Hochschule Neu-Ulm
University of Applied Sciences

Bachelorarbeit

im Bachelorstudiengang Game-Produktion und Management

an der Hochschule für angewandte Wissenschaften Neu-Ulm

Thema

Spielwelten auf Knopfdruck: Wie viel Zeitersparnis ermöglichen PCG-Graphs bei der Levelerstellung im Vergleich zu manuellen Methoden und wie schätzen Nutzende deren Usability ein?

Erstprüfer/in: Prof. Dr. Weilemann

Betreuer: Prof. Dr. Antje Wild

Verfasser: Marc Steiner (303561)

Thema erhalten: 04.08.2025

Arbeit abgeliefert: 04.12.2025

1. Inhaltsverzeichnis

1. Inhaltsverzeichnis I
1.1. Abkürzungs- und Symbolverzeichnis IV
1.2. Liste der Fachbegriffe V
1.3. Abbildungsverzeichnis VIII
1.4. Tabellenverzeichnis XI
2. Einleitung 1
3. Methodologie und methodisches Vorgehen 3
3.1. Forschungsfrage und Hypothese 3
3.2. Forschungsdesign und Datenerhebung 3
3.3. Bewertungskriterien 3
3.4. Übersicht verwendeter Hilfsmittel 4
4. State of the Art 5
4.1. Traditionelle Levelgestaltung 5
4.1.1. Herangehensweise 5
4.1.2. Herausforderungen 7
4.2. Procedural Content Generation 8
4.2.1. Herangehensweise 8
4.2.2. Herausforderungen 9
4.2.3. Verwendungen in der Industrie 11
4.3. PCG-Graphen in der Unreal Engine 14
4.4. Umfrage unter Indie Studios 15
4.5. Forschungslücke und Begründung der Fragestellung 17
5. Prototypisierung 18
5.1. Methodische Einordnung des experimentellen Vorgehens 18
5.2. Unabhängige und abhängige Variablen 18
5.3. Versuchsplan und Designentscheidung 18
5.4. Operationalisierung und Messprozeduren 18
5.5. Kontrolle von Störvariablen und Validität 19
5.6. Gütekriterien (Objektivität, Reliabilität, Validität) 19
5.7. Einschränkungen und Hinweise zur Interpretation 19
5.8. Planung 20
5.9. Pilotversuch 22

6. Implementierung und Leitfaden 23
6.1. Manuelle Levelgestaltung: Leitfaden 25
6.1.1. Foliage Mode 25
6.1.2. Create Static Foliage Meshes 26
6.1.1. Foliage Brushes 28
6.1.2. Static Foliage Meshes 29
6.1.3. Vegetation platzieren 32
6.2. PCG-Graphen-Levelgestaltung: Leitfaden 33
6.2.1. Landscape Data und Surface Sampler Node 33
6.2.2. Spatial Noise Node 35
6.2.3. Density Filter Node 35
6.2.4. Transform Points Node 36
6.2.5. Static Mesh Spawner Node 37
6.2.6. Random Choice Node 37
6.2.7. Graph Parameters 38
6.2.8. Normal to Density Node 39
6.2.9. Projection Node 41
6.2.10. PCG Subgraphen 42
6.2.11. Create Points Grid und Copy Points Node 44
6.2.12. Distance Node 46
6.2.13. Blumenfelder 47
6.2.14. Büsche und Pflanzen 50
7. Anpassungsfähigkeit 51
8. Veränderungen: Leitfaden 52
8.1. PCG – Graphen 52
8.1.1. Pilze 52
8.1.2. Bäume 53
8.1.3. Büsche 54
8.1.4. Dorf 54
8.1.5. Neues Level 60
8.2. Foliage Tool 61
8.2.1. Pilze 61
8.2.2. Bäume 61

8.2.3. Büsche ... 61
8.2.4. Dorf ... 61
8.2.5. Neues Level ... 62
9. Durchführung und experimentelle Evaluation ... 63
9.1. Ablauf ... 63
9.2. Einordnung der Teilnehmer ... 64
9.3. Rohdaten ... 65
9.4. Auswertung der Tabelle ... 65
9.5. Auswertung des Fragebogens ... 66
9.6. Limitationen ... 68
9.6.1. Externe Validität der Online-Umfrage ... 68
9.6.2. Begrenzte Stichprobengröße ... 68
9.6.3. Projekt Setup ... 68
9.6.4. Begrenzte Datentiefe und Messgenauigkeit ... 68
9.6.5. Lern- und Reihenfolgeeffekte ... 68
9.6.6. Remote-Durchführung über Discord ... 68
10. Fazit ... 69
10.1. Zusammenfassung der zentralen Erkenntnisse ... 69
10.2. Praxisempfehlungen für Entwicklerteams ... 70
11. Anhang ... XII
11.1. A Survey on Procedural Content Generation in Indie Game Development ... XII
11.2. Participant Questionnaire - A Survey on Procedural Content Generation in Indie Game Development ... XVII
12. Quellenverzeichnis ... XX

1.1. Abkürzungs- und Symbolverzeichnis

PCG	Prozedurale Content Generierung
2D	zweidimensional
3D	dreidimensional
Ggf.	gegebenenfalls
Indie	Independent
NPC	Non Playable Character
Vgl.	vergleiche

1.2. Liste der Fachbegriffe

Asset	Einzelnes Element wie Modell, Textur oder Sound, das in einem Spiel verwendet wird
Blueprint / Blueprint Class / Blueprint Actor	Skriptbasierte Klassen in UE zur Definition von Verhalten und vorkonfigurierten Objekten
Bounds Modifier Node	Node, die die Grenzen oder Skalierung von Punktbereichen verändert, um Abdeckung sicherzustellen
Brush Size / Paint Density	Parameter zur Steuerung der Pinselgröße und Dichte der platzierten Objekte
Copy Points Node	Node zum Duplizieren von Punkten
Create Points Grid Node	Node zum Erzeugen von Punktgittern
Debugging	Prozess des Auffindens und Behebens von Fehlern (Bugs) in Software
Density Filter Node / Density Node	Node zum Filtern oder Steuern der Dichte von generierten Punkten
Distance Node	Node zum Filtern von Punkten basierend auf Distanzkriterien
Foliage	Vegetationstools in der Unreal Engine zur Platzierung von Pflanzen, Bäumen und Bodenbewuchs
Get Landscape Data Node	Node zum Auslesen von Landscape Informationen wie Höhe
Heightmap	Graustufenbild, das Höheninformationen einer Landschaft enthält. Helle Bereiche stehen für hohe, dunkle für niedrige Geländepunkte
Indie	Bezeichnung für unabhängige Spieleentwickler oder -studios
Inspect Tool/ Outliner / Content Browser	Hilfsfenster und Panels in der Unreal Engine zur Inspektion und Organisation von Assets und Szenen
Kompilieren / Compile	Übersetzen von Blueprints in ausführbare Logik innerhalb der Engine
Level	Spielumgebung oder Spielabschnitt, den die Spielerinnen und Spieler durchlaufen
Node	Ein Baustein im *PCG-Graphen*, der eine Funktion oder Operation ausführt
Non Playable Character	Nicht spielbare Figur im Spiel, gesteuert durch die Software
Normal to Density Node	Node zur Ableitung der Platzierungsdichte aus Flächennormalen

Paint Brush	Werkzeug im Foliage Mode, mit dem Assets direkt in die Umgebung „gemalt“ werden
PCG Subgraph	Wiederverwendbarer Teilgraph, der komplexe Abläufe kapselt
PCG-Debugger	Werkzeuge zur Fehleranalyse
PCG-Graph / Graph	Visuelle Repräsentation prozeduraler Abläufe in Form von Knoten und Verbindungen
Projection Node	Node, welche Punkte auf eine definierte Oberfläche projiziert, zum Beispiel auf das Terrain
Prozedurale Content Generierung	Algorithmische Erzeugung von Spielinhalten wie Terrain, Vegetation oder Levelstrukturen
Random Choice Node	Node, die zufällig zwischen Optionen wählt
Seed	Startwert für Zufallsfunktionen. Gleicher Seed ergibt reproduzierbare Ergebnisse
Selection Mode	Modus in der Unreal Engine, in dem Objekte im Editor ausgewählt und bearbeitet werden können
Single Brush	Pinselmodus, der jeweils genau ein Asset pro Klick platziert
Spatial Noise Node	Node, die räumliches Rauschen erzeugt und damit Zufallsmuster für Platzierungen liefert
Spline / Spline Sampler / Get Spline Data Node	Spline ist eine Kurve im Level. Die zugehörigen Nodes lesen Splinedaten aus und samplen Punkte innerhalb der Kurve
Static Foliage Mesh	Gespeicherte Foliage-Konfiguration eines Static Meshes zur wiederholten Verwendung
Static Mesh	Fertiges 3D-Modell ohne eingebaute Bewegungslogik, häufig für Umweltobjekte verwendet
Static Mesh Spawner Node	Node, die an Punkten Meshes spawnt beziehungsweise platziert
Surface Sampler Node	Node, die Punkte basierend auf der Oberfläche sammelt und als Basis für Spawns nutzt

Tile / Tileset	Sammlung von wiederverwendbaren Kacheln oder Texturen, mit denen Umgebungen modular aufgebaut werden
Transform Points Node / Transform Node	Node zur Verschiebung, Rotation oder Skalierung der generierten Punkte im Raum
Unreal Engine	Entwicklungsumgebung zur Erstellung von Spielen und interaktiven 3D-Anwendungen

1.3. Abbildungsverzeichnis

Abb. 1 Graustufen Textur in Heightmap umgewandelt. Quelle: Ahearn, 2008. 5
Abb. 2 Blockout eines Uncharted Levels. Quelle: Klepek, P., VICE, 5. Okt. 2017. 6
Abb. 3 Perlin Noise 8
Abb. 4 Simplex Noise 8
Abb. 5 Worley Noise 8
Abb. 6 Screenshot einer Minecraft Welt. Quelle: Minecraft Backrooms Wiki (Fandom). Zugriff: 12.09.2025. 9
Abb. 7 Screenshot von zufällig generierter Welt aus No Man's Sky. Quelle: No Man's Sky Wiki. Zugriff: 12.09.2025. 10
Abb. 8 Screenshot von zufällig generiertem Level aus Diablo IV. Quelle: MeinMMO. Zugriff: 12.09.2025. 12
Abb. 9 Screenshot aus Skyrim Dialog einer zufälligen Assassinen Quest. Quelle: Namu Wiki. Zugriff: 12.09.2025. 13
Abb. 10 Darstellung von PCG-Punkten Quelle: Epic Games (2025), Unreal Engine 5.6 Documentation. 14
Abb. 11 Screenshot der Anforderungen aus den Notizen. 20
Abb. 12 Screenshot der Veränderungsanforderungen aus den Notizen. 21
Abb. 13 Moodboard der Assets 23
Abb. 14 Alle in Blender modellierten Assets 24
Abb. 15 Beispiellevel mit allen angepassten Parametern in der Unreal Engine 24
Abb. 16 Auswahl des Foliage Tools 25
Abb. 17 Erstellen von Static Foliage Meshes 26
Abb. 18 Aktiven und Speichern der Static Foliage Meshes 27
Abb. 19 Speicherort der Static Foliage Meshes 27
Abb. 20 Paint Brush 28
Abb. 21 Single Brush 28
Abb. 22 Density/1Kuu Einstellungen 29
Abb. 23 Scale Einstellungen 30
Abb. 24 Align to Normal Einstellung 30
Abb. 25 Ground Slope Angle Einstellungen 31
Abb. 26 Anzahl der platzierten Instanzen 32
Abb. 27 Alle plazierten Instanzen 32
Abb. 28 Darstellung des fertigen Levels 32
Abb. 29 Kreierung eines leeren Graphens 33
Abb. 30 Content Browser PCG Graph 33
Abb. 31 Get Landscape Data Node 33
Abb. 32 Surface Sampler Node 34
Abb. 33 PCG Debugger 34
Abb. 34 PCG Unbound Einstellung 34
Abb. 35 Debugger mit aktiver Unbound Einstellung 34
Abb. 36 Spatial Noise Node 35
Abb. 37 Spatial Noise in Level 35

Abb. 38 Density Node Einstellungen 35
Abb. 39 Filtered Points in Level 35
Abb. 40 Transform Node in Level 36
Abb. 41 Transform Points Node Einstellungen 36
Abb. 42 Static Mesh Spawner Node Einstellungen 37
Abb. 43 Random Choice Node mit Einstellungen 37
Abb. 44 Graph Parameter Einstellungen 38
Abb. 45 Random Choice Node mit Parameter 39
Abb. 46 Normal to Density Node in Level 39
Abb. 47 Normal to Density Node Einstellungen 40
Abb. 48 Normal to Density Node in Level 40
Abb. 49 Get Landscape Data Node mit Height only in Level 41
Abb. 50 Get Landscape Data Node mit Height Only Einstellung 41
Abb. 51 PCG Subgraphen Erstellung 42
Abb. 52 PCG Subgraph in einer Node 43
Abb. 53 Create Points Grid in Level 44
Abb. 54 Copy Points Node 44
Abb. 55 Grids an jedem Baum im Level 44
Abb. 56 Projection Node in PCG_Subgraph_Remove_Cliffs 45
Abb. 57 Finaler Aufbau der Äste 45
Abb. 58 Distance Node Einstellungen 46
Abb. 59 Distance Node in Level 46
Abb. 60 Punkte in Level nachdem Distance Filter Node eingebaut wurde 47
Abb. 61 Cell Size in der Create Points Grid Node 48
Abb. 62 Inspect Tool 48
Abb. 63 Finaler Aufbau der Blumenfelder 49
Abb. 64 Blumenfelder im Level 49
Abb. 65 Spawnen der Büsche im PCG Graph 50
Abb. 66 Büsche im Level 50
Abb. 67 Screenshot der Veränderungen für jedes Level aus den Notizen 51
Abb. 68 Spawnen der Pilzfelder im PCG Graph 52
Abb. 69 Zypressenbaum in Static Mesh Spawner 53
Abb. 70 Vergleich der Tree Amount Parameter in Leveln 53
Abb. 71 Busch Varianten in PCG Graph 54
Abb. 72 Busch Varianten in Level 54
Abb. 73 Town Level Asset im Content Browser 54
Abb. 74 Dorf Location auf 0, 0, 0 Position 54
Abb. 75 Blueprint Class 55
Abb. 76 Parent Blueprint Class 55
Abb. 77 Name der Spline Blueprint Class 55
Abb. 78 Spline in Components Tab 55
Abb. 79 Closed Loop Einstellung in Spline 55
Abb. 80 Spline in Kreis Form 56
Abb. 81 Blueprint Actor in Components Tab ausgewählt 56

Abb. 82 Tag hinzufügen 56
Abb. 83 Kompilierknopf 56
Abb. 84 Erfolgreiches Kompilieren 56
Abb. 85 Spline im Level 57
Abb. 86 Dorf umrandet von Spline 57
Abb. 87 Get Spline Data Einstellungen 57
Abb. 88 Spline Sampler Node Einstellungen 58
Abb. 89 Bounds Modifier Einstellungen 58
Abb. 90 Spline Data Nodees verbunden mit dem Hauptgraphen 59
Abb. 91 Jegliche Vegetation innerhalb der Spline entfernt 59
Abb. 92 Seed Parameter 60
Abb. 93 Seed Parameter in Spatial Noise 60
Abb. 94 PCG Forest im Outliner 60
Abb. 95 PCG Parameter Override 60
Abb. 96 Static Foliage Mesh im Content Browser 62
Abb. 97 Foliage Tool Drag and Drop 62
Abb. 98 Alle Foliage Meshes 62

1.4. Tabellenverzeichnis

Tabelle 1 „Do you use any Procedural Content Generation (PCG) techniques in your game development?” 15
Tabelle 2 "Which game engine do you primarly use for your projects?" 15
Tabelle 3 "What are your main reasons for using PCG?" 15
Tabelle 4 "What are your main reasons for not using PCG?" 16
Tabelle 5 "Have you used Unreal Engine before?" 64
Tabelle 6 "How frequently do you work with the engine?" 64
Tabelle 7 “How familiar are you with the following topics (PCG Graph)?” 64
Tabelle 8 “How familiar are you with the following topics (Foliage Tool)?” 64
Tabelle 9 Time spent on Level Creation in Minutes 65
Tabelle 10 Linien Graph aller Zeiten in Minuten 65
Tabelle 11 Balkendiagramm der Teilnehmer Antworten nach dem Experiment 66
Tabelle 12 "Which workflow felt better to use overall?" 67

2. Einleitung

Die Entwicklung moderner Computerspiele führt zu immer komplexeren virtuellen Welten, die als entscheidendes Verkaufsargument in der Branche gelten(Cox, 2014). Herkömmliche Verfahren zur Levelerstellung, die auf manueller Platzierung beruhen, sind zeitaufwendig und stellen vor allem kleine und unabhängige Entwicklerteams vor große Herausforderungen(Consalvo and Paul, 2018). In diesem Zusammenhang bietet die prozedurale Content-Generierung (*PCG*) eine vielversprechende Alternative, die nicht nur den Entwicklungsaufwand reduziert(Rodrigues, Bonidia and Brancher, 2020), sondern auch neue kreative Möglichkeiten eröffnet(Korn *et al.*, 2017).

Die während des Praktikums bei Active Fungus gewonnenen praktischen Erfahrungen lassen darauf schließen, dass der Einsatz von *PCG-Graphen* in der Levelgestaltung nicht nur zu einer Verkürzung der Produktionszeiten führt, sondern auch die Flexibilität im Entwicklungsprozess verbessert. Vor diesem Hintergrund befasst sich diese Arbeit mit der Frage, inwieweit *PCG-Graphen* in der *Unreal Engine* den Zeitaufwand bei der Erstellung von Spielwelten im Vergleich zu traditionellen Methoden verringern können.

Ziel der Untersuchung ist es, den potenziellen Effizienzgewinn durch den Einsatz von *PCG-Graphen* quantitativ zu erfassen und damit einen Beitrag zur Optimierung der Produktionsprozesse in der Spieleentwicklung zu leisten. Zwar deuten einige Studien darauf hin, dass prozedurale Verfahren den Levelerstellungsprozess beschleunigen, jedoch fehlen bislang belastbare quantitative Beweise. Ma et al. (2014) und Roden und Parberry (2004) sprechen im allgemein von Generierungszeiten im Bereich von „Sekunden“ oder „Echtzeit“, während González Duque et al. (2020) lediglich auf eine geringe Zahl benötigter Iterationen verweist. Auch Cardamone et al. (2011) berichten von einer erheblichen Beschleunigung bei der Simulation, beispielsweise einer zehnminütigen Spielsequenz, die in zehn Sekunden abgebildet werden konnte. Zwar legen diese Arbeiten nahe, dass algorithmische Optimierungen und automatisierte Prozesse zu schnelleren Entwicklungszyklen führen, ein direkter zeitlicher Vergleich mit manuellen Methoden wird jedoch nicht vorgenommen. Die tatsächliche Effizienzsteigerung durch den Einsatz von *PCG*-Techniken bleibt somit weitgehend unbelegt(Roden and Parberry, 2004; Cardamone *et al.*, 2011; Ma *et al.*, 2014; González-Duque *et al.*, 2020). An genau dieser Stelle setzt die vorliegende Arbeit an und zielt darauf ab, den Zeitaufwand beider Ansätze systematisch gegenüberzustellen.

Hierfür wird ein Mixed-Method-Ansatz verfolgt. Zunächst erfolgt eine Analyse des bestehenden Forschungsstands und eine quantitative Befragung von Entwicklern durch Fragebögen, um ein fundiertes Bild der aktuellen Verwendung von *PCG-Graphen* in der Spielentwicklung zu erhalten. Anschließend werden zwei Prototypen in der *Unreal Engine* entwickelt, einer basierend auf manuellen Methoden, der andere unter Einsatz von *PCG-Graphen*. Diese Prototypen dienen als Grundlage für einen Leitfaden, mit dem anschließend Entwickler eigene Levels erstellen. Dabei werden die Entwicklungszeiten der Entwickler beider Ansätze systematisch verglichen, wobei auch Anpassungen während des Entwicklungsprozesses berücksichtigt werden. Ziel ist es herauszufinden welche Methode

unter den Festgelegten Parametern einen Effizienzgewinn in Bezug auf die Entwicklungszeit bietet.

3. Methodologie und methodisches Vorgehen

3.1. Forschungsfrage und Hypothese

Forschungsfrage:

Wie viel Zeitersparnis ermöglichen PCG-Graphs bei der Levelerstellung im Vergleich zu manuellen Methoden und wie schätzen Nutzende deren Usability ein?

Hypothese:

Der Einsatz von *PCG-Graphen* in der *Unreal Engine* reduziert den Entwicklungszeitaufwand bei der Levelerstellung signifikant im Vergleich zur manuellen Gestaltung, insbesondere bei mehrfachen Anpassungen während der Entwicklung. Zusätzlich wird die Usability von den Nutzenden als überwiegend positiv wahrgenommen.

3.2. Forschungsdesign und Datenerhebung

Die Untersuchung basiert auf einem experimentellen Mixed-Methods-Ansatz. Ziel ist es, den Effizienzgewinn und die wahrgenommene Usability bei der Levelerstellung mit *PCG-Graphen* im Vergleich zur manuellen Methode zu analysieren. Das Experiment umfasst die Erstellung zweier Prototypen (manuell und *PCG*-basiert) sowie Testdurchläufe mit Entwicklerinnen und Entwicklern unterschiedlicher Erfahrungsstufen. Während der Aufgabenbearbeitung werden Bearbeitungszeiten erfasst und zusätzlich liefern standardisierte Fragebögen quantitative und qualitative Daten. Eine detaillierte Beschreibung des Versuchsaufbaus und der Erhebungsinstrumente folgt in Kapitel 6 und die Fragebögen im Anhang.

3.3. Bewertungskriterien

Die Bewertung erfolgt anhand folgender Kriterien:

- Zeitaufwand bei Erstaufbau: Gemessene Dauer zur Erstellung des ersten funktionsfähigen Prototyps
- Zeitaufwand bei Änderungen: Zeit zur Durchführung kleiner und mittlerer Änderungen am bestehenden Level
- Flexibilität: Fähigkeit zur schnellen Anpassung des Levels auf sich ändernde Anforderungen
- Skalierbarkeit: Eignung des Systems zur Erweiterung auf größere Welten oder zusätzliche Inhalte
- Usability: Subjektive Bewertung der Benutzerfreundlichkeit durch die Entwickler*innen
- Kreative Kontrolle: Subjektive Einschätzung des kreativen Handlungsspielraums

3.4. Übersicht verwendeter Hilfsmittel

Software (Entwicklung & Prototypen)

- *Unreal Engine* 5.6 - Hauptentwicklungsumgebung zur Erstellung der Prototypen und Implementierung der *PCG-Graphen*
- *PCG* Framework Plugin (*Unreal Engine*) - Verwendung der *PCG-Node*s
- Blender - Modellierung und Export der statischen *Mesh-Asset*s
- PureRef - Erstellung und Bearbeitung des Moodboards

Mess- und Erhebungsinstrumente

- LimeSurvey - Administration und Ausspielung der Fragebögen an Indie-Studios (siehe Anhang) und zur Erhebung subjektiver Usability Bewertungen (siehe Anhang)
- Zeitmessung / Protokollierung: Excel / Stoppuhren
- Protokollblätter / Beobachtungsnotizen - Dokumentation von Besonderheiten während der Sessions

Zusätzliche Tools für Recherche, Literaturmanagement und Visualisierung

- Zotero - Literaturverwaltung und Erstellung des Quellenverzeichnisses
- Lucidchart - Erstellung von Zeitplänen und Diagrammen zur Visualisierung von Prozessabläufen und Graph-Strukturen
- Elicit - Unterstützung bei der Recherche und beim Auffinden relevanter wissenschaftlicher Quellen
- ChatGPT - Unterstützung bei Formulierungen und als erste Hilfe bei der Übersetzung fremdsprachiger Quellen. Alle erzeugten Texte wurden eigenständig überarbeitet, sprachlich und fachlich angepasst und auf Richtigkeit geprüft

4. State of the Art

4.1. Traditionelle Levelgestaltung

4.1.1. Herangehensweise

Vielseitige und visuell ansprechende Level sind wichtige Elemente eines Videospiels, da sie maßgeblich zur Spielerfahrung und zur Immersion beitragen. Um ein bestimmtes Spielerlebnis gezielt zu gestalten, greifen Level-Designer häufig auf vorgefertigte Bausteine zurück, aus denen sie die Spielwelt manuell aufbauen. Dieses Vorgehen erlaubt es ihnen, den Spielverlauf präzise zu steuern sowie die visuelle Ästhetik des Spiels genau anzupassen(Ma *et al.*, 2014; Khalifa *et al.*, 2019). Ein weiterer wesentlicher Vorteil des manuellen Leveldesigns besteht darin, dass Spielabschnitte bewusst auf bestimmte Herausforderungen oder Spielmechaniken zugeschnitten werden können(Khalifa *et al.*, 2019).

