verschiebung der Chunk Nodes. Um zu verhindern, dass ein Octree bei dieser Verschiebung jedes Mal die Unterteilung eines benachbarten Octrees kopieren muss, um weiterhin Blätter zu besitzen, die mit den momentanen Chunks korrespondieren, werden nur die entsprechenden Chunk Nodes verschoben. Aufgrund der Verschiebung werden die betroffenen Octrees bei der darauffolgenden Unterteilung jedoch nicht zwingend zu demselben Grad unterteilt. Entsprechend fehlen eventuell Zellen mit dessen Positionen daraufhin Chunks referenziert und gelöscht werden können. Ein Weg dies zu umgehen wäre es die Octrees in diesem Fall zuerst weit genug zu unterteilen, damit dieser Fall nicht auftritt, und daraufhin den korrekten Unterteilungsgrad zu bestimmen. Dabei müsste jeder betroffene Octree allerdings mehr oder weniger zwei Mal unterteilt werden. In dieser Implementation existiert stattdessen die zuvor erwähnte Liste mit Chunk, die potenziell gelöscht werden müssen. Nach der Verschiebung der Chunk Nodes enthält diese alle weiterhin existierenden Chunks. Nach der Iteration über die beiden Blätterlisten der Octreeaktualisierung enthält diese Liste dann nur noch die Chunks, auf die keine direkte Referenz durch Octreeblätter mehr vorhanden ist. Folglich kann diese Liste durchlaufen werden, um die restlichen überflüssigen Chunks zu löschen.

Das Löschen und Aktualisieren von Chunks, in den beschriebenen Schritten, erfolgt nicht sofort. Da die Chunkmeshes auf der GPU erzeugt werden, würde es beim sofortigen Löschen dazu kommen, dass kurzzeitig kein Mesh für den Ersatzchunk vorhanden wäre. Entsprechend wären ständig viele Löcher im Terrain zu sehen, sobald sich das LOD-Level der Chunks ändert. Stattdessen werden alle dieser Chunks entsprechend der nötigen Aktionen in zwei Listen gespeichert. Das Löschen der alten und Anzeigen der neuen Chunkmeshes erfolgt erst in einem Satz, sobald die Ergebnisse aller Compute Shader der zu aktualisierenden Chunks zur Verfügung stehen. Die alten Chunk-Objekte werden ebenfalls nicht wirklich gelöscht und neu benötigte Chunk-Objekte auch nicht komplett neu erzeugt. Das ständige Löschen und Erzeugen von vielen Objekten würde sich negativ auf die Framerate der Flugsimulation auswirken. Stattdessen werden ungenutzte Chunk-Objekte deaktiviert und in einen Pool verschoben. Bei der Anforderung eines neuen Chunks kann dann folglich einfach ein Chunk-Objekt aus diesem Pool genutzt werden.

7.5 SEAMS

Nach den bisherigen Schritten ist die Erzeugung eines Chunks jedoch noch nicht abgeschlossen. Ähnlich zu Dual Contouring und Schaefer und Warrens Dual Marching Cubes kommt es auch bei Nielsons Verfahren, aufgrund der Platzierung der Vertices innerhalb der gesampelten Zellen, dazu, dass die Chunkränder nicht komplett durch das erstellte Mesh abgedeckt werden. An den Übergängen, im Folgenden als Seams bezeichnet, von einem zum nächsten Chunk entstehen entsprechend also Lücken im Terrain. Wenn alle Chunks die gleiche Größe besitzen würden, wäre das Schließen dieser Lücken trivial. Die Größe der Chunks müsste in diesem Fall lediglich um eine Zelle je Achse erweitert werden. Dank der unterschiedlichen Chunkgrößen in der Flugsimulation funktioniert dieser Ansatz jedoch nicht. Die Lücken selbst würden zwar teilweise geschlossen werden, aufgrund der unterschiedlichen Abstände zwischen Sampeln würden sich die Vertices der jeweiligen Meshes jedoch immer noch nicht an denselben Positionen befinden. Um die Lücken zwischen Chunks mit unterschiedlichen Auflösungen zu schließen, gibt es daher verschiedene Methoden.

Ein erster Ansatz wäre es die Seams mittels sogenannter „Skirts“ oder „Flanges“ darzustellen (Lengyel, 2010, S. 5). Die Idee dabei ist es die benachbarten Meshes nicht direkt miteinander zu verbinden, sondern weitere Dreiecke zu verwenden, welche die Lücken zwischen den

Meshen verdecken. Ein Vorteil dieses Ansatzes ist es, dass keinerlei direkte Informationen über die benachbarten Chunks bekannt sein müssen. Als problematisch können sich jedoch die Texturierung und folgende Beleuchtung der zusätzlichen Dreiecke erweisen. Die Normalen der zusätzlichen Dreiecke stimmen meist nicht mit den eigentlichen Terrainnormalen überein und müssen zur Vermeidung von merkbaren Beleuchtungsänderungen entsprechend angepasst werden. Auch die eigentliche Platzierung der zusätzlichen Vertices erweist sich im Zusammenhang mit der Extraktion von Isoflächen als problematisch. Die Wahl der Vertexpositionen bei Heightmap-basierten Terrainmeshen fällt vergleichsweise leicht, da die Vertices beliebig weit nach unten verschoben werden können, ohne sich mit einem anderen Bereich des Terrains zu überschneiden. Können hingegen Überhänge und Höhlen entstehen ist dies nicht mehr der Fall.

Eine alternative und eventuell bessere Vorgehensweise in Kombination mit der Extraktion von Isoflächen ist daher das Erweitern des eigentlichen Meshes oder Erzeugen eines Übergangsmeshes, weitergehend Seammesh genannt, welches die Meshe benachbarter Chunks nahtlos miteinander verbindet. Je nachdem, welcher Algorithmus zur Extraktion der Isofläche verwendet wird, erweist sich diese Vorgehensweise als mehr oder weniger aufwändig. Basierend auf Marching Cubes nutzt der Transvoxel Algorithmus von Lengyel beispielsweise Übergangszellen, welche, ähnlich den Zellen in Marching Cubes, unterschiedliche Zellkonfigurationen besitzen können (Lengyel, 2010, S. 28). Jede dieser Konfigurationen definiert die Vertexverbindungen zwischen den Zellen zweier Chunks mit potenziell unterschiedlichen Auflösungen. Die Auflösung eines benachbarten Chunks kann dabei immer nur maximal doppelt und minimal halb so groß sein wie die des behandelten Chunks (Lengyel, 2010, S. 28). Durch diese Begrenzung können die möglichen Konfigurationen auf 512 Fälle beschränkt und vorberechnet in einer Tabelle abgespeichert werden. Lengyels Methode ist aufgrund dieser Vorberechnung relativ effizient. Es muss allerdings immer sichergestellt sein, dass benachbarte Chunks die richtigen Auflösungen besitzen. Auch das initiale Erzeugen der Tabellen mit den Verbindungsdaten ist aufwändig und keineswegs trivial. Neben der Verwendung von Übergangszellen könnte auch ein rekursiver Traversal-Algorithmus, wie er bei Dual Contouring oder Schaefer und Warrens Dual Marching Cubes Anwendung findet, verwendet werden, um die für das Seamesh benötigten Daten zu sammeln. Im Zusammenhang mit Nielsons Dual Marching Cubes ergibt sich dabei allerdings ein Problem aufgrund der Möglichkeit mehrere Vertices in einer Zelle zu generieren. Innerhalb einer Zelle können folglich mehrere Terrainbereiche ohne eine Verbindung zueinander enthalten sein. Zwischen diesen Bereichen kann ohne weitere Informationen jedoch nicht immer unterschieden werden, was zu eventuell falschen Dreiecken im Seammesh führen würde. Ju et al. lösen dieses Problem in ihrer Methode Manifold Dual Contouring mittels der getrennten Gruppierung von Vertices verschiedener Terrainbereiche in einem Octree (Ju et al., 2007, S. 3).

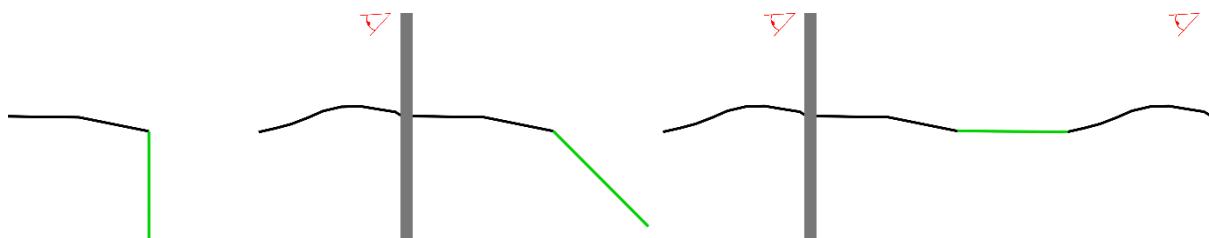


ABBILDUNG 15: DARSTELLUNG DER VERBINDUNG BENACHBARER MESHE ÜBER SKIRTS (LINKS), FLANGES (MITTE) UND EIN ÜBERGANGSMESH (RECHTS).

Der Ansatz, der in der Flugsimulation dieser Arbeit verwendet wird, macht sich hingegen eine Eigenschaft von Nielsons Methode zu nutzen, um die Lücken an den Seams zu schließen. Dabei findet ebenfalls ein Seammesh Anwendung, um die Übergänge zu erzeugen. Die Eigenschaft, die genutzt wird, ist die Zuweisung von Vertices zu den Kanten der Zellen.

Jede Zellkante kann dabei maximal mit vier Vertices benachbarter Zellen assoziiert werden und jede Zelle kann maximal einen Vertex zu einer Kante beitragen. Die Kanten an den Rändern benachbarter Chunks enthalten folglich jeweils Teilinformationen für die Verbindung der Zellvertices. Dies wird genutzt, um das letztendliche Seammesh zu erzeugen.

Sammeln der seamrelevanten Chunks

Zum Erstellen des Seammeshes werden als erstes für jeden Chunk Informationen zu den benachbarten Chunks benötigt. Um performant an diese Informationen zu gelangen kann der Octree aus dem vorherigen Kapitel verwendet werden. Jede Zelle des Octrees erhält hierfür zunächst eine ID mittels welcher direkt auf eine Octreezelle zugegriffen werden kann. Die IDs besitzen 64 Bits in denen die Octreezugehörigkeit, Tiefe und eigentlichen X-, Y- und Z-Koordinaten gespeichert werden. Die Bits sind wie in Abbildung 16 verteilt. Für die Koordinaten einer Zelle sind je 19 Bits reserviert. Als Basis für die Koordinaten dient die Unterteilung von Octrees. Jede Zelle eines Octrees kann acht direkte Kinderzellen besitzen. Die Verschiebung dieser Kinderzellen innerhalb der Zelle kann je Achse über eine 0 oder 1 angegeben werden. Insgesamt kann die Position jeder dieser Zellen also mittels drei Bits dargestellt werden. Die ID gruppiert diese drei Bits nach den Achsen. Da die Spielwelt über insgesamt neun Octrees dargestellt ist, muss zur Identifikation einer Zelle ebenfalls die Zugehörigkeit zu diesen Octrees abgespeichert werden. Dies findet ebenfalls in den Koordinatengruppen statt. Je Achse werden dazu zwei Bits benötigt. In der Praxis sind die beiden Bits der Y-Achse aber immer auf 0 gesetzt, da sich die Octrees nur in die X- und Z-Achse erstrecken. Die Tiefe der Zellen könnte in der Theorie aus den gesetzten Bits abgeleitet werden. Es würde dabei allerdings vorkommen, dass eine Zelle die gleiche ID besitzt wie ihre Kinderzelle an der Stelle 000, da in diesem Fall nicht zwischen nicht gesetzten Bits und keiner Verschiebung unterschieden werden kann. Durch das explizite Abspeichern der Tiefe ist dies nicht der Fall, da Kinderzellen eine höhere Tiefe als ihre Elternzellen und somit auch eine andere ID besitzen. Für die Tiefe stehen die restlichen sieben Bits zur Verfügung, obwohl diese, wie die Bits der Koordinatengruppen, nie vollständig genutzt werden.



ABBILDUNG 16: AUFBAU DER ID EINER OCTREEZELLE.

Beim Erstellen der Octrees basieren die initial übergebenen IDs auf der Position des Octrees. Die beiden entsprechenden Bits jeder Achse stellen also 0, 1 oder 2 als Binärwert dar und die Tiefe entspricht 0. Bei jeder weiteren Unterteilung der Octrees werden für die IDs der Kinderzellen die Bits der Koordinatengruppen um eine Stelle nach links verschoben, um die Verschiebungsinformationen der Kinder am rechten Ende der jeweiligen Gruppe anhängen zu können. Da sich die Tiefe in den rechten Bits der Eltern-ID befindet kann diese einfach inkrementiert werden, um die Tiefe der Kinderzellen zu erhalten. Am Ende müssen die beiden erhaltenen Werte noch mit entsprechenden Masken verodert werden, um die endgültige ID einer Kinderzelle zu erhalten. Dank der Größe der Octrees hält sich die Gesamtzahl an Zellen in Grenzen und es kann ein Dictionary verwendet werden, um mittels der ID auf alle Zellen der Octrees zuzugreifen.

Nachdem die Konstruktion der eigentlichen Octrees abgeschlossen ist, können mittels der ID je Zelle rekursiv die Nachbarn bestimmt und Referenzen auf diese in einem Array gespeichert werden. Da die Verschiebungsinformationen neuer Kinderzellen immer an der rechten Seite der Koordinatengruppen angehangen werden, kann zum Bilden derselben Gruppe in der ID der Nachbarzelle einfach ein entsprechender Wert auf die Gruppe addiert

werden. Sollte beispielsweise der rechte Nachbar einer Zelle gefunden werden, wird 1 auf die Gruppe der X-Koordinaten addiert. Ist die momentane Zelle eines der linken Kinder der Elternzelle entspricht der erhaltene Wert der X-Koordinatengruppe der rechts davon liegenden Kinderzelle. Ist die momentane Zelle hingegen bereits auf der rechten Seite, ändert sich das entsprechende Bit an der Tiefe der Zelle auf 0 und das nächste Bit erhöht sich, was der Verschiebung der Elternzelle nach rechts entspricht. Befindet sich die Elternzelle ebenfalls rechts in deren Elternzelle wiederholt sich der komplette Vorgang den Octree hinauf. Wird der linke Nachbar gesucht wird stattdessen 1 subtrahiert, um den richtigen Wert zu erhalten. Dieser Ablauf funktioniert auf die gleiche Weise für alle drei Achsen. Die letztendliche ID des Nachbarn ergibt sich aus der Kombination der erhaltenen Koordinatengruppen und der Tiefe der behandelten Zelle. Zu Problemen kommt es dabei nur an den Rändern, an denen der 3×3 -Bereich der Octrees überschritten wird. Die erzeugte ID wäre in diesen Fällen ungültig und würde eventuell auf eine falsche Zelle verweisen. Entsprechend muss daher in diesen Fällen Null zurückgegeben werden.

Zum Sammeln der Chunkinformationen der Seams werden die Nachbarreferenzen der zum behandelten Chunk gehörenden Octreezelle genutzt. Die Referenzen liefern jedoch nur Octreezellen auf derselben Tiefe wie die zum Chunk gehörende Zelle. In diesem Kontext wird der zuvor erwähnte Boolean für die Blattrichtung verwendet. Ausgehend von der erhaltenen Zelle wird der Octree in die angegebene Richtung durchlaufen. Alle für den Seam relevanten Blätter werden in eine Liste geschrieben und jeder Nachbar gibt diese Liste am Ende zurück. Es kommen jedoch nicht alle Blätter für einen Seam in Frage. In der Praxis werden nur Blätter benötigt, welche sich an der zum Seam zeigenden Seite befinden. Um bestimmen zu können welche Blätter diese Bedingung erfüllen, findet wieder der in Kapitel 7.1 beschriebene Bitcode Anwendung. Der Bitcode beschreibt diesmal jedoch nicht die Randzugehörigkeit einer Zelle in einem Octree, sondern nur die lokale Positionierung in ihrer Elternzelle.

Beim Sammeln der Chunkinformationen ist es ebenfalls nicht nötig alle 26 direkten Nachbarn einer Octreezelle zu durchlaufen. Die Seammesh für einen Chunk werden immer nur in Richtung der positiven Achsenrichtungen erzeugt. Dadurch wird das mehrfache Generieren derselben Seammeshbereiche in benachbarten Chunks verhindert. Einen Beitrag zum Seammesh liefern daher nur die Chunks der sechs benachbarten Bereiche, welche sich in der positiven X-, Y-, Z-, XY-, XZ- und YZ-Richtung befinden. Der siebte Bereich in XYZ-Richtung wird nicht benötigt, da Verbindungen bei Nielsons Verfahren nur zwischen Zellen mit gemeinsamen Kanten auftreten können und das bei diesem Bereich nie der Fall ist.

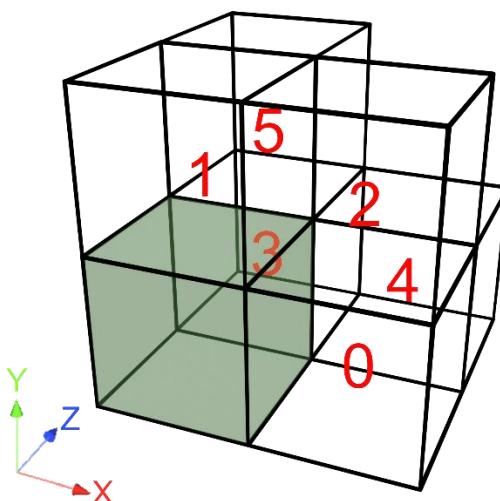


ABBILDUNG 17: DIE SEAMBEREICHE AUS DENEN SEAMCHUNKS GEWÄHLT WERDEN, UM DAS SEAMMESH ZU BILDET. DER BEHANDELTE CHUNK IST GRÜN GEKENNZIEHNET.

Am Ende der Octreeaktualisierung aus dem vorherigen Kapitel werden die Seams ebenfalls noch aktualisiert. Für alle Chunks, die aktualisiert werden, werden dazu mit der gerade beschriebenen Vorgehensweise Informationen zu den Seams gesammelt. Aufgrund der durch die Seams entstehenden Abhängigkeit benachbarter Chunks, müssen alle Chunks, dessen Seams sich durch die Aktualisierung eines Chunks potenziell verändert haben, ebenfalls aktualisiert werden. Bei den betroffenen Chunks handelt es sich dabei um die sechs Nachbarn entlang der negativen Achsenrichtungen.

Datendarstellung für die GPU

Pro Chunk sind nun zwar in der Theorie alle Daten zu den am Seam beteiligten Chunks vorhanden, ein weiteres Problem entsteht allerdings beim Erzeugen des Seammeshes aus diesen Daten. Da keinerlei Samples zu den Chunks auf der CPU gespeichert werden, müssen auch hier relativ viele Punkte gesampelt werden, um das finale Mesh zu erzeugen. Die Implementation der Flugsimulation nutzt daher wieder einen Compute Shader, um das Seammesh zu generieren. Im Folgenden wird der Chunk für den das Seammesh generiert wird lediglich als Chunk und alle benachbarten Chunks als Seamchunks bezeichnet.

Die erste Herausforderung entsteht in diesem Kontext durch das Übertragen der Seamdaten auf die GPU. Die Datendarstellung über Listen und Referenzen auf der CPU ist ungeeignet für die Darstellung auf der GPU. Zudem kommt, dass keine würfelförmigen Chunks behandelt werden müssen, sondern nur die Zellen an den Schnittflächen zwischen dem Chunk und den Seamchunks. Ebenfalls als problematisch erweist sich die Tatsache, dass die Anzahl an Zellen der Seamchunks nicht konstant ist, sondern je nach Positionierung und Auflösung der Seamchunks stark variieren kann. Es wird also eine Repräsentation für die relevanten Daten benötigt, mit der eine variierende Anzahl an Zellen mit unterschiedlichen Größen auf der GPU behandelt werden kann. Die Implementation teilt die Daten der Seamchunks dazu in drei Buffer auf, welche Metadaten zu den Seambereichen, die eigentlichen Seamchunkdaten und Mappinginformationen beinhalten. Wenn eine Seamaktualisierung für einen Chunk angefordert wird, werden sechs Listen mit den benachbarten Blättern der Octrees übergeben. Jede dieser Listen stellt einen Bereich des Seams dar. Für alle Listen werden im Anschluss alle in der jeweiligen Liste enthaltenen Blätter durchlaufen. Jede Liste generiert dabei je eine Instanz an Metadaten und Mappingdaten und jedes Blatt eine Instanz an Seamchunkdaten. Die Seamchunkdaten enthalten einerseits direkte Chunkinformationen in der Form der Position und Größe des Seamchunks, andererseits aber auch die Anzahl der Zellen, welche im Compute Shader für diesen Seamchunk behandelt werden müssen, und das Startoffset dieser Zellen im entsprechenden Buffer. Die Metadaten enthalten ebenfalls einen Zellenoffset, welcher den Startindex der Zellen des zugehörigen Seambereiches angibt. Des Weiteren enthalten die Metadaten weitere Informationen zu der Anzahl an Seamchunkdaten dieses Bereiches und des Offsets dieser Daten im zugehörigen Buffer. Für das Mapping enthalten die Metadaten ebenfalls die minimale Chunkgröße aller zur Liste gehörenden Seamchunks, das Startoffset im Mappingbuffer und die Größe des Mappingarrays. Das Mappingarray ist im Grunde nur ein zweidimensionales Array an Seamchunkindices, welches auf eine einzige Dimension gemappt wird. Die Mappingdaten werden benötigt, da ein Seambereich aus einer stark variierenden Anzahl an Seamchunks mit unterschiedlichen Größen bestehen kann. Das Bestimmen einer zu einer beliebigen Position gehörenden Zelle wäre auf der GPU aufgrund dieser Variation ziemlich aufwändig. Durch das Mappingarray kann diese Zugehörigkeit hingegen vergleichsweise leicht berechnet werden.

Damit der Compute Shader das Seammesh generieren kann, müssen in diesem die Vertices aller Zellen, die am Seam beteiligt sind, erzeugt werden. Der Compute Shader muss also alle dieser Zellen durchlaufen. Der Chunk trägt dabei immer 2977 Zellen entlang des Seams bei. Dies ist daher der Startwert für die Gesamtzahl der Zellen. Bei der Bearbeitung jedes Seambereiches werden direkt die momentanen Anzahlen an Einträgen in den jeweiligen Buffern als entsprechende Startoffsets in den Metadaten gesetzt. Die Anzahl an Seamchunks entspricht der Länge der zum Seambereich gehörenden Liste an Octreezellen und kann

daher ebenfalls direkt gesetzt werden. Im Anschluss werden alle in der jeweiligen Liste enthaltenen Blätter durchlaufen. Bei jedem Durchlauf werden hierbei die benötigten Seamchunkdaten für den zum Blatt gehörenden Seamchunk konstruiert und in die entsprechende Liste eingefügt. Pro Durchlauf wird anschließend zusätzlich die Gesamtzahl an zu behandelnden Zellen um die für den Chunk benötigte Anzahl erhöht und die minimale Seamchunkgröße für die Metadaten mit der Größe des behandelten Seamchunks aktualisiert. Nach der Abarbeitung aller Seamchunks eines Seambereiches stehen die restlichen Metadaten für das Mapping zur Verfügung. Diese sind die minimalen Seamchunkgröße für die Abstände der Mappingwerte und das Verhältnis der Größe des Chunks zu dieser minimalen Seamchunkgröße als Größe des Mappingarrays. Die Metadaten werden dann ebenfalls in die entsprechende Liste eingefügt. Nachdem sowohl die Meta- als auch die Seamchunkdaten konstruiert sind, werden die Mappingdaten ebenfalls erzeugt und in die Mappingliste geschrieben.

Die Konstruktion der Seamchunkdaten benötigt neben dem behandelten Blatt die Startposition des Seambereiches, die Achsen, die für den Seambereich verwendet werden, die Endposition des Chunks und die Gesamtzahl der Zellen vor der Bearbeitung des Seamchunks. Während der Startoffset der Zellen der übergebenen Gesamtzahl der Zellen entspricht und sich die Größe des Seamchunks direkt aus der Größe des Blattes erschließt, wird dessen Position anhand der Positionierung des Blattes entlang der verwendeten Achsen erschlossen. Ist die Blattposition entlang einer der Achsen größer als die Startposition des Seambereiches wird der Wert der Blattposition für die Komponente der Seamchunkposition verwendet. Ist dies für eine Achse nicht der Fall wird die entsprechende Komponente der Startposition genutzt. Auf die sich daraus ergebende Position wird anschließend noch das Ursprungsoffset der Spielwelt addiert, bevor sie als Position in den Seamchunkdaten gesetzt wird. Die Anzahl der Zellen, die der Compute Shader für einen Seamchunk behandeln muss, ergibt sich aus der Differenz zwischen der maximalen Chunkposition und der gerade in den Daten gesetzten Position dividiert durch die Größe des Seamchunks. Nachdem es bei Seamchunks, welche kleiner sind als der eigentliche Chunk, dazu kommen würde, dass mehr als 32 Zellen je Achse behandelt werden könnten, wird die Anzahl entsprechend auf maximal 32 limitiert.

Die Mappingdaten berechnen sich aus der Arraygröße, der minimalen Seamchunkgröße, der Startposition des Seambereiches, den daran beteiligten Achsen und den Blättern der entsprechenden Liste. Das Mappen aller Seamchunks trotz der variierenden Größen wird durch die minimale Seamchunkgröße ermöglicht. Keiner der Seamchunks ist kleiner als die minimale Größe. Nachdem alle anderen Seamchunks größtmäßig der gleichen oder einer höheren Zweierpotenz entsprechen, müssen sie auch immer ein Vielfaches der minimalen Größe sein. Das Mappingarray unterteilt den Seambereich in Bereiche dieser minimalen Größe, wodurch folglich genug Platz für alle Seamchunkindizes des Seambereiches im Array entsteht. Die zuvor berechnete Arraygröße stellt diese Unterteilung dar. Je nachdem ob ein oder zwei Achsen des Seambereiches zum Ergebnis beitragen, wird die Größe des letztendlichen Mappingarrays auf die Arraygröße oder die quadrierte Arraygröße gesetzt. Im Anschluss werden alle Blätter des Bereiches erneut durchlaufen. Die Position im Mappingarray, an welche der Index eines Blattes geschrieben wird, berechnet sich aus der Blattposition im Verhältnis zur Startposition des Seambereiches. Wenn die Blattgröße der minimalen Größe entspricht, wird nur eine Position im Mappingarray auf den Blattindex gesetzt. Ist das Blatt jedoch größer wird derselbe Index entsprechend mehreren Positionen zugewiesen.

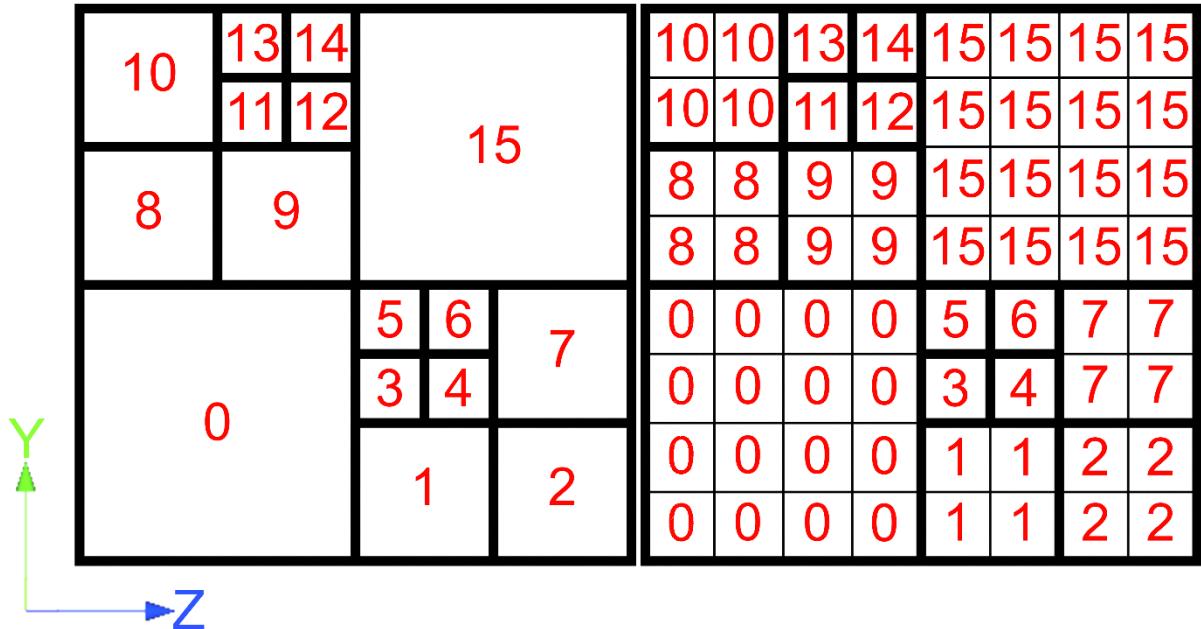


ABBILDUNG 18: BEISPIELHAFTE ZWEIDIMENSIONALE DARSTELLUNG EINES ERSTELLTEN MAPPINGARRAYS (RECHTS) FÜR DEN SEAMBEREICH IN DER POSITIVEN X-RICHTUNG (LINKS). DIE ZAHLEN REPRÄSENTIEREN DIE INDICES DER SEAMCHUNKS IN DIESEM SEAMBEREICH.

Konstruktion des Seammeshes

Nachdem alle Seambereiche abgearbeitet sind, sind sowohl die drei Listen mit den Metadaten, Seamchunkdaten und Mappingdaten, als auch die gesamte Anzahl an zu behandelnden Zellen vorhanden. All diese Daten werden an den Compute Shader übergeben. Der Compute Shader erzeugt das Seammesh dann in zwei Pässen. Ein sehr grober Ablauf für den zweiten Pass ist in Abbildung 20 dargestellt. Etwas detailreichere Flussdiagramme für sowohl den ersten als auch den zweiten Pass sind im Anhang dieser Arbeit in Abbildung 34 und 35 enthalten. Der erste der beiden Pässe initialisiert alle Daten der Zellen des Chunks und der Seamchunks. Basierend auf der zuvor berechneten Gesamtzahl wird dieser Pass für alle zu behandelnden Zellen ausgeführt. Die Eingabe für jede Zelle besteht dabei nur aus dem Threadindex. Als erstes werden daher einige benötigte Zelldaten ermittelt. Dies ist die Zugehörigkeit zu den Chunk- oder Seamzellen, der Chunk- oder Metaindex, die Seamchunkdaten, und natürlich die Position der Zelle. Die Zugehörigkeit erschließt sich je nachdem, ob der übergebene Index kleiner als 2977 ist oder nicht. Ist dies der Fall handelt es sich um eine Chunkzelle, anderenfalls um eine Zelle eines Seamchunks. Der Chunk- und Metaindex gibt jeweils Aufschluss über die Zugehörigkeit zu den Seambereichen und dem Index der Daten im Buffer. Handelt es sich um eine Seamzelle wird der Metaindex berechnet. Dieser ergibt sich aus der Iteration über die letzten fünf der sechs Einträge des Metadatenbuffers. Ist der Eingabeindex größer oder gleich dem im Buffereintrag stehenden Zellenoffset wird der Metadatenindex erhöht. Ein ähnlicher Ablauf wird für den Chunkindex im Falle von Chunkzellen verwendet. Da für die Chunkzellenoffsets keine Metadaten in einem Buffer bereitstehen, die Daten dafür aber immer dieselben sind, werden die Zellenoffsets aus einer Tabelle ausgelesen.

Auch bei der Ermittlung der Seamchunkdaten wird zwischen Chunk- und Seamzellen unterschieden. Für Seamzellen können hierbei über den Metaindex die Metadaten aus dem entsprechenden Buffer gelesen werden. Diese Metadaten liefern sowohl die Startposition als auch, durch die Addition der Datenanzahl, die Endposition im Buffer für die Seamchunkdaten. Mit diesen Daten kann dann analog zu der Berechnung des Metaindexes die Position der Seamchunkdaten berechnet werden. Für die Zellen des Chunks stehen keine

Seamchunkdaten zur Verfügung. Damit die Initialisierung jedoch einheitlich für alle Zellen gehalten werden kann, werden diese Daten mit den zur Verfügung stehenden Chunkdaten konstruiert. Die Größe in den Daten entspricht dabei direkt der Chunkgröße, das Zellenoffset entspricht dem konstanten Offset in der Tabelle, welche bei der Bildung des Chunkindexes Anwendung findet, und die Anzahl an Zellen je Achse entspricht 31. Die Position ergibt sich aus der Addition der Chunkposition und der Verschiebung zum Startpunkt der Chunkzellen je Seambereich.

Die eigentliche Position einer Zelle berechnet sich aus diesen Daten. Zur Bestimmung der Position wird als erstes der lokale Index im zum Chunk oder Seamchunk gehörenden Datensatz benötigt. Dieser bildet sich über die Subtraktion des in den Seamchunkdaten stehenden Zellenoffsets vom Eingabeindex. Im Anschluss können Achsenoffsets mittels des Chunk- oder Metaindexes aus einer Tabelle gelesen werden. Diese Achsenoffsets stellen die für einen Seambereich relevanten Achsen und die Richtung des Seambereiches dar. Für den Bereich in die X-Richtung würden die Offsets beispielsweise (0, 0, 1), (0, 1, 0) und (1, 0, 0) entsprechen. Die lokale Verschiebung der Zellposition berechnet sich folglich wie in Abbildung 19 dargestellt aus der Modulo-Rechnung oder Division des lokalen Indexes durch die Anzahl an Zellen je Achse und der darauffolgenden Multiplikation mit den entsprechenden Achsenoffsets und dem Abstand zwischen Samplepunkten einer Zelle. Die finale Position einer Zelle ergibt sich aus der Addition der Verschiebung auf die in den Daten stehende Position.

```
SeamChunkData GetSeamChunkData(uint idx, uint metaIndex)
{
    SeamChunkMeta meta = GetSeamChunkMeta(metaIndex);
    uint dataOffset = meta.chunkDataOffset;
    uint endDataOffset = dataOffset + meta.chunkDataCount;
    for (uint i = dataOffset + 1; i < endDataOffset; i++)
    {
        dataOffset += idx >= dataBuffer[i].cellOffset;
    }
    return dataBuffer[dataOffset];
}

int3 GetCellPosition(uint idx, uint metaOrChunkIndex)
{
    uint localIndex = idx - data.cellOffset;
    int3x3 axisMapping = SEAM_AXIS_MAPPING[metaOrChunkIndex];
    int3 positionOffset = axisMapping[0] * (localIndex % data.cellsPerAxis);
    positionOffset += axisMapping[1] * (localIndex / data.cellsPerAxis);
    return data.position + positionOffset * data.size;
}
```

ABBILDUNG 19: CODE FÜR DAS AUSLESEN DER SEAMCHUNKDATEN EINER SEAMCHUNKZELLE AUS DEM BUFFER UND DER BERECHNUNG DER POSITION EINER CHUNK- ODER SEAMCHUNKZELLE.

Damit stehen die grundlegenden Daten einer Zelle zur Verfügung. Die Erzeugung von Vertices und Zuweisung dieser zu den Kanten einer Zelle entsprechen größtenteils derselben Vorgehensweise wie bei der Konstruktion des eigentlichen Chunkmeshes. Es besteht allerdings auch eine Handvoll an Unterschieden. Aufgrund der variablen Anzahl und Größen der bearbeiteten Zellen werden die Werte der Terrainfunktion an den Eckpunkten der Zellen nicht einmalig vorberechnet. Stattdessen findet die Auswertung der Funktion individuell je Zelle statt. Nachdem der Abstand zwischen Samplepunkten nichtmehr einheitlich ist, muss zudem anstelle der Chunkgröße immer die in den Seamchunkdaten stehende Größe verwendet werden. Ein weiterer Unterschied besteht bei der Entscheidung, wann ein Vertex erstellt und einer Kante zugewiesen werden sollte. Dies kann hierbei nur geschehen, wenn die Kante, mit der der Vertex assoziiert werden sollte, relevant für das Seammesh ist. Eine

Kante ist relevant, wenn beide ihrer Eckpunkte an der Schnittstelle zwischen dem Chunk und einem Seamchunk liegen. Da diese Information für die Kanten eines Bereiches immer dieselbe ist, kann die Relevanz einer Kante mittels des Chunk- oder Metaindexes, der Zellart und dem Kantenindex aus einer weiteren Tabelle ausgelesen werden. Würde diese Einschränkung bei der Assoziiierung nicht stattfinden, wäre dies kein direktes Problem, es würden allerdings Vertices generiert werden, welche letztendlich nie zu den Dreiecken des Seameshes beitragen würden. Die Zwischenspeicherung der Kanteninformationen findet ebenfalls unterschiedlich statt. Anstelle der Gruppierung von Kanten zu Kantengruppen von je drei Kanten pro Zelle, wird der mit einer Kante assoziierte Vertex lokal je Kante gespeichert. Jede Zelle speichert dabei Daten zu ihren zwölf Kanten ab. Da auf diese Weise je Kante immer nur ein Vertex zugewiesen werden kann, entsteht insgesamt kein höherer Speicheraufwand als bei der Verwendung von Kantengruppen. Zusammen mit der Tatsache das für den Seam irrelevante Kanten nie einen Vertex zugewiesen bekommen, hat dies den zusätzlichen Vorteil, dass stattdessen weitere Informationen in den ungenutzten Kanten gespeichert werden können. In der Implementation ist dies das Speichern der Information, ob sich ein bestimmter Eckpunkt innerhalb oder außerhalb des Terrains befindet. Darüber kann später die Vertexreihenfolge der Dreiecke bestimmt werden. Bei der Verwendung der Eckpunkt- und Kantenreihenfolge dieser Arbeit entspricht das für Chunkzellen dem Speichern der Terrainzugehörigkeit von Eckpunkt 6 in Kante 0 und für Seamzellen dem Speichern der Zugehörigkeit von Eckpunkt 0 in Kante 5.

Nach dem ersten Pass stehen die Vertices und Verbindungsinformationen für die Erzeugung der Dreiecke im zweiten Pass zur Verfügung. Die Bildung dieser Dreiecke geschieht ausgehend von den Zellen des Chunks. Entsprechend wird der zweite Pass nur für diese ausgeführt. Für jede betroffene Zelle des Chunks werden auch in diesem Pass wieder als erstes die grundlegenden Zellinformationen berechnet. Dies sind der Chunkindex, die Position der Zelle und die Kantendaten der Zelle aus dem vorherigen Pass. Sowohl der Chunkindex als auch die Position der Zelle werden auf dieselbe Art und Weise wie im ersten Pass bestimmt. Da allerdings nur Zellen des Chunks behandelt werden, werden keine Seamchunkdaten benötigt, um die Bearbeitung allgemein zu halten. Daher werden direkt die vorhandenen Chunkdaten genutzt, um die Position der Zelle zu bestimmen. Die Kantendaten können aus dem Buffer mittels des Zellindexes gelesen werden. Als nächstes wird bestimmt, welche Seambereiche für die Zelle behandelt werden müssen. Hierbei ist es nicht nötig alle sechs Bereiche des Seams separat voneinander zu betrachten. Stattdessen reicht es aus nur die Bereiche entlang der Hauptachsen zu separieren. Der inkrementierte Chunkindex gibt dazu Aufschluss über die Bereichszugehörigkeit einer Zelle. Die Zugehörigkeit zur X-, Y- und Z-Achse entsprechen jeweils dem gesetzt sein des ersten, zweiten und vierten Bits im inkrementierten Chunkindex. Für jede dieser Bereichszugehörigkeiten werden Zellen in der entsprechenden Richtung miteinander verbunden. Dabei dienen als allgemeine Eingabe die ermittelte Position und Kantendaten der Zelle. Abhängig von der behandelten Richtung werden zusätzlich Informationen über die zu behandelnden Achsenrichtungen und Kantenverbindungen übergeben. Letzteres besteht aus den Indices der Kanten einer Zelle auf der Seite des Seams für sowohl Chunk- als auch Seamzellen.