Grundsätzlich basiert dieses System auf Graustufen-Texturen, den sogenannten *Heightmaps*. Diese können entweder in einer beliebigen 2D-Software, wie Photoshop, erstellt werden, oder wie in den meisten Game Engines, direkt in der Engine gemalt werden. Landschaften bestehen, wie alle anderen 3D-Objekte, aus einem *Mesh*. Die *Heightmap* wird verwendet, um dieses Mesh je nach Graustufe zu verformen(Ahearn, 2008). In der Abbildung 1 ist zu erkennen, wie die Landschaft durch die Heightmap angelegt wird.

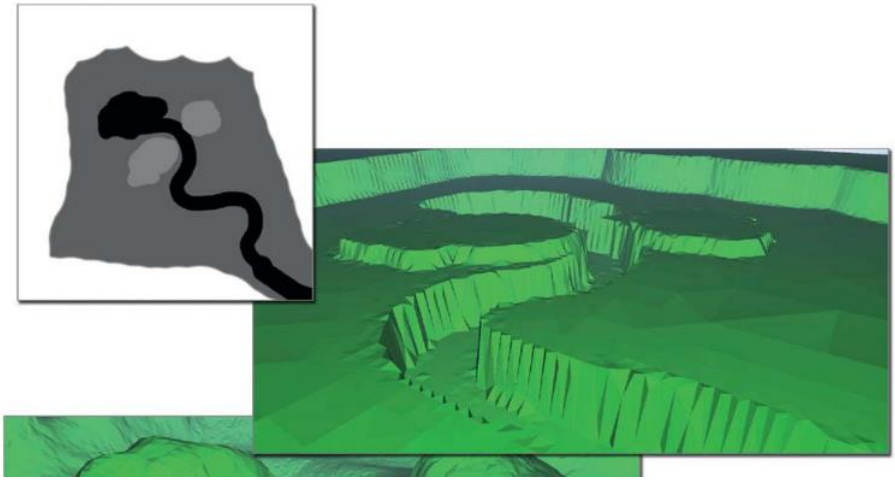


Abb. 1 Graustufen Textur in Heightmap umgewandelt. Quelle: Ahearn, 2008.

Viele Spiele-Engines, darunter auch die *Unreal Engine*, bieten zusätzlich eigene Terrain-Editoren an. Ein zentraler Bestandteil davon sind die sogenannten Brushes. Diese funktionieren wie herkömmliche Pinsel und ermöglichen es dem Leveldesigner, direkt auf die Landschaft zu malen. Oberflächen können dadurch angehoben oder abgesenkt werden, und auch Übergänge, etwa an Klippen, lassen sich glätten. Ein weiterer Aspekt dieser Pinsel sind die sogenannten Alpha Brushes. Diese erlauben es, Graustufenkarten zu

importieren und als Pinsel zu verwenden, so kann beispielsweise ein Berg oder See direkt in die Landschaft „gemalt“ werden(Ahearn, 2008; *Landscape Brushes in Unreal Engine | Unreal Engine 5.6 Documentation | Epic Developer Community*, 2025).

Um realistische Vegetation in die Spielwelt zu integrieren, können entweder *Asset*s wie Bäume, Gras oder Büsche manuell in das Level gesetzt werden, wie es beispielsweise in *Uncharted* der Fall ist. Diese Vorgehensweise gibt den Level Designern die Möglichkeit, den Spieler durch gezielte Platzierung von Vegetation subtil zu lenken (*Level Designers Reveal Early Stages in 'Blocktober' Hashtag*, 2017). Im folgenden Bild (Abbildung 2) ist ein früher Prototyp eines Levels zu sehen, ein sogenannter *Blockout* aus *Uncharted*, bei dem überprüft wird, ob das Level sinnvoll aufgebaut ist.

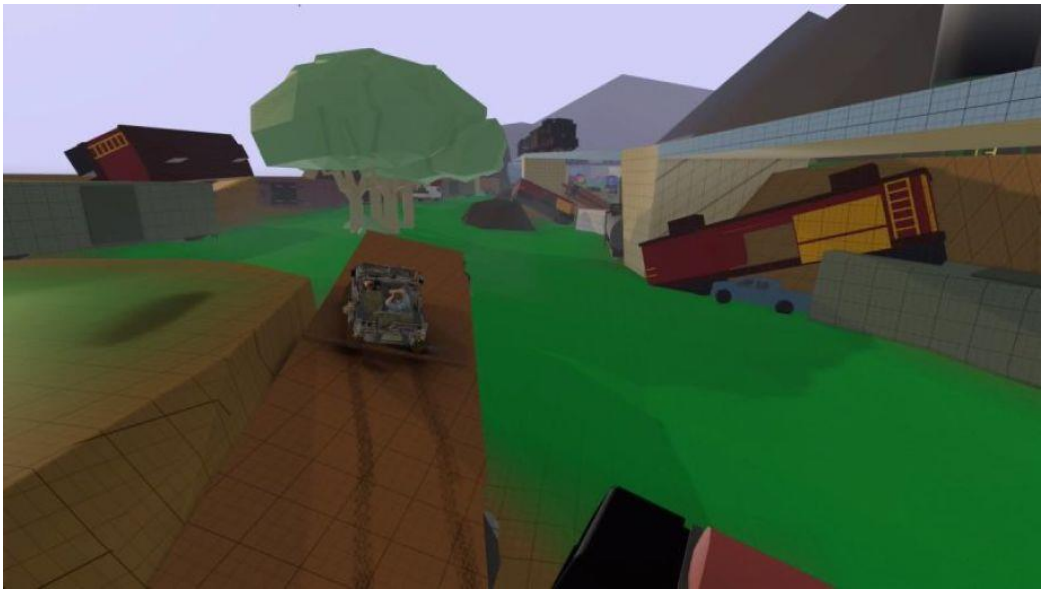


Abb. 2 Blockout eines Uncharted Levels*. Quelle: Klepek, P., VICE, 5. Okt. 2017.*

Um hingegen schnell größere Waldgebiete zu erstellen, ohne jedes Objekt einzeln zu platzieren, bietet die *Unreal Engine* ein Tool namens *Foliage* Tool an. Mit diesem Werkzeug lassen sich wie bei der Landschaftsgestaltung Bäume, Büsche und andere *Asset*s, die vorher in *Foliage-Actor* umgewandelt wurden, direkt auf die Landschaft malen. Auch hier gibt es zahlreiche Einstellungsmöglichkeiten wie etwa die Dichte oder den maximalen Neigungswinkel, auf dem platziert wird. Das *Foliage* Tool stellt somit eine einfache und unkomplizierte Möglichkeit dar, großflächig Vegetation in Levels einzufügen (*Foliage Tool | Unreal Engine 4.27 Documentation | Epic Developer Community*, 2025).

4.1.2. Herausforderungen

Wenn Level manuell aus einzelnen, per Hand platzierten Bausteinen gebaut werden, treten typischerweise Wiederholungen in den verschiedenen Elementen auf. Eine Balance zwischen wiederkehrenden Mustern und neuen Bausteinen, ist dabei besonders wichtig, da diese für Abwechslung sorgen (Ma *et al.*, 2014). Ein weiterer Nachteil manuell erstellter Level ist, dass sie statisch sind. Ein einmal gestaltetes Level bleibt unverändert und bietet beim erneuten Durchspielen keine neuen Inhalte oder Überraschungen. Dies kann den Wiederspielwert erheblich einschränken, da der Spieler bei jedem Durchlauf mit denselben Herausforderungen und Abläufen konfrontiert wird.

Auch bei der Verwendung von *Heightmaps* zur Erstellung der Landschaft gibt es einige wichtige Aspekte zu beachten. Durch Bildkomprimierung kann es leicht zu Artefakten kommen, die bereits durch geringe Unterschiede in der Schattierung entstehen(Ahearn, 2008).

Außerdem ist es entscheidend, eine passende Anzahl an Polygonen für die Landschaft zu wählen. Wenn mehr Polygone verwendet werden, wird die Landschaft zwar detailreicher dargestellt, gleichzeitig erhöht sich jedoch auch der Rechenaufwand(Ahearn, 2008).

Bei der Verwendung des *Foliage* Tools gibt es natürlich auch Herausforderungen. Eine davon ist der Mangel an präziser Kontrolle. Da die *Asset*s auf die Landschaft gemalt werden, ist es schwierig, sie an exakt bestimmten Positionen zu platzieren. Auch das nachträgliche Bearbeiten ist aufwendig, da sich die platzierten *Asset*s nicht einfach verschieben lassen.

Ein weiterer Nachteil besteht darin, dass sich gemalte Landschaften nicht ohne Weiteres in andere Level übertragen lassen. Wenn beispielsweise ein Wald gestaltet wurde und anschließend neue *Asset*s wie Pilze hinzugefügt werden sollen, müssen diese in jedem Level erneut manuell platziert werden.

Zudem erlauben die platzierten *Asset*s keine Veränderungen zur Laufzeit und besitzen keine physikalischen Eigenschaften(*Foliage Tool | Unreal Engine 4.27 Documentation | Epic Developer Community*, 2025).

4.2. Procedural Content Generation

4.2.1. Herangehensweise

Noise Funktionen

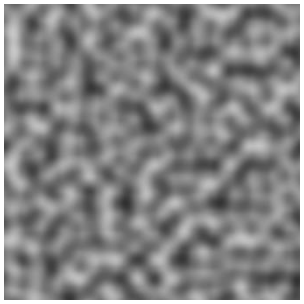


Abb. 3 Perlin Noise

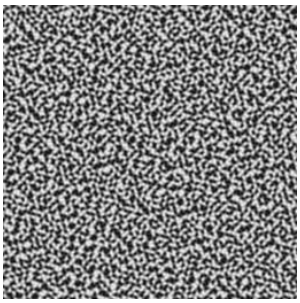


Abb. 4 Simplex Noise

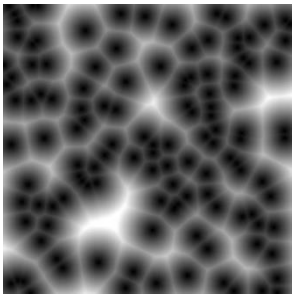


Abb. 5 Worley Noise

Perlin (Abbildung 3), Simplex (Abbildung 4) und Worley (Abbildung 5) Noise Algorithmen zählen zu den am häufigsten verwendeten Verfahren bei der prozeduralen, realistischen Landschaftsgestaltung(Parberry, 2014; Wijaya and Rahman, 2018). Sie basieren alle auf zufallsverteilten Rauschfunktion und erzeugen Graustufen-Höhenkarten, welche als Grundlage für Geländemodelle dienen können. Durch das Verwenden von Parametern können sie leicht angepasst werden, um verschiedenste Landschaften abzubilden. Simplex Noise, eine weiterentwickelte Variante, verringert den Speicherbedarf und hält am selben Graustufenprinzip fest(Wijaya and Rahman, 2018).

Worley Noise generiert Landschaften basierend auf den abständen der zufällig verteilten Zellkerne, welches sich besonders für Flüsse und andere organische Strukturen eignet. Für eine detailliertere Beschreibung der einzelnen Algorithmen wird auf (Burhöi *et al.*, 2017) verwiesen.
Ein häufig genanntes Beispiel für diese Art der Welten Generierung ist *Minecraft*. *Minecraft* ist ein Open-World-Sandbox-Spiel, dessen *Asset*s hauptsächlich aus Blöcken bestehen und eine fast unendlich große Spielwelt besitzt. Die verschiedenen Arten der Blöcke sind durch ihre Texturen erkennbar(Burhöi *et al.*, 2017; Zheng, 2024). Die Spielwelt wird durch eine skalierte Perlin Noise mit linearer Interpolation erstellt(*Minecraft - Procedural Content Generation Wiki*, 2012), welches die Spieler abwechslungsreiche Biome mit Bergen, Tälern und Höhlen erkunden lässt.

Abb. 6 Screenshot einer Minecraft Welt. Quelle: Minecraft Backrooms Wiki (Fandom). Zugriff: 12.09.2025.

4.2.2. Herausforderungen

Prozedural generierte Inhalte (engl. Procedural Content Generation, kurz *PCG*) in Videospielen öffnen zahlreiche Lösungsansätze für Herausforderungen, die unter anderem aus dem Wunsch nach inhaltlicher Vielfalt sowie der Erhöhung des Wiederspielwerts resultieren. Jedoch bringt ihre Implementierung sowohl technische als auch kreative Herausforderungen mit sich. Insbesondere Designer stehen vor der Schwierigkeit, das *PCG*-Framework zu verstehen als auch ihre eigenen gestalterischen Wünsche in dieser Methodik ausdrücken(Craveirinha and Roque, 2015).

Darüber hinaus besteht ein Spannungsfeld zwischen optimierungsgetriebenen Ansätzen und der Notwendigkeit von Erkundungsphasen im kreativen Prozess(Craveirinha and Roque, 2015). *PCG* wirft außerdem die Frage der Autorenschaft und der Verantwortlichkeit für die im Spiel implementierten Rhetorik auf(Phillips *et al.*, 2016).

Eine weitere Herausforderung besteht zudem im Verlust der Kontrolle. Bei zu geringer Kontrolle können unerwünschte Szenarios entstehen, und es wird schwierig, Strukturen in den Inhalt einzubinden. Daher ist es entscheidend, einen guten Mittelweg zwischen Zufälligkeit und Anpassbarkeit zu finden, um zum einen dem Spieler ein immersives Erlebnis zu bieten und zum andern dem prozedural generierten Inhalt genügend Freiraum für zufällige Erzeugungen zu lassen(Gervás *et al.*, 2005).

Das Videospiel *No Man´s Sky* ist ein Paradebeispiel für den Einsatz von prozedural generierten Inhalten in einem Videospiel (Abbildung 7). Das Spiel umfasst eine fast unendlich große Spielwelt mit ebenso vielen Planeten sowie eine ebenso vielfältige Flora und Fauna (Tait and Nelson, 2022). All dies wird durch vordefinierte *PCG*-Algorithmen erzeugt. Obwohl dieser Ansatz zunächst für zahlreiche Variationen und inhaltliche Diversität sorgt, führt er in der Spielerfahrung jedoch oft zu einer empfundenen Leere und langweile. So ist das Spiel zwar inhaltlich groß, jedoch zugleich inhaltlich beliebig(Douglas Heaven, 2016).

Abb. 7 Screenshot von zufällig generierter Welt aus No Man's Sky. Quelle: No Man's Sky Wiki. Zugriff: 12.09.2025.

4.2.3. Verwendungen in der Industrie

No Man's Sky ist natürlich nicht das einzige Spiel, das prozedural generierte Inhalte verwendet. In der folgenden Tabelle sind zahlreiche Spiele aufgeführt sowie die jeweiligen Bereiche, die prozedural generiert werden. Zusätzlich ist die Größe der Studios ergänzt, um zu verdeutlichen, dass *PCG*-generierter Content bereits in Spielen unterschiedlichster Größenordnung Anwendung findet. Aus der Tabelle lässt sich ebenfalls erkennen, dass Spiele in der Regel nicht vollständig auf prozedural generierte Inhalte setzen, sondern nur bestimmte Bestandteile des Spiels mithilfe von *PCG* erstellen.

Games	**Release**	**Game Bits**	**Game Space**	**Game Systems**	**Game Scenarios**	**Studio Size**
Borderlands	2009	X				AAA
Diablo I	2000		X			AAA
Diablo II	2008		X		X	AAA
Dwarf Fortress	2006		X	X	X	Indie
Elder Scrolls IV: Oblivion	2007	X				AAA
Elder Scrolls V: Skyrim	2011				X	AAA
Elite	1984		X	X	X	Indie
EVE Online	2003	X	X		X	AA
Facade	2005				X	Indie
FreeCiv and Civilization IV	2004		X			AA
Fuel	2009		X			AAA
Gears of War 2	2008	X				AAA
Left4Dead	2008				X	AAA
.kkrieger	2004	X				Indie
Minecraft	2009		X	X		Indie
Noctis	2002		X			Indie
RoboBlitz	2006	X				Indie
Realm of the Mad God	2010	X				Indie
Rogue	1980		X		X	Indie
Spelunky	2008	X	X			Indie
Torchlight	2009		X			AA
X-Com: UFO Defense	1994		X			AA

Table 1 Use of PCG-G Techniques in Games (Adaptiert nach (Hendrikx et al., 2013) mit zusätzlichen Daten vom Verfasser)

Ein weiteres Beispiel aus Tabelle 1 ist *Diablo*, das dafür bekannt ist, Level und Karten prozedural zu generieren, welches den wiederspielwert des Spiels deutlich erhöht (Abbildung 8). Diese Art der Levelerstellung hat den Vorteil, dass auch erfahrene Spieler immer wieder neue Herausforderungen und unerwartete Elemente entdecken können, selbst wenn diese das Spiel seit längerer Zeit spielen.

Allerdings wirken solche prozedural erzeugten Level häufig generisch, da ihnen wiedererkennbare Merkmale oder charakteristische Gestaltungselemente fehlen, wie sie in klassischen *Tilesets* vorkommen(Pereira de Araujo and Souto, 2017). Um dennoch eine dichte Atmosphäre und eine starke Handlung bieten zu können, enthält *Diablo* auch vorgenerierte Kartenbereiche, in denen der Spieler mit der Szenerie interagieren kann ohne, dass diese Elemente Einfluss auf die prozedural generierten Dungeons haben.

Durch diese geschickte Trennung von Story relevanten Abschnitten und zufällig erzeugten Spielabschnitten gelingt es *Diablo*, einen besonders hohen Wiederspielwert zu erreichen(Pereira de Araujo and Souto, 2017).

Abb. 8 Screenshot von zufällig generiertem Level *aus Diablo IV. Quelle: MeinMMO. Zugriff: 12.09.2025.*

Natürlich sind Landschaften und Level nicht die einzigen Anwendungsbereiche prozeduraler Generierung. Ein Beispiel aus einem anderen Bereich bietet *The Elder Scrolls V: Skyrim*. Viele Rollenspiele basieren auf einem sogenannten Quest-System, bei dem der Spieler in der Regel von nicht spielbaren Charakteren (*NPC*s) auf Missionen geschickt wird, für deren Abschluss er eine Belohnung erhält.

Für *Skyrim* wurde ein sogenanntes Radiant-Quest-System entwickelt. Dabei handelt es sich um prozedural generierte Nebenquests, die unabhängig von der Haupthandlung sind und sich dynamisch an die aktuelle Spielwelt und den Fortschritt des Spielers anpassen. Ein Beispiel dafür sind die Aufträge der Assassinen-Gilde, in denen der Spieler zufällig ausgewählte *NPC*s eliminieren muss (Abbildung 9). Diese Nebenquests sind komplett vertont und fühlen sich damit sehr hochwertig an.

Abb. 9 Screenshot aus Skyrim Dialog einer zufälligen Assassinen Quest. Quelle: Namu Wiki. Zugriff: 12.09.2025.

Auch wenn diese Quests inhaltlich oft als „Filler Content“ angesehen werden, bieten sie dem Spieler dennoch eine Möglichkeit, Erfahrung zu sammeln, Gegenstände zu finden und die Spielwelt von *Skyrim* weiter zu erkunden. Zudem sind sie häufig interessanter gestaltet als klassische Standardquests nach dem Schema „Töte zehn Ratten, sammle zehn Blumen, etc.“ (‘*Skyrim*: Radiant Quest System | Terminally Incoherent’, 2011).

4.3. PCG-Graphen in der Unreal Engine

Unreal Engine bietet seit Version 5.6 ein eigenes Procedural Content Generation (*PCG*) Framework an. Dieses Framework basiert, wie auch der Material- und *Blueprint*-Editor, auf einem *Node*-System. Das bedeutet, dass kein tatsächlicher Code geschrieben werden muss, sondern lediglich *Node*-Blöcke miteinander verbunden werden, die intern vordefinierte Funktionen ausführen. Dadurch können auch Artists ohne umfangreiche Programmierkenntnisse das System relativ einfach erlernen und nutzen.

Um das *PCG*-Framework verwenden zu können, muss zunächst das Procedural Content Generation Framework Plugin aktiviert werden. Eine Erweiterung des Frameworks mit C++ ist ebenfalls möglich.

Ein zentraler Aspekt des *PCG*-Editors ist die Echtzeitausgabe sowie die Verwendung sogenannter Points (Abbildung 10). Diese enthalten Transformationsdaten sowie zusätzliche Informationen wie Farbe, Dichte, ein *Bounding*-Volumen und einen *Seed*-Wert. Die Punktdichte steuert beispielsweise die Wahrscheinlichkeit, mit der Punkte tatsächlich gespawnt werden(*Procedural Content Generation Overview | Unreal Engine 5.6 Documentation | Epic Developer Community*, 2025).

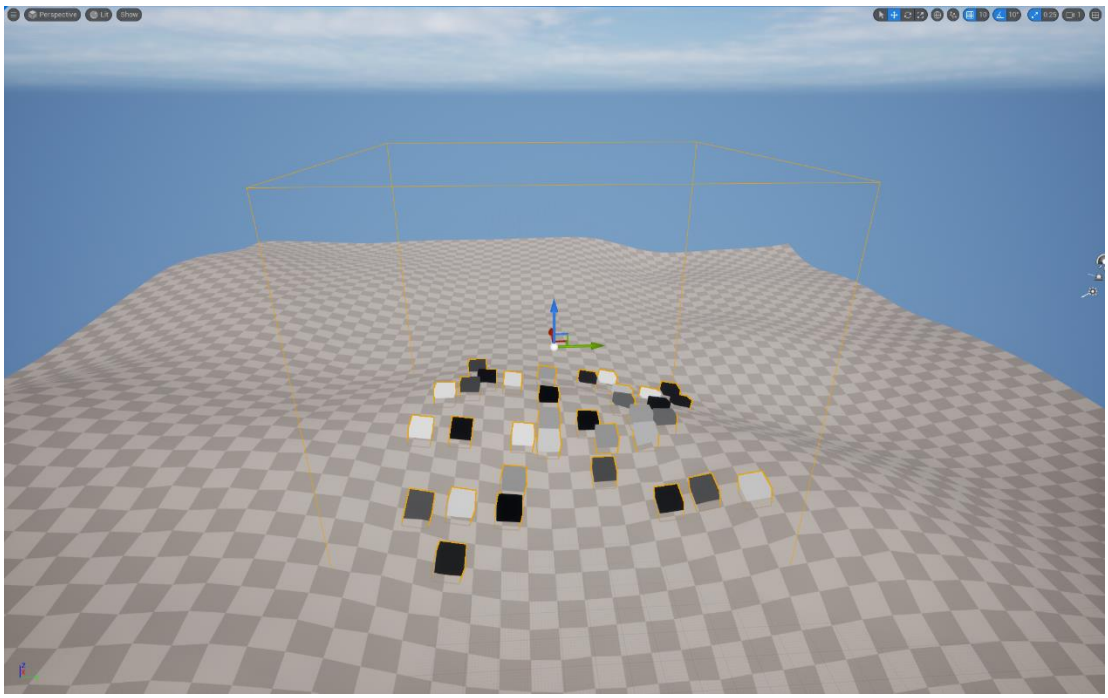


Abb. 10 Darstellung von PCG-Punkten Quelle: Epic Games (2025), Unreal Engine 5.6 Documentation.

In Kapitel 6.2 werden wir die verschiedenen *Node*s und Aspekte des *PCG*-Frameworks noch genauer betrachten.

4.4. Umfrage unter Indie Studios

Im Rahmen dieser Arbeit wurde eine Online-Umfrage mit sieben Indie-Studios durchgeführt, um praktische Erfahrungen und Einschätzungen zur Nutzung prozeduraler Inhaltsgenerierung (*PCG*) zu erfassen. Der vollständige Fragebogen befindet sich im Anhang. Von den sieben teilnehmenden Studios gaben zwei an, *PCG* in ihren Projekten einzusetzen, während fünf Studios angaben, kein *PCG* zu verwenden.

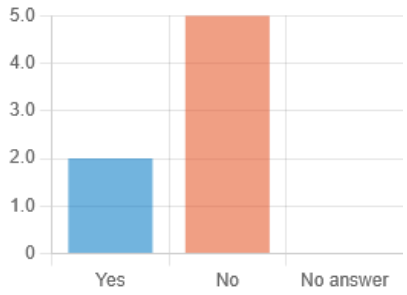


Tabelle 1 „Do you use any Procedural Content Generation (PCG) techniques in your game development?"

Auffällig ist, dass fünf der befragten Studios mit der *Unreal Engine* arbeiten, was auf eine technologische Präferenz innerhalb der Umfrage hinweist.

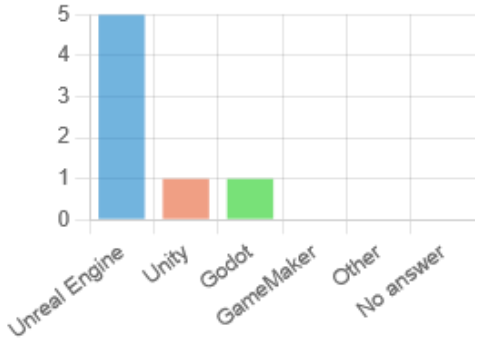


Tabelle 2 "Which game engine do you primarly use for your projects?"

Die Studios, die *PCG* einsetzen, nannten mehrere, annähernd gleich starke Motivationen:

- Zeitersparnis in der Entwicklung
- größere Variationsbreite der erstellten Inhalte
- erhöhte Wiederspielbarkeit
- Automatisierung repetitiver Aufgaben

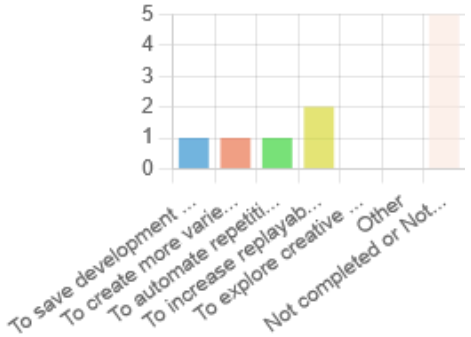


Tabelle 3 "What are your main reasons for using PCG?"

Diese Gründe deuten darauf hin, dass *PCG* in der Praxis vor allem dort eingesetzt wird, wo Effizienzgewinne und eine größere Vielfalt an Inhalten einen direkten Mehrwert für das Projekt bieten. Demgegenüber begründeten die Studios, die auf *PCG* verzichten, ihre Entscheidung überwiegend pragmatisch. Entweder sei das jeweilige Spielkonzept mit handgefertigten Inhalten besser bedient („das Spiel braucht es nicht“), oder *PCG* wird als zu komplex eingeschätzt bzw. es fehlt an entsprechendem Fachwissen. Mehrere Befragte gaben außerdem an, dass sie aus gestalterischer Präferenz lieber Inhalte manuell erstellen.

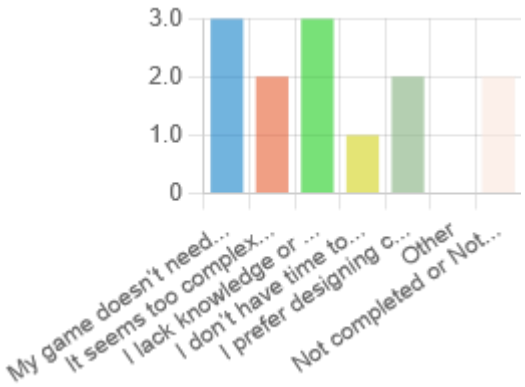


Tabelle 4 "What are your main reasons for not using PCG?"

Ein weiterer, aus den Antworten ableitbarer Befund betrifft die Einschätzung der Vor- und Nachteile von *PCG*. Die Anwender von *PCG* stimmen darin überein, dass prozedurale Verfahren Entwicklungszeit sparen, die Vielfalt und Flexibilität von Content erhöhen können, aber auch das *Debugging* erschweren kann.

Dieser Zwiespalt zwischen Effizienz und Variation versus erhöhten technischen Aufwand und Komplexität spiegelt die in der Fachliteratur beschriebene Diskussion wider. Für die Fragestellung dieser Arbeit ist wichtig, dass die Entscheidung für oder gegen *PCG* offenbar weniger eine rein theoretische Abwägung ist, sondern stark von konkreten Projektanforderungen, vorhandenen Ressourcen und Expertise abhängt.

Abschließend ist die geringe Stichprobengröße als Limitation zu nennen. Die Ergebnisse liefern eine praxisnahe Momentaufnahme und ermöglichen qualitative Einsichten, sind jedoch nur bedingt repräsentativ und sollten daher mit Vorsicht generalisiert werden.

4.5. Forschungslücke und Begründung der Fragestellung

Die bisherigen Ausführungen in Kapitel 4 fassen die technischen Grundlagen und praktischen Erfahrungen zur manuellen und prozeduralen Levelgestaltung zusammen. Die Untersuchungen und Praxisbeispiele in diesem Kapitel zeigen nicht nur konkrete Vor- und Nachteile der beiden Ansätze, sondern verdeutlichen zugleich eine grundlegende Forschungslücke im gesamten Feld. Bisher existieren nur wenige Studien, die prozedurale Ansätze in der *Unreal Engine* systematisch und quantitativ mit klassischen, manuellen Workflows vergleichen. Vorhandene Arbeiten beschreiben häufig technische Verfahren, liefern jedoch selten reproduzierbare Zeitmessungen, belastbare Usability-Analysen oder Aussagen zur Wiederverwendbarkeit und Skalierbarkeit unter praxisnahen Bedingungen.

Diese Lücke bestätigt sich auch in den Ergebnissen der im Rahmen dieser Arbeit durchgeführten Umfrage unter Indie Studios. Von diesen gaben lediglich zwei an, *PCG* aktiv in ihren Projekten zu nutzen, während fünf darauf verzichten. Als Hauptgründe für den Einsatz wurden Zeitersparnis, eine größere inhaltliche Variationsbreite sowie die Automatisierung repetitiver Aufgaben genannt. Studios, die auf *PCG* verzichten, verwiesen dagegen auf fehlendes Fachwissen, den hohen technischen Aufwand oder gestalterische Präferenzen zugunsten handgefertigter Inhalte. Diese Rückmeldungen verdeutlichen, dass *PCG* in der Praxis zwar Potenzial zur Effizienzsteigerung bietet, seine Anwendung jedoch stark von der vorhandenen Expertise und den jeweiligen Projektanforderungen abhängt.

Insgesamt zeigt sich damit, dass es bislang an praxisnahen, empirischen Vergleichen fehlt, die Rückschlüsse darauf erlauben, unter welchen Bedingungen *PCG-Graphen* tatsächlich schneller, flexibler oder nutzerfreundlicher sind als manuelle Verfahren. Ohne solche Ergebnisse bleibt es ins besonders für kleinere Entwicklerteams schwierig, den tatsächlichen Nutzen von *PCG*-Systemen im Verhältnis zum erforderlichen Zeitaufwand realistisch einzuschätzen.

Vor diesem Hintergrund leitet sich die Fragestellung dieser Arbeit aus dem aktuellen Forschungsstand ab. Untersucht werden soll dabei nicht nur die technische Umsetzung von *PCG-Graphen*, sondern insbesondere, ob auch weniger erfahrene Entwickler mithilfe eines strukturierten Leitfadens in der Lage sind, innerhalb kurzer Zeit überzeugende Ergebnisse zu erzielen. Ziel ist es zu prüfen, ob der Einsatz von *PCG-Graphen* nicht ausschließlich für erfahrene Fachkräfte, sondern auch für kleinere Indie-Studios einen praktischen Mehrwert darstellt. Von besonderem Interesse ist dabei, ob die Methode im Vergleich zu herkömmlichen Workflows eine messbare Zeitersparnis ermöglicht, ohne dass Qualität oder kreative Gestaltungsfreiheit beeinträchtigt werden. Die gewählte Mixed-Methods-Vorgehensweise ergibt sich folglich direkt aus der identifizierten Forschungslücke und soll aufzeigen, in welchem Umfang *PCG-Graphen* für kleinere Entwicklerteams ein realistisch einsetzbares Tool zur Effizienzsteigerung und Prozessoptimierung in der Levelgestaltung sein können.

5. Prototypisierung

5.1. Methodische Einordnung des experimentellen Vorgehens

Ein Experiment ist eine systematisch kontrollierte empirische Untersuchungsmethode, die der Prüfung von Hypothesen durch gezielte Variation einer unabhängigen Variable und gleichzeitige Messung der Auswirkungen auf abhängige Variable(n) dient(Lang, 2010).

5.2. Unabhängige und abhängige Variablen

Für die vorliegende Arbeit bildet die Methode der Levelerstellung (manuelle Gestaltung mittels *Foliage-Tool* vs. prozedurale Erzeugung mittels *PCG-Graph*) die unabhängige Variable. Als abhängige Variablen werden primär der gemessene Zeitaufwand (Initialaufbau, Zeit für vorgegebene Änderungen) und die subjektiv wahrgenommene Usability (Erfassung mittels standardisiertem Fragebogen) herangezogen. Ergänzend werden Indikatoren wie wahrgenommene Flexibilität, Wiederverwendbarkeit und Zufriedenheit erfasst.

5.3. Versuchsplan und Designentscheidung

Als Versuchsplan wurde ein Within-Subject / Repeated-Measures-Design gewählt, bei dem dieselben Teilnehmenden beide Bedingungen bearbeiten. Ein solches Design erlaubt direkte intraindividuelle Vergleiche und reduziert die zwischen-Personen-Varianz, da individuelle Faktoren wie allgemeine Arbeitsgeschwindigkeit, Feinmotorik oder Vorerfahrung in beiden Bedingungen gleichermaßen wirken und somit als Kontrollmechanismus fungieren(Bortz and Döring, 2006).

Die Entscheidung für ein Within-Subject-Design folgt der empfohlenen Abwägung zwischen interner Effizienz, das heißt höhere Teststärke bei geringerer Stichprobengröße und zusätzlich bietet es die Möglichkeit, individuelle Präferenzen in Bezug auf Usability unmittelbar zu erfassen. Entsprechende Gegenmaßnahmen (vgl. Kapitel 5.5) wurden geplant, um die interne Validität zu stützen.

5.4. Operationalisierung und Messprozeduren

Die Überführung abstrakter Konstrukte in messbare Indikatoren (Operationalisierung) ist zentraler Bestandteil der Versuchsplanung(Bortz and Döring, 2006).
Konkret bedeutet das für diese Studie:

- Zeitaufwand: wird anhand klar definierter Start-/Stopp-Regeln gemessen (Start = Beginn der ersten produktiven Aktion nach Instruktion, Stopp = Abschluss der letzten geforderten Handlung).
- Usability: wird durch einen standardisierten Fragebogen (siehe Anhang) geprüft. Die Fragen orientieren sich an praxisrelevanten Aspekten wie Verständlichkeit der Tools, Aufwand für Änderungen, und subjektive Zufriedenheit.

5.5. Kontrolle von Störvariablen und Validität

Ein zentrales Merkmal experimenteller Forschung ist die Kontrolle von Störvariablen, um kausale Schlussfolgerungen zu ermöglichen(Bortz and Döring, 2006; Lang, 2010).
Zur Minimierung möglicher Verzerrungen wurden folgende Maßnahmen getroffen:

- Standardisierte Arbeitsumgebung (gleiche *Unreal-Engine*-Version, identisches Template-Projekt, vergleichbare Hardware).
- Einheitliche, schriftliche Aufgabenbeschreibungen.
- Erfassung relevanter Hintergrundvariablen (Vorerfahrung mit *Unreal Engine*) zur späteren statistischen Kontrolle.
- Durchführung eines Pilotversuchs zur Feinjustierung von Instruktionen und Messprozeduren.

Diese Maßnahmen dienen der Erhöhung der internen Validität, die Auseinandersetzung mit Validitäts- und Reliabilitätsfragen ist zentral in der Methodenliteratur und wird hier entsprechend berücksichtigt(Bortz and Döring, 2006; Lang, 2010).

5.6. Gütekriterien (Objektivität, Reliabilität, Validität)

Die Datenerhebung und Datenauswertung orientieren sich an den klassischen Gütekriterien:

- Objektivität: Standardisierte Messprozeduren, klar definierte Start/Stop-Regeln und schriftliche Protokolle reduzieren messtechnische Einflüsse(Lang, 2010).
- Reliabilität: Konsistente Messregeln, ggf. Wiederholungsmessungen und die Durchführung eines Pilotversuchs dienen der Absicherung gegen Zufallsfehler(Bortz and Döring, 2006).
- Validität: Die Auswahl der Messindikatoren (Zeitaufwand, inhaltsnahe Usability-Fragen) ist theoriegeleitet und auf das Untersuchungsziel ausgerichtet, um die inhaltliche Validität zu stärken(Bortz and Döring, 2006; Lang, 2010).

5.7. Einschränkungen und Hinweise zur Interpretation

Trotz der genannten Kontrollmaßnahmen bleiben Einschränkungen bestehen. Reihenfolge- bzw. Übungseffekte beim Within-Subject-Design lassen sich zwar durch Pausen mindern, jedoch nicht vollständig ausschließen. Die Stichprobengröße sowie mögliche Selbstselektion der Teilnehmenden schränken die externe Validität ein. Solche Limitationen werden bei der Auswertung offen kommuniziert und bei Schlussfolgerungen berücksichtigt(Bortz and Döring, 2006).

5.8. Planung

Zur Untersuchung der Unterschiede in der manuelle Levelgestaltung und der mittels automatisch prozedural generierten Generierung wurde ein Experiment konzeptioniert. Ziel ist es, beide Arbeitsweisen hinsichtlich Effizienz und Usability zu vergleichen. Im Folgenden werden die Vorbereitungen, der genaue Ablauf sowie die Aufgabenstellung des Experiments im Detail beschrieben.

Im ersten Schritt wurden geeignete Teilnehmer rekrutiert. Dabei wurde darauf geachtet, eine möglichst heterogene Gruppe bezüglich der Erfahrung mit der *Unreal Engine* zusammenzustellen, um ein realistisches Abbild potenzieller Nutzergruppen zu erhalten. Alle Teilnehmenden arbeiteten mit derselben *Unreal Engine* Version (5.6) und möglichst vergleichbarer Hardware, um technische Einflussfaktoren zu minimieren. Zur Erhebung grundlegender Informationen wurde ein Eingangsfragebogen erstellt (vgl. Kapitel 10.1), die demografischen Daten, das Vorwissen in Bezug auf die Engine sowie die Vorerfahrung im Bereich der prozeduralen Generierung abfragt.

Für die Durchführung des Experiments wurde ein Template-Projekt vorbereitet, das alle notwendigen Plugins enthält. Dieses Projekt diente als einheitliche Arbeitsumgebung für alle Teilnehmenden. Zudem wurde vor Beginn der eigentlichen Erhebung ein Pilotversuch mit einer Person durchgeführt. Dieser diente dazu, potenzielle Unklarheiten in der Aufgabenstellung oder technische Schwierigkeiten frühzeitig zu identifizieren und das Versuchskonzept entsprechend anzupassen.

Vor dem eigentlichen Experiment erhielten alle Teilnehmenden einen Fragebogen (siehe Anhang) zur Selbsteinschätzung ihrer Kenntnisse und Fähigkeiten im Umgang mit der *Unreal Engine* sowie den relevanten Tools.

Im Anschluss daran begann der Hauptteil des Experiments, der aus zwei getrennten Aufgaben bestand. Diese waren eine manuelle Umsetzung (Aufgabe 1) und einer prozedural generierten Umsetzung (Aufgabe 2) um ein Level zu gestallten.

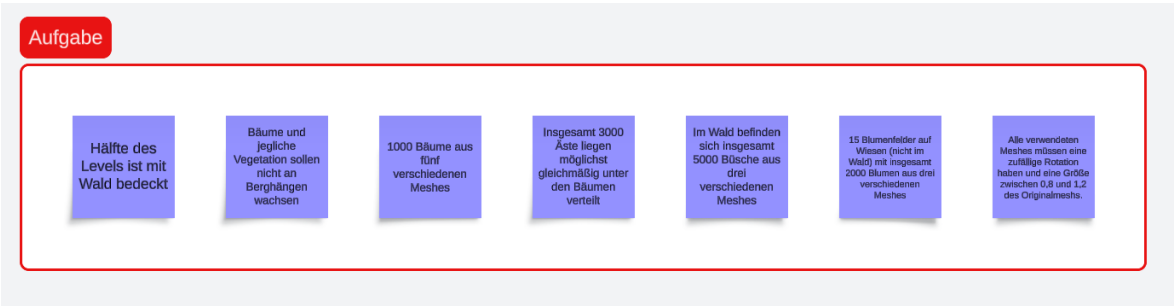


Abb. 11 Screenshot der Anforderungen aus den Notizen.

Beide Aufgaben basierten auf einem zuvor vorbereiteten Blockout-Layout. Die Teilnehmer erhielten die Aufgabe, ein Gelände mit den folgenden Merkmalen vollständig auszugestalten. Ein Kriterium war, die Hälfte der Karte mit Wald zu bedecken. Dieser sollte aus insgesamt 1000 Bäumen bestehen, die sich aus fünf unterschiedlichen Baummeshes zusammensetzen.

Unter den Bäumen sollten insgesamt 3000 Äste verteilt werden, die möglichst gleichmäßig platziert sind und auf drei verschiedene *Meshes* zurückgreifen. Innerhalb des Waldes sollten darüber hinaus drei verschiedene Arten von Büschen wachsen, insgesamt 5000 Stück.

Auf den Wiesenflächen zwischen den bewaldeten Gebieten sollten 15 Blumenfelder angelegt werden. Diese sollten sich aus insgesamt 2000 Blumen zusammensetzen und auf drei unterschiedliche Blumenmeshes zurückgreifen. Sämtliche Vegetation durfte nur auf Grasflächen wachsen, nicht auf Klippen oder anderem steilem Untergrund. Zusätzlich sollten alle verwendeten *Meshes* eine zufällige Rotation haben und eine Größe zwischen 0,8 und 1,2 des Originalmeshs.

Während der Bearbeitung wurden die Bearbeitungszeiten jeder teilnehmenden Person individuell erfasst. Zunächst für Aufgabe 1 und 2 (Abbildung 11).

Abb. 12 Screenshot der Veränderungsanforderungen aus den Notizen.

Nach der Erstellung des initialen Layouts wurden zusätzliche Änderungen vorgenommen, um die Flexibilität beider Methoden zu testen (Abbildung 12). Zu diesen Veränderungen gehörte die Platzierung von 15 neuen Pilzfeldern, bestehend aus zwei unterschiedlichen Pilzarten und insgesamt 500 Pilzen. Diese sollten ausschließlich innerhalb des Waldes erscheinen.

Zusätzlich wurde ein weiterer Baumtyp hinzugefügt, wodurch sich die Gesamtzahl der verwendeten Baummeshes auf sechs erhöhte. Gleichzeitig wurde die Gesamtzahl der Bäume halbiert. Zwei weitere Buscharten wurden ergänzt, die Gesamtanzahl der Büsche blieb jedoch unverändert.

Darüber hinaus wurde in der Mitte der Karte ein vorgefertigtes Dorf platziert. In diesem Bereich durfte keinerlei Vegetation wachsen, also weder Bäume noch Büsche, Blumen oder andere Elemente.

Abschließend wurde ein neues Level mit anderem Layout und mit leicht veränderten Höhen- und Tiefenwerten erstellt, um die Wiederverwendbarkeit beider Methoden unter geänderten Bedingungen zu untersuchen.

Nach Abschluss beider Aufgaben füllten die Teilnehmenden einen Nachfragebogen aus (vgl. Kapitel 10.2), der die subjektive Bewertung beider Arbeitsmethoden erfasste.

Abgefragt wurde unter anderem der wahrgenommene Aufwand, die Verständlichkeit der Werkzeuge sowie die Zufriedenheit mit dem erzielten Ergebnis.

5.9. Pilotversuch

Zur Überprüfung der Durchführbarkeit des Experiments wurde ein Pilotversuch mit einer Testperson durchgeführt, die als Amateur ohne vertiefte Vorerfahrung im Bereich der Levelgestaltung einzustufen ist. Ziel war es, das Vorgehen bei der Nutzung des *PCG Graphs* und des *Foliage Tools* zu evaluieren und mögliche Verständnisschwierigkeiten zu identifizieren. Es ging dabei nicht darum, Zeiten zu erfassen, sondern ausschließlich darum, die Verständlichkeit der Anleitungen zu überprüfen.

Während des Versuchs zeigte sich, dass die Testperson Probleme hatte, unterschiedliche *Density*-Werte für Büsche und andere Vegetationstypen korrekt zuzuweisen. Infolgedessen wurde das entsprechende Kapitel überarbeitet, um die Vorgehensweise klarer darzustellen und die Parameterzuweisung verständlicher zu gestalten.

Des Weiteren traten beim Arbeiten mit dem *PCG Graph* Schwierigkeiten im Umgang mit Verbindungen zwischen *Nodes* auf. Die Anleitung wurde entsprechend angepasst, um auf diese Problematik aufmerksam zu machen und den Umgang mit Verbindungen zu verdeutlichen.

Ein weiterer Aspekt betraf die Wiederverwendung von *Subgraphen* im *PCG Graph*. Die Anleitung wurde ergänzt, sodass nun erläutert wird, wie Subgraphen aus dem *Content Browser* erneut in den Graphen eingefügt werden können, um sie mehrfach zu verwenden, ohne sie neu erstellen zu müssen.

Zudem zeigte sich beim Überprüfen der Punktanzahl im *Copy Points Grid*, dass die Testperson die erzeugten Punkte zunächst schwer einschätzen konnte. Ebenso wurde festgestellt, dass beim Einsatz der *Spline* das *Kompilieren* erforderlich ist, um Änderungen korrekt zu übernehmen.

Auf Grundlage dieser Erkenntnisse wurde die Anleitung überarbeitet, um die Nutzung beider Tools für zukünftige Anwender nachvollziehbarer zu gestalten.