Beim Verbinden der Vertices der Zellen in einer der drei Richtungen werden zunächst einige, teilweise richtungsabhängige, Informationen benötigt. Die erste dieser Informationen sind die Seamchunkdaten des Seamchunks dessen Zelle an der behandelten Chunkzelle anliegt. Da der Index der Seamzelle diesmal nicht bekannt ist, werden die Daten des Seamchunks über eine Position in diesem ermittelt. Von der übergebenen Position der Chunkzelle wird dazu eine Chunkzellenbreite in Richtung des Seams gegangen. Die daraus entstehende Position liegt im gesuchten Seamchunk. Der Metaindex des zugehörigen Seambereiches kann mit dieser Position bestimmt werden, indem geprüft wird, welche der Positionskomponenten den Komponenten der maximalen Chunkposition entsprechen. Je nachdem für welche Komponenten dies der Fall ist, werden dann 1, 2, und/oder 4 miteinander summiert und das Ergebnis anschließend dekrementiert, um den Metaindex zu erhalten. Mit den durch den Metaindex erhaltenen Metadaten berechnen sich dann die

Seamchunkdaten, indem als erstes die Startposition des zugehörigen Seambereiches durch eine entsprechende Verschiebung der Chunkposition berechnet wird. Mit der Startposition kann dann die lokale Position des Seamchunks innerhalb des Seambereiches bestimmt werden. Dies geschieht über die Division der Differenz, zwischen der Position im Seamchunk und der Startposition, durch die Ausmaße des kleinsten Seamchunks in diesem Seambereich. Mit der Hilfe des Mappingoffsets in den Metadaten und den relevanten Komponenten der lokalen Position ergibt sich daraus die Position der benötigten Daten im Mappingbuffer. Da im Mappingbuffer die Indices der Seamchunks eines Bereiches stehen, kann der erhaltene Wert auf den in den Metadaten stehenden Offset für die Seamchunkdaten addiert werden, um die eigentlich gesuchten Seamchunkdaten aus dem Buffer zu lesen. Das beschriebene Finden der Daten kann allerdings nur geschehen, wenn der Seambereich in der entsprechenden Richtung auch Seamchunks enthält. Ist dies nicht der Fall wird eine neue Instanz an Seamchunkdaten zurückgegeben, in der alle Werte 0 entsprechen. Besitzen die erhaltenen Daten eine Größe von 0 kann das Generieren der Dreiecke folglich abgebrochen werden. Die wirkliche Position, der an der Chunkzelle anliegenden Seamzelle, ergibt sich mittels der Seamchunkdaten. Dazu wird die Startposition des Seamchunks von der zuvor ermittelten Position innerhalb des Seamchunks abgezogen und das Ergebnis dividiert durch die Größe einer Zelle. Dadurch entsteht die Anzahl an Zellen entlang jeder Positions komponente. Da dabei Integer-Positionen verwendet werden, ergibt sich durch die anschließende Multiplikation mit der Zellgröße und Addition der Seamchunkposition die gesuchte Position der Seamzelle.

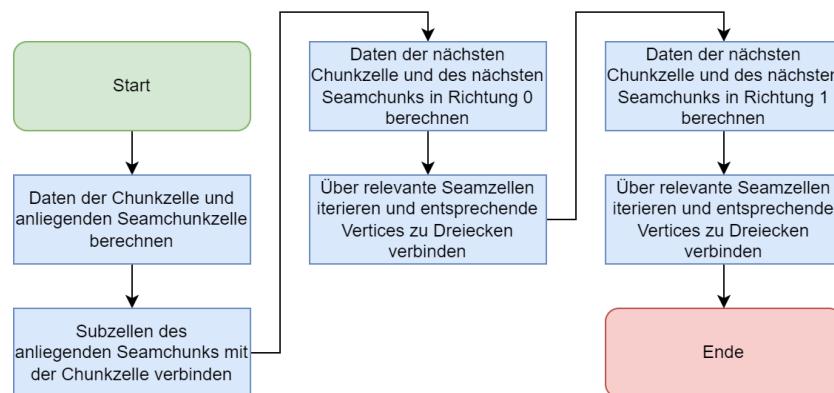


ABBILDUNG 20: FLUSSDIAGRAMM FÜR DEN GROBEN ABLAUF DES ZWEITEN PASSES ZUR ERSTELLUNG DES SEAMMESHES.

Das eigentliche Verbinden erfolgt in zwei groben Schritten. Dem Verbinden der Chunkzelle mit kleineren Seamzellen in einem Bereich der gleichen Zellgröße im Seamchunk und dem Verbinden mit Seamzellen außerhalb der Ränder der Chunkzelle. Diese Aufteilung ist nötig, da für eine Chunkzelle eine Reihe an Zellen in einem anliegenden kleineren Seamchunk existieren können. Für die weitere Behandlung der Verbindungen wird daher das Verhältnis zwischen der Größe der Chunkzelle und der Größe der Zelle des Seamchunks benötigt. Dieses Verhältnis wird auf einen minimalen Wert von 1 limitiert. Ist das Zellenverhältnis größer als 1 enthält der Seamchunk mehrere Zellen pro Chunkzelle. Um diese miteinander zu verbinden, wird über den entsprechenden zweidimensionalen Bereich der Seamzellen iteriert. Das Verhältnis der Zellen liefert dabei die Anzahl an nötigen Iterationen je Achse. In jeder Iteration werden die Vertices der Chunkzelle und der behandelten Seamzelle mit den Vertices der nächsten Seamzelle entlang einer der beiden Achsenrichtungen verbunden. Für die Seamzellen des Seambereiches in der X-Richtung würden diese Richtungen der Z- und Y-Richtung entsprechen. Zur Bestimmung der Kantendaten der Seamzellen werden deren Indices benötigt. Diese ergeben sich mittels der Berechnung der Seamchunkdaten, wie zuvor beschrieben, und der darauffolgenden Ermittlung der lokalen Position der Zellen im zugehörigen Seamchunk. Diese lokalen Komponenten können wie in Abbildung 21 auf das in

den Seamchunkdaten stehende Bufferoffset für die Zellen addiert werden, um die Indices der Zellen zu erhalten. Mittels der Zellindices können dann die benötigten Vertices der Seamzellen bestimmt werden. Da keinerlei direkte Informationen zur Separierung der Bereiche des Terrains bekannt sind, werden die Vertices der Seamzellen mit allen Vertices der Chunkzelle verbunden, falls diese nicht dieselben Vertices sind. In Nielsons Verfahren kann eine Zelle bis zu vier Vertices besitzen. Da die Vertices allerdings nur an den Seiten des Seams erzeugt werden, enthält jede Zelle maximal zwei Vertices. Zusätzlich zu den Vertices eines Dreieckes muss dessen Orientierung in Form der Vertexreihenfolge bestimmt werden. Die Orientierung bestimmt sich dabei aus der Klassifikation einer der beiden Eckpunkte der durch die Zellen geschnittenen Kante. Da der Seamchunk die höhere Auflösung besitzt, ergibt sich dieser Wert aus der zuvor gespeicherten Klassifikation von Eckpunkt 0 der nächsten Seamzelle. Beim Hinzufügen der Dreiecke wird sicherheitshalber geprüft, ob alle Vertices ungleich 0 sind und es sich bei ihnen um unterschiedliche Vertices handelt. Ist dies der Fall werden die Vertices dekrementiert, um die eigentlichen Vertexindices zu erhalten, und anschließend in den Dreiecksbuffer geschrieben.

```
uint GetSeamCellIndex(int3 cellPosition, uint metaIndex, SeamChunkData data)
{
    int3 localPosition = (cellPosition - data.position) / data.size;
    int3x3 axisMapping = SEAM_AXIS_MAPPING[metaIndex];
    uint indexOffset = dot(axisMapping[0], localPosition);
    indexOffset += dot(axisMapping[1], localPosition) * data.cellsPerAxis;
    return data.cellOffset + indexOffset;
}
```

ABBILDUNG 21: CODE FÜR DIE BESTIMMUNG DER INDICES VON SEAMCHUNKZELLEN ANHAND DER POSITION EINER ZELLE.

Damit sind die Zellen im Bereich der Chunkzelle innerhalb des Seamchunks verbunden. Nun fehlen noch die chunkzellenübergreifenden Verbindungen mit den nächsten Chunkzellen und Seamzellen. Die Verbindungen werden getrennt voneinander für die jeweiligen Achsen berechnet. Der Ablauf ist jedoch grundsätzlich der gleiche für beide Achsen. Hierbei muss beachtet werden, dass der zur initialen Chunkzelle gehörende Seamchunk nicht zwingend der gleiche ist wie der der nächsten Chunkzelle entlang einer Achse. Die Seamchunks der beiden Chunkzellen können also ebenfalls Zellen mit unterschiedlichen Größen besitzen. Um alle Zellen zu behandeln, müssen neben der Aufteilung je Achse, zusätzlich für alle Zellen des einen Seamchunks alle zugehörigen Zellen des nächsten Seamchunks behandelt werden. Dazu werden, ausgehend von der nächsten Chunkzellenposition, wieder die Seamchunkdaten des nächsten Seamchunks über die nächste Position in diesem Seamchunk berechnet. Ist die Größe in den erhaltenen Daten ungleich 0 existiert ein Seamchunk an dieser Stelle und die Verbindungen können erzeugt werden. Zum Generieren der übergreifenden Dreiecke werden die Positionen der vier beteiligten Zellen benötigt. Die Positionen auf Seiten des Chunks sind konstant. Die Position der initialen Chunkzelle ist bereits bekannt und die Position der nächsten Chunkzelle kann über eine Verschiebung in die entsprechende Richtung ermittelt werden. Die Positionen der Seamzellen müssen berechnet werden. Als Startposition für die Seamzellen des initialen Seamchunks dient die zuvor berechnete Startposition der Zellen im Seamchunk, verschoben entlang der entsprechenden Achse um die Größe einer Seamzelle multipliziert mit dem dekrementierten Verhältnisfaktor zwischen Chunk und Seamchunk. Daraus entsteht die Position der ersten zu behandelnden Seamzelle auf der Seite des initialen Seamchunks. Da das Verhältnis auf minimal 1 limitiert wird entsteht keine Verschiebung der Startposition, wenn die Seamzellen größer als die Chunkzellen sind. Die Startposition der Seamzellen im nächsten Seamchunk ermittelt sich gleich der ursprünglichen Startposition im initialen Seamchunk. Über die Größen des initialen und des nächsten Seamchunks kann anschließend das Verhältnis der Zellen zwischen diesen gebildet werden. Für alle Zellen des initialen Seamchunks, innerhalb

einer Chunkzelle, wird dann über alle Zellen des nächsten Seamchunks, innerhalb einer Zelle des initialen Seamchunks, iteriert. Damit werden alle Verbindungen zwischen den Zellen abgedeckt.

In jeder Iteration ist es nötig zunächst die Kantendaten und daraus die Vertices der vier beteiligten Zellen zu ermitteln. Die Vertices der beiden Chunkzellen sind dabei konstant über alle Iterationen. Der Vertex der nächsten Chunkzelle in einer Richtung kann allerdings nur existieren, wenn es sich bei dieser Zelle auch um eine Chunkzelle handelt. Befindet sich die initiale Chunkzelle bereits am Rand des Chunks ist die nächste Zelle ein Teil eines Seamchunks und der Vertex muss entsprechend auf 0 gesetzt werden. Eine Zelle ist eine Seamzelle, wenn sich die berechnete Position der Zelle mindestens eine Komponente mit der maximalen Chunkposition teilt. Aus den Vertices der vier Zellen werden dann zwei Dreiecke generiert. Die Vertices dieser beiden Dreiecke sind einerseits die entsprechenden Vertices der Chunkzelle, der nächsten Seamzelle und der initialen Seamzelle, und andererseits die Vertices der Chunkzelle, der nächsten Chunkzelle und der nächsten Seamzelle. Durch den Check auf gleiche Vertices und der Ungleichheit der Vertices zu 0 wird hierbei wieder sichergestellt, dass keine ungültigen Dreiecke produziert werden. Die Orientierung der Dreiecke erschließt sich ebenfalls wieder durch die Klassifikation der Zelleckpunkte. Ist das Verhältnis zwischen Chunk und Seamchunk ungleich 1 wird wieder die Klassifikation der nächsten Seamzelle als Basis verwendet. Ist dies nicht der Fall findet stattdessen die gespeicherte Klassifikation von Eckpunkt 6 in der Chunkzelle Anwendung. Ein Fall, welcher ebenfalls noch gehandhabt werden muss, kann auftreten, wenn die Zellen des Seamchunks größer sind als die Zellen der Chunks. In diesem Fall kann die Seamzelle und die nächste Seamzelle derselben Zelle entsprechen. Die Verwendung der bisherigen Kantenindices zur Bestimmung der an den Dreiecken beteiligten Vertices liefert in diesen Fällen potenziell falsche Werte und der Vertex der nächsten Seamzelle muss daher mit dem entsprechend gültigen Vertex ersetzt werden. Nach der Abarbeitung einer Zelle des initialen Seamchunks werden die Positionen zur Berechnung der nächsten Seamzellen um entsprechende Offsets verschoben, um im nächsten Iterationsdurchlauf die richtigen Zellpositionen zu ermitteln. Nachdem dieser Ablauf die Dreiecke in beiden Achsenrichtungen des Seambereiches generiert hat, ist die Generation des Seammeshes abgeschlossen. Die Meshdaten werden darauffolgend analog zur Erstellung des Chunkmeshes asynchron auf die CPU geladen und dem Mesh zugewiesen.

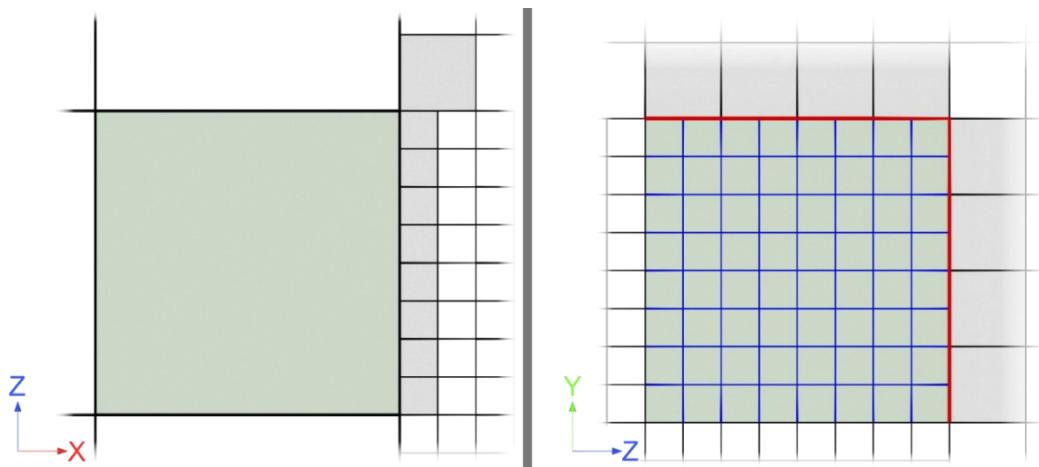


ABBILDUNG 22: BEISPIEL DER IN KOMBINATION MIT EINER CHUNKZELLE BEARBEITETEN KANTEN DER ZELLEN VON SEAMCHUNKS. DIE BEARBEITE CHUNKZELLE IST GRÜN HINTERLEGT UND DIE ZUGEHÖRIGEN SEAMZELLEN GRAU. DIE KANTEN, DIE IM ERSTEN VERBINDUNGSSCHRITT MITEINANDER VERBUNDEN WERDEN, SIND IN DER RECHTEN HÄLFTE BLAU EINGEZEICHNET UND DIE CHUNKZELLENÜBERGREIFENDEN KANTEN DES ZWEITEN SCHRITTES ROT.

7.6 TERRAINFUNKTION

Die Beschreibung der eigentlichen Terrainfunktion zur Strukturierung der Spielwelt und Extraktion der Isofläche fehlt bisher. Perlin Noise stellt den grundlegenden Baustein dieser Funktion dar. Da die Terrainfunktion auf der GPU ausgewertet wird, muss Perlin Noise ebenfalls auf der GPU implementiert werden. Der Code der letztendlichen Implementation von Perlin Noise in dieser Arbeit orientiert sich dabei stark an der in Kapitel 2 erwähnten Referenzimplementation des verbesserten Algorithmus von Perlin. Zur Implementation auf der GPU werden in der Praxis zudem nur minimale Änderungen an Perlins Referenzcode benötigt. Der grundlegende Ablauf entspricht folglich dem bereits erklärten Ablauf und wird daher zur Vermeidung der redundanten Beschreibung nicht nochmals erklärt.

Die Terrainfunktion selbst muss im Kontext der Flugsimulation einige spezifische Eigenschaften besitzen. Da sich die Spielwelt der Flugsimulation endlos erstrecken soll, muss die Terrainfunktion zunächst dafür ausgelegt sein. Dies ist aufgrund der periodischen Wiederholung von Perlin Noise bereits größtenteils gegeben. Im Vergleich dazu erweist sich hingegen die gewünschte labyrinthartige Struktur der Schluchten, durch die der Spieler fliegt, zumindest auf den ersten Blick als problematisch. Die Struktur der Schluchten sollte einerseits nicht nur möglichst natürlich wirken, sondern andererseits auch immer einen Weg für den Spieler bereithalten. Letzteres bedeutet vorwiegend, dass der Verlauf der Schluchten keine allzu schlagartigen Richtungsänderungen und keinerlei Sackgassen enthalten sollte. Perlin Noise bietet auch hier eine mögliche Grundlage. Ein grundlegender Pfad einer Labyrinthstruktur kann bereits relativ einfach über eine Potenzbildung mit den Noise-Werten als Basis und einem weiteren Faktor als Exponent erreicht werden. Über den Exponenten kann hierbei die Breite der Schluchten eingestellt werden. Wird im Anschluss ebenfalls noch der Wertebereich des Betrags mittels einer unteren und oberen Grenze angepasst, entsteht eine bereits schluchtenähnliche Landschaft mit relativ steilen Wänden. Das Ergebnis dieser Vorgehensweise für den zweidimensionalen Fall ist in Abbildung 23 dargestellt. Wie ersichtlich entstehen dabei Kurven, welche sich relativ konstant durch den gesamten Bereich bewegen ohne direkte Sackgassen zu erzeugen. Aus der Visualisierung des zweidimensionalen Falles geht eine weitere positive Eigenschaft dieser Methode jedoch nicht hervor. In der Regel sind Schluchten nicht ausschließlich vertikal. Stattdessen existieren höhenabhängige Variationen in der Breite und Ausrichtung des Schluchtverlaufes. Im dreidimensionalen Fall erzeugt diese Methode ebenfalls diese Schluchteneigenschaften aufgrund der Variation der Noise-Werte entlang der Y-Achse. Das Ergebnis ist allerdings auch nicht komplett perfekt. Wie ebenfalls erkennbar entstehen ab und zu Bereiche, in denen der Spieler in einem kleinen Bereich gefangen wäre und endlos im Kreis fliegen würde. Um dieses Problem zu lösen, existiert der in Kapitel 8.3 beschriebene Drill-Ray als Spielelement, mit dem der Spieler einen temporären Tunnel durch die Landschaft erzeugen kann. Insgesamt liefert die beschriebene Methode also eine grundsätzlich gute Basis für die Terrainfunktion dieser Arbeit. Für die Verwendung in der Flugsimulation ist das direkte Ergebnis jedoch noch etwas zu regelmäßig und viel zu glatt, um eine wirkliche Herausforderung für den Spieler zu bieten. Zudem wiederholen sich die Schluchtenbereiche merkbar periodisch dank der Verwendung von nur einer Oktave. Abhilfe schafft hierbei die Kombination mehrerer Oktaven an Perlin Noise vor der Potenzierung. Durch jede weitere Oktave entsteht jedoch auch ein weniger reguläres Ergebnis. In der Praxis werden daher nur zwei Oktaven kombiniert, damit zwar eine Herausforderung beim Durchfliegen der Landschaft geboten und Variation in den generierten Pfad eingebracht wird, die Spielwelt aber dennoch spielbar bleibt.