6. Implementierung und Leitfaden

Für die Erstellung des Template-Projekts und die spätere Umsetzung mussten zunächst alle benötigten *Asset*s erstellt werden. Als Inspiration für Vegetation und Gebäude dienten die Spiele *Assassin's Creed Odyssey* und *Sea of Thieves*. Mein Moodboard (Abbildung 13) diente als klare Referenz und half dabei, stilistisch nah an tatsächlichen Umsetzungen in modernen Videospielen zu bleiben.

Abb. 13 Moodboard der Assets

Daraufhin wurden *Asset*s im ähnlichen Stil in Blender modelliert und anschließend in die *Unreal Engine* importiert (Abbildung 14). Dabei war es wichtig, dass die *Asset*s visuell konsistent sind und nicht zu komplex, um mögliche Leistungseinbußen zu vermeiden.

Abb. 14 Alle in Blender modellierten Assets

Für das Grundlevel in der *Unreal Engine* wählte ich ein einfaches Level-Setup und modellierte mithilfe des Sculpting-Tools einige Berge und Täler (Abbildung 15).

Die Beleuchtung basiert auf einer angepassten Standardbeleuchtung mit Bloom-Effekt, Lichtstrahlen durch die Baumkronen und einem Post-Process-Material für eine leicht cartoonhafte Ästhetik.

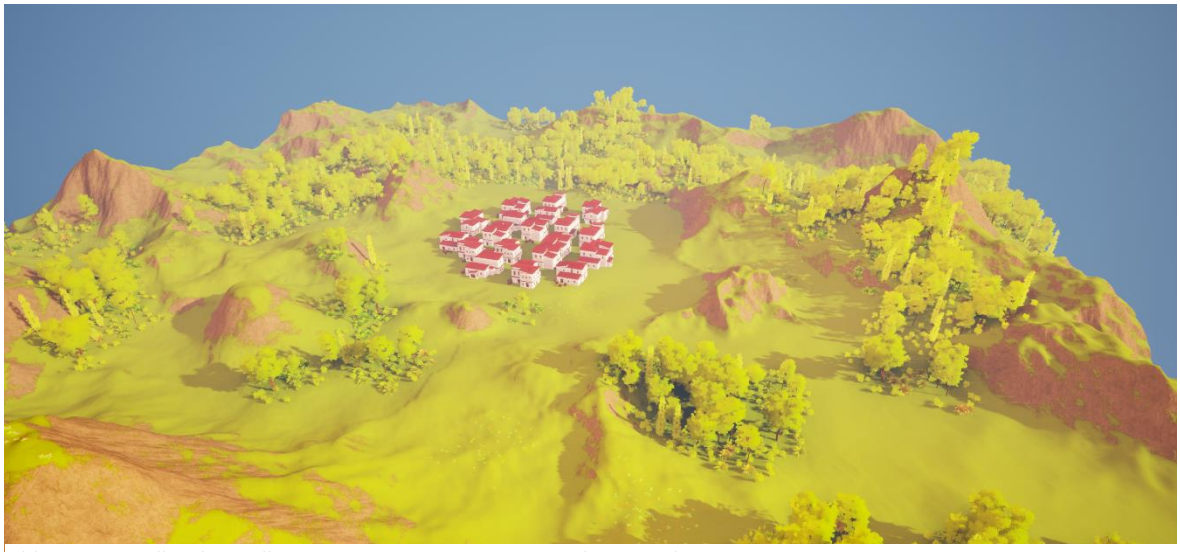


Abb. 15 Beispiellevel mit allen angepassten Parametern in der Unreal Engine

6.1. Manuelle Levelgestaltung: Leitfaden

6.1.1. Foliage Mode

Um das *Foliage-Tool* zu öffnen, kann oben links auf *Selection Mode* geklickt und anschließend *Foliage* ausgewählt werden (Abbildung 16). Alternativ lässt sich das Tool auch schnell über die Tastenkombination *Shift + 3* aktivieren.

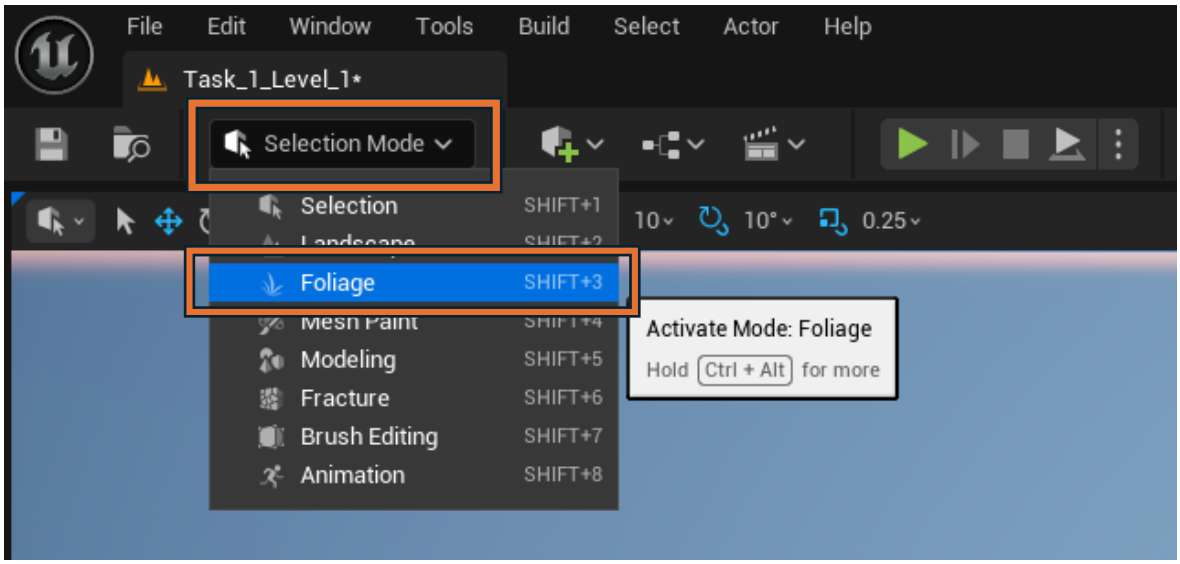


Abb. 16 Auswahl des Foliage Tools

6.1.2. Create Static Foliage Meshes

Nun können die gewünschten *Bäume*, *Büsche* und anderen *Static Meshes* aus dem *Content Browser* in den Ordner *+Drop Foliage Here* gezogen werden (Abbildung 17).

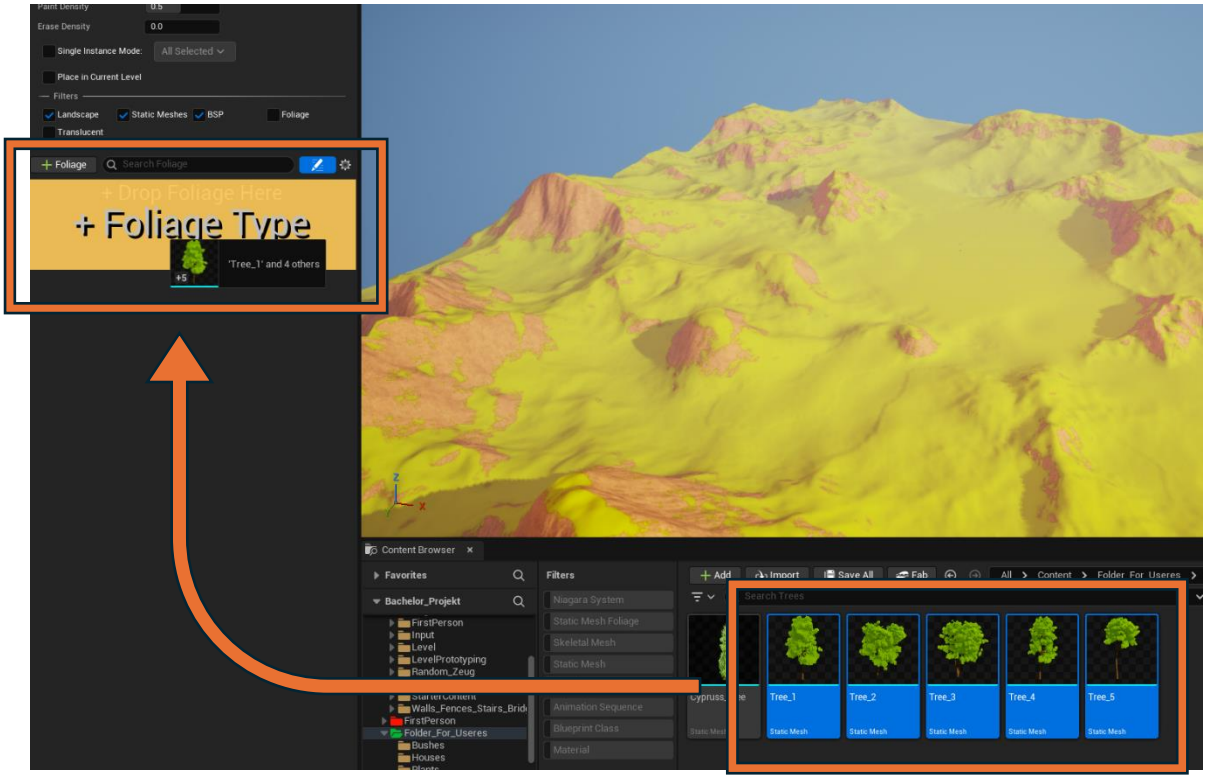


Abb. 17 Erstellen von Static Foliage Meshes

Aktivierte *Meshes* haben oben in der Ecke einen kleinen Haken und können dadurch an- oder ausgeschaltet werden. Wenn *Foliage Meshes* ausgewählt sind, können diese gespeichert werden, um spätere Änderungen ebenfalls zu übernehmen. Dazu fährt man mit der Maus über das ausgewählte *Foliage Asset* und klickt auf den *Save Foliage Asset* Button (Abbildung 18).

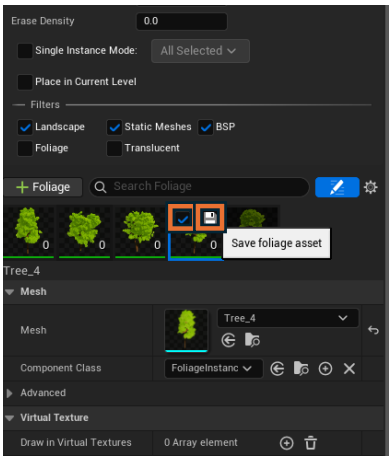


Abb. 18 Aktiven und Speichern der Static Foliage Meshes

Anschließend öffnet sich ein Fenster, in dem der Speicherort und der Name des neuen *Static Mesh Foliage* festgelegt werden können (Abbildung 19). Es wird empfohlen, diesen Vorgang für jedes *Asset* durchzuführen. Dadurch müssen später nicht mehr die einzelnen *Static Meshes* in den *+Foliage Type* Ordner gezogen werden, sondern nur noch die gespeicherten *Static Mesh Foliages*, die bereits alle voreingestellten Einstellungen enthalten.

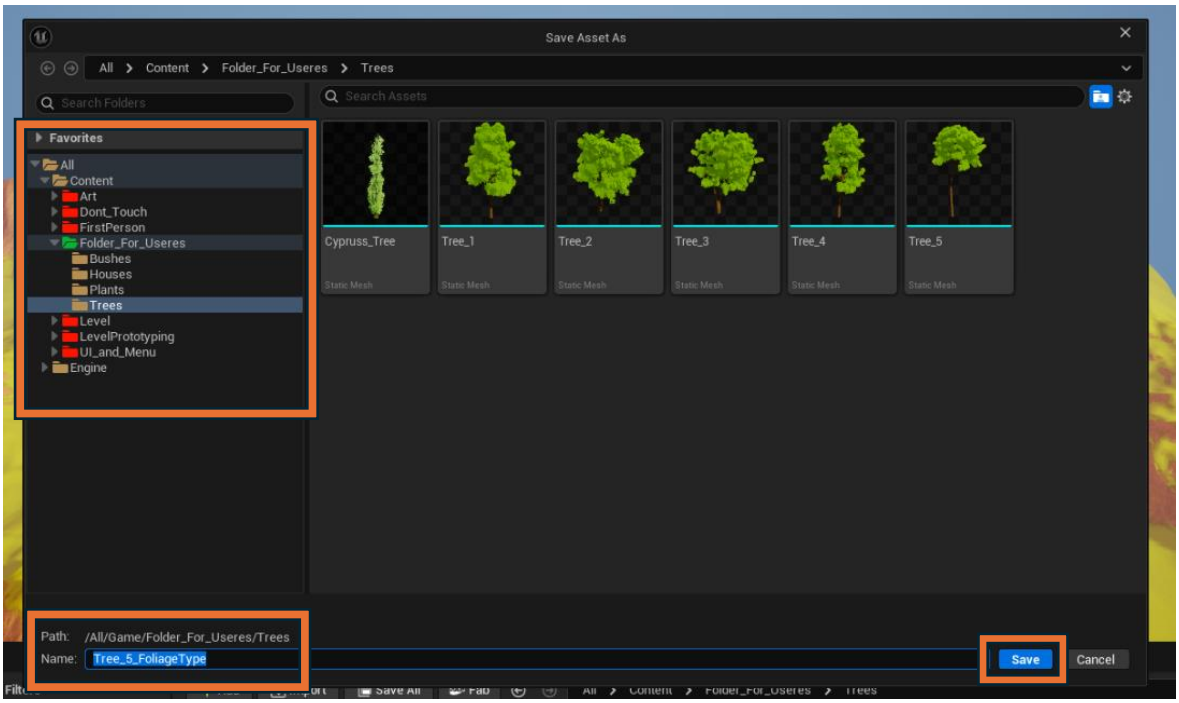


Abb. 19 Speicherort der Static Foliage Meshes

6.1.1. Foliage Brushes

Im *Foliage*-Menü kann oben zwischen verschiedenen *Brush*-Arten gewählt werden. Der *Paint Brush* ist dabei wahrscheinlich das Werkzeug, das am häufigsten genutzt wird. Mit der *Brush Size* lässt sich die Größe des *Brushes* anpassen, während die *Paint Density* bestimmt, wie viele *Meshes* gleichzeitig platziert werden. Es wird empfohlen, die *Paint Density* nicht direkt hier zu verändern, sondern stattdessen später in den einzelnen *Foliage Meshes* die *Density*-Einstellungen anzupassen, um mehr Kontrolle zu haben (Abbildung 20).

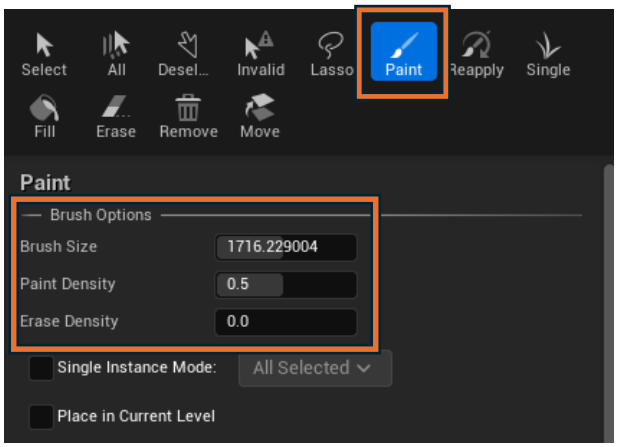


Abb. 20 Paint Brush

Wichtig ist außerdem der *Single Brush*, der von allen ausgewählten *Assets* jeweils genau ein *Foliage Asset* an einer Stelle platziert. Soll stattdessen durch die ausgewählten *Assets* rotiert werden, kann in den Einstellungen des *Brushes* der *Single Instance Mode* von *All Selected* auf *Cycle Through Selected* umgestellt werden (Abbildung 21).

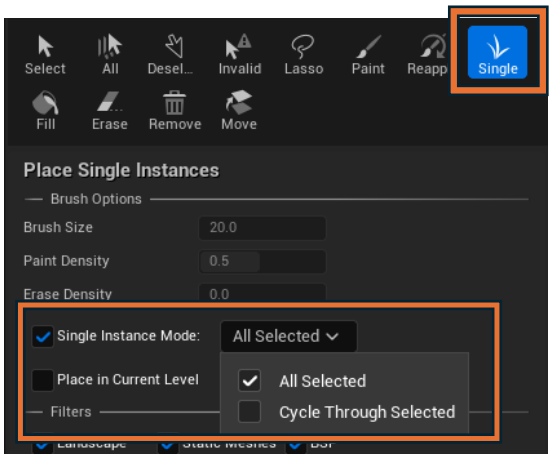


Abb. 21 Single Brush

Der *Erase Brush* entfernt platzierte, ausgewählte *Foliage Meshes*. Befindet man sich im *Paint Brush*, kann man zudem durch das Halten der *Shift*-Taste temporär in den *Erase Brush* wechseln, was ein schnelleres und effizienteres Arbeiten ermöglicht.

6.1.2. Static Foliage Meshes

Jedes der *Static Mesh Foliage Assets* kann individuell bearbeitet werden. Dies kann entweder direkt über den *Content Browser* erfolgen, indem die einzelnen *Static Mesh Foliage Assets* geöffnet werden, oder direkt im *Foliage Tool*. Wenn beispielsweise mehrere Baum-*Meshes* vorhanden sind, können diese im *Foliage Tool* gleichzeitig bearbeitet werden. Dazu werden alle gewünschten Bäume ausgewählt, woraufhin im unteren Bereich ein Einstellungsmenü erscheint. Werden hier Änderungen vorgenommen, müssen anschließend alle betroffenen *Static Mesh Foliage Assets* gespeichert werden, damit die Anpassungen dauerhaft übernommen werden. Im Folgenden wird dies am Beispiel der Bäume erklärt, die Vorgehensweise gilt jedoch gleichermaßen für alle anderen *Foliage*-Elemente wie Blumen, Büsche oder andere Pflanzen.

6.1.2.1. Density

Bevor mit dem Platzieren der Bäume begonnen wird, müssen zunächst einige Einstellungen angepasst werden. Eine dieser Einstellungen ist *Density / 1Kuu*. Dieser Wert legt fest, wie viele Instanzen des ausgewählten *Meshes* pro Fläche gesetzt werden. Ein hoher Wert führt dazu, dass viele und dicht beieinanderstehende Bäume platziert werden, während ein niedriger Wert eine geringere Baumdichte erzeugt. Für unser Experiment empfiehlt sich eine *Density* von 0.3 (Abbildung 22).

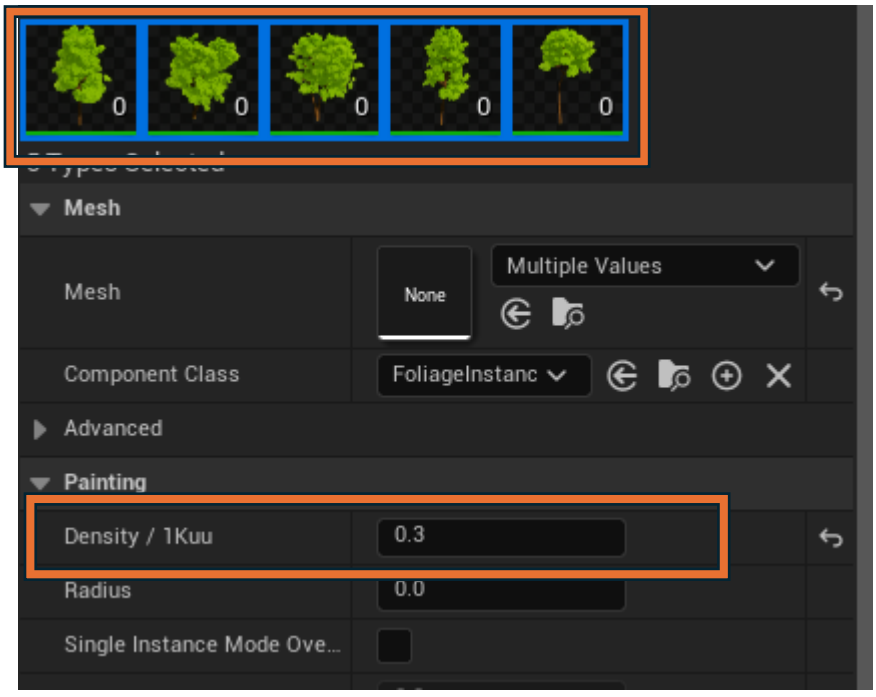


Abb. 22 Density/1Kuu Einstellungen

6.1.2.2. Scale

Die *Scaling*-Einstellung legt fest, ob die platzierten Instanzen unterschiedliche Größen haben sollen. Da die Bäume in einer Größe von 0.8 bis 1.2 erscheinen sollen, wird das *Scaling* auf *Uniform* gesetzt. Dadurch werden alle Achsen gleichmäßig skaliert. Anschließend wird der *Scale X*-Wert auf die gewünschten Min- und Max-Werte von 0.8 bis 1.2 eingestellt, um die gewünschte Variation in der Baumgröße zu erzielen (Abbildung 23).

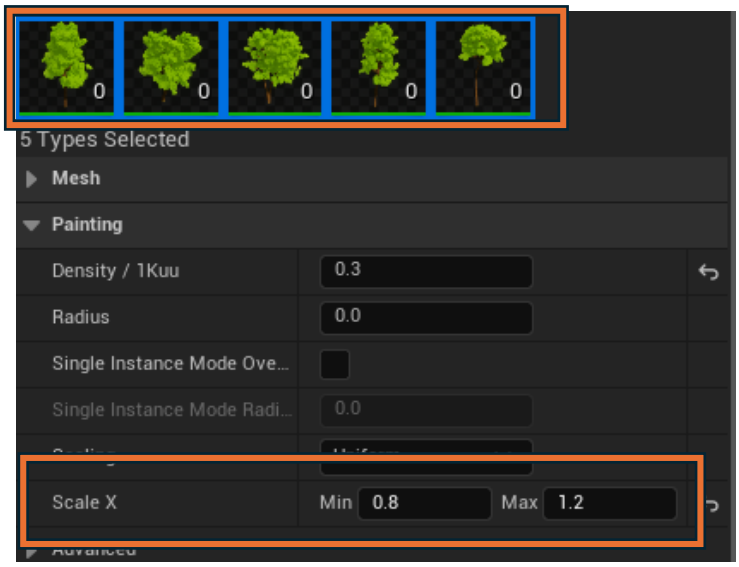


Abb. 23 Scale Einstellungen

6.1.2.3. Align to Normal

Die *Align to Normal*-Einstellung sorgt dafür, dass sich die platzierten Instanzen an der Ausrichtung des Bodens orientieren. Elemente wie Gras oder Blumen sollten beispielsweise der Neigung eines Hangs folgen. Da die Bäume jedoch gerade nach oben wachsen sollen, muss die *Align to Normal*-Einstellung deaktiviert werden (Abbildung 24).

Abb. 24 Align to Normal Einstellung

6.1.2.4. Ground Slope Angle

Die *Ground Slope*-Einstellung legt fest, auf welchen Steigungen Instanzen platziert werden können. Je höher der Maximalwert, desto steiler dürfen die Flächen sein, auf denen Instanzen gespawnt werden. Für die Bäume empfiehlt sich ein Minimalwert von 0, damit sie auf allen flachen Flächen erscheinen, und ein Maximalwert von 35, um steilere Hänge zu vermeiden (Abbildung 25).

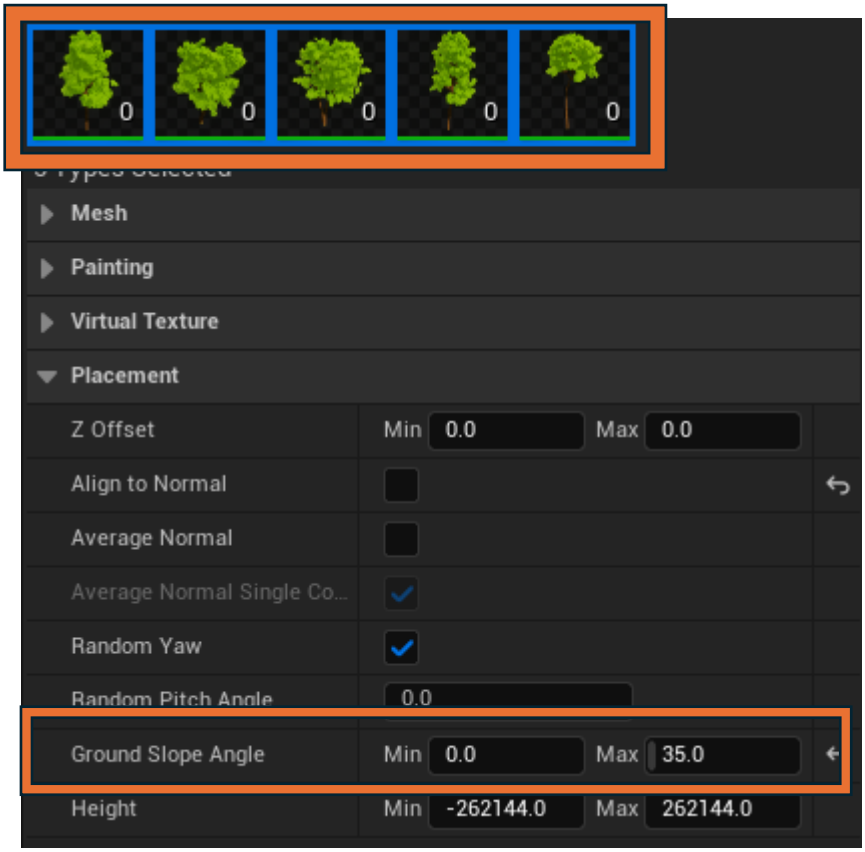


Abb. 25 Ground Slope Angle Einstellungen

6.1.3. Vegetation platzieren

Nachdem alle Einstellungen für die Bäume festgelegt wurden, muss dieser Prozess ebenfalls für alle weiteren *Asset*s wiederholt werden, die für die Landschaftsgenerierung erforderlich sind. Dazu zählen unter anderem Büsche, Äste und Blumen, für die jeweils individuelle Parameter wie beispielsweise die geeignete *Density* definiert werden müssen.

Sobald alle erforderlichen Konfigurationen vorgenommen wurden, können die Bäume im *Foliage Tool* ausgewählt und mithilfe des *Paint Brush* in der Landschaft platziert werden. Dabei ist es wichtig, die Anzahl der platzierten Instanzen im Blick zu behalten. Diese wird als kleine Zahl direkt im *Foliage Tool* angezeigt (Abbildung 26).

Abb. 26 Anzahl der platzierten Instanzen

Sind nahezu alle erforderlichen Bäume jeder Kategorie platziert, kann der *Single Brush* genutzt werden. Hierbei wird ausschließlich das gewünschte Baum-Mesh aktiviert, sodass einzelne Bäume gezielt positioniert werden können. Auf diese Weise lässt sich die gewünschte Gesamtanzahl exakt erreichen, ohne die zuvor festgelegten Verteilungen zu verändern.

Dieser Vorgang wird anschließend für alle weiteren Meshes auf die gleiche Weise wiederholt, bis sämtliche Anforderungen erfüllt sind (Abbildung 27) und das Level fertig gestaltet ist (Abbildung 28).

Abb. 27 Alle plazierten Instanzen

Abb. 28 Darstellung des fertigen Levels

6.2. PCG-Graphen-Levelgestaltung: Leitfaden

Zunächst wird im *Content Browser* ein leerer *PCG-Graph* erstellt und in *PCG_Forest* umbenannt (Abbildung 29, 30). Dieses *PCG-Element* wird anschließend in die Szene gezogen. Die genaue Platzierung ist in diesem Fall nicht relevant.

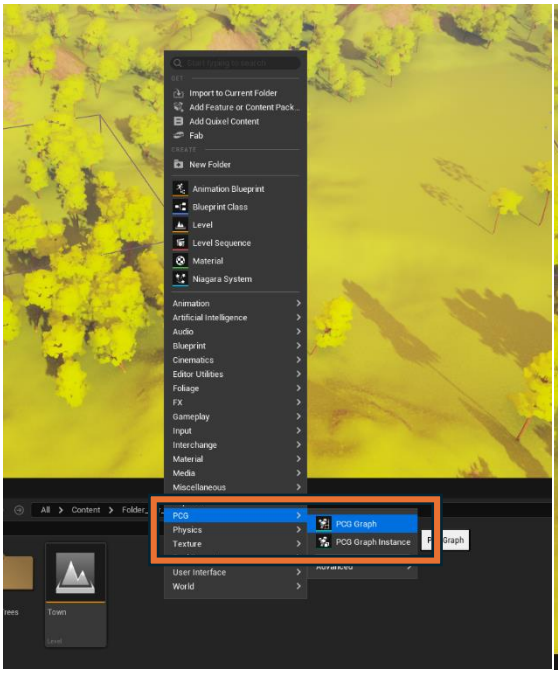


Abb. 30 Content Browser PCG Graph

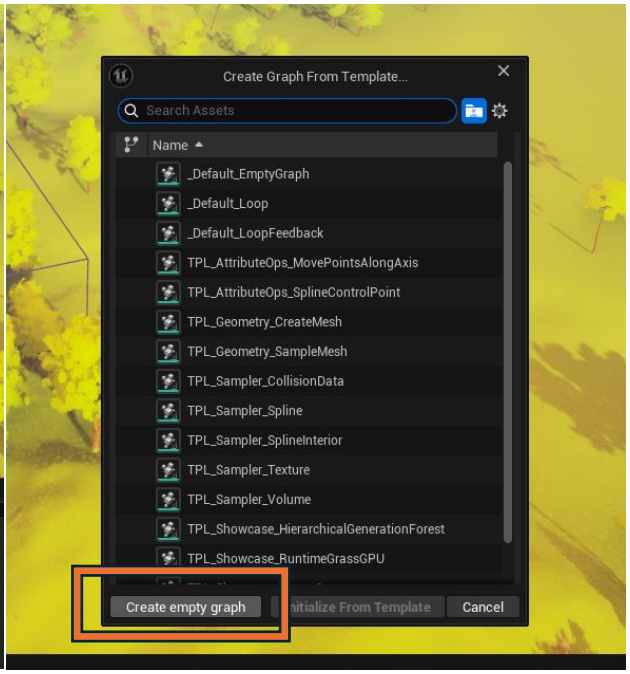


Abb. 29 Kreierung eines leeren Graphens

6.2.1. Landscape Data und Surface Sampler Node

Im nächsten Schritt wird der *PCG-Graph* geöffnet und mit dem Aufbau begonnen. Über einen Rechtsklick im Editor wird die *Node Get Landscape Data* hinzugefügt (Abbildung 31). Diese wird verwendet, um das gesamte *Landscape Terrain* zu samplen.

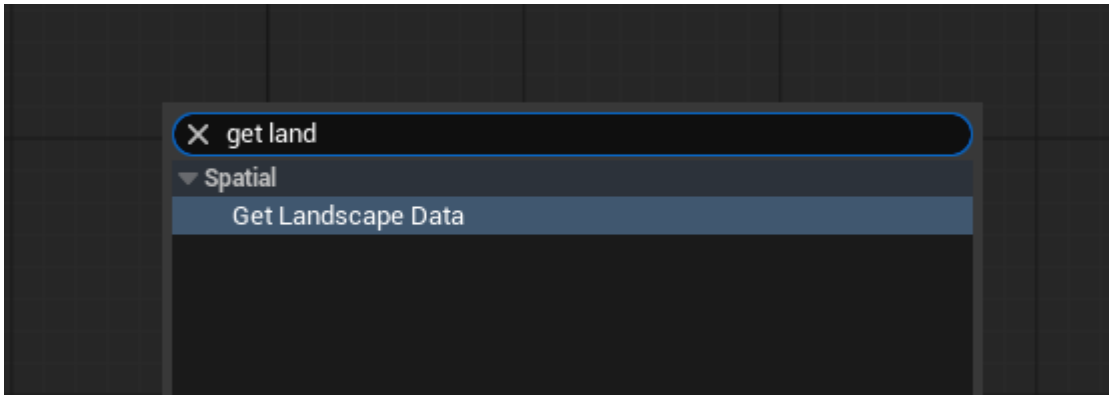


Abb. 31 Get Landscape Data Node

Die *Node Get Landscape Data* wird mit der *Node Surface Sampler* verbunden (Abbildung 32). Durch Drücken der Taste D auf dem *Surface Sampler* wird der *Debugger* aktiviert, wodurch alle erzeugten Punkte angezeigt werden (Abbildung 33). Diese Funktion ist besonders hilfreich bei der Kontrolle und dem weiteren Aufbau des *PCG-Graphen*.

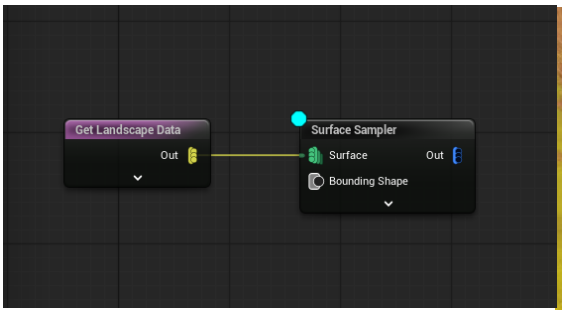


Abb. 32 Surface Sampler Node

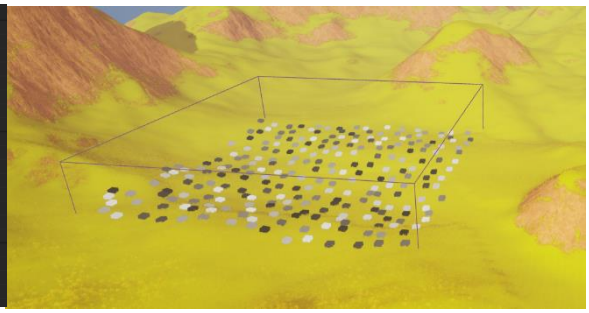


Abb. 33 PCG Debugger

Um nun die gesamte *Landscape* zu samplen, wird im *Details Panel* des *Surface Samplers* die Option *Unbound* gesucht (Abbildung 34). Durch Aktivieren dieser Option wird die Abhängigkeit von der *PCG Bound Box* aufgehoben, wodurch das gesamte Level gesampelt werden kann.

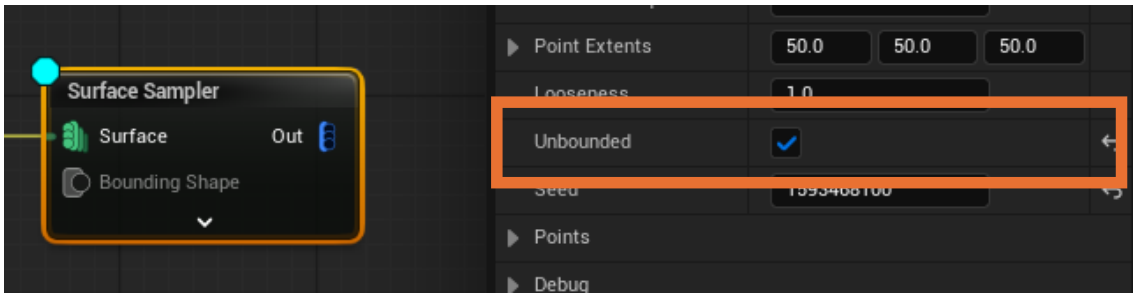


Abb. 34 PCG Unbound Einstellung

Nach erfolgreichem Aktivieren der Option sollte das Level nun entsprechend dargestellt werden (Abbildung 35).

Abb. 35 Debugger mit aktiver Unbound Einstellung

6.2.2. Spatial Noise Node

Es ist zu erkennen, dass jeder Punkt eine Farbe zwischen Schwarz und Weiß besitzt. Dies stellt die *Density* des jeweiligen Punktes dar. Um mehr Ordnung in diese Verteilung zu bringen, kann eine *Spatial Noise Node* verwendet werden (Abbildung 36). Diese weist jedem Punkt eine *Density* auf Basis eines *Noise Patterns* zu und erzeugt dadurch gleichmäßigere und weichere Übergänge (Abbildung 37).

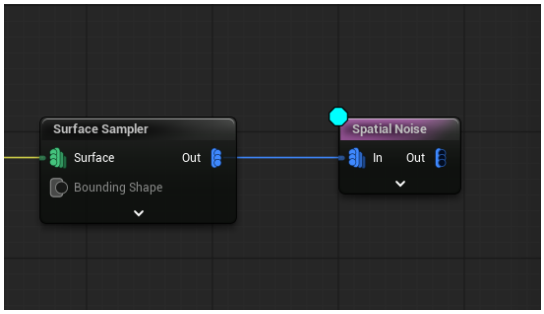


Abb. 36 Spatial Noise Node

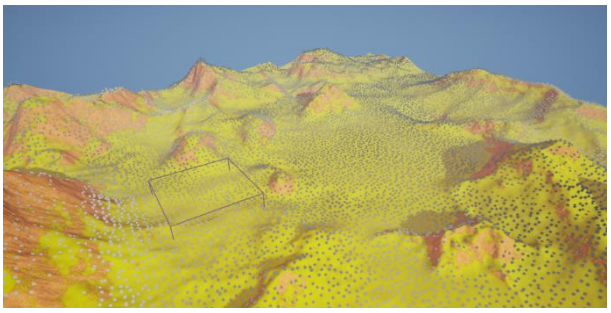


Abb. 37 Spatial Noise in Level

6.2.3. Density Filter Node

Mit einer *Density Filter Node* kann gesteuert werden, welche Punkte basierend auf ihrer *Density* gefiltert werden und welche erhalten bleiben. Da nur die Hälfte der Karte mit Bäumen bedeckt sein soll, können alle Punkte mit einer *Density* von 0.5 oder weniger entfernt werden. Im *Details Panel* der *Density Filter Node* kann hierfür unter *Lower Bound* der Wert 0.5 eingetragen werden (Abbildung 38). Übrig bleibt etwa die Hälfte der Punkte in einem organisch wirkenden Muster (Abbildung 39).

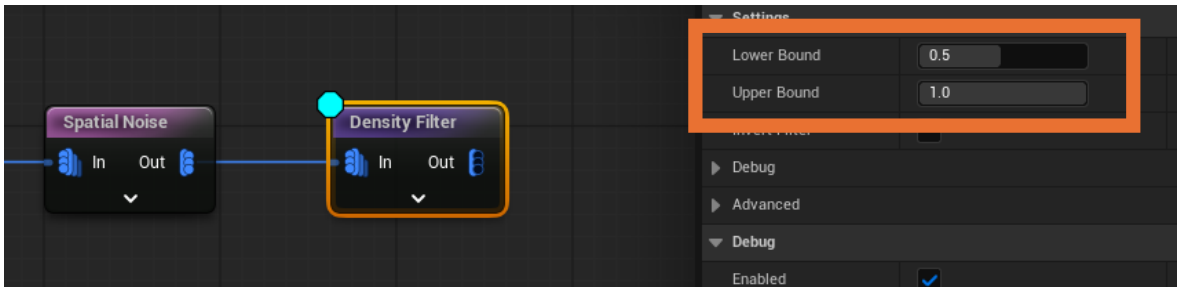


Abb. 38 Density Node Einstellungen

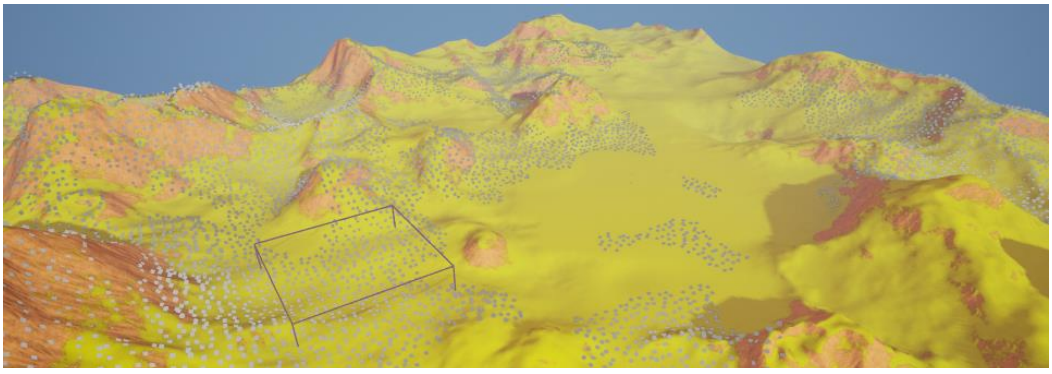


Abb. 39 Filtered Points in Level

6.2.4. Transform Points Node

Da an jedem dieser Punkte ein Baum gespawnt werden soll, sollten die Punkte weniger gleichmäßig und nicht in einem *Grid*-Muster angeordnet sein. Um dieses Muster aufzubrechen, kann jedem Punkt ein zufälliger *Offset* im Bereich von –100 bis 100 zugewiesen werden. Zusätzlich sollen die Bäume eine zufällige *Rotation* zwischen 0 und 360 Grad sowie ein variierendes *Scaling* zwischen 0.8 und 1.2 erhalten (Abbildung 40).

Abb. 40 Transform Node in Level

Zur Umsetzung wird eine *Transform Points Node* verwendet. Im *Details Panel* dieser *Node* können minimale und maximale Werte für *Position Offset*, *Rotation* und *Scale* definiert werden, beispielsweise eine minimale *Rotation* von 0 und eine maximale von 360 Grad (Abbildung 41).

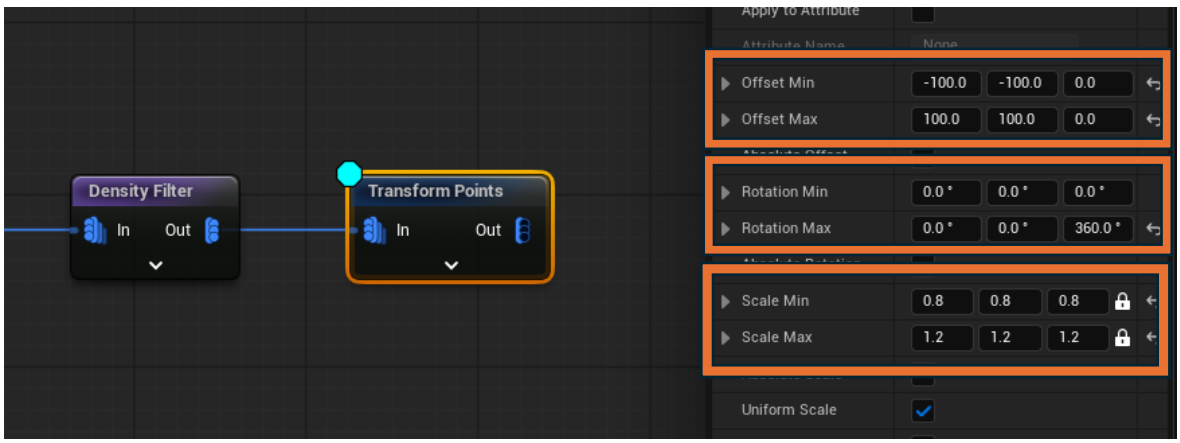


Abb. 41 Transform Points Node Einstellungen

6.2.5. Static Mesh Spawner Node

Damit ist das Baum-Setup nahezu abgeschlossen. Nun muss nur noch an jedem Punkt ein Baum gespawnt werden. Dafür wird eine *Static Mesh Spawner Node* verwendet . Im *Details Panel* dieser *Node* kann unter *Mesh Selector* und *Mesh Entries* eine Auswahl der gewünschten Baum-*Meshes* eingetragen werden (Abbildung 42).

Für jedes unterschiedliche *Mesh* muss ein eigener Eintrag erstellt werden. Dazu wird das Plus-Symbol betätigt, anschließend der Index des jeweiligen *Entries* geöffnet und im *Descriptor* unter *Static Mesh* das gewünschte Baum-*Mesh* hinzugefügt. Dieser Vorgang muss für jedes *Mesh Entry* wiederholt werden, bis alle gewünschten *Meshes* integriert sind.

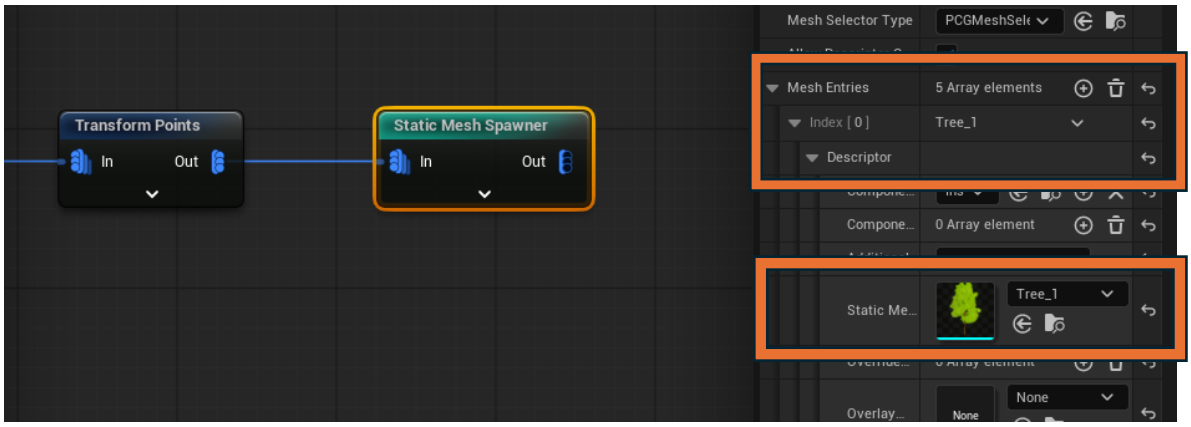


Abb. 42 Static Mesh Spawner Node Einstellungen

6.2.6. Random Choice Node

Da bisher keine Kontrolle über die Gesamtanzahl der Bäume vorhanden ist, wird eine *Random Choice Node* zwischen der *Transform Points Node* und der *Static Mesh Spawner Node* eingefügt (Abbildung 43). Bereits bestehende Verbindungen zwischen den *Nodes* können mit der Tastenkombination *ALT + Linksklick* entfernt werden.

Mit dieser *Node* kann im *Details Panel* unter *Fixed Number* festgelegt werden, wie viele der vorhandenen Punkte beibehalten werden sollen. In diesem Fall wird der Wert auf 1000 gesetzt, sodass 1000 zufällig ausgewählte Punkte für das spawnen von Bäumen verwendet werden (Abbildung 43).

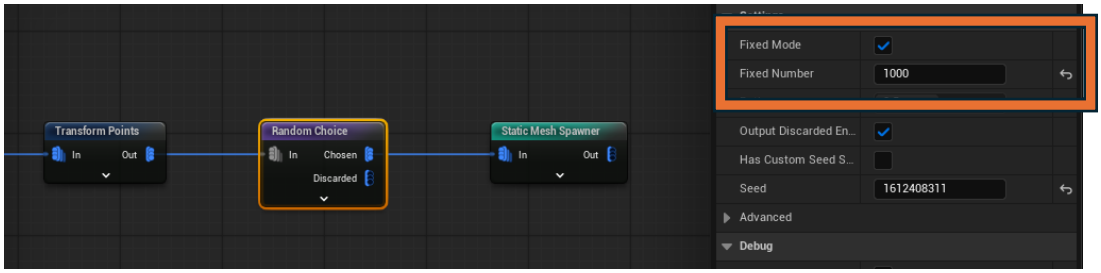


Abb. 43 Random Choice Node mit Einstellungen

6.2.7. Graph Parameters

Um später mehr Kontrolle über den *PCG Graph* zu erhalten, empfiehlt es sich, für die Anzahl der Bäume einen Parameter zu erstellen. Dazu wird im *Graph Parameters Panel* über das Plus-Symbol ein neuer Parameter angelegt. Durch Klicken auf den Pfeil neben dem neu erstellten Parameter kann dieser umbenannt werden, zum Beispiel in *Tree Amount.* Zusätzlich kann im selben Menü kann der *Value Type* von *Float* auf *Integer* geändert und der *Value* von 0 auf den gewünschten Wert, beispielsweise 1000, gesetzt werden (Abbildung 44).