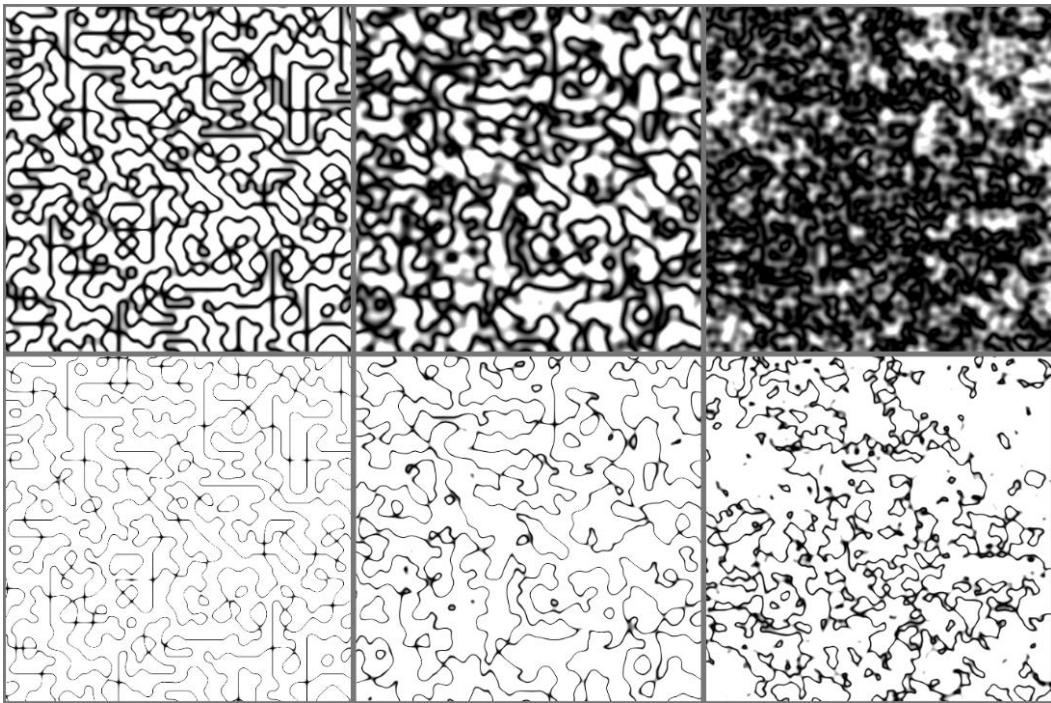


ABBILDUNG 23: DARSTELLUNG DER GRUNDLEGENDEN PFADE DURCH DIE SCHLUCHTEN. VON LINKS NACH RECHTS IST DIE VERWENDUNG VON JEWELLS 1, 2 UND 6 OKTAVEN AN PERLIN NOISE DARGESTELLT. IN DEN OBEREN ERGEBNISSEN WIRD DABEI EIN EXPONENT VON 1 UND IN DEN UNTEREN EIN EXPONENT VON 7 FÜR DIE POTENZIERUNG GENUTZT.

Während der grundlegende Pfad nicht von weiteren Noise-Oktaven profitiert, ist dies nicht der Fall für das allgemeine Terrain. Im Gegensatz ist es hierbei erwünscht weitere Oktaven zu nutzen, damit das letztendliche Terrain detailreicher ist. Dazu können die beiden Noise-Oktaven, die für den grundlegenden Pfad verwendet werden, wiederverwendet und um drei weitere Oktaven erweitert werden. Die direkte Addition der Terrainbasis und des resultierenden Noise-Wertes liefert, zumindest im Kontext dieser Implementation, keine allzu guten Resultate. Stattdessen wird zwischen den beiden Werten basierend auf dem grundlegenden Pfadwert interpoliert. Dieser Pfadwert liegt nahe 0 auf dem Pfad und nahe 1 an Punkten die weiter entfernt vom Pfad liegen. In anderen Worten trägt auf dem Pfad größtenteils die Terrainbasis zum Ergebnis bei und mit zunehmender Entfernung zum Pfad fällt der Noise-Wert immer weiter ins Gewicht.

Die bisher erzeugten Schluchten enthalten zudem noch keinen festen Boden. Um die Schluchten etwas interessanter zu gestalten, sammelt sich am Boden der Schluchten Wasser an. Für die Generierung von Becken für das Wasser wird ebenfalls wieder der Pfad der Schluchten über die Potenzierung und Betragsbildung berechnet. Die X- und Z-Achse der dabei verwendeten Position entsprechen wieder der eigentlichen Eingabeposition. Die Y-Achse wird jedoch durch einen statischen Wert ersetzt. Dadurch entsteht ein fester 2D-Ausschnitt des Pfades. Durch einige weitere Operationen bilden sich aus diesem Ausschnitt die Einkerbungen für das Wasser. Im Anschluss werden die fünf vorherigen Oktaven an Noise auf das Ergebnis addiert, um die Struktur des Bodens aufzubrechen. Der Wert des Bodens und der bisherige Terrainwert werden am Ende über eine weitere Interpolation kombiniert. Als Basis für den Interpolationsfaktor dient dabei der Y-Wert der Eingabeposition.

Aufgrund der Interpolationen kommt es leider auch zu größeren Hohlräumen im Terrain an Punkten, an denen das Ergebnis stark auf den Noise-Werten basiert. Die Hohlräume stellen kein direktes Problem für den Spieler dar, für das prozedurale Platzieren von Objekten

hingegen schon, da diese eventuell in diesen platziert werden könnten. In der Praxis muss eine solche Hohlraumbildung also vermieden werden. Dies geschieht relativ einfach, indem der Terrainwert nach den vorherigen Interpolationen stark erhöht wird, wenn die Terrainbasis über einem gewissen Schwellwert liegt.

Durch das Nutzen von Noise kann es auch dazu kommen, dass kleinere fliegende Objekte nahe der Schluchtenwände generiert werden. Für die Spielbarkeit erweisen sich diese als problematisch, da der Spieler solche kleinen Objekte leicht übersehen kann und folglich ebenfalls leicht mit diesen kollidieren könnte. Das Verhindern der Generation aller solcher fliegenden Objekte ist ohne eine weitere Analyse der Terrainfunktion kein leichtes Unterfangen. Die Bildung extrem kleiner Objekte kann jedoch zum größten Teil verhindert werden, indem der vorliegende Terrainwert negiert wird, wenn er positiv ist und sich nahe genug am Isolevel befindet. Dies beeinflusst aber den gesamten Detailgrad des Terrains. Der obere Schwellwert sollte hierbei also nicht zu hoch gewählt werden.

Damit sich die Landschaft nicht endlos entlang der Y-Achse erstreckt wird der letztendliche Terrainwert vor der Rückgabe noch um einen entsprechend hohen oder niedrigen Wert erweitert, sobald er sich außerhalb der festgelegten Höhengrenzen befindet.

```
float GetTerrainValue(float3 position)
{
    float noise = PerlinNoise(position * 0.001f) + PerlinNoise(position * 0.0023f) * 1.36f;

    float pathValue = 1.0f - pow(1.0f - abs(noise * 0.42372881355f), terrainWidthFactor);
    float baseValue = saturate((pathValue - 0.02f) * 5.56f);

    noise += PerlinNoise(position * 0.039142f) * 0.5f;
    noise += PerlinNoise(position * 0.0871f) * 0.25f;
    noise += PerlinNoise(position * 0.16337f) * 0.125f;

    float terrainValue = lerp(baseValue * 2.0f - 1.0f, noise, pow(pathValue, 0.4f));

    float3 groundPos = float3(position.x, 10.0f, position.z);
    float groundValue = PerlinNoise(groundPos * 0.001f) + PerlinNoise(groundPos * 0.0023f) * 1.36f;
    groundValue = 1.0f - pow(1.0f - abs(groundValue * 0.42372881355f), terrainWidthFactor);
    groundValue = (saturate(groundValue * 10.0f) * 10.0f - position.y) * 0.1f;
    groundValue += noise;

    terrainValue = lerp(groundValue, terrainValue, pow(saturate(position.y * 0.1f), 7.0f));
    terrainValue += (baseValue > 0.8f) * 50.0f;
    terrainValue -= (0.0f < terrainValue && terrainValue < 0.1f) * 2.0f * terrainValue;
    terrainValue += (position.y > 110.0f) * -1000.0f + (position.y < 2.0f) * 1000.0f;
    return terrainValue + GetDrillRayValue(position);
}
```

ABBILDUNG 24: DER CODE DER TERRAINFUNKTION.

7.7 TEXTURIERUNG

Neben der allgemeinen Generierung des Terrains muss dieses ebenfalls noch texturiert werden. Dazu verwendet der Shader des Terrainmaterials die in Kapitel 5 beschriebenen Methoden. Für die Texturierung des Terrains werden fünf verschiedene Texturen verwendet. Jede dieser Texturen besteht in der Praxis aus mehreren Texturarten, welche jeweils Informationen zu dem dargestellten Material beinhalten. Damit der Terrainshader folglich nicht die Informationen zu jeder dieser Varianten als einzelne Texturen erhalten muss, werden diese zu insgesamt drei Texturarrays zusammengefasst. Im ersten dieser Arrays ist die Farbe der Terrainbereiche enthalten, das zweite enthält die zugehörigen Normal-Maps und das letzte Array besitzt je Channel weitere Informationen, wie die Ambient Occlusion der Textur. Des Weiteren sind im Sinne dieser Arbeit die einzelnen Texturen prozedural

erstellt. Für die Generierung dieser Texturen findet Blender¹³ Anwendung. Da sich die Flugsimulation vom Setting her auf einem Wüstenplaneten abspielt, besitzen die Texturen der Schluchtwände und -überhänge ein entsprechend steiniges Aussehen. Die Texturen für die Böden sind hingegen eher sandig gehalten. Bei der Generation aller Texturen werden mehrere Oktaven an Noise kombiniert. Für eine der Bodentexturen wird beispielsweise eine Oktave an hochfrequentem Perlin Noise verwendet, um zwischen zwei sandigen Farben zu mischen und somit die grundlegende Texturfarbe festzulegen. Über die Normal-Map werden gröbere Unebenheiten eingebracht. Dazu werden zwei Oktaven an Perlin Noise mit eher niedrigen Frequenzen miteinander multipliziert und als Höhe interpretiert. Daraus kann darauffolgend eine entsprechende Normal-Map abgeleitet werden. Ein ähnlicher Ansatz wird für die Texturen der Schluchtenwände und -überhänge verwendet. Dabei findet jedoch nicht nur Perlin Noise Anwendung. Es wird zusätzlich Worley Noise in der Form von Blenders Voronoi-Texture-Node eingebracht, um grundlegend eine gewisse steinähnliche Struktur in den Texturen zu erzeugen. Dazu werden die aus den Nodes beider Noise-Arten erzeugten Farbwerte miteinander gemischt. Anschließend wird die Differenz mehrerer verschiedener dieser Kombination gebildet. Mit dieser Vorgehensweise als Grundlage kann über die Anpassung der Parameter und leichten Weiterverarbeitung der Ergebnisse eine Vielzahl an wüsten- und canyontypischen Texturen erzeugt werden. Worauf jedoch bei der Generierung dieser Texturen geachtet werden sollte, ist, dass die letztendliche Textur keine Merkmale besitzt, die direkt und zu stark ins Auge springen. Nachdem die Texturen fortlaufend über das gesamte Terrain wiederholt werden, fallen Wiederholungen solcher prominenten Merkmale meist trotz der verwendeten Methoden vergleichsweise stark auf.

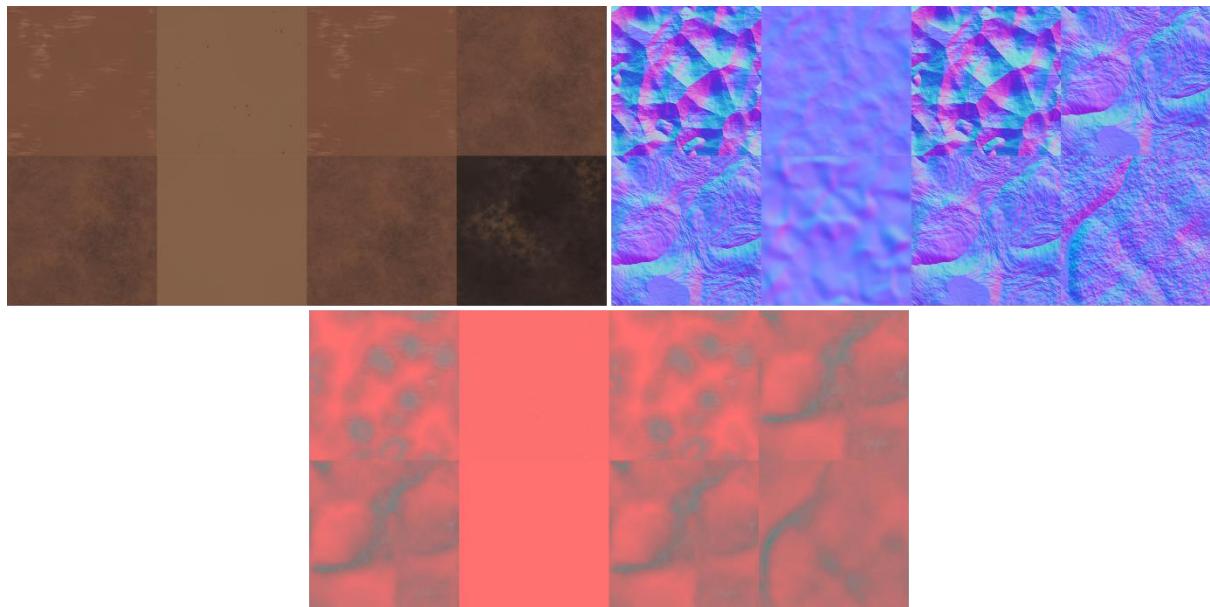


ABBILDUNG 25: DIE TEXTURARRAYS DES TERRAINS DER FLUGSIMULATION.

Die drei Texturarrays des Terrainshaders enthalten Informationen zu je acht Texturen. Je zwei dieser Texturen stellen die Varianten einer Projektionsrichtung dar. Die ersten drei Projektionsrichtungen bestehen dabei aus den für Triplanar Mapping üblichen Projektionen von vorne, oben, und rechts. Zusätzlich wird zwischen der Projektion von unten und oben unterschieden, da ansonsten Überhänge ebenfalls mit der Bodentextur texturiert werden würden, was aufgrund des sandigen Bodens, zumindest im Falle dieser Arbeit, ungeeignet wäre. Als grundlegende Eingaben für den Terrainshader dienen je Fragment die Position und Normale im Koordinatensystem der Spielwelt. Dazu kommen die Terraintexturen und

¹³ Blender Webseite - <https://www.blender.org/>

weitere Parameter, über die das Aussehen des Terrains angepasst werden kann. Die UV-Koordinaten für Triplanar Mapping ergeben sich aus den in Kapitel 5.1 erwähnten Koordinatenpaaren der Weltposition. Die vier Mischfaktoren werden zunächst für die drei grundlegenden Achsen über den Betrag der Normalenkomponenten gebildet. Je nachdem, ob die Normale nach oben oder unten zeigt, wird der Faktor der Y-Achse anschließend für die Projektion von oben oder unten gesetzt. Damit die Schärfe des Blendings einstellbar ist, wird die Potenz der Faktoren vor der Normalisierung mit einem weiteren Wert gebildet. Als nächstes werden Texture Splatting und Texture Bombing verwendet, um die Struktur der Terraintexturen aufzubrechen. Als erstes wird dazu eine Variationstextur an den vorliegenden UV-Koordinatenpaaren gesampelt. Diese Textur besteht aus einer zuvor erstellten Perlin-Noise-Textur. Danach werden dieselben Koordinatenpaare genutzt, um die entsprechenden Channel einer weiteren Perlin-Noise-Textur zu sampeln. Diese Textur enthält unterschiedliche Noise-Werte in jedem ihrer vier Channel. Die daraus entstehenden Werte stellen die Faktoren für die Texturmischung der Variationen in den Texturarrays dar. Mit diesen Variations- und Mischwerten werden im Anschluss die Informationen innerhalb der jeweiligen Texturarrays miteinander vermischt. Die Vorgehensweise ist hierbei größtenteils gleich zu dem in Kapitel 5.3 beschriebenen Verfahren von Quilez. Über den berechneten Variationswert wird anfangs ein Index und aus diesem wiederum zwei Offsets berechnet. Anstatt aber im Anschluss diese Offsets direkt zu nutzen, um zwischen zwei Variationen der gleichen Textur zu mischen, wird zuerst der zuvor berechnete Mischfaktor genutzt, um zwischen den beiden Texturvariationen im Texturarray zu interpolieren. Erst im Anschluss werden die Ergebnisse des Texture Splattings weiter miteinander gemischt. Je Projektionsrichtung entsteht dadurch ein Farbwert für jedes der drei Texturarrays.

Während die letztendlichen Farbwerte und meisten anderen Texturinformationen direkt über die Mischfaktoren des Triplanar Mappings vermischt werden können, entsteht im Zusammenhang mit den Normalen der Normal-Maps ein Problem mit deren Ausrichtung. Die grundlegende Ursache dieses Problems und verschiedene Mischmethoden, die bessere Normalen erzeugen, werden im Detail von Golum und Hill beschrieben (Golum, 2017, Hill, 2012). Das Blending der Normalen verwendet die von Golum beschriebene Version des Whiteout-Blends (Golum, 2017, S. 1). Da die Tangenten und Bitangenten nicht vorhanden sind, werden diese, wie von Golum beschrieben, über das Swizzling der Komponenten der eigentlichen Weltnormale approximiert (Golum, 2017, S. 1). Daraufhin wird der Whiteout-Blend durchgeführt.

Wie bereits erwähnt, enthalten die Schluchten der Flugsimulation ebenfalls Ansammlungen an Wasser, um sie etwas weniger monoton zu gestalten. Die Wasserobjekte selbst bestehen dabei aus nicht mehr als einer Plane. Der Materialshader, über den der Wassereffekt erzielt wird, entspricht größtenteils dem Shader wie er in einem von Unity veröffentlichten Video¹⁴ beschrieben ist. Dieser Shader nutzt im Grunde zwei Normal-Maps zur Erzeugung von Welleneffekten. Die UV-Koordinaten zum Sampeln der beiden Normal-Maps bewegen sich dazu über die Zeit hinweg mit unterschiedlichen Geschwindigkeiten in entgegengesetzte Richtungen. Der Tiefeneffekt des Wassers wird im Shader zudem über die Tiefe der Szene und den Bildschirmpositionen der Fragments erzeugt.

Um den Spieler zudem visuell davon abzuhalten aus der Schlucht zu fliegen, werden an der oberen Grenze der Schluchten Sandsturm-Objekte genutzt. Diese Objekte verwenden Unitys Partikelsystem, um die Illusion eines Sandsturms zu erzeugen. Die für die einzelnen Partikel genutzten Sprites basieren hierbei ebenfalls wieder auf Noise. Dazu werden mehrere Oktaven an Perlin Noise kombiniert und die entstehenden Werte als Alphawerte einer Textur interpretiert, um bestimmte Bereiche transparent wirken zu lassen. Durch eine leichte Weiterverarbeitung entsteht eine Art Wolkentextur, über die die letztendlichen Sandpartikel des Sandsturms erzeugt werden.

¹⁴ Youtube - Unity - Making a Water Shader in Unity with URP! (Tutorial) - <https://www.youtube.com/watch?v=gRq-IdShxpU>

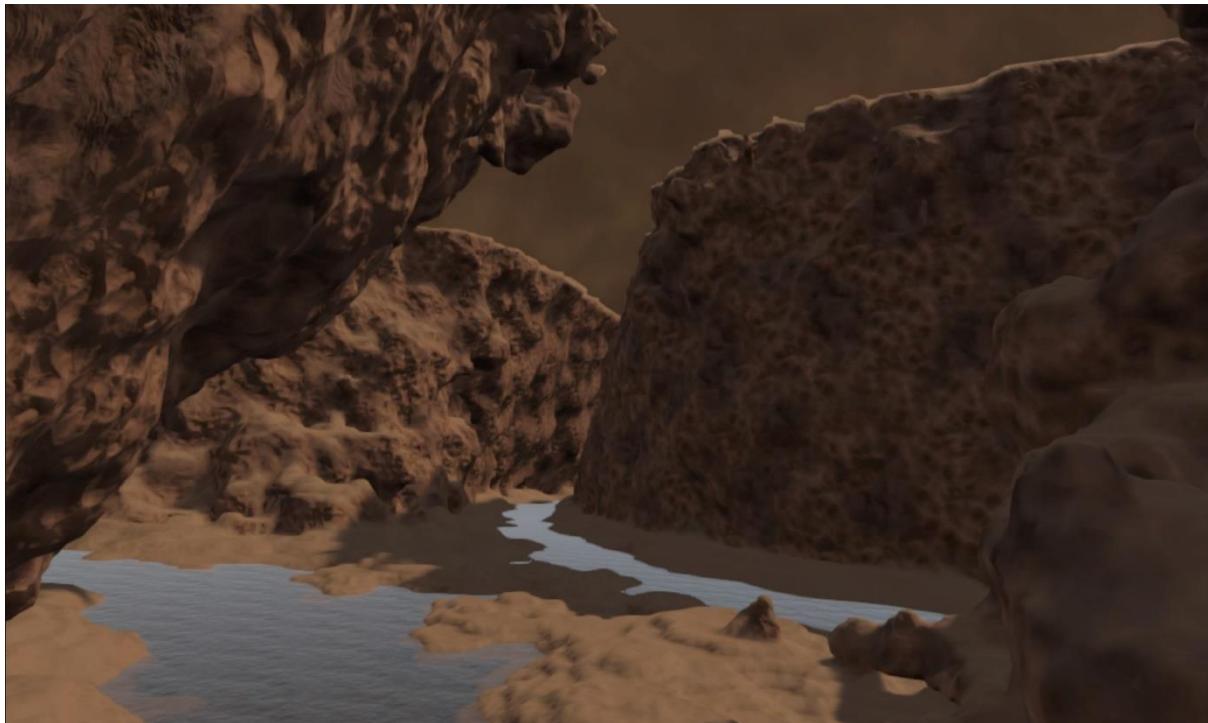


ABBILDUNG 26: AUSSCHNITT AUS DEM TEXTURIERTEN TERRAIN.