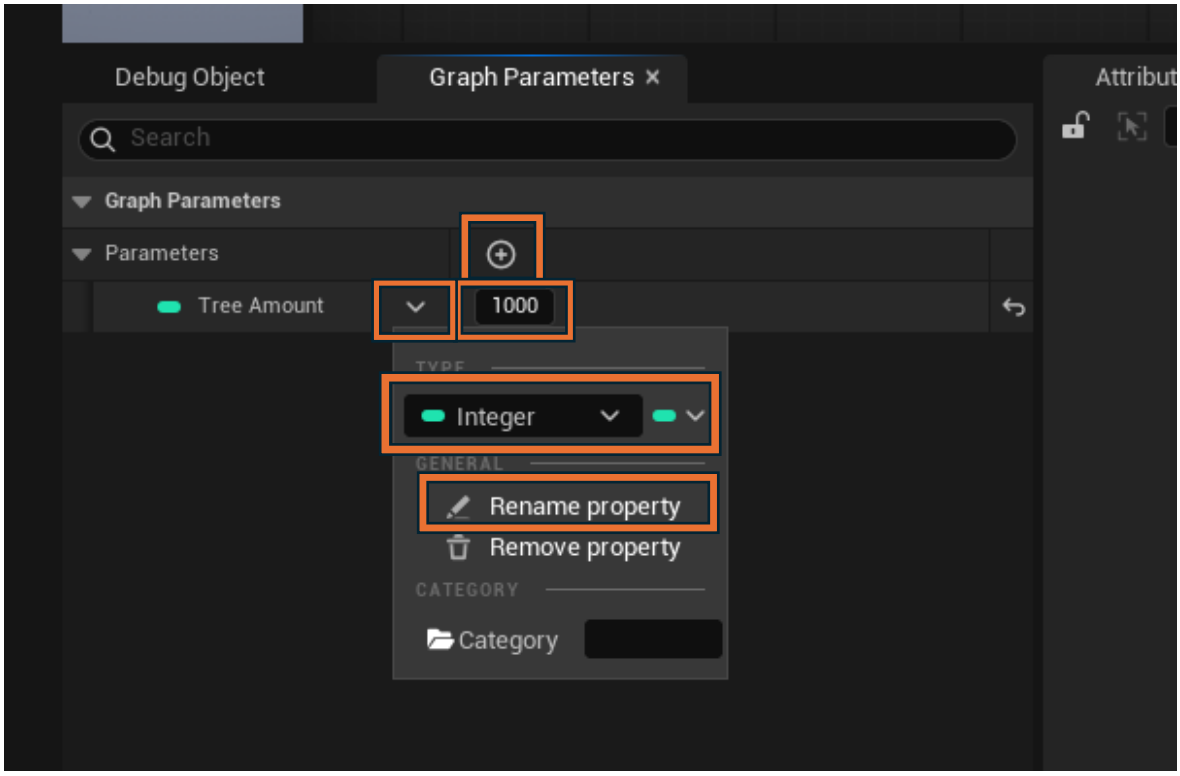


Abb. 44 Graph Parameter Einstellungen

Anschließend wird die zuvor erstellte *Random Choice Node* aufgeklappt. Vom *Fixed Number* Attribut wird eine Verbindung gezogen und nach dem *Tree Amount* Parameter gesucht (Abbildung 45). Nachdem die Verbindung hergestellt wurde, kann die *Random Choice Node* wieder eingeklappt werden, um den *PCG Graph* übersichtlich zu halten.

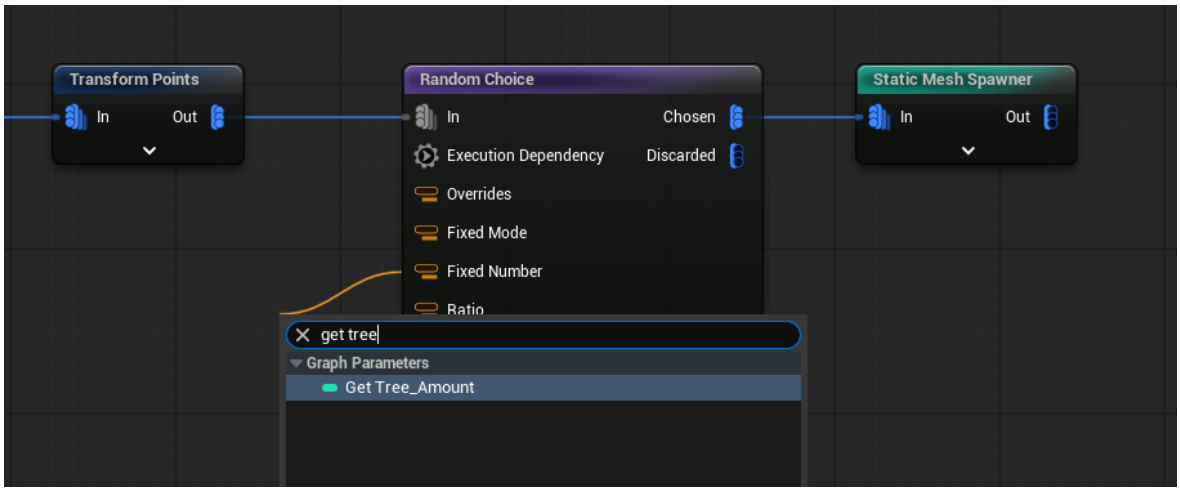


Abb. 45 Random Choice Node mit Parameter

6.2.8. Normal to Density Node

Ein Problem, das bisher noch besteht, ist, dass Punkte auch auf steilen Klippen gespawnt werden. Um dieses Verhalten zu vermeiden, kann die *Normal to Density Node* verwendet werden.

Diese *Node* weist jedem Punkt eine *Density* zu, abhängig von der Ausrichtung der Fläche, auf der er liegt. Flächen, die flach und nach oben gerichtet sind, erhalten dabei einen hohen *Density*-Wert und werden hell (weiß) eingefärbt, während steile Flächen einen niedrigeren Wert erhalten und entsprechend dunkler dargestellt werden (Abbildung 46).

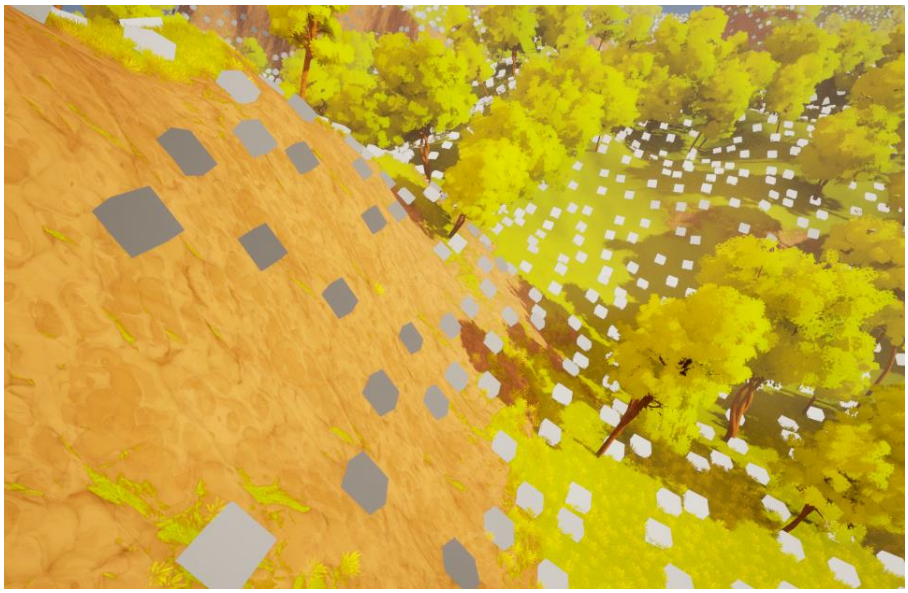


Abb. 46 Normal to Density Node in Level

Die *Normal to Density Node* wird zwischen die *Surface Sampler Node* und die *Spatial Noise Node* eingefügt, um die Punktmenge bereits frühzeitig basierend auf der Flächenausrichtung zu reduzieren. Anschließend kann, wie bereits zuvor beschrieben, eine *Density Filter Node* verwendet werden, um alle dunklen Punkte an steilen Klippen herauszufiltern. In diesem Fall bietet sich ein *Lower Bound* Wert von 0.8 an, um ausschließlich Punkte auf ausreichend flachen Flächen zu behalten (Abbildung 47). Dadurch werden alle Punkte die an Klippen liegen entfernt (Abbildung 48).

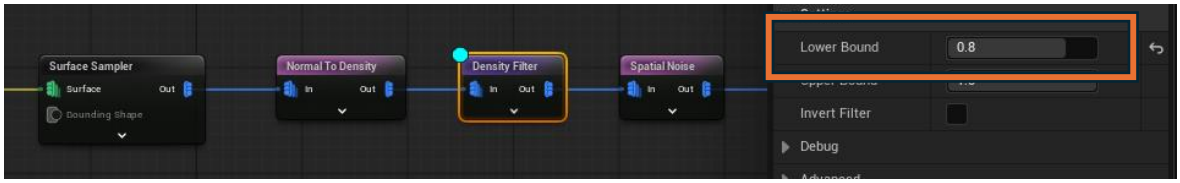


Abb. 47 Normal to Density Node Einstellungen

Abb. 48 Normal to Density Node in Level

6.2.9. Projection Node

Aktuell richtet sich das Wachstum der Bäume noch nach der Ausrichtung der jeweiligen Oberfläche. Um dieses Verhalten zu korrigieren, kann eine *Projection Node* verwendet werden. Diese *Node* projiziert alle Punkte auf eine definierte Oberfläche. Diese Oberfläche kann ein *Mesh* sein oder, wie in diesem Fall, die *Landscape* selbst.

Die *Projection Node* wird nach dem ersten *Density Filter* in den Graph eingefügt. Als *Projection Target* dient eine *Get Landscape Data Node*, die mit der *Projection Node* verbunden wird.

Zunächst erscheint das Ergebnis unverändert. Um jedoch die gewünschte Ausrichtung zu erreichen, wird im *Details Panel* der *Get Landscape Data Node* die Option *Height Only* aktiviert (Abbildung 49). Dadurch werden nur noch die Positionen der Punkte auf der *Landscape* übernommen, unabhängig von deren Normalenrichtung. Das Ergebnis ist, dass alle Punkte für die Bäume senkrecht nach oben wachsen, unabhängig von der Geländeausrichtung (Abbildung 50).

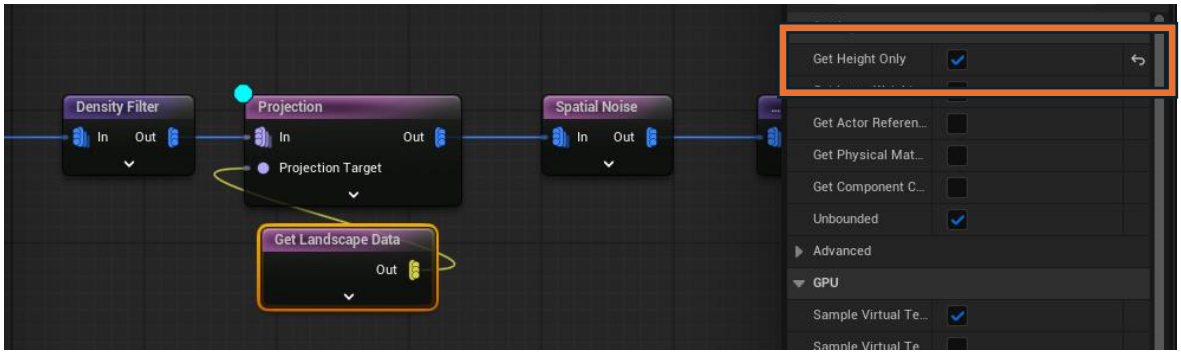


Abb. 50 Get Landscape Data Node mit Height Only Einstellung

Abb. 49 Get Landscape Data Node mit Height only in Level

6.2.10. PCG Subgraphen

Die Funktion zur Entfernung von Punkten an steilen Klippen wird im weiteren Verlauf des Projekts mehrfach benötigt. Um den entsprechenden Abschnitt nicht bei jeder Verwendung neu aufbauen zu müssen, empfiehlt sich die Erstellung eines *Subgraphs*. *Subgraphs* sind wiederverwendbare Mini-Graphen, die in ihrer Funktionsweise mit den bisherigen *Nodes* vergleichbar sind.

Zur Erstellung eines *Subgraphs* werden alle relevanten *Nodes* ausgewählt, in diesem Fall die *Normal to Density*, *Density Filter* und *Projection Node*, die für das Entfernen der Punkte auf Klippen verantwortlich sind. Die *Get Landscape Data Node* wird dabei bewusst nicht mit einbezogen, um bei der späteren Verwendung des *Subgraphs* flexibel entscheiden zu können, ob lediglich die *Landscape Height* oder zusätzlich auch die *Rotation* berücksichtigt werden soll.

Nach Auswahl der gewünschten *Nodes* kann über einen Rechtsklick die Funktion *Collapse into Subgraph* aufgerufen oder alternativ der Shortcut *ALT + J* verwendet werden (Abbildung 51). Anschließend wird ein Speicherort sowie ein passender Name, beispielsweise *PCG_Subgraph_Remove_Cliffs*, vergeben und der *Subgraph* gespeichert.

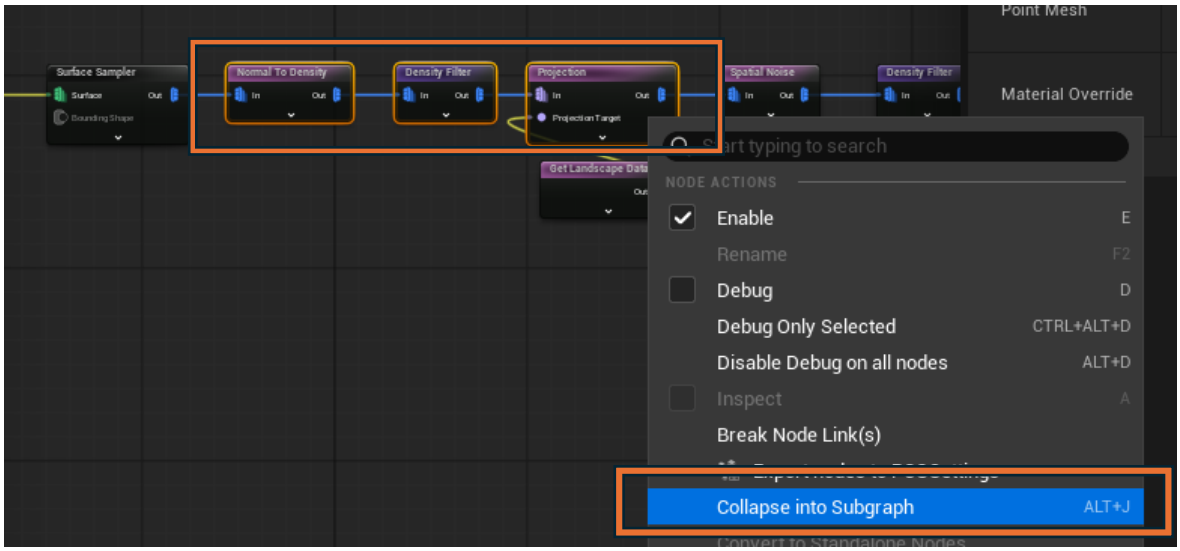


Abb. 51 PCG Subgraphen Erstellung

Im Anschluss erscheinen die zuvor ausgewählten *Nodes* zusammengefasst in einer einzigen neuen *Node* (Abbildung 52), die beliebig oft wiederverwendet werden kann. Durch einen Doppelklick auf den *Subgraph* im *Content Browser* oder direkt auf die *Node* kann dieser geöffnet und bei Bedarf bearbeitet werden. Der *PCG_Subgraph_Remove_Cliffs* verfügt nun zusätzlich über einen *Input* für die *Get Landscape Data Node*.

Subgraphen können aus dem *Content Browser* an der Stelle, an der sie gespeichert wurden, erneut in den *PCG-Graphen* eingefügt werden, um sie wiederzuverwenden.

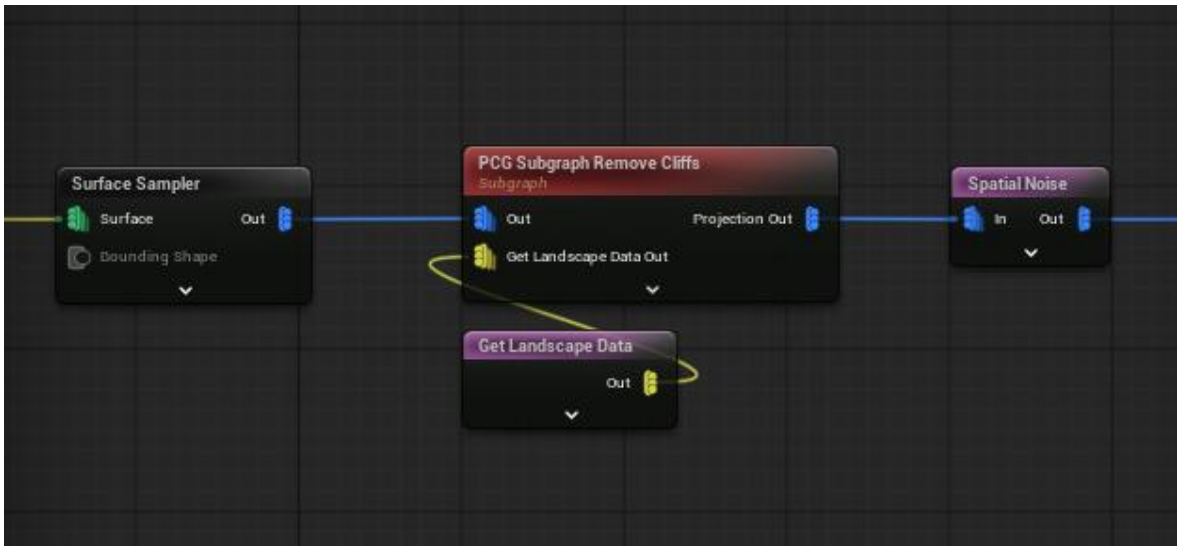


Abb. 52 PCG Subgraph in einer Node

6.2.11. Create Points Grid und Copy Points Node

Zur Platzierung von Ästen unter den Bäumen wird zunächst eine *Create Points Grid Node* verwendet. Diese *Node* erzeugt ein regelmäßiges Raster aus Punkten, standardmäßig an der *World Location* (0, 0, 0) (Abbildung 53). Die Anzahl der Punkte innerhalb dieses Rasters kann im *Details Panel* angepasst werden.

Abb. 53 Create Points Grid in Level

Dieses Raster wird anschließend mithilfe einer *Copy Points Node* an jedem Baum repliziert. Als *Source* dient dabei die *Create Points Grid Node*, da deren Punkte kopiert werden sollen. Als *Target* wird die *Random Choice Node* vor dem *Static Mesh Spawner* gewählt, da sie die finalen Positionen der Bäume enthält (Abbildung 54). Auf diese Weise werden die Rasterpunkte an jedem Baumstandort dupliziert und bilden die Grundlage für das Platzieren von Ästen (Abbildung 55).

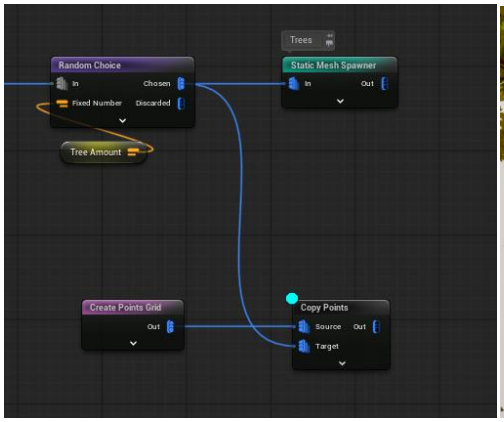


Abb. 54 Copy Points Node

Abb. 55 Grids an jedem Baum im Level

Zum abschließenden Erstellen der Äste wird ein ähnlicher Aufbau wie bei den Bäumen verwendet. Zunächst wird eine *Transform Points Node* eingefügt, um Variation in Position, Rotation und Skalierung der Punkte zu ermöglichen.

Anschließend werden die Punkte mithilfe einer *Projection Node* und einer *Get Landscape Data Node* auf den Boden projiziert. Um diese Punkte ebenfalls von steilen Klippen zu entfernen, kann der bereits erstellte *Subgraph PCG_Subgraph_Remove_Cliffs* genutzt werden. Damit dieser auch die Projektion auf die Oberfläche beinhaltet, können die vorher verwendeten *Projection* und *Get Landscape Data Node*s direkt am Anfang des *Subgraphs* ergänzt werden (Abbildung 56). So lässt sich dieser Schritt in Zukunft vollständig über *PCG_Subgraph_Remove_Cliffs* abbilden, ohne ihn im Hauptgraph erneut aufbauen zu müssen.

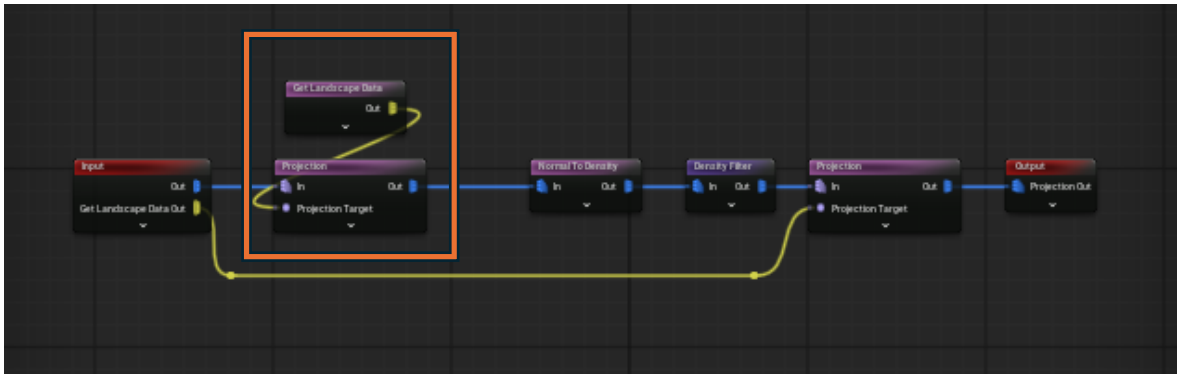


Abb. 56 Projection Node in PCG_Subgraph_Remove_Cliffs

Im Hauptgraph wird nach dem *PCG_Subgraph_Remove_Cliffs* eine *Random Choice Node* eingefügt. Diese wird mit einem neuen Parameter verbunden, der die Gesamtanzahl der zu spawnenden Äste bestimmt. Abschließend erfolgt das Platzieren der Äste durch eine *Static Mesh Spawner Node*. Damit stehen nun Äste zur Verfügung, die ausschließlich unter Bäumen gespawnt werden und über einen Parameter kann gesteuert werden, wie viele Äste insgesamt platziert werden (Abbildung 57).

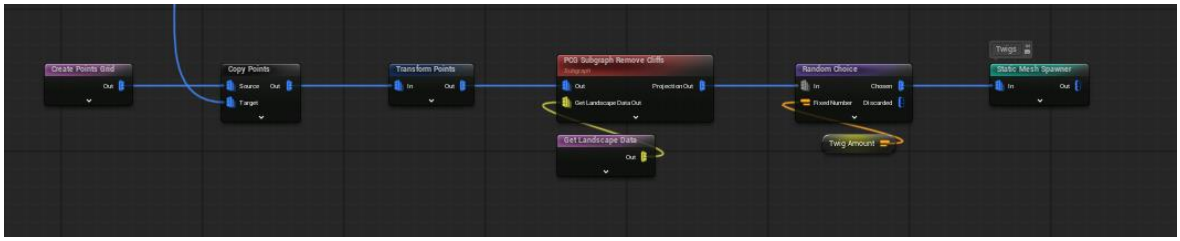


Abb. 57 Finaler Aufbau der Äste

6.2.12. Distance Node

Um Blumenfelder gezielt auf offenen Wiesen und nicht im Wald zu platzieren, muss zunächst ermittelt werden, wo sich Bäume befinden. Zur Umsetzung dieses Schritts wird eine *Distance Node* verwendet. Diese berechnet den Abstand zwischen zwei Punkten.

In diesem Fall sollen alle Punkte auf der *Landscape*, an denen potenziell Blumenfelder platziert werden können, auf ihre Entfernung zu den Baumstandorten überprüft werden. Als *Source* dienen daher alle Punkte auf der *Landscape*, die nicht auf Klippen liegen. Diese Punkte repräsentieren potenzielle Standorte für Blumenfelder. Als *Target* werden die Punkte verwendet, an denen Bäume gespawnt werden (Abbildung 58).

Um die Punkte nach Entfernung zu klassifizieren, wird im *Details Panel* der *Distance Node* die Option *Set Density* aktiviert. Über den Parameter *Maximum Distance* lässt sich nun definieren, ab welchem Abstand die Punkte unterschiedlich eingefärbt werden. In diesem Fall wird ein Wert von 2500 verwendet (Abbildung 58).

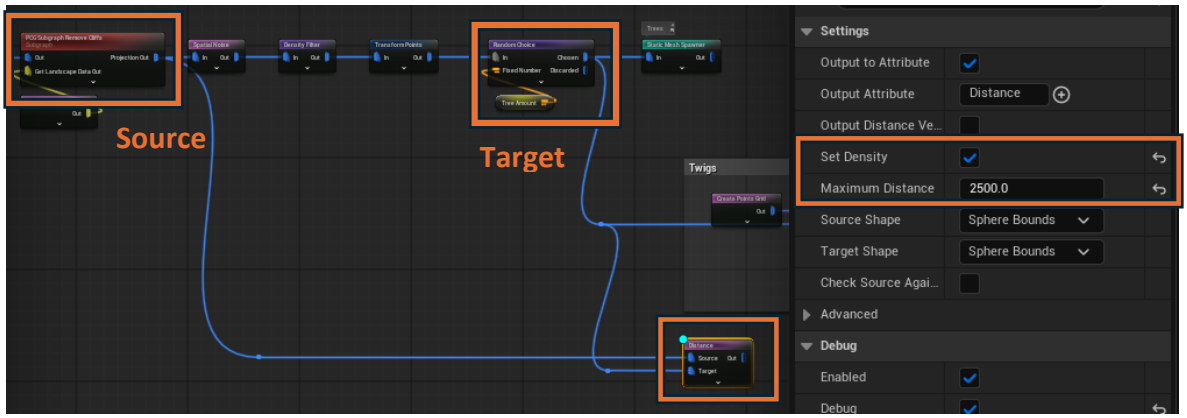


Abb. 58 Distance Node Einstellungen

Als Ergebnis sind Punkte in unmittelbarer Nähe zu Bäumen dunkel eingefärbt, während weiter entfernte Punkte heller erscheinen (Abbildung 59).

Abb. 59 Distance Node in Level

6.2.13. Blumenfelder

Nachdem nun festgelegt wurde, an welchen Stellen Blumenfelder entstehen können, werden im nächsten Schritt alle Punkte in der Nähe von Bäumen mithilfe eines *Density Filter* entfernt. So bleiben nur jene Punkte übrig, die ausreichend weit von Waldflächen entfernt liegen (Abbildung 60).

Abb. 60 Punkte in Level nachdem Distance Filter Node eingebaut wurde

Anschließend wird eine *Random Choice Node* eingefügt und mit einem neuen Parameter verbunden, der die gewünschte Anzahl an Blumenfeldern definiert.

Für jedes dieser Blumenfelder wird ein *Grid* durch eine *Create Points Grid Node* erzeugt. Dieses Raster wird mithilfe einer *Copy Points Node* an alle Punkte kopiert, an denen zuvor Blumenfelder bestimmt wurden.

Um Variation innerhalb der Felder zu erzielen, wird eine *Transform Points Node* hinzugefügt und der Offset, Rotation und Scale Wert angepasst. Anschließend sorgt der bestehende *Subgraph PCG_Subgraph_Remove_Cliffs* dafür, dass keine Blumen auf steilen Klippen platziert werden.

Im Anschluss wird eine weitere *Random Choice Node* eingefügt, die über einen zusätzlichen Parameter die Gesamtanzahl der tatsächlich zu spawnenden Blumen steuert. Dabei ist darauf zu achten, dass im vorherigen *Grid* ausreichend viele Punkte erzeugt wurden. Sollte dies nicht der Fall sein, kann im *Details Panel* der *Create Points Grid Node* die *Cell Size* angepasst werden (Abbildung 61). Eine kleinere *Cell Size* führt zu einer höheren Punktdichte innerhalb des Rasters .

Gleichzeitig sollte vermieden werden, unnötig viele Punkte zu erzeugen, da dies die Performance negativ beeinflussen und zu Verzögerungen oder Rucklern im Editor führen kann. Ein ausgewogenes Verhältnis zwischen Punktdichte und Performance ist daher entscheidend.

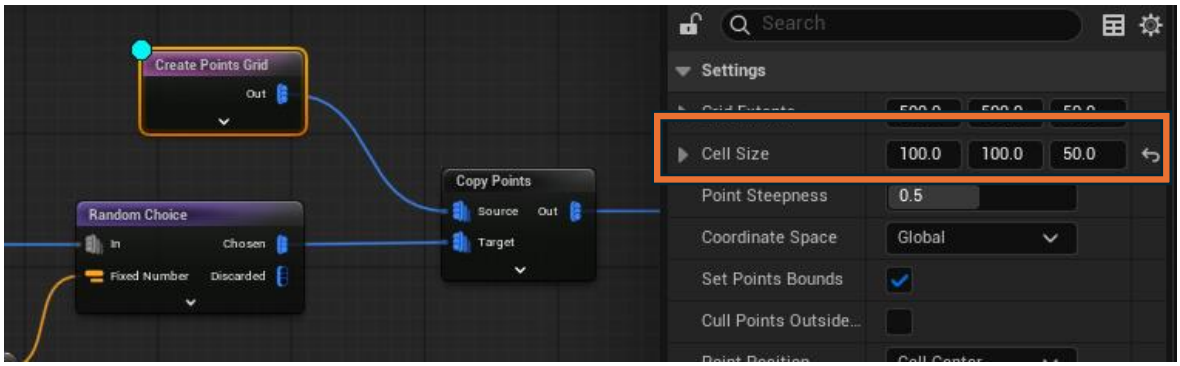


Abb. 61 Cell Size in der Create Points Grid Node

Zur Überprüfung, ob eine ausreichende Anzahl an Punkten generiert wurde, kann die *Random Choice Node* ausgewählt werden. Durch das Drücken der Taste *A* oder über einen Rechtsklick und die Auswahl der Option *Inspect* werden im unteren Bereich des *Attribute Panels* alle generierten Punkte angezeigt. Anschließend muss der *Output* auf *Discarded* umgestellt werden, um zu prüfen, ob Punkte verworfen werden (Abbildung 62). Falls dies der Fall sein sollte, bedeutet dies, dass insgesamt genügend Punkte generiert wurden.

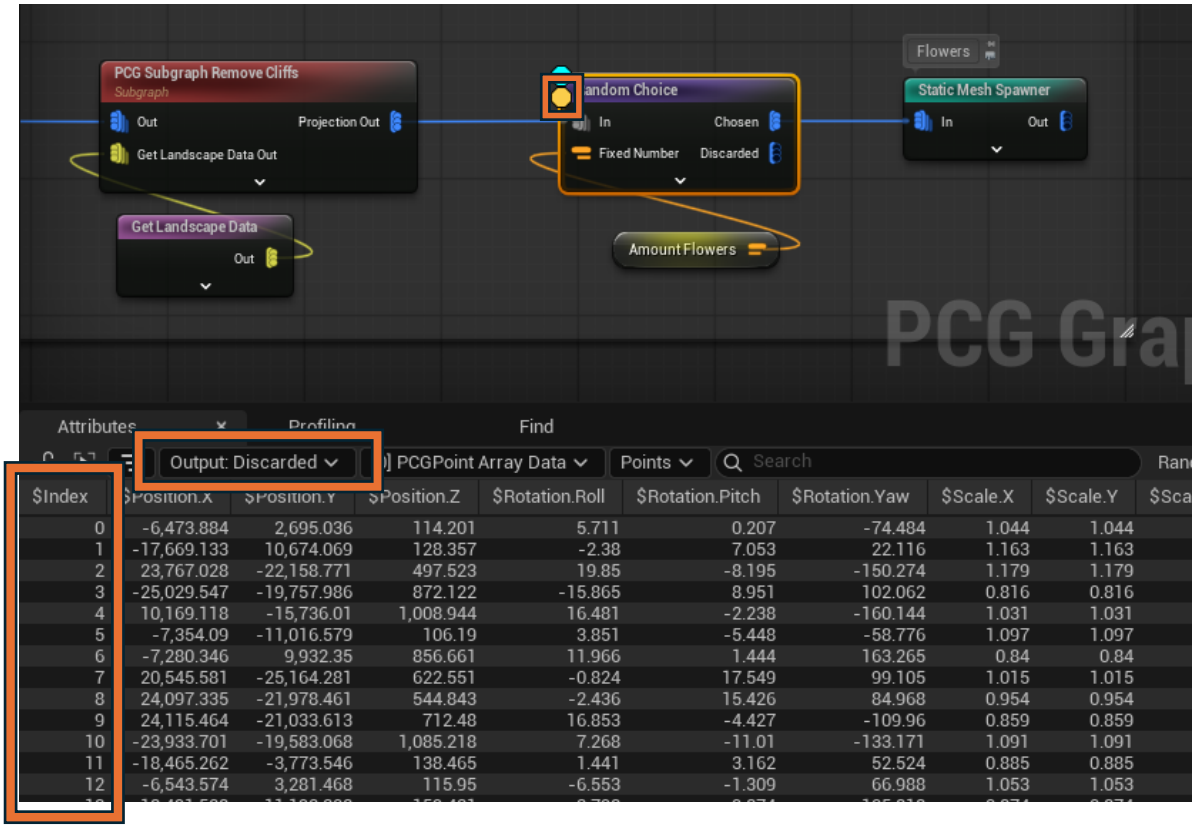


Abb. 62 Inspect Tool

Abschließend werden die Blumen über eine *Static Mesh Spawner Node* am *Chosen*-Ausgang platziert (Abbildung 63). Der finale Aufbau platziert dann Blumenfelder im Level (Abbildung 64).

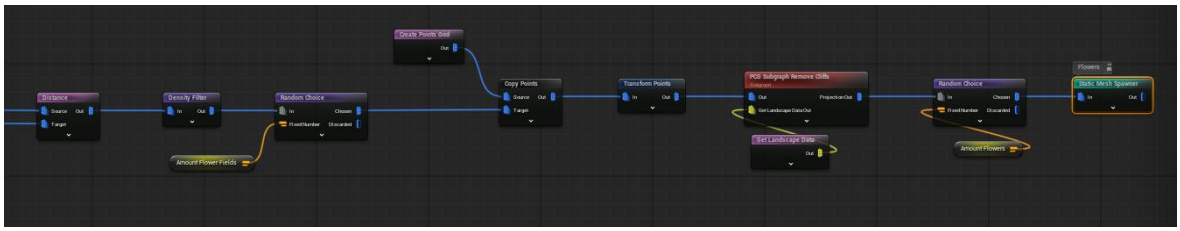


Abb. 63 Finaler Aufbau der Blumenfelder

Abb. 64 Blumenfelder im Level

6.2.14. Büsche und Pflanzen

Für das Platzieren von Büschen und Pflanzen im Wald wird zunächst festgestellt, welche Punkte sich innerhalb des Waldbereichs befinden. Aus dem bestehenden *Distance Filter* Output wird hierfür eine weitere *Density Filter Node* erstellt. Anstatt schwarze Punkte zu entfernen, wie bei den Blumenfeldern, werden diesmal die weißen Punkte gefiltert, indem der *Lower Bound* auf 0 und der *Upper Bound* auf 0.2 gesetzt wird. Anschließend wird mit einer *Random Choice Node* und einem zugehörigen Parameter die gewünschte Anzahl an Punkten für das spawnen ausgewählt. Eine *Transform Points Node* sorgt für Variation in Position, Rotation und Skalierung, bevor die Büsche mit einer *Static Mesh Spawner Node* in der Szene platziert werden (Abbildung 65, 66).

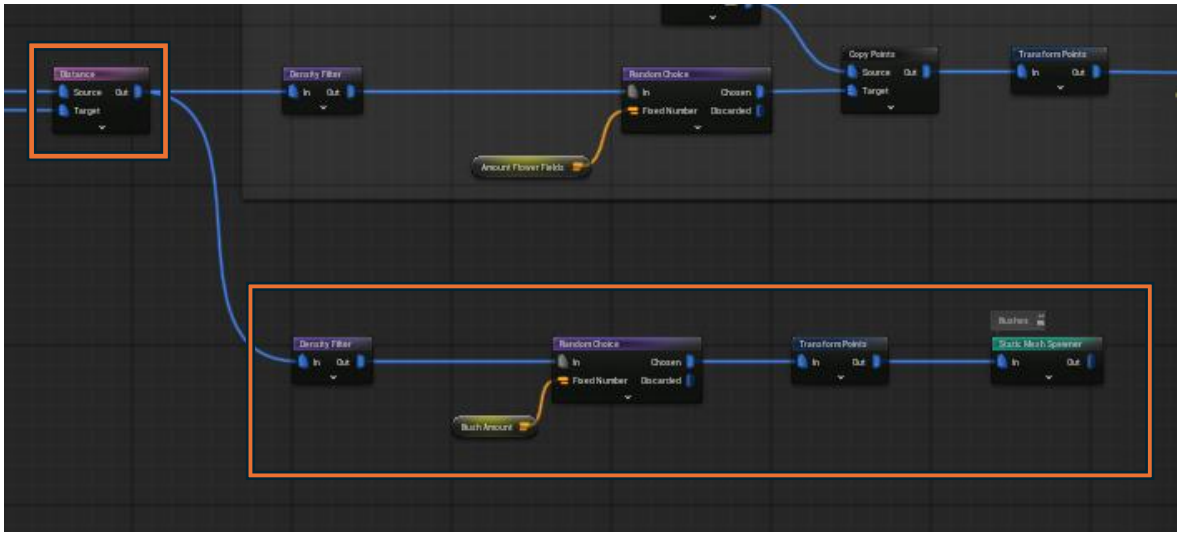


Abb. 65 Spawnen der Büsche im PCG Graph

Abb. 66 Büsche im Level

7. Anpassungsfähigkeit

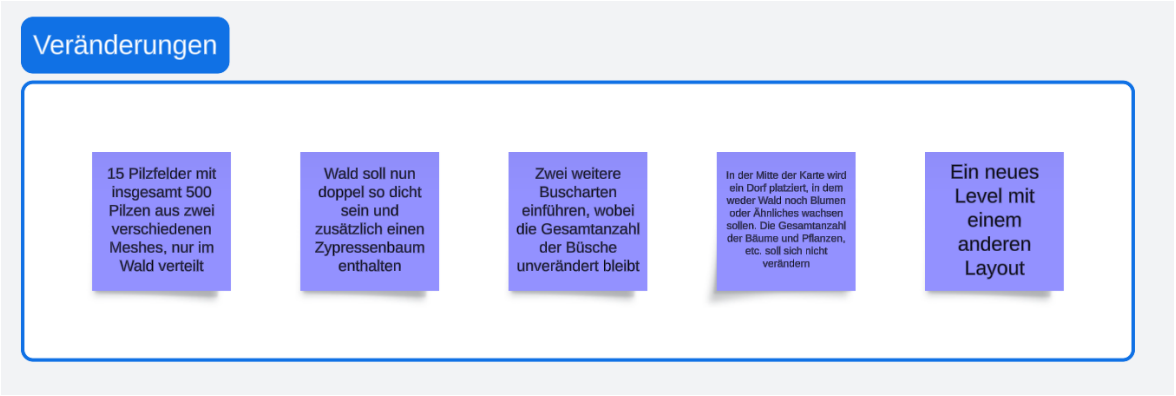


Abb. 67 Screenshot der Veränderungen für jedes Level aus den Notizen

Die nächste Aufgabe bestand darin, Anpassungen am bestehenden Level vorzunehmen, um zu testen, wie vielseitig beide Systeme einsetzbar sind. Zunächst wurde eine neue Ressource für den Spieler integriert. Dabei handelt es sich um Pilze, die ausschließlich innerhalb des Waldes wachsen. Diese Pilze sollen in insgesamt 15 Feldern verteilt sein und eine Gesamtzahl von 500 Instanzen umfassen, wobei zwei unterschiedliche Pilzarten verwendet werden.

Im nächsten Schritt wurde die Dichte des Waldes erhöht, indem die Anzahl der platzierten Bäume verdoppelt wurde. Zusätzlich wurde eine neue Baumart hinzugefügt, der Zypressenbaum, um die Baumvielfalt innerhalb des Levels zu erweitern.

Um die Vegetation abwechslungsreicher zu gestalten, wurden den bestehenden Busch-Meshes zwei weitere Varianten hinzugefügt, ein Farn und ein Sprössling. Dabei wurde darauf geachtet, dass die Gesamtanzahl der maximal platzierten Büsche nicht überschritten wird.

In die Mitte des Levels wurde außerdem ein vorgefertigtes Dorf platziert, in dem weder Bäume noch andere Vegetation wachsen, wobei die Gesamtanzahl der Vegetation im Level unverändert blieb.

Abschließend wurde ein komplett neues Level mit einem anderen Layout erstellt. Jede dieser Änderungen wurde einzeln umgesetzt und die benötigte Zeit individuell erfasst.

8. Veränderungen: Leitfaden

8.1. PCG – Graphen

8.1.1. Pilze

Um Pilze in das bestehende System zu integrieren, muss zunächst überprüft werden, an welchen Stellen im Wald Bäume wachsen. Da bereits Büsche an denselben Positionen platziert werden, kann direkt auf die bestehende *Density Node* der Büsche zurückgegriffen werden. Die Vorgehensweise zum Erstellen von Feldern ist bereits aus dem Prozess der Blumenfelder bekannt, sodass dieser Schritt nicht erneut von Grund auf umgesetzt werden muss. Es müssen lediglich neue Variablen angelegt und die entsprechenden Meshes durch Pilz-Meshes ersetzt werden (Abbildung 68).

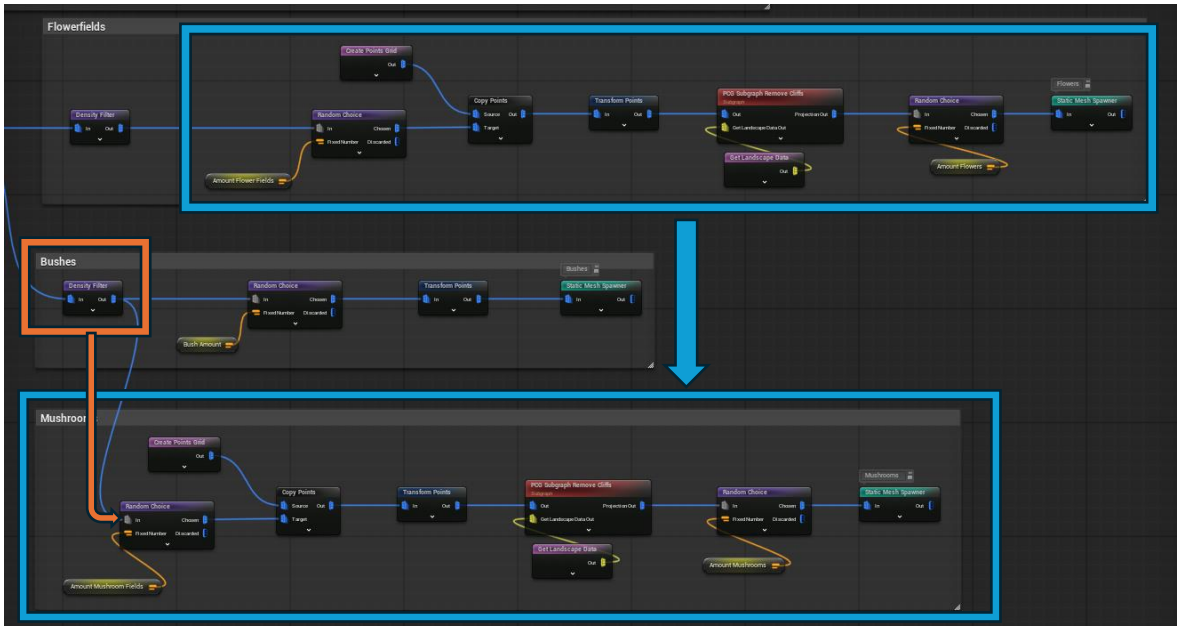


Abb. 68 Spawnen der Pilzfelder im PCG Graph

8.1.2. Bäume

Um eine weitere Baumart wie den Zypressenbaum in das bestehende System einzufügen, wird im *PCG-Graphen* die *Static Mesh Spawner Node* geöffnet, die für das Platzieren der Baum-Meshes zuständig ist. Dort wird das vorgegebene Zypressenbaum-Mesh hinzugefügt, sodass es gemeinsam mit den bestehenden Baumarten gespawnt wird (Abbildung 69).

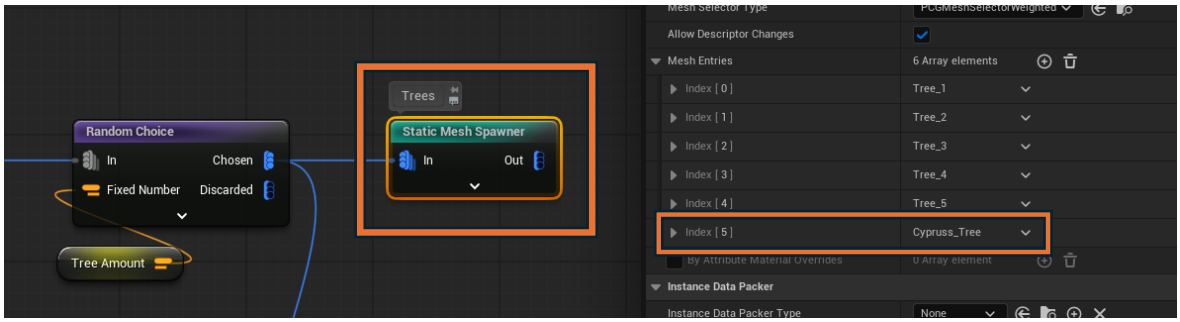


Abb. 69 Zypressenbaum in Static Mesh Spawner

Um die Dichte des Waldes zu erhöhen, wird der Wert der *Tree Amount*-Variable angepasst, indem er auf das Doppelte des ursprünglichen Wertes gesetzt wird (Abbildung 70).