8 IMPLEMENTATION DER SPIELELEMENTE

Neben der generierten Landschaft benötigt die Flugsimulation natürlich auch verschiedene Spielemente, um wirklich spielbar zu sein. Der Term Spielemente ist hierbei eher vage gehalten und bezeichnet jegliche Elemente, welche direkt oder indirekt zum letztendlichen Erlebnis des Spielers beitragen.

8.1 HAUPTMENÜ

Damit der Spieler nicht direkt in die Spielwelt geworfen wird, existiert ein Hauptmenü, in dem der Spieler verschiedene Optionen wählen kann. VR-freundlich befinden sich alle UI-Elemente des Hauptmenüs im Koordinatensystem der Welt. Das Design der Menüfläche ist futuristisch gehalten und stellt eine Art Hologramm dar. Der Shader des Hologramm-Materials nutzt ebenfalls wieder Perlin Noise, um den Hologrammeffekt zu erzielen. Dazu werden zwei Oktaven an Perlin Noise mit einem Fresneleffekt und leichtem Flickern kombiniert. Die Samplekoordinaten für das Perlin Noise werden über die Zeit verschoben und mittels Modulo-Rechnungen modifiziert, um die schlagartigen Helligkeitswechsel der horizontalen Streifen im Hologramm zu erzeugen. Das Hauptmenü selbst enthält verschiedene Menüpunkte. Die ersten beiden erlauben es dem Spieler entweder ein neues Level zu generieren oder das vorherige Level erneut zu spielen. Der dritte Menüpunkt öffnet das Settingsmenü. In diesem kann der Spieler verschiedene Einstellungen vornehmen. Dazu zählen unter anderem das Zurücksetzen des Highscores, das Anpassen der Missionslänge und das Ändern der Effektlautstärke. Alle dieser Settings werden, zusammen mit dem Highscore, zwischen Spielsessions in einer separaten Datei gespeichert. Damit der Spieler eine grundsätzliche Idee über die Spielwelt und eine Einführung in die Mechaniken des Spieles bekommen kann, enthält das Hauptmenu ebenfalls Menüpunkte für ein Story- und ein Tutorialmenü, in denen entsprechender Weise die Geschichte des Spieles und die grundlegenden Mechaniken beschrieben werden. Als letztes ist zum Beenden der Simulation natürlich auch ein Quit-Button im Hauptmenü vorhanden. Neben dem UI für das

Hauptmenü befinden sich links und rechts davon ebenfalls noch weitere UI-Elemente, welche die Effekte der aufhebbaren Items in der Spielwelt beschreiben und den Spieler über die Tastenbelegung des Icaros-Controllers informieren.



ABBILDUNG 27: DAS HAUPTMENÜ UND DIE BESCHREIBUNG DER ITEMS UND TASTENBELEGUNGEN.

8.2 SPAWNEN DES SPIELERS

Das Spawning des Spielers stellt eine weitere Herausforderung für die Implementation der Flugsimulation dar. Damit der Spieler nicht immer wieder an derselben Position spawnt, wird jedes Mal, wenn ein neues Level generiert wird, eine neue Startposition in der Spielwelt benötigt. Um immer dieselbe Position für ein bestimmtes Level erhalten zu können, wird ein Seed in der Form eines Integer-Wertes verwendet. Mit diesem Seed wird beim Erzeugen der Spielwelt der Zustand von Unitys Random-Klasse initialisiert. Im Anschluss können die Komponenten der Startposition zuverlässig auf pseudozufällige Werte in einem bestimmten Bereich gesetzt werden. Durch die Anpassung des Seeds können dadurch zufällige Startpositionen gewählt werden, wenn ein neues Level generiert wird, und die letzte Startposition beibehalten werden, wenn der Spieler das letzte Level wiederholt.

Die gewählte Startposition garantiert jedoch nicht, dass der Spieler nicht im Terrain oder ausgerichtet in die Richtung des Terrains spawnt. Im Gegenteil ist es aufgrund des Schluchtterrains extrem wahrscheinlich, dass die bisherige Startposition innerhalb des Terrains liegt. Um dies zu verhindern, wird die Terrainfunktion herangezogen. Ausgehend von der erhaltenen Startposition wird die Terrainfunktion schrittweise in die X-Richtung abgetastet, bis eine Position gesampelt wird, welche außerhalb des Terrains liegt. Diese Position liegt zwar nichtmehr direkt im Terrain aber je nach Schrittgröße immer noch relativ nahe an diesem. Daher wird derselbe Ablauf ausgehend von dieser Position nochmals wiederholt, bis eine Position erreicht wird, welche wieder innerhalb des Terrains liegt. Wird anschließend wieder ein Schritt in die entgegengesetzte Richtung getätigt, entstehen, zusammen mit der ersten Position, zwei Punkte dessen verbindender Bereich außerhalb des Terrains liegt. Der Mittelpunkt der verbindenden Gerade zwischen diesen Punkten ist eine erste potenzielle Spawnposition für den Spieler. Da jedoch mit dieser Vorgehensweise nur garantiert ist, dass die Samplepunkte entlang der X-Achse außerhalb des Terrains liegen, ist noch nicht garantiert, dass das Umfeld dieser Position auch genug Platz bietet, um den Spieler zu spawnen. Dies muss also ebenfalls noch geprüft werden. Dazu wird die Terrainfunktion einfach in bestimmten Abständen an mehreren Positionen um die

potenzielle Spawnposition gesampelt. Sobald eine dieser Positionen innerhalb des Terrains liegt ist die Position nicht mehr valide und der bisherige Ablauf muss ausgehend von der letzten Endposition wiederholt werden. Befinden sich alle gesampelten Positionen außerhalb des Terrains, ist der Bereich um den Spawnpunkt höchstwahrscheinlich offen genug für den Spieler. Je nach den Eigenschaften des Terrains funktioniert dieser Ansatz mehr oder weniger gut. Im Kontext der Schluchten der Flugsimulation ist ein Sampleabstand von zwei Einheiten gut genug, um keine falschen Ergebnisse zu liefern. Bei einem Terrain mit feineren Objekten könnte es jedoch dazu kommen, dass solche feinen Objekte zwischen den Samples liegen und daher nicht einbezogen werden. Folglich könnte der Spieler weiterhin teilweise oder komplett im Terrain spawnen. In diesen Fällen müsste eine feinere Samplerate oder ein anderes Verfahren gewählt werden. Neben der Spawnposition muss auch noch die Orientierung des Spielers angepasst werden, damit dieser nicht ausgerichtet in die Richtung des Terrains spawnt. Dafür kann der Gradient der Terrainfunktion an der Spawnposition verwendet werden. Indem die X- und Z-Koordinaten des resultierenden Gradienten vertauscht werden, entsteht ein Richtungsvektor, welcher mehr oder weniger entlang der Schlucht zeigt. Die erhaltene Spawnposition und -orientierung kann anschließend zurückgegeben werden, um den Spieler an der richtigen Position und mit der richtigen Ausrichtung zu spawnen. Nachdem sich die Spielwelt um den Weltursprung abspielt, wird beim folgenden Aktualisieren des Terrains das Ursprungsoffset der Spielwelt basierend auf der gewählten Spawnposition aktualisiert und alle Chunks erhalten direkt die richtigen Samplepositionen.

8.3 DRILL-RAY

Nachdem der Spieler in manchen Fällen in einem relativ abgegrenzten Bereich spawnen kann, enthält die Flugsimulation eine weitere Mechanik, mit der der Spieler diese Bereiche verlassen kann. Diese Mechanik ist der sogenannte Drill-Ray. Dieser ermöglicht es dem Spieler einen temporären Tunnel durch das Schluchtterrain zu bohren. Wenn der Spieler den Drill-Ray feuert, werden zunächst einmal nur Informationen zur Position des Drill-Rays gespeichert. Beim nächsten Terrainupdate werden diese Informationen dann genutzt, um die betroffenen Chunks zu aktualisieren. Der Drill-Ray besitzt einen Collider in der Spielwelt. Die grundlegende Form dieses Colliders ist eine langgezogene Kapsel. Um alle betroffenen Chunks mit diesem Collider zu finden, werden die Octrees durchlaufen und für jede Octreezelle geprüft, ob der Collider des Drill-Rays mit der Zelle kollidiert. Da die Octreezellen immer entlang der Hauptachsen ausgerichtet sind, ist dies letztendlich ein einfacher Kollisionscheck mit den Boxen, welche die Octreezellen repräsentieren. Wird eine Zelle nicht vom Drill-Ray geschnitten, kann anschließend das Durchlaufen aller Kinder dieser Zelle abgebrochen werden, da diese ebenfalls nicht getroffen werden können. Nach dem Durchlaufen der Octrees kann für die betroffenen Chunks ebenfalls eine Aktualisierung angefordert werden. Da der erzeugte Tunnel kein permanenter Teil der Spielwelt ist, muss der gleiche Vorgang ebenfalls ausgeführt werden, wenn der Tunnels des Drill-Rays wieder verschwindet. Die Daten des Drill-Rays beeinflussen das Ergebnis der Terrainfunktion und werden daher beim Aktualisieren der Chunk- und Seammeshe an die Compute Shader übergeben. An der bisherigen Terrainfunktion selbst ist keine direkte Änderung nötig. Stattdessen wird für eine Position ein separater Terrainwert für den Drill-Ray berechnet und auf den eigentlichen Terrainwert addiert. Der Terrainwert des Drill-Rays berechnet sich hierbei basierend auf der Distanz der Eingabeposition zu der Kapsel des Drill-Rays. Liegt die Position innerhalb der Kapsel wird ein sehr niedriger Wert zurückgegeben, wodurch garantiert wird, dass der letztendliche Terrainwert an dieser Stelle als außerhalb des Terrains klassifiziert ist. Liegt die Position nicht zu nahe am Drill-Ray wird 0 zurückgegeben und der eigentliche Terrainwert folglich nicht verändert.

Das plötzliche Verschwinden eines Teiles des Terrains würde ohne jegliche Effekte merkwürdig wirken und schlecht für das allgemeine Spielerlebnis des Spielers sein. Daher werden mehrere Effekte genutzt, um die schlagartige Terrainmodifizierung größtenteils zu verdecken. Wie auch der Sandsturm nutzen diese Effekte ebenfalls Unitys Partikelsysteme.

Wenn der Tunnel verschwindet, kann es zusätzlich dazu kommen, dass sich der Spieler noch komplett oder teilweise in diesem befindet. In diesen Fällen muss der Spieler abstürzen und die Mission als fehlgeschlagen gelten. Während Fälle, in denen sich der Spieler zu diesem Zeitpunkt nur teilweise im Terrain befindet, durch die Collider der Chunks gehandhabt werden, ist dies nicht der Fall, wenn sich der Spieler komplett innerhalb des Terrains befindet. Das Fehlgeschlagen der Mission wird in einem solchen Fall in zwei Schritten geprüft. Der erste Schritt prüft, ob sich die Position des Spielers in dem kapselförmigen Collider des Drill-Rays befindet. Ist dies nicht der Fall, befindet sich der Spieler nicht innerhalb des erzeugten Tunnels und kann sich daher auch nicht innerhalb des Terrains befinden. Ist dies jedoch schon der Fall, wird der zweite Prüfungsschritt durchgeführt. Der Kapselcollider des Drill-Rays deckt nicht nur einen Bereich innerhalb des Terrains ab. Daher muss für den zweiten Schritt zusätzlich die Terrainfunktion verwendet werden, um sicherzustellen, dass sich der Spieler nicht im Collider aber außerhalb des Terrains befindet.

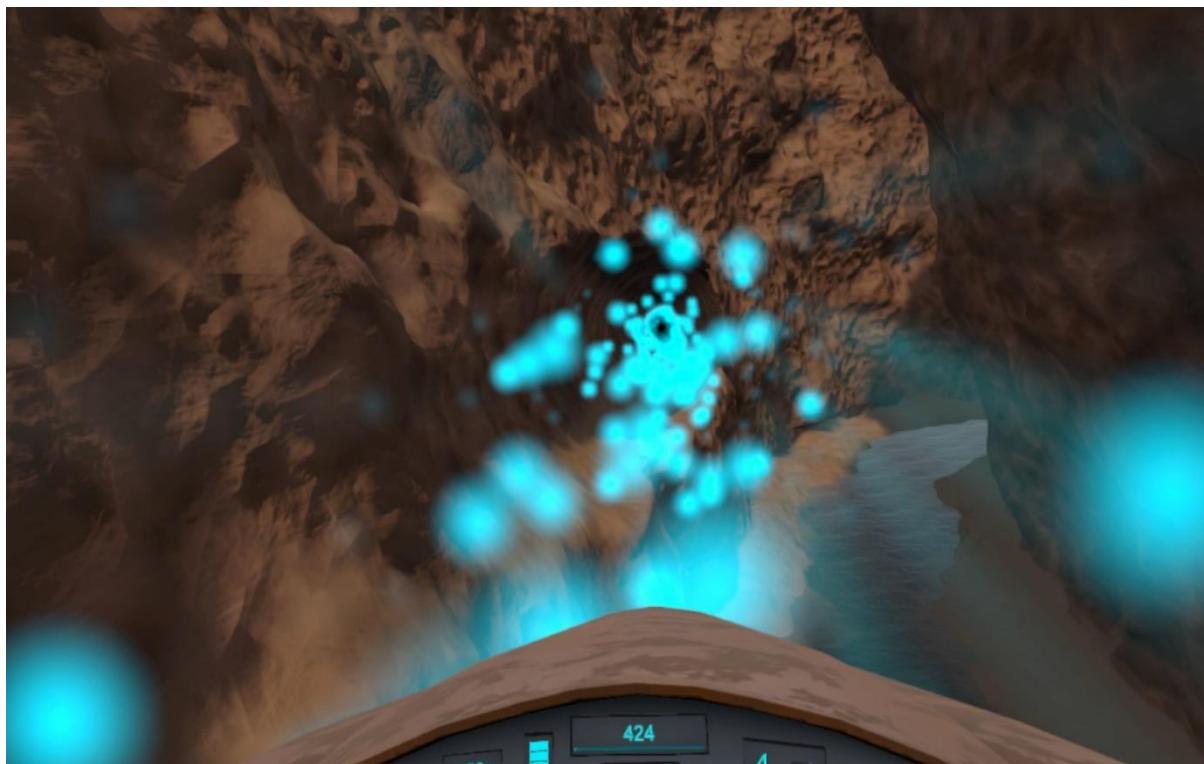


ABBILDUNG 28: DER EFFEKT NACH DEM SCHIEßen DES DRILL-RAYS.

8.4 ITEMS

In der Spielwelt kann der Spieler insgesamt drei verschiedene Items finden und aufsammeln. Das erste dieser Items ist eine Kiste welche Mineralproben enthält. Diese Kiste gibt dem Spieler eine bestimmte Anzahl an Punkten. Das zweite Item ist ein Multiplikator für die Punkte. Sobald der Spieler diesen einsammelt, werden alle Punkte, die er sammelt, für eine bestimmte Zeit verdoppelt. Das letzte Item ist eine Drill-Ray-Charge, welche es dem Spieler beim Einsammeln erlaubt den Drill-Ray ein weiteres Mal zu feuern. Damit die Items in der Spielwelt leichter zu erkennen sind, besitzen diese ein Energieschild. Dieses Schild besteht lediglich aus einer Kugel mit einem transparenten Material. Der Materialshader nutzt ebenfalls wieder eine Perlin-Noise-Textur und Triplanar Mapping um ein passendes Muster ohne Verzerrungen auf die Kugel zu projizieren.

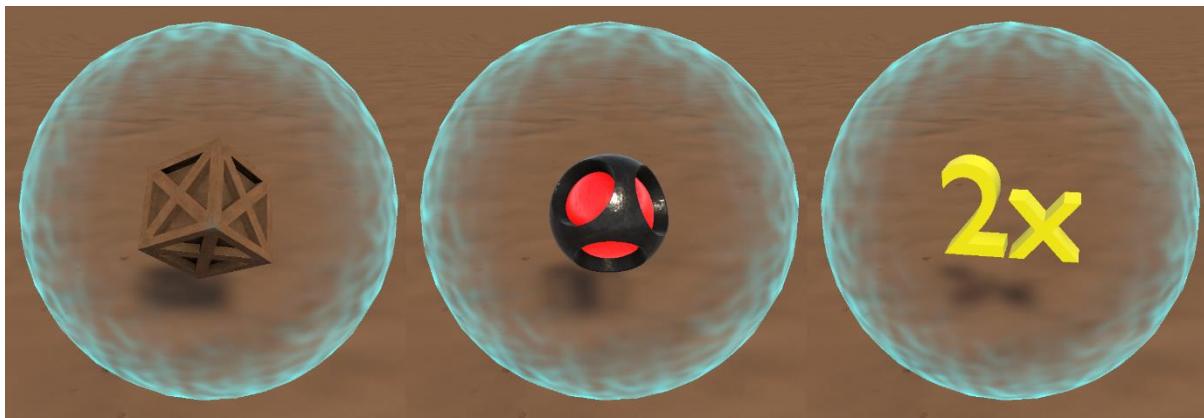


ABBILDUNG 29: DIE DREI ITEMS DER FLUGSIMULATION INKLUSIVE DER ENERGIESCHILDER.

Zum Platzieren der Items innerhalb der Spielwelt findet dieselbe Vorgehensweise wie beim Finden der Spawnposition des Spielers Anwendung. Die einzigen Unterschiede bestehen darin, dass die Orientierung der Items an sich irrelevant ist und als solches nicht berechnet werden muss, und dass die initialen Positionen für die Suche der Itempositionen nicht komplett zufällig gewählt werden. Zur Bestimmung der initialen Positionen werden stattdessen die neun Teilbereiche der Spielwelt von oben betrachtet diagonal durchwandert. Entlang dieser Diagonalen werden Schrittweise Positionen ausgewählt. Die Höhe der initialen Positionen wird über Perlin Noise innerhalb eines bestimmten Bereiches gewählt.

Die vorhandenen Items in der Spielwelt werden nur dann aktualisiert, wenn sich die Spielwelt verschiebt. In diesem Fall ist es zunächst nötig alle theoretisch benötigten initialen Positionen in den Teilbereichen zu berechnen. Diese Position ist immer dieselbe für ein bestimmtes Item und kann daher als Key in einem Dictionary und einem HashSet genutzt werden, um bereits vorhandene und aufgehobene Items auszusortieren. Ähnlich werden alle Items, welche nichtmehr benötigt sind, zur Löschung freigegeben. Für alle neuen Items, die nach der Aktualisierung benötigt werden, findet dann die Berechnung der tatsächlichen Positionen für die Platzierung statt. Welches der drei Itemarten dabei gespawnt wird, erfolgt ebenfalls wieder über Perlin Noise. Dazu wird relativ hochfrequentes Perlin Noise genutzt, um einen pseudozufälligen Wert zwischen 0 und 1 zu erhalten. Aus diesem kann dann, mittels gesetzter Itemwahrscheinlichkeiten, die Wahl der entsprechenden Itemart stattfinden.

8.5 EXPEDITIONSBIKE

Das Fortbewegungsmittel in der Flugsimulation ist ein altes Expeditionsbike. Das generelle Aussehen dieses Bikes ist aufgrund des Settings der Flugsimulation auch eher futuristisch gehalten. Nachdem der Spieler die Spielwelt mittels VR wahrnimmt und daher die Immersion des Spielers im Vordergrund steht, ist das Aussehen so gewählt, dass die Lage auf dem Bike der Lage des Spielers auf dem Icaros entspricht. Zusätzlich enthält das Bike weitere Elemente, welche die Immersion erhöhen. Während der Spieler im Icaros liegt, stützt er sich mit den Schienbeinen und Unterarmen auf diesem. Diese Ablagen sind auch im Modell des Bikes wiederzufinden. Das gleiche gilt für die Griffe, an denen sich der Spieler am Icaros festhält. Weiterhin zur Immersion beitragend ist das gesamte UI in das Bike integriert. Der Großteil der Informationen wird dabei über Displays vermittelt. Über diese erhält der Spieler beispielsweise Aufschluss über den momentanen Punktemultiplikator, die verbleibende Anzahl an Drill-Ray-Charges und deren Dauer, und natürlich dem momentanen Punktestand. Das einzige UI-Element, das nicht über diese Displays dargestellt wird, ist das Pausenmenü. Dieses öffnet sich als größeres Hologramm vor dem Spieler, sobald er das Spiel pausiert.

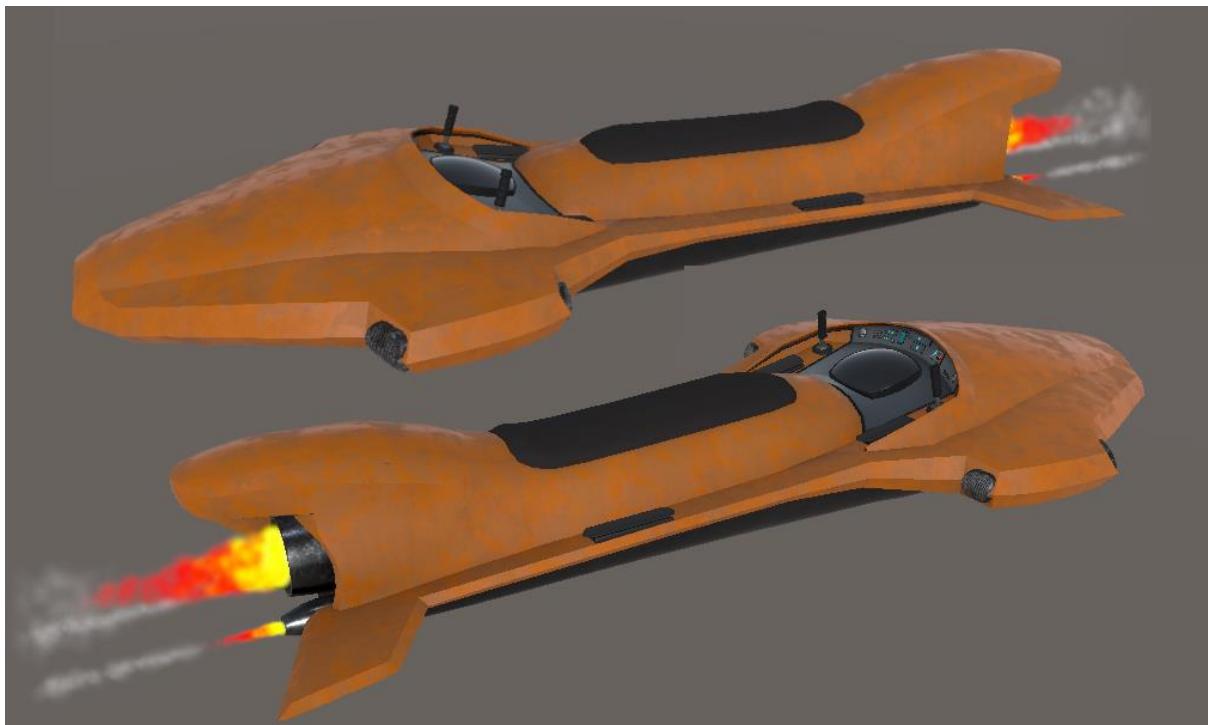


ABBILDUNG 30: ZWEI ANSICHTEN DES EXPEDITIONSBIKES.

Damit der Spieler einerseits wissen kann wo sich Items in der Spielwelt befinden, um diese gezielt einzusammeln, und damit er andererseits das grobe Layout der Schluchten in seiner Umgebung kennen kann, um sich auf eventuelle Kurven vorzubereiten, besitzt das Expeditionsbike eine Minimap. Die Minimap wird über einen Compute Shader in eine Textur gerendert, welche am Ende als Textur auf dem Minimapdisplay gesetzt wird. Als Basis für diese Minimap dient ein Teil der Terrainfunktion, genauer genommen die Potenzierung der beiden ersten Oktaven an Perlin Noise. Dadurch ergibt sich der grobe Pfad des Terrains. Dieser ist ausreichend für die Textur der Minimap, da der Spieler keine zu detaillierten Informationen über die Gegebenheiten des Terrains benötigt. Der grundlegende Pfad wird im Compute Shader an der Höhe des Spielers entlang der X- und Z-Achse ausgewertet. Anschließend werden den erhaltenen Werten an jeder Texturposition über einen Schwellwert eine von zwei Farben zugewiesen. Der Spieler selbst wird mittels einer weiteren Farbe dargestellt. Dazu wird die Farbe aller Texturpositionen dessen Samplepunkte sich nahe genug an der Spielerposition befinden durch eine weitere Farbe ersetzt. Die gleiche Vorgehensweise wird für die Itempositionen verwendet. Während es bei den Farben der Displays und Menüs aufgrund der unterschiedlichen Farbintensitäten nicht zu Problemen im Zusammenhang mit eventuellen Sehschwächen des Spielers kommen sollte, ist dies nicht zwingend der Fall für die Minimap. Im Gegenteil sind Unterscheidungsprobleme bei einer entsprechenden Sehschwäche sogar sehr wahrscheinlich, da standardmäßig Grün-, Rot- und Gelbtönen verwendet werden. Deshalb gibt es in den Settings eine Option aus verschiedenen Presets an Farbkombinationen für die Minimap zu wählen.

Die Kontrolle des Bikes erfordert Informationen über die Positionierung des Spielers im Icaros und den Eingaben mittels des Controllers des Icaros. Diese Daten können über die vom Icaros-SDK bereitgestellten Komponenten abgefragt werden. In jedem Aktualisierungsschritt wird die Rotation des Bikes um die X-Achse mit der X-Achsenrotation des Icaros aktualisiert. Damit das Bike nach links und rechts lenkt, sobald sich der Spieler in eine der beiden Richtungen lehnt, wird die Rotation des Bikes um die Y-Achse mit der Rotation des Icaros um die Z-Achse aktualisiert. Visuell spiegelt sich die Rotation um die Z-Achse zusätzlich in der Rotation des Bikemodells um die Z-Achse wider.

Die Beschleunigung des Bikes erfolgt über einen der Knöpfe des Controllers. In jedem Aktualisierungsschritt wird das Bike basierend auf der aktuellen Geschwindigkeit um einen weiteren Schritt in die momentane Vorwärtsrichtung verschoben. Um zu verhindern, dass der Spieler aus der Schlucht fliegen kann, nähert sich die Y-Komponente des Geschwindigkeitsvektors immer näher an 0 an, sobald sich der Spieler nahe genug an der oberen Schluchtgrenze befindet und versucht nach oben zu fliegen. Neben der Beschleunigungsfunktion existierte zwischenzeitlich auch eine Abbremsfunktion. Aufgrund der limitierten Anzahl an nutzbaren Knöpfen ist diese Funktion allerdings durch ein passives Abbremsen ersetzt worden.

Die allgemeine Kontrolle des Icaros erfordert in der Praxis viel Übung. Beim Testen der Flugsimulation viel direkt auf, dass sich die feine Kontrolle beim Lehnen in Kurven als problematisch erweist. Um dieses Problem zu lösen, existiert ein weiteres Spielelement in der Form des Turn-Speed-Boosters. Dieser erlaubt es dem Spieler die Rate, mit der sich das Bike dreht, bei Bedarf mit einem Knopfdruck zu erhöhen. Folglich kann die standardmäßige Sensitivität der Rotationen relativ niedrig gehalten werden, wodurch es dem Spieler leichter fällt feine Rotationen durchzuführen. Damit der Spieler den Booster nicht durchgehend verwenden kann und stattdessen entscheiden muss wann es notwendig ist diesen zu nutzen, besitzt der Booster einen beschränkten Zeitraum, für den er aktiv bleiben kann. Wird der Booster nicht genutzt wird der nutzbare Zeitraum langsam wieder auf den ursprünglichen Wert zurückgesetzt.



ABBILDUNG 31: DAS IN DAS EXPEDITIONSBIKE INTEGRIERTE UI.

8.6 SAMMELN VON PUNKTEN

Der Spieler kann in der Flugsimulation auf zwei Weisen Punkte sammeln. Die erste der beiden wurde bereits in der Form des Kisten-Items erwähnt. Die zweite Möglichkeit stellt eines der grundlegenden Spielelemente der Flugsimulation dar. Sollte sich der Spieler nahe genug am Terrain befinden sammelt das Expeditionsbike Stichproben des Terrains. In anderen Worten ist die zweite Möglichkeit also das Sammeln von Punkten über die Nähe zur Terrainoberfläche der Spielwelt. Umgesetzt wird das Sammeln der Punkte über einen Collider, der am Expeditionsbike hängt, und die Collider der Chunks, an denen der Spieler

vorbeifliegt. Während sich ein Chunkcollider und der Collider des Bikes schneiden, erhöht sich der Punktestand des Spielers. Damit der Spieler einen Grund hat möglichst lange in der Nähe des Terrains zu fliegen existiert zusätzlich ein Multiplikator für die erhaltenen Punkte, welcher sich erhöht, wenn sich der Spieler in der Nähe des Terrains befindet, und sich langsam wieder zurücksetzt, wenn sich der Spieler vom Terrain entfernt. Ein ähnlicher Multiplikator wird verwendet, um den Spieler dazu zu bewegen möglichst schnell durch die Schluchten zu fliegen. Dieser Multiplikator ist entsprechend umso höher desto schneller der Spieler fliegt. Um also möglichst viele Punkte zu sammeln, muss der Spieler möglichst schnell und nahe entlang des Terrains fliegen.

8.7 SCHWIERIGKEITSGRAD

Damit sowohl Spieler ohne Erfahrung als auch Spieler mit mehr Übung in der Kontrolle des Icaros die Flugsimulation spielen können, ohne ein zu schweres oder zu leichtes Spielerlebnis zu erleben, enthält die Flugsimulation neben dem Turn-Speed-Booster ebenfalls noch drei verschiedene Schwierigkeitsgrade. Der gewählte Schwierigkeitsgrad beeinflusst dabei verschiedene Aspekte der Simulation, um ein mehr oder weniger anforderndes Erlebnis zu bieten. Anfangend mit den Funktionen des Expeditionsbikes sind die ersten dieser Aspekte die Daten des Drill-Rays. Der Schwierigkeitsgrad beeinflusst in diesem Kontext nicht nur die Anzahl der maximal zur Verfügung stehenden Drill-Ray-Charges, sondern auch den Radius des erzeugten Tunnels und die Dauer für den dieser offenbleibt. Neben den Daten des Drill-Rays wird auch die maximale Dauer, Aufladrate und Stärke des Turn-Speed-Boosters beeinflusst. Damit in höheren Schwierigkeitsgraden das allgemeine Gameplay der Simulation auch anspruchsvoller ist, werden grundlegendere Daten der Flugsimulation ebenfalls beeinflusst. Neben einer höheren Mindestgeschwindigkeit des Bikes und einem kleineren Radius zum Sammeln der Punkte, fallen die Schluchten bei höheren Schwierigkeitsgraden allgemein enger aus. Dazu muss, wie in Kapitel 7.6 erwähnt, lediglich der Exponent bei der Bildung des grundlegenden Schluchtenpfades erhöht werden, um eine engere Schlucht zu erzeugen. Da es für den Spieler keinen direkten Grund gäbe die Schwierigkeit zu erhöhen, wird ein weiterer Punktemultiplikator beim Überstehen der Missionsdauer eingebracht. Höhere Schwierigkeitsgrade erlauben es dem Spieler also einerseits leichter den bisherigen Highscore zu knacken und andererseits einen insgesamt höheren Highscore zu erreichen.

9 EVALUIERUNG DER SPIELBARKEIT

Der Grad der Spielbarkeit der Flugsimulation wurde über eine Handvoll Tester evaluiert. Jeder der Testpersonen hat dazu zwei Sessions, mit einer zwischenzeitigen kurzen Pause, in der Flugsimulation verbracht und anschließend Feedback über das Spielerlebnis geliefert. Die Testpersonen haben die Flugsimulation dabei insgesamt je zwischen 10 bis 20 Minuten ausprobiert. Der Konsens aller Tester war in der anschließenden Befragung, dass die Flugsimulation als gesamtes spielbar ist. Es wurden allerdings im Feedback individuell auch Probleme auf sowohl der Hardware- als auch auf der Softwareseite angemerkt, durch welche die letztendliche Spielbarkeit eingeschränkt wird.

Auf Seiten der Hardware entstehen die einschränkenden Probleme aufgrund der Nutzung des Icaros zur Interaktion mit der Flugsimulation. Alle Tester, die keinerlei Vorerfahrung mit der Kontrolle des Icaros besaßen, hatten vor allem am Anfang der ersten Session merkbare Probleme mit der Steuerung des Expeditionsbikes durch die präzise Verlagerung des Gleichgewichts. Nach einer relativ kurzen Eingewöhnungszeit fiel dies allerdings um einiges milder aus. Vor allem in der zweiten Session hatten die Tester viel geringere Probleme mit der Kontrolle des Icaros. Fast alle Tester merkten jedoch an, dass die Kontrolle sehr gewöhnungsbedürftig und anstrengend sei, und ein Großteil ihrer Aufmerksamkeit auf die feinen Gleichgewichtsverlagerungen fokussiert werden musste. Dies hat unter anderem

dazu geführt, dass es den Nutzern in beiden Sessions schwerfiel das UI des Expeditionsbikes zu beachten, da sie sich auf das Lenken des Bikes konzentrieren mussten. In diesem Kontext wurde die Minimap von den meisten Testern komplett außer Acht gelassen. In der zweiten Session war das Beachten des UIs allerdings ein geringeres Problem als in der ersten Session.

Ebenfalls als problematisch erwies sich für die meisten Tester die Empfindlichkeit bei den seitlichen Rotationen des Icaros. Ab einem bestimmten Rotationswinkel fielen die Tester sozusagen in eine Richtung und es war schwer für sie Rotationen in diesem Bereich auszuführen. Durch den Turn-Speed-Booster und nach der Anpassung der Rotationsempfindlichkeit in den Settings fiel dieses Problem laut den Testern jedoch viel geringer aus. Auch die Neigungen nach vorne und hinten machten einigen Testern anfangs Probleme. Nach der Anpassung der Armlehnen löste sich dieses Problem allerdings ebenfalls größtenteils.

Auf Seiten der Software entstanden Limitierungen der Spielbarkeit größtenteils aufgrund der Positionierung und Ausrichtung von virtuellen Objekten. Aufgrund des leichten Blicks nach unten, welcher während des Liegens im Icaros auftritt, meldeten die Tester Probleme mit der Auswahl einiger Menüpunkte des Hauptmenüs. Vor allem die weiter oben gelegenen Optionen benötigten laut den Testern eine eher unangenehme Kopfneigung. Im Anschluss ist das gesamte Hauptmenü daher etwas rotiert und verschoben worden, um die Auswahl der Menüpunkte weniger anstrengend zu gestalten.

Ein ähnliches Problem entstand bei der Ausrichtung des Modells des Expeditionsbikes. Durch den leichten Blick nach unten verdeckte das Modell hierbei einen etwas zu großen Bereich des Sichtfeldes, wodurch es manchen Testern in bestimmten Situationen schwer fiel ihre Position im Vergleich zu dem Terrain der Schluchten einzuschätzen. Dies wurde zwischen der ersten und zweiten Session behoben, indem das Modell des Bikes leicht nach vorne rotiert wurde. Dies hat laut den Testern einen relativ großen Unterschied gemacht und die Rotation des Modells selbst war nahezu nicht bemerkbar. Hierbei wäre jedoch die Wahl einer besseren Form für das Modell des Expeditionsbikes eine wahrscheinlich bessere Option gewesen.

Abgesehen von der problematischen Ausrichtung bestimmter Objekte, erwies sich die horizontale Ausrichtung des Bikes am Start des Gameplayloops ebenfalls als problematisch. Vorgesehen war es dabei eigentlich, dass der Nutzer die Menüpunkte in der nach hinten gelehnten Ruhelage auswählt und sich dann in die horizontale Ruhelage bewegt bevor das Level startet. Nach dem Laden des Levels hat der Spieler dafür einen kurzen Zeitraum, um sich horizontal im Icaros auszurichten. Für die Tester war dieser Zeitraum in der Praxis allerdings nicht immer ausreichend, was entsprechend zu einem schlechteren Spielerlebnis geführt hat. Nach der Evaluierung wurden daher Änderungen vorgenommen damit das Level nichtmehr zeitbasiert startet, sondern der Spieler einen beliebigen Knopf des Icaros-Controllers drücken muss, um das Level zu starten. Entsprechend wurde ebenfalls ein weiteres Hologramm-UI eingebaut, um dem Spieler mitzuteilen, dass das Drücken eines Knopfes erforderlich ist.

Fünf Punkte, bei denen es vor allem Bedenken bezüglich der Spielbarkeit gab, waren der LOD-Wechsel, die Seams, das Auftreten von Motion Sickness, die Framerate und der allgemeine Spaßfaktor der Flugsimulation. Der Wechsel zwischen den unterschiedlichen LOD-Stufen der Chunks ist allen Testern aufgefallen. Die Tester haben diesen Wechsel allerdings nicht wirklich als allzu störend empfunden. Folglich wurde die Spielbarkeit, wie erwartet, zwar durch die LOD-Wechsel eingeschränkt, jedoch nicht zu einem Punkt ab dem dies problematisch wäre.

Ähnliche Bedenken bestanden bei den Meshen der Seams, durch welche die Chunks miteinander verbunden sind. Zum Zeitpunkt der Evaluierung gab es bei diesen noch Probleme in bestimmten Fällen, wodurch vereinzelt Dreiecke des Seammeshes nicht

korrekt generiert wurden und das Terrain entsprechend Löcher besaß. Diese Löcher fielen den Testern natürlich auf. Entgegen den Erwartungen empfanden die Tester diese Löcher allerdings nicht als ein enormes Problem für die Spielbarkeit der Flugsimulation.

Dank der Wahrnehmung der Spielwelt über VR entstanden ebenfalls Bedenken bezüglich des Auftretens von Motion Sickness in den Testern. Bis auf einen Tester äußerte sich Motion Sickness jedoch in keiner der Testpersonen. Der Tester, bei dem sich Symptome von Motion Sickness äußerten, beschrieb hingegen, dass diese nur kurzzeitig am Anfang auftraten und nach einer kurzen Eingewöhnungszeit kein Problem mehr darstellten. Ob dies allerdings auch über eine längere Spieldauer der Fall ist, kann aufgrund der vergleichsweise kurzen Sessions nicht beurteilt werden.

Da das Spielerlebnis eines Spiels auch stark von der Framerate beeinflusst wird und eine gute Framerate in Kombination mit VR eine noch höhere Bedeutung hat, bestanden hierbei ebenfalls einige Bedenken. Auf dem Rechner, auf dem die Flugsimulation entwickelt wurde, läuft diese im Durchschnitt mit etwa 60 bis 80 Frames pro Sekunde. Der Rechner, auf dem die Evaluierung stattfand, besitzt nochmals bessere Hardware und liefert entsprechend eine noch höhere Framerate. Wie erwartet haben alle Tester daher die Framerate der Flugsimulation als hoch genug empfunden, um ein gutes Spielerlebnis zu liefern.

Der letzte und wichtigste Punkt für die Spielbarkeit, bei denen Bedenken entstanden, war der Spaßfaktor der Simulation. Wenn das Spielen des Endprodukts keinen Spaß macht, kann ein Spiel noch so viele Mechaniken zur Erhöhung der Spielbarkeit besitzen, ohne das letztendliche Ziel wirklich zu erreichen. Glücklicherweise empfanden alle Tester die Flugsimulation als spaßig. Einerseits aufgrund des generellen Gameplayloops und der Mechaniken, andererseits aber auch aufgrund der Wahrnehmung über VR und nicht zuletzt auch der interaktiven Kontrolle des Bikes über den Icaros.

10 DISKUSSION DER ERKENNTNISSE

Abschließend sollte erwähnt werden, dass die Implementation der Flugsimulation für diese Arbeit größtenteils ohne Probleme ab lief und das Endprodukt befriedigende Ergebnisse liefert. Mit den Erkenntnissen dieser Arbeit gibt es im Rückblick allerdings auch einige Punkte die Verbesserungswürdig sind. Die beiden Hauptpunkte stellen dabei das mit der Nutzung der parallelen Rechenkraft einer Grafikkarte verbundene Zurücklesen der Meshdaten auf die CPU und der Ansatz für die Generierung der Seams dar.

Die Nutzung der Grafikkarte für die Erzeugung des Terrainmeshes stellt allgemein gesehen kein Problem dar. Im Gegenteil löst die Grafikkarte durch die parallele Implementierung von Nielsons Dual Marching Cubes die vergleichsweise schlechte Performanz auf der CPU, welche durch die hohe Anzahl an zu prüfenden Punkten entsteht. Potenzielle Probleme entstehen durch diese Vorgehensweise allerdings, sobald das Terrainmesh an bestimmten Punkten direkt (oder zumindest möglichst schnell) nach der Aktualisierungsbeauftragung der Chunks benötigt wird. Während die Grafikkarte sowohl das Chunk- als auch das Seammesh für eine große Menge an Chunks relativ schnell berechnen kann, erfolgt das Rücklesen der generierten Daten asynchron in einem späteren Frame. Modifikationen des Terrains benötigen daher eine gewisse Zeit bis sie sich im Mesh äußern. In der Flugsimulation führt dies zu einigen Problemen. Ein erstes Problem entsteht bei der Terrainmodifikation über den Drill-Ray. Der Tunnel, der mit diesem gebohrt wird, wird nicht direkt generiert, wodurch es einerseits nötig ist die Änderung des Terrains mit einem entsprechenden Effekt zu verdecken, bis das Terrainmesh modifiziert ist, und andererseits dazu führen kann, dass der Spieler mit dem Terrain kollidiert bevor der Tunnel im Terrain auftaucht. Aufgrund der Asynchronität entsteht ein zweites Problem bei der Selektion der zu generierenden Chunks. Im besten Fall würden nur die Chunks generiert werden müssen, welche sich im Sichtfeld des Spielers befinden. Da die Chunks allerdings nicht direkt zur Verfügung stehen, sobald sie angefordert werden, ist es nötig die Chunks außerhalb des

Sichtfelds ebenfalls zu generieren, um zu verhindern, dass der Spieler während der Rotation Lücken im Terrainmesh sieht. Im Nachhinein wäre es daher eventuell besser gewesen einen Ansatz zu wählen, bei dem sich diese Probleme nicht oder zumindest weniger ausgeprägt äußern. Ein solcher Ansatz könnte beispielsweise Threads auf der CPU nutzen oder die Generation eines Chunks über einen oder mehrere Frames aufteilen, um Performanzprobleme zu verhindern. Ebenfalls ausnutzbar könnte die Tatsache sein, dass das Mesh direkt nur für die Berechnung des Colliders auf der CPU nötig ist. Durch eine Anpassung der in dieser Arbeit beschriebenen Implementation könnten die Meshdaten eventuell direkt auf der GPU gerendert werden, sobald diese generiert sind, und separat davon zu einem späteren Zeitpunkt für die Erstellung der Collider auf der CPU ausgelesen werden.

Weitere Probleme entstehen bei der Erzeugung der Übergangsmeshe zwischen den Chunks. Keines der Verfahren, die während der anfänglichen Recherche gefunden wurden, wirkte als ob es sich ohne Probleme auf die Chunkübergänge der Flugsimulation anwenden ließ. Daher fiel die Entscheidung darauf zu versuchen ein eigenes Verfahren für die Generierung der Seammeshe zu entwerfen. Dies hat einen nicht unerheblichen Zeitraum der Arbeit in Anspruch genommen und während der letztendlich entstandene Algorithmus nahezu alle Chunks der Flugsimulation nahtlos miteinander verbindet, gibt es dennoch Fälle, in denen dies nicht der Fall ist. Einer dieser Fälle kann beispielsweise auftreten, wenn sich ein bestimmter Terrainbereich eines Chunks komplett innerhalb einer Zelle eines benachbarten Chunks mit einer niedrigeren Auflösung befindet. Bei der Erzeugung des Seammeshes würden in diesem Fall alle Eckpunkte der größeren Zelle außerhalb des Terrains liegen und keine Vertices generieren. Folglich könnten an diesen Stellen keine korrekten Verbindungen für das Seammesh erzeugt werden. In der Praxis ist dies glücklicherweise kein allzu großes Problem da diese Fälle nur an den Übergängen von Chunks zu Chunks mit größeren Sampleabständen entstehen. In der Flugsimulation zeigen die Lücken daher immer weg vom Spieler und sind für diesen nicht sichtbar. Ein ähnliches Problem entsteht in der Theorie aufgrund der Betrachtung einer limitierten Umgebung bei der Verbindung der Vertices. Bereiche, in denen sich das Terrain an den Seams über die betrachtete Nachbarschaft einer Zelle erstreckt, führen ebenfalls zu Lücken im Terrain, da die für die Verbindung benötigten Vertices nicht in Betracht gezogen werden. Im Code der Implementation existiert zudem ein Fehler, durch den in bestimmten Fällen Dreiecke nicht korrekt generiert werden und daher vereinzelt Löcher im Seammesh auftreten. Die Ursache dieses Fehlers konnte trotz extensiver Suche nicht gefunden werden. Im Rückblick wäre es daher auch hier von eventuellem Vorteil gewesen ein anderes Verfahren zu nutzen. Ein guter Kandidat dafür wäre unter anderem das in Kapitel 7.5 kurz erwähnte Verfahren von Ju et al. (2007), welches im Zusammenhang mit ihrer Manifold Dual Contouring Methode verwendet wird. Dafür müsste allerdings ein Großteil der Implementation umstrukturiert werden und es wäre eine Weiterverarbeitung und Analyse der Vertices erforderlich, welche nicht trivial umzusetzen ist. Für zukünftige Arbeiten wäre daher wahrscheinlich eher ein Ansatz von Vorteil der Übergangszellen nutzt. Eine Adaption von Lengyels Methode sollte aufgrund der Ähnlichkeit von Nielsons Verfahren zu Marching Cubes möglich sein. Die initiale Erstellung der Tabellen wäre zwar relativ aufwändig, das Ergebnis würde allerdings eine ziemlich effiziente Möglichkeit für die Verbindung von Chunks bieten, welche die genannten Probleme dieser Implementation ebenfalls komplett lösen sollte.

Kleinere Verbesserungsmöglichkeiten würden sich ebenfalls im Zusammenhang mit dem LOD-Wechsel bieten. LOD-Popping ist ein Problem in den meisten Videospielen. Es gibt daher auch eine Reihe an Möglichkeiten diese Wechsel weniger auffällig zu gestalten. Eine bessere Implementation könnte beispielsweise zwischen den alten und neuen Chunkmeshen interpolieren, um einen weniger ins Auge stechenden Übergang zu erzeugen.

Ein weiterer Punkt, der zwar kein direktes Problem in der Implementation der Flugsimulation darstellt, aber dennoch erwähnenswert ist, da er in bestimmten Implementationskontexten von Vorteil sein kann, ist die Erzeugung der Werte von Berlin

Noise. Während an den meisten Stellen dieser Arbeit, an denen Noise-Werte benötigt werden, Perlin Noise ausgewertet wird, ist es nicht zwingend nötig diese Werte zu berechnen. Stattdessen könnte ein Ansatz wie der von Geiss genutzt werden. Dieser Ansatz nutzt eine oder mehrere kleine dreidimensionale Texturen, in denen zuvor berechnete Noise-Werte gespeichert sind, um das Noise an bestimmten Positionen zu erhalten (Geiss, 2007, Kapitel 1.3.3). Da sich Perlin Noise ohnehin periodisch wiederholt, könnte damit beispielsweise das Auswerten der Terrainfunktion weniger aufwändig ausfallen, ohne den Detailgrad merkbar zu beeinflussen.

Für die Spielbarkeit der Flugsimulation stellt die initial nötige Einarbeitungszeit zur präzisen Kontrolle des Icaros ein definitives Problem dar. Dadurch haben vor allem neue Spieler anfangs enorme Probleme überhaupt einen merkbaren Fortschritt in der Flugsimulation zu erzielen. Wie die Evaluation zeigt, kann dieses Problem zwar durch entsprechende Spielmechaniken, wie dem Turn-Speed-Booster und der einstellbaren Rotationssensitivität, und einer mehr oder weniger kurzen Einarbeitungszeit bereits zum größten Teil gelöst werden, es erweist sich aber dennoch als problematisch für das Spielerlebnis der Nutzer. Insgesamt liefert der Icaros jedoch eine sehr gute und effektive Möglichkeit das Fitness-Training des Nutzers mit dem Spielspaß immersiver VR-Spiele zu kombinieren. Vor allem in der Kombination mit prozedural generierten Spielwelten zeigt sich ein enormes Potenzial, dem Spieler ein stetig herausforderndes und immer neues Spielerlebnis zu liefern und somit eine spaßige Erfahrung mit positiven Effekten auf dessen Gesundheit zu bieten.

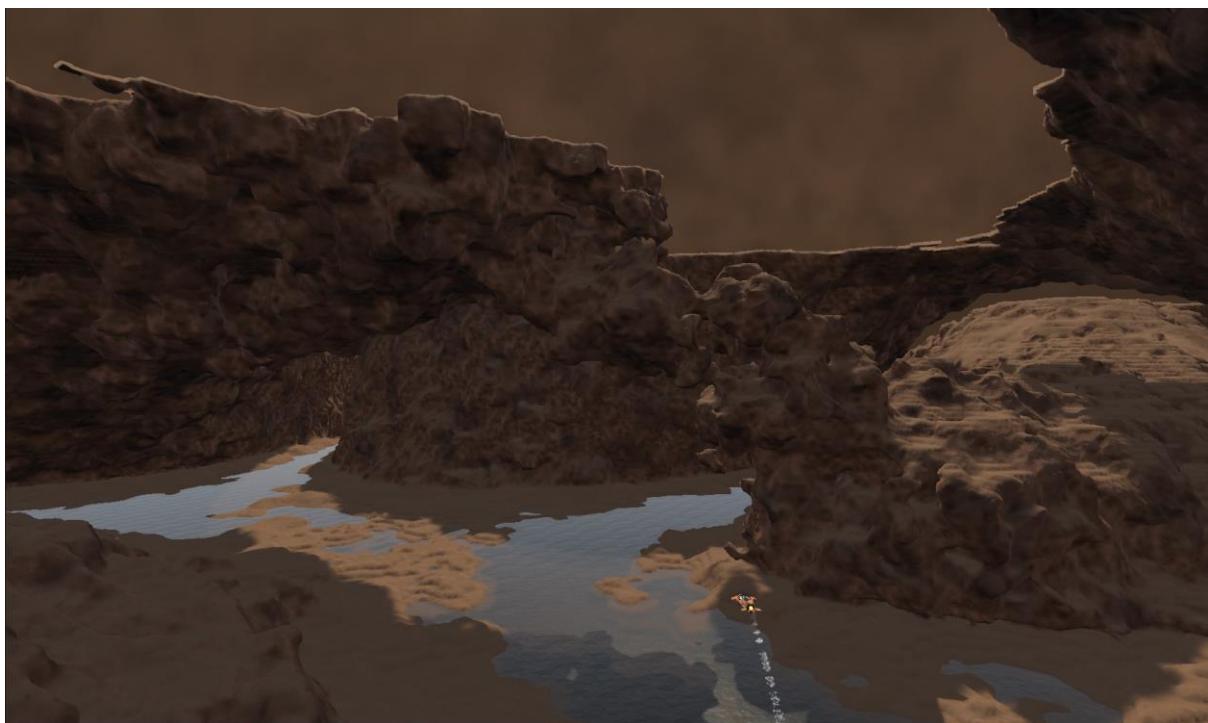


ABBILDUNG 32: EIN LETZTER AUSSCHNITT AUS DER SPIELWELT DER FLUGSIMULATION.

11 ABBILDUNGSVERZEICHNIS

ABBILDUNG 1: AUSSCHNITT AUS DER LETZTENDLICHEN FLUGSIMULATION. EIGENE ABBILDUNG.	5
ABBILDUNG 2: ZWEIDIMENSIONALES BEISPIEL FÜR DIE SCHLECHTE EXTRAKTION EINES MESHES DURCH MARCHING CUBES (RECHTS) AUS EINEM OBJEKT MIT SCHARFEN KANTEN (LINKS). WIE ERSICHTLICH WIRD DAS MESH ZWAR GRUNDLEGEND APPROXIMIERT, DIE SCHARFEN KANTEN GEHEN ALLERDINGS VERLOREN. EIGENE ABBILDUNG.	14
ABBILDUNG 3: ZWEIDIMENSIONALES BEISPIEL FÜR DIE EXTRAKTION EINES MESHES DURCH DUAL CONTOURING (RECHTS) AUS EINEM OBJEKT MIT SCHARFEN KANTEN (LINKS), IN WELCHEM DIE KANTEN DURCH DIE PLATZIERUNG DER VERTICES INNERHALB DER ZELLEN BESSER APPROXIMIERT WERDEN ALS ES BEI MARCHING CUBES DER FALL WÄRE. EIGENE ABBILDUNG.	16
ABBILDUNG 4: BEISPIEL FÜR DEN UNTERSCHIED ZWISCHEN EINER ZELLKONFIGURATION VON MARCHING CUBES UND NIELSONS DUAL MARCHING CUBES. EIGENE ABBILDUNG.	20
ABBILDUNG 5: VERGLEICH ZWISCHEN DEN WERTEN EINER OKTAVE (LINKS) UND VIER OKTAVEN (RECHTS) AN PERLIN NOISE. EIGENE ABBILDUNG.	22
ABBILDUNG 6: VERGLEICH ZWISCHEN EINEM TERRAIN, WELCHES AUF NUR EINER OKTAVE AN NOISE BASIERT (LINKS), UND EINEM TERRAIN, WELCHES HÖHENABHÄNGIG MEHRERE OKTAVEN AN NOISE KOMBINIERT (RECHTS). EIGENE ABBILDUNG.	23
ABBILDUNG 7: VERGLEICH DER TEXTURPROJEKTIONEN IN DER XY-EBENE (LINKS) UND XZ-EBENE (MITTE) MIT DEM ERGEBNIS DES TRIPLANAR MAPPINGS MIT EINEM ERHÖHTEN SCHÄRFEFAKTOREN (RECHTS). EIGENE ABBILDUNG.	25
ABBILDUNG 8: DIE PIMAX VISION 8K X, DAS ICAROS-SYSTEM UND DER ZUGEHÖRIGE CONTROLLER, WELCHE IN KOMBINATION MIT DER FLUGSIMULATION VERWENDET WERDEN. EIGENE ABBILDUNG.	30
ABBILDUNG 9: DIE GEWÄHLTE ECKPUNKT- (LINKS) UND KANTENREIHENFOLGE (RECHTS). EIGENE ABBILDUNG.	32
ABBILDUNG 10: VERGLEICH ZWISCHEN DEM DURCH MARCHING CUBES (LINKS) UND SCHAEFER UND WARRENS DUAL MARCHING CUBES (RECHTS) ERZEUGTEN MESH DES GLEICHEN TERRAINAUSSCHNITTS. EIGENE ABBILDUNG.	35
ABBILDUNG 11: AUSSCHNITT EINES TERRAINMESHES WIE ES DURCH NIELSONS DUAL MARCHING CUBES ERZEUGT WIRD. EIGENE ABBILDUNG.	36
ABBILDUNG 12: DER CODE DER BEIDEN FUNKTIONEN ZUM BERECHNEN DER SAMPLEPOSITION UND DEM EIGENTLICHEN SAMPLEN AN DIESER POSITION. EIGENE ABBILDUNG.	38
ABBILDUNG 13: DER CODE DES ZWEITEN PASSES ZUM ERSTELLEN DER GRUNDLEGENDEN MESHINFORMATIONEN. EIGENE ABBILDUNG.	40
ABBILDUNG 14: DIE UNTERTEILUNG DER SPIELWELT IN NEUN TEILBEREICHE INKLUSIVE DER UNTERSCHIEDLICHEN AUFLÖSUNGEN VON CHUNKS JE NACH DER BLICKRICHTUNG DES SPIELERS. EIGENE ABBILDUNG.	43
ABBILDUNG 15: DARSTELLUNG DER VERBINDUNG BENACHBARER MESHE ÜBER SKIRTS (LINKS), FLANGES (MITTE) UND EIN ÜBERGANGSMESH (RECHTS). EIGENE ABBILDUNG.	47
ABBILDUNG 16: AUFBAU DER ID EINER OCTREEZELLE. EIGENE ABBILDUNG.	48
ABBILDUNG 17: DIE SEAMBEREICHE AUS DENEN SEAMCHUNKS GEWÄHLT WERDEN, UM DAS SEAMMESH ZU BILDEN. DER BEHANDELTE CHUNK IST GRÜN GEKENNZIEHNET. EIGENE ABBILDUNG.	49
ABBILDUNG 18: BEISPIELHAFT ZWEIDIMENSIONALE DARSTELLUNG EINES ERSTELLTEN MAPPINGARRAYS (RECHTS) FÜR DEN SEAMBEREICH IN DER POSITIVEN X-RICHTUNG (LINKS). DIE ZAHLEN REPRÄSENTIEREN DIE INDICES DER SEAMCHUNKS IN DIESEM SEAMBEREICH. EIGENE ABBILDUNG.	52
ABBILDUNG 19: CODE FÜR DAS AUSLESEN DER SEAMCHUNKDATEN EINER SEAMCHUNKZELLE AUS DEM BUFFER UND DER BERECHNUNG DER POSITION EINER CHUNK- ODER SEAMCHUNKZELLE. EIGENE ABBILDUNG.	53
ABBILDUNG 20: FLUSSDIAGRAMM FÜR DEN GROBEN ABLAUF DES ZWEITEN PASSES ZUR ERSTELLUNG DES SEAMMESHES. EIGENE ABBILDUNG.	55
ABBILDUNG 21: CODE FÜR DIE BESTIMMUNG DER INDICES VON SEAMCHUNKZELLEN ANHAND DER POSITION EINER ZELLE. EIGENE ABBILDUNG.	56
ABBILDUNG 22: BEISPIEL DER IN KOMBINATION MIT EINER CHUNKZELLE BEARBEITETEN KANTEN DER ZELLEN VON SEAMCHUNKS. DIE BEARBEITE CHUNKZELLE IST GRÜN HINTERLEGT UND DIE ZUGEHÖRIGEN SEAMZELLEN GRAU. DIE KANTEN, DIE IM ERSTEN VERBINDUNGSSCHRITT MITEINANDER VERBUNDEN WERDEN, SIND IN DER RECHTEN HÄLFTE BLAU EINGEZEICHNET UND DIE CHUNKZELLENÜBERGREIFENDEN KANTEN DES ZWEITEN SCHRITTES ROT. EIGENE ABBILDUNG.	57

ABBILDUNG 23: DARSTELLUNG DER GRUNDLEGENDEN PFADE DURCH DIE SCHLUCHTEN. VON LINKS NACH RECHTS IST DIE VERWENDUNG VON JEWELS 1, 2 UND 6 OKTAVEN AN PERLIN NOISE DARGESTELLT. IN DEN OBEREN ERGEBNISSEN WIRD DABEI EIN EXPONENT VON 1 UND IN DEN UNTEREN EIN EXPONENT VON 7 FÜR DIE POTENZIERUNG GENUTZT. EIGENE ABBILDUNG.	59
.....
ABBILDUNG 24: DER CODE DER TERRAINFUNKTION. EIGENE ABBILDUNG.....	60
ABBILDUNG 25: DIE TEXTURARRAYS DES TERRAINS DER FLUGSIMULATION. EIGENE ABBILDUNG.....	61
ABBILDUNG 26: AUSSCHNITT AUS DEM TEXTURIERTEM TERRAIN. EIGENE ABBILDUNG.....	63
ABBILDUNG 27: DAS HAUPTMENÜ UND DIE BESCHREIBUNG DER ITEMS UND TASTENBELEGUNGEN. EIGENE ABBILDUNG.....	64
ABBILDUNG 28: DER EFFEKT NACH DEM SCHIEßen DES DRILL-RAYS. EIGENE ABBILDUNG.	66
ABBILDUNG 29: DIE DREI ITEMS DER FLUGSIMULATION INKLUSIVE DER ENERGIESCHILDERR. EIGENE ABBILDUNG.....	67
ABBILDUNG 30: ZWEI ANSICHTEN DES EXPEDITIONSBIKES. EIGENE ABBILDUNG.....	68
ABBILDUNG 31: DAS IN DAS EXPEDITIONSBIKE INTEGRIERTE UI. EIGENE ABBILDUNG.....	69
ABBILDUNG 32: EIN LETZTER AUSSCHNITT AUS DER SPIELWELT DER FLUGSIMULATION. EIGENE ABBILDUNG.	74
ABBILDUNG 33: FLUSSDIAGRAMME DER GROBEN ABLÄUFE DER DREI PÄSSE FÜR DIE ERSTELLUNG DER CHUNKMESHDATEN. EIGENE ABBILDUNG.	80
ABBILDUNG 34: FLUSSDIAGRAMM DES GROBEN ABLAUFES DES ERSTEN PASSES FÜR DIE ERSTELLUNG DER SEAMMESHDATEN. EIGENE ABBILDUNG.	81
ABBILDUNG 35: FLUSSDIAGRAMME DER GROBEN ABLÄUFE DES ZWEITEN PASSES FÜR DIE ERSTELLUNG DER SEAMMESHDATEN. EIGENE ABBILDUNG.....	82

12 LITERATURVERZEICHNIS

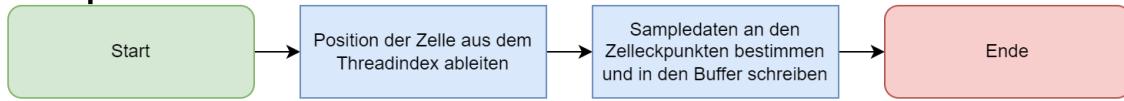
- Bajaj C., Sohn B. & Zhang Y. (2005). 3D Finite Element Meshing from Image Data. *Computer Methods in Applied Mechanics and Engineering*, 194 (48-49), pp. 5083-5106.
- Bakaoukas A. G. & Rose T. (2016). Algorithms and Approaches for Procedural Terrain Generation - A Brief Review of Current Techniques. *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, pp. 1-2.
- Berechet T., Mahlmann T., Mark B. & Togelius J. (2015). Procedural Generation of 3D Caves for Games on the GPU. *Foundations of Digital Games 2015*.
- Bloom C. (06.03.22). Terrain Texture Compositing by Blending in the Frame-Buffer. <https://www.cblomm.com/3d/techdocs/splatting.txt>
- Botsch M., Kobelt L. P., Schwancke U. & Seidel H. (2001). Feature Sensitive Surface Extraction from Volume Data. *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp 57-66.
- Bourke P. (04.03.22). Polygonising a scalar field. <http://paulbourke.net/geometry/polygonise/>
- Cline H. E. & Lorensen W. E. (1987). Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21 (4), pp. 163-169.
- Dębska M., Mynarski A., Polechoński J. & Polechoński P. (2019). Enjoyment and Intensity of Physical Activity in Immersive Virtual Reality Performed on Innovative Training Devices in Compliance with Recommendations for Health. *International Journal of Environmental Research and Public Health* 2019, 16 (19).
- Denis F., Dupont. F. & Lobello R. U. (2012). Multi-Resolution dual contouring from volumetric data. *Proceedings of the International Conference on Computer Graphics Theory and Applications and International Conference on Information Visualization Theory and Applications – GRAPP*, pp. 163-168.
- Ebert D. S., Musgrave F. K., Peachey D., Perlin K. & Worley S. (2002). *Texturing & Modeling. A Procedural Approach*. Morgan Kaufmann.
- Feodoroff B., Konstantinidis I. & Froböse I. (2019). Effects of Full Body Exergaming in Virtual Reality on Cardiovascular and Muscular Parameters: Cross-Sectional Experiment. *JMIR Serious Games*, 7 (3).
- Geiss R. (05.03.22). NVIDIA – GPU Gems 3 – Chapter 1. Generating Complex Procedural Terrains Using the GPU. <https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu>
- Gibson S. F. F. (1998). Using Distance Maps for Accurate Surface Representation in Sampled Volumes. *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pp 23-30.
- Glanville R. S. (06.03.22). NVIDIA – GPU Gems – Chapter 20. Texture Bombing. https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch20.html
- Golum B. (06.03.22). Normal Mapping for a Triplanar Shader. <https://bgolum.medium.com/normal-mapping-for-a-triplanar-shader-10bf39dca05a#d715>

- Green S. (04.03.22). NVIDIA – GPU Gems 2 – Chapter 26. Implementing Improved Perlin Noise. <https://developer.nvidia.com/gpugems/gpugems2/part-iii-high-quality-rendering/chapter-26-implementing-improved-perlin-noise>
- Hamann B. & Nielson G. M. (1991). The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes. IEEE Conference on Visualization 1991, pp. 83-91.
- Hill, S. (09.03.22). Self Shadow - Blending in Detail. <https://blog.selfshadow.com/publications/blending-in-detail/>
- Holmlid E. (2010). Manifold Contouring of an Adaptively Sampled Distance Field. Chalmers University of Technology, Gothenburg.
- Ju T., Losasso F., Schaefer S. & Warren J. (2002). Dual Contouring of Hermite Data. ACM Transactions on Graphics, 21 (3), pp. 339-346.
- Ju T., Schaefer S. & Warren J. (2007). Manifold Dual Contouring. IEEE Transactions on Visualization and Computer Graphics, 13, pp. 610-619.
- Ju T. & Udeshi T. (2006). Intersection-free Contouring on An Octree Grid. Proc. 14th Pacific Conf. Computer Graphics and Applications (PG '06).
- Lehmann-Böhm P. (06.03.22) (a). Volume GFX – Octree Generation. <https://www.volume-gfx.com/volume-rendering/dual-marching-cubes/octree-generation/>
- Lehmann-Böhm P. (06.03.22) (b). Volume GFX – Deriving the Dualgrid. <https://www.volume-gfx.com/volume-rendering/dual-marching-cubes/deriving-the-dualgrid/>
- Lengyel E. S. (2010). Voxel-Based Terrain for Real-Time Virtual Simulations. University of California at Davis, Division of Computer Science 4455 Chem. Annex Davis, CA, United States.
- Löffler F. & Schumann H. (2012). Generating Smooth High-Quality Isosurfaces for Interactive Modeling and Visualization of Complex Terrains. Vision, Modeling & Visualization, pp. 79-86.
- Mishkinis A. (06.03.22). Advanced Terrain Texture Splatting. <https://www.gamedeveloper.com/programming/advanced-terrain-texture-splatting>
- Nielson G. M. (2004). Dual Marching Cubes. VIS '04: Proceedings of the conference on Visualization '04, pp. 489-496.
- Olsen J. (2004). Realtime Procedural Terrain Generation.
- Palko M. (06.03.22). Triplanar Mapping. <http://www.martinpalko.com/triplanar-mapping/>
- Perlin K. (1985). An Image Synthesizer. ACM SIGGRAPH Computer Graphics, 19 (3), pp. 287-296.
- Perlin K. (2002). Improving Noise. ACM Transactions on Graphics, 21 (3), pp. 681-682.
- Schaefer S. & Warren J. (2004). Dual Marching Cubes: Primal Contouring of Dual Grids. 12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.

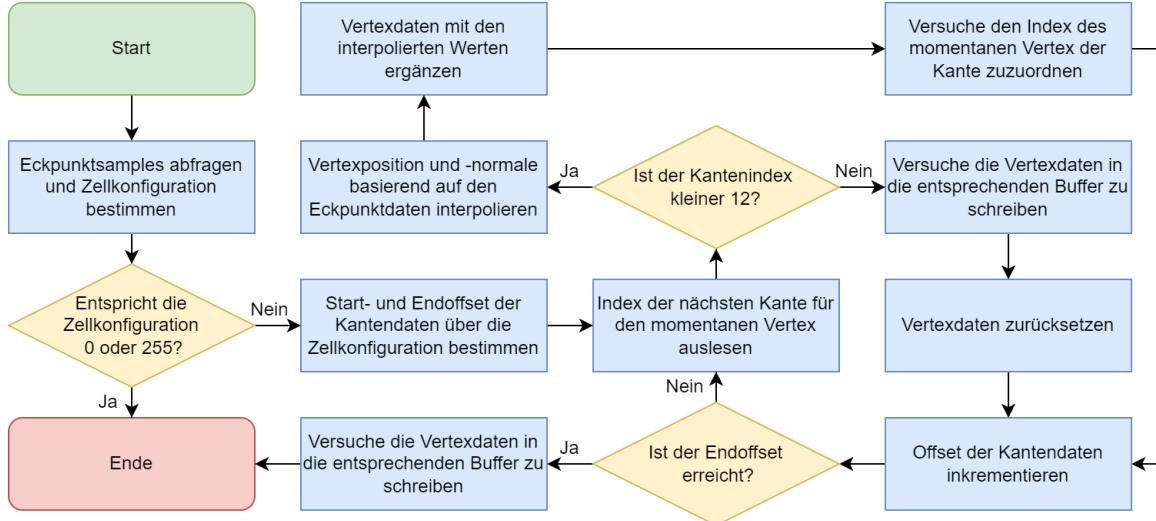
- Thorne C. (2005). Using a Floating Origin to Improve Fidelity and Performance of Large, Distributed Virtual Worlds. 2005 International Conference on Cyberworlds (CW'05), pp. 263-270.
- Wenger R. (2013). Isosurfaces. Geometry, Topology and Algorithms. CRC Press.
- Quilez I. (06.03.22). Texture repetition.
<https://www.iquilezles.org/www/articles/texturerepetition/texturerepetition.htm>

13 ANHANG

Sample:



Create:



Connect:

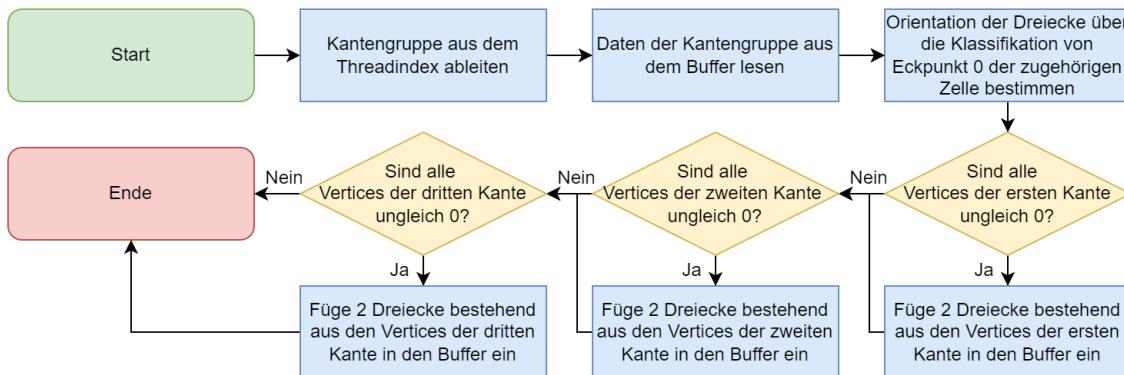


ABBILDUNG 33: FLUSSDIAGRAMME DER GROBEN ABLÄUFE DER DREI PÄSSE FÜR DIE ERSTELLUNG DER CHUNKMESHDATEN.

Seammesh Pass 1:

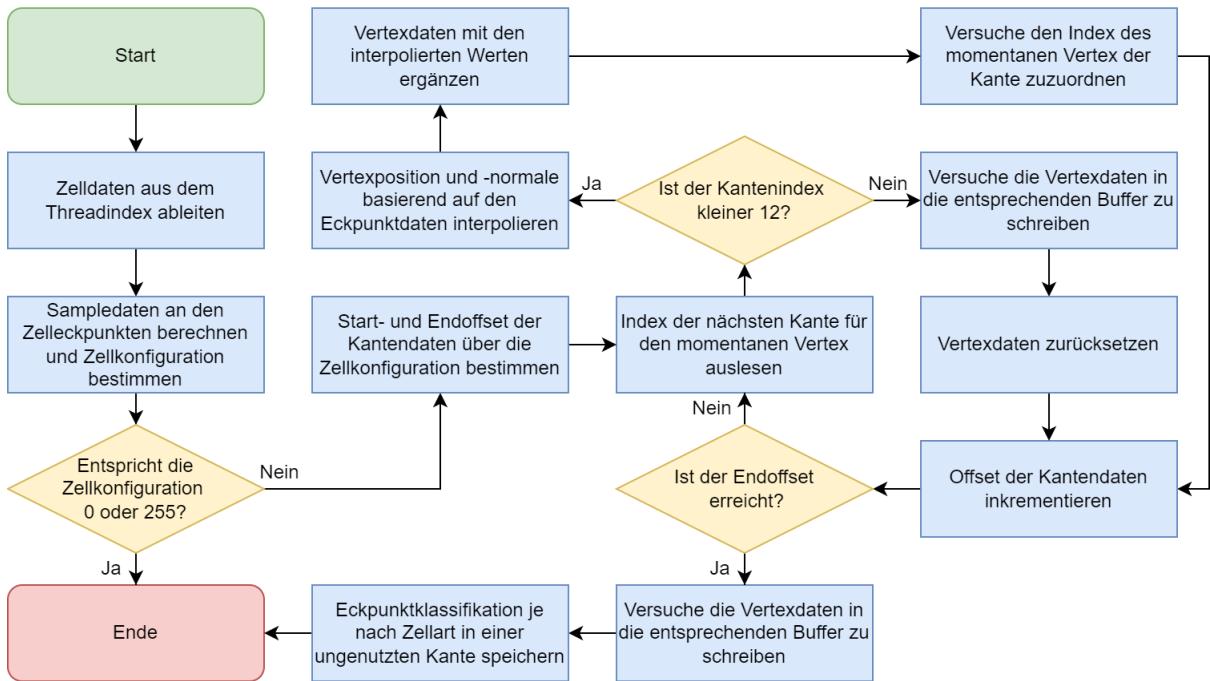
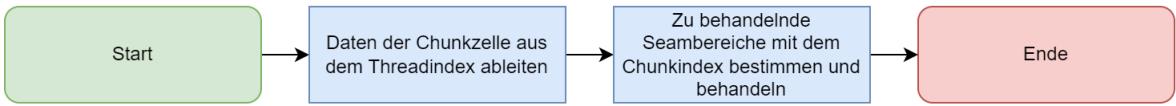
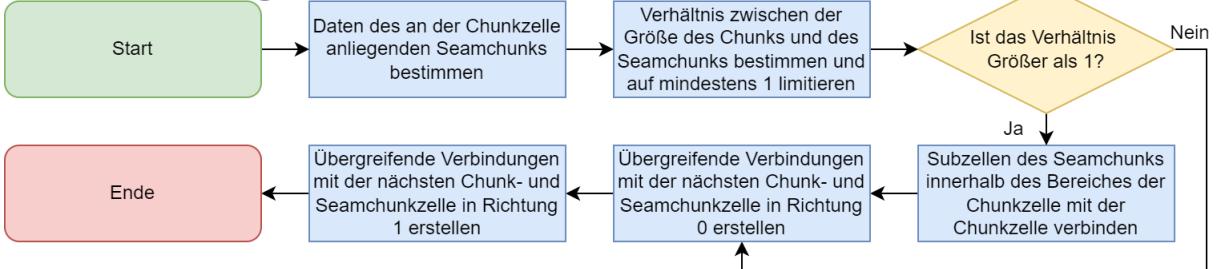


ABBILDUNG 34: FLUSSDIAGRAMM DES GROBEN ABLAUFES DES ERSTEN PASSES FÜR DIE ERSTELLUNG DER SEAMMESHDATEN.

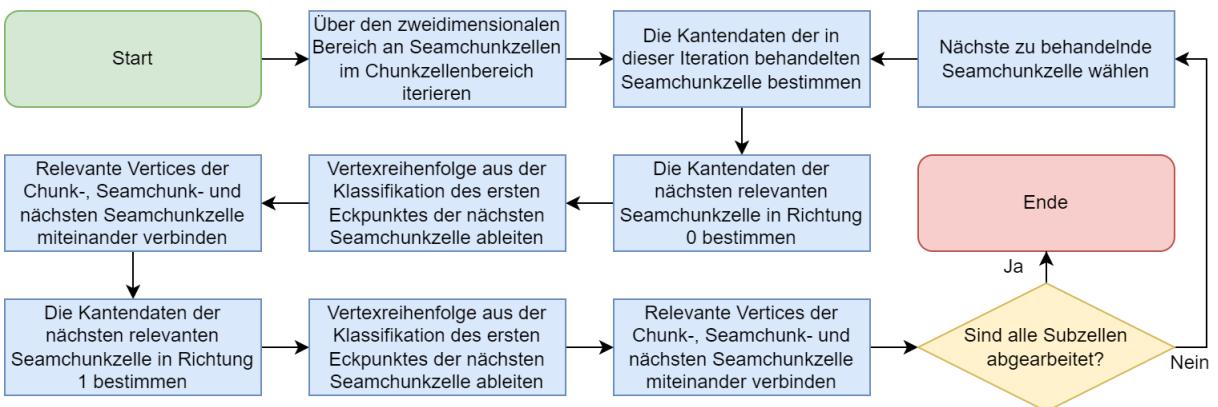
Seammesh Pass 2:



Behandlung eines Seambereiches:



Verbinden von Subzellen:



Erstellen von übergreifenden Verbindungen:

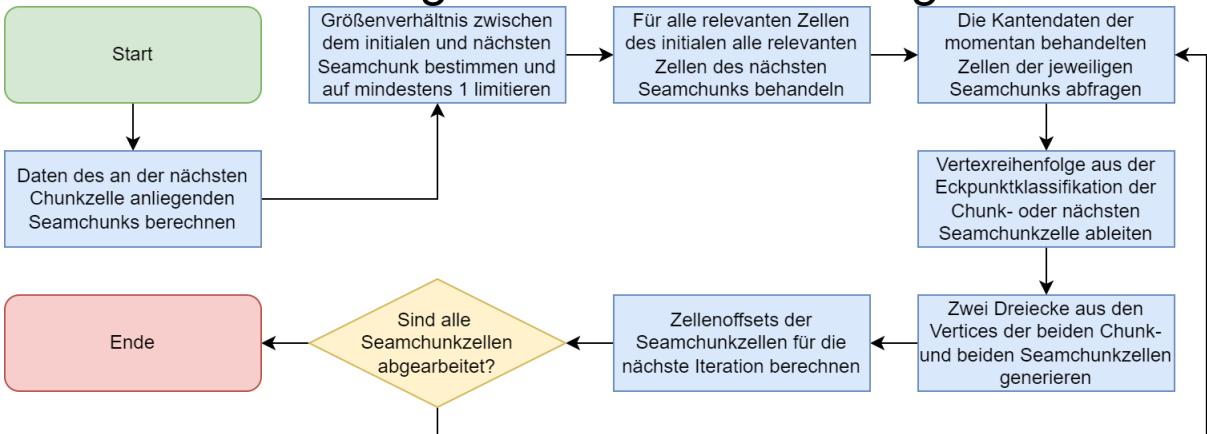


ABBILDUNG 35: FLUSSDIAGRAMME DER GROBEN ABLÄUFE DES ZWEITEN PASSES FÜR DIE ERSTELLUNG DER SEAMMESHDATEN.

14 ERKLÄRUNGEN

14.1 SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit erkläre ich, dass ich die Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe.

Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Ort, Datum Unterschrift

Unterschrift

14.2 ERMÄCHTIGUNG

- Hiermit ermächtige ich/wir die Hochschule Kempten zur Veröffentlichung einer Kurzzusammenfassung sowie Bilder/Screenshots und ggf. angefertigte Videos meiner studentischen Arbeit z. B. auf gedruckten Medien oder auf einer Internetseite der Hochschule Kempten zwecks Bewerbung des Bachelorstudiengangs „Game Engineering“ und des Masterstudiengangs „Game Engineering und Visual Computing“.

Dies betrifft insbesondere den Webauftritt der Hochschule Kempten inklusive der Webseite des Zentrums für Computerspiele und Simulation. Die Hochschule Kempten erhält das einfache, unentgeltliche Nutzungsrecht im Sinne der §§ 31 Abs. 2, 32 Abs. 3 Satz 3 Urheberrechtsgesetz (UrhG).

Ort, Datum

Unterschrift