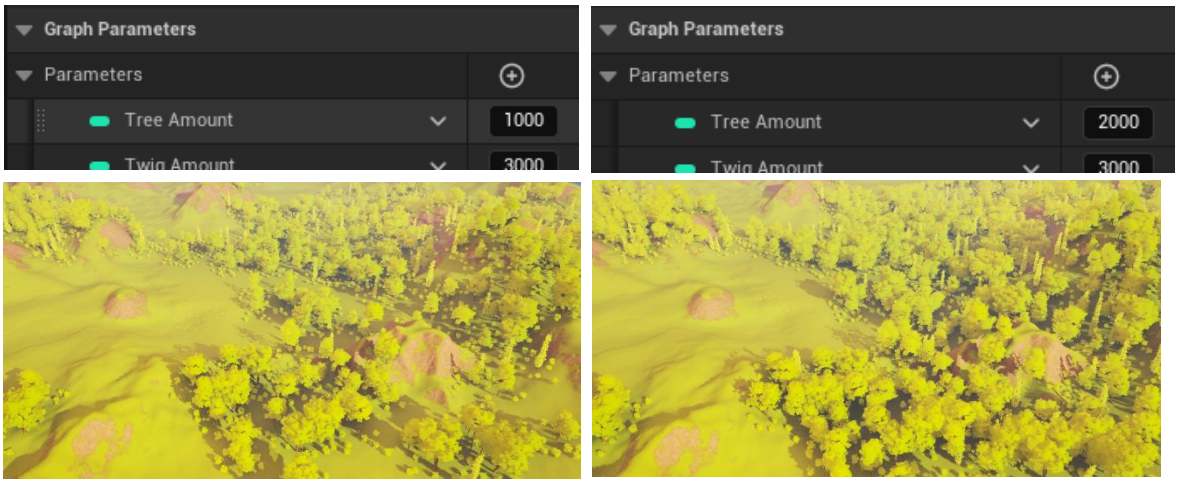


Abb. 70 Vergleich der Tree Amount Parameter in Leveln

8.1.3. Büsche

Bei den Büschen erfolgt die Anpassung auf die gleiche Weise wie bei den Bäumen. Dazu wird die entsprechende *Static Mesh Spawner Node* der Büsche im *PCG-Graphen* geöffnet, und die zusätzlichen Meshes für Farn und Sprössling werden hinzugefügt (Abbildung 71). Dadurch werden diese Varianten automatisch in den bestehenden Busch-Spawning-Prozess integriert (Abbildung 72).

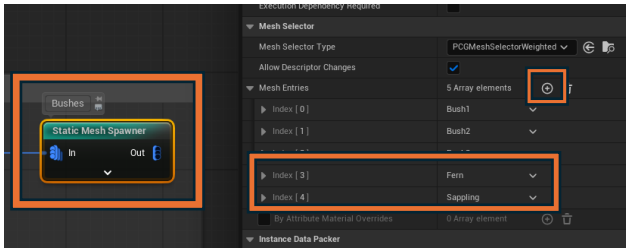


Abb. 71 Busch Varianten in PCG Graph

Abb. 72 Busch Varianten in Level

8.1.4. Dorf

Um das Dorf zu platzieren, wird zunächst das *Town* Level im *Content Browser* aufgesucht und in das bestehende Level gezogen (Abbildung 73). Anschließend wird die Position des *Asset*s auf die Koordinaten 0, 0, 0 gesetzt (Abbildung 74).

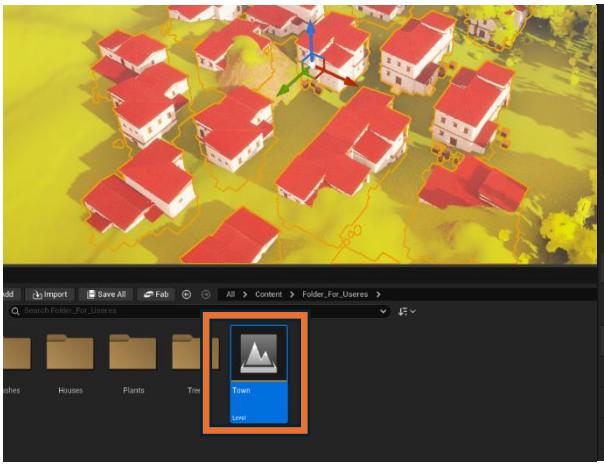


Abb. 73 Town Level *Asset im Content Browser*

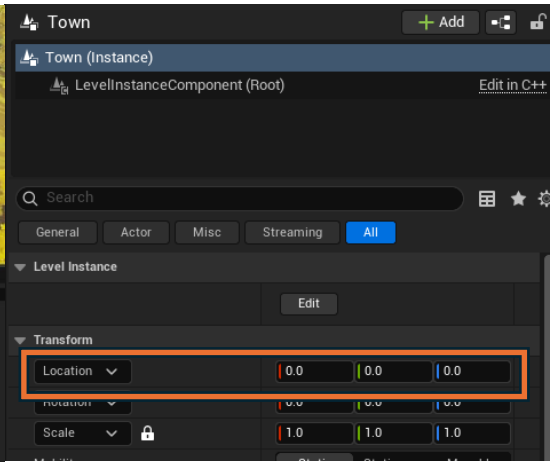


Abb. 74 Dorf Location auf 0, 0, 0 Position

8.1.4.1. Spline erstellen

Um die Vegetation innerhalb des Dorfbereichs zu entfernen, wird eine sogenannte *Spline* verwendet. Zunächst wird im *Content Browser* eine neue *Blueprint Class* erstellt (Abbildung 75) und mit dem Namen *BP_No_Forest_Spline* versehen (Abbildung 77). Hierfür wird im *Content Browser* mit der rechten Maustaste in einen freien Bereich geklickt, anschließend die Option *Blueprint Class* ausgewählt und als Parent Class *Actor* festgelegt (Abbildung 76). Diese *Blueprint* dient später dazu, eine *Spline* zu erstellen, die die Vegetation in einem definierten Bereich ausschließt.

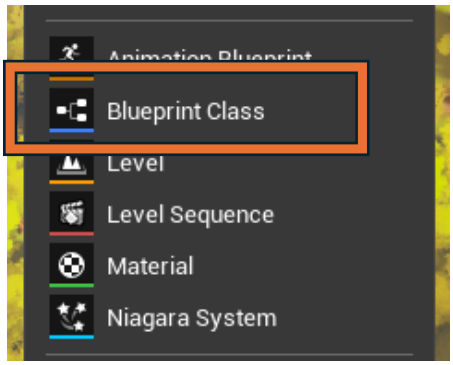


Abb. 75 Blueprint Class

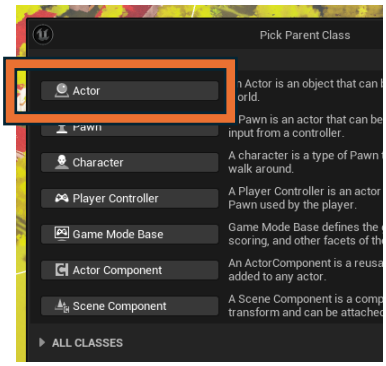


Abb. 76 Parent Blueprint Class

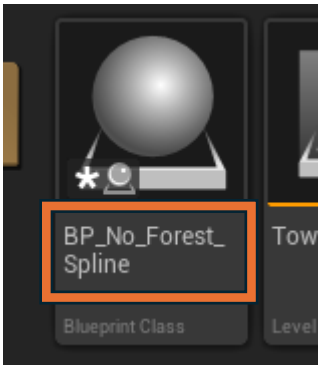


Abb. 77 Name der Spline Blueprint Class

Als nächstes wird die erstellte *Blueprint Class* geöffnet und über das *Components Tab* eine *Spline* hinzugefügt (Abbildung 78). In den Einstellungen der *Spline* im *Details Panel* wird die Option *Closed Loop* aktiviert (Abbildung 79).

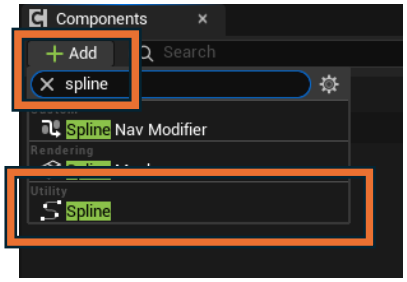


Abb. 78 Spline in Components Tab

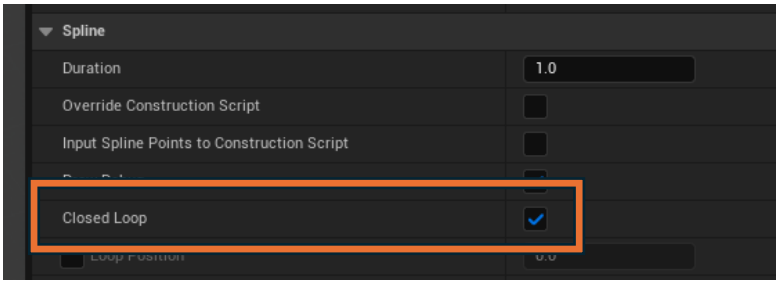


Abb. 79 Closed Loop Einstellung in Spline

Die einzelnen Punkte der *Spline* können nun verschoben werden, und durch das Drücken der *ALT*-Taste lassen sich neue Punkte hinzufügen. Es empfiehlt sich, die *Spline* in Form eines Kreises zu erstellen (Abbildung 80).

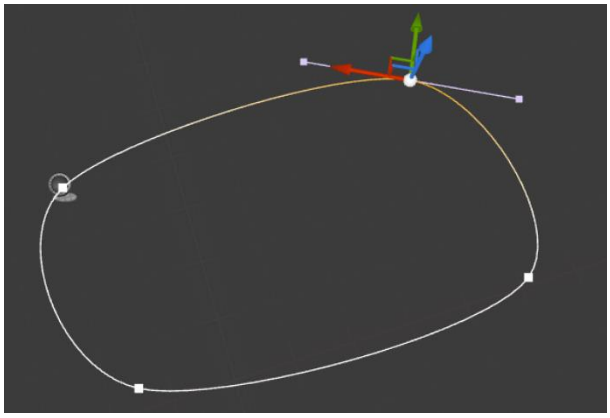


Abb. 80 Spline in Kreis Form

Abschließend wird im *Components Panel* die *BP_No_Forest_Spline* ausgewählt (Abbildung 81) und im *Details Panel* der Tag *No_Forest* hinzugefügt (Abbildung 82).

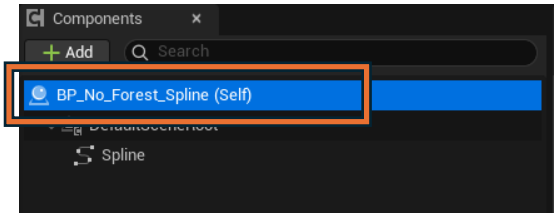


Abb. 81 Blueprint Actor in Components Tab ausgewählt

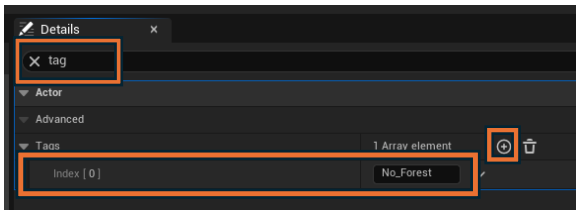


Abb. 82 Tag hinzufügen

Im Anschluss muss die erstellte *Spline* kompiliert werden (Abbildung 83). Wenn alle Schritte korrekt ausgeführt wurden, sollte das angezeigte *Fragezeichen* durch ein *Häkchen* ersetzt werden (Abbildung 84).

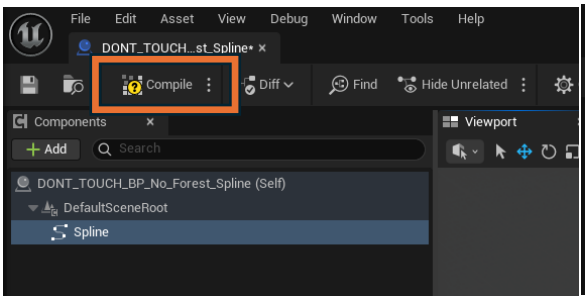


Abb. 83 Kompilierknopf

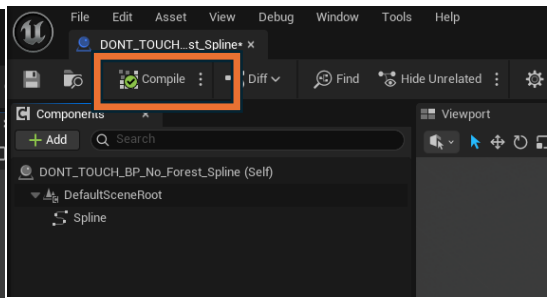


Abb. 84 Erfolgreiches Kompilieren

Die *Spline* kann nun in das Level gezogen werden, um das Dorf zu umkreisen (Abbildung 85). Dazu werden die einzelnen *Spline*-Punkte ausgewählt und so verschoben, dass das Dorf vollständig innerhalb der *Spline* liegt (Abbildung 86).

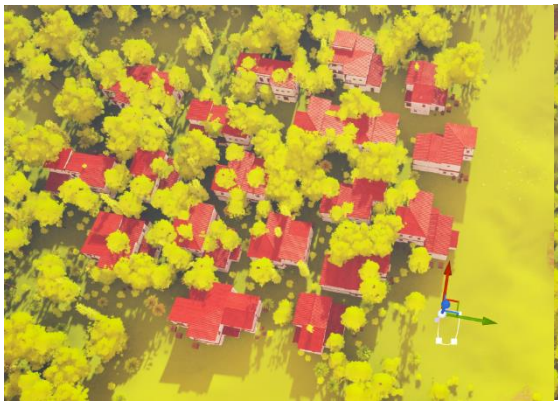


Abb. 85 Spline im Level

Abb. 86 Dorf umrandet von Spline

8.1.4.2. Spline in PCG-Graphen integrieren

Um die *Spline* nun in den *PCG Graph* zu integrieren, wird dieser erneut geöffnet. Ziel ist es, alle Punkte zu entfernen, die sich innerhalb der *Spline* befinden, sodass diese nicht für das Spawning zur Verfügung stehen. Zunächst muss überprüft werden, ob im Level eine *Spline* vorhanden ist. Dazu wird eine *Get Spline Data Node* verwendet, mit der nach *Splines* gesucht werden kann. Im *Details Panel* wird bei *Actor Filter* die Option *All World Actors* ausgewählt und bei *Actor Selection Tag* der Tag *No_Forest* eingetragen, der zuvor dem *Spline Actor* vergeben wurde. Zusätzlich sollte die Option *Select Multiple* aktiviert werden, damit die *Spline* mehrfach innerhalb desselben Levels verwendet werden kann (Abbildung 87).

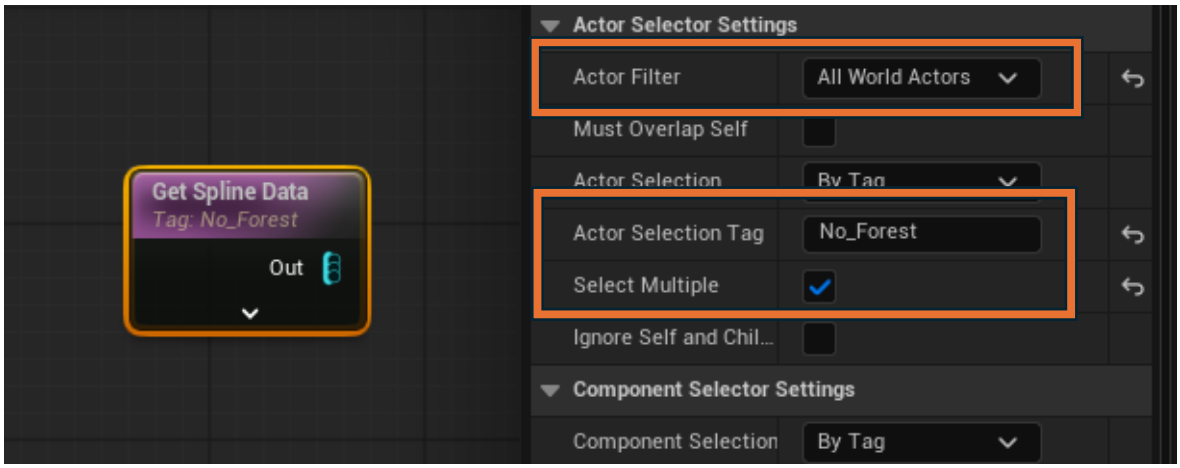


Abb. 87 Get Spline Data Einstellungen

Die *Get Spline Data Node* wird mit einer *Spline Sampler Node* verbunden. Damit alle Punkte innerhalb der *Spline* erfasst werden, wird im *Details Panel* unter *Dimension* die Einstellung auf *On Interior* gesetzt. Um die Performance zu optimieren, werden *Interior Sample Spacing* und *Interior Border Sample Spacing* jeweils auf 500 festgelegt. Zusätzlich muss die *Unbounded* Einstellung aktiviert werden, damit die Punkte unabhängig von der ursprünglichen Begrenzung der *Spline* erzeugt werden (Abbildung 88).

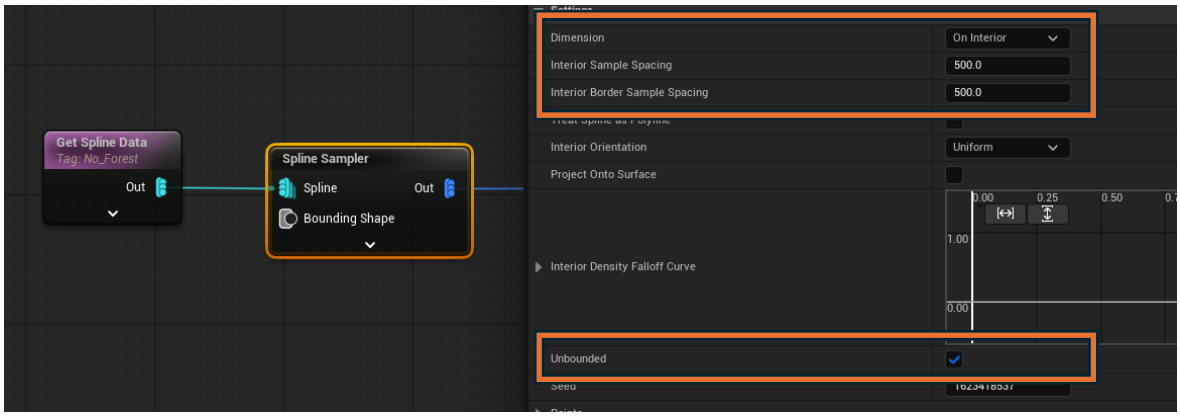


Abb. 88 Spline Sampler Node Einstellungen

Für den nächsten Schritt wird eine *Bounds Modifier Node* erstellt. In dieser wird die *Scale* aller Punkte auf das Doppelte erhöht, um sicherzustellen, dass alle Flächen innerhalb der *Spline* abgedeckt sind, selbst wenn sich das Terrain später verändert (Abbildung 89).

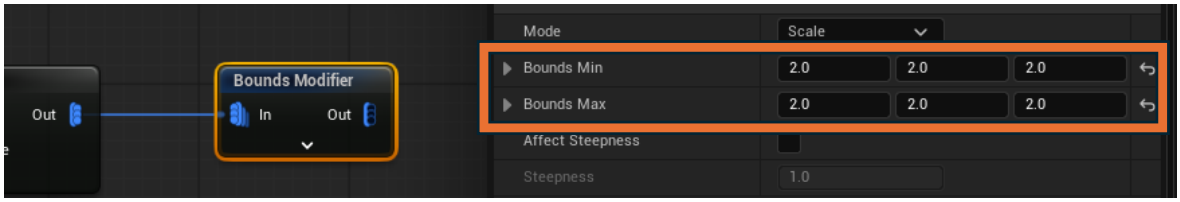


Abb. 89 Bounds Modifier Einstellungen

Um die Punkte nun tatsächlich an das Terrain anzupassen, wird erneut eine *Projection Node* mit einer *Get Landscape Data Node* hinzugefügt.

Nun müssen alle Punkte aus dem *Surface Sampler* entfernt werden, bevor die Baum- und anderen *Meshes* gespawnt werden. Dazu wird von der ursprünglich verwendeten *Surface Sampler Node* eine *Difference Node* abgeleitet. Da alle Punkte innerhalb der *Spline* von der Landschaft entfernt werden sollen, wird als *Source* die Verbindung zwischen *Surface Sampler Node* und *Get Landscape Data Node* verwendet, während die *Difference* von der *Projection Node* der *Get Spline Data Node* stammt. Der *Out Output* der *Difference Node* wird anschließend wieder in den Hauptgraphen verbunden (Abbildung 90). Dabei ist besonders darauf zu achten, dass keine direkte Verbindung mehr zwischen dem *Surface Sampler* und dem *PCG Subgraph* besteht.

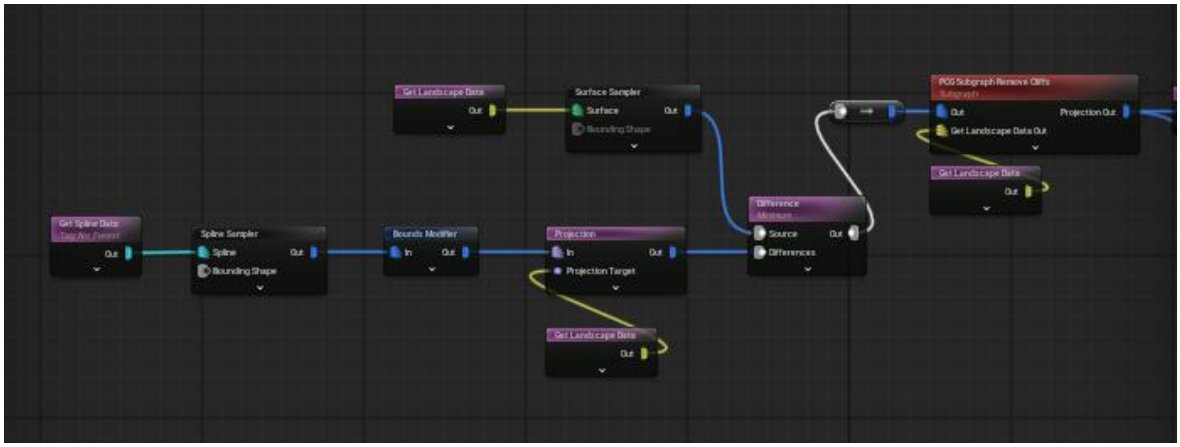


Abb. 90 Spline Data Nodees verbunden mit dem Hauptgraphen

Nun sollten alle *Meshes* innerhalb der *Spline* entfernt sein (Abbildung 91). Die *BP_No_Forest_Spline* kann auch für weitere Bereiche verwendet werden, in denen keine Vegetation wachsen soll. Da dabei alle Punkte des *Surface Samplers* gelöscht werden, bleibt die Gesamtanzahl der Bäume und anderen *Meshes* unverändert.

Abb. 91 Jegliche Vegetation innerhalb der Spline entfernt

8.1.5. Neues Level

Um ein neues Level zu gestalten, wird zunächst der bestehende *PCG Graph* ausgewählt und in das neue Level gezogen. Um eine andere Version des Waldes zu erzeugen, wird der *PCG Graph* geöffnet. Anschließend wird die *Spatial Noise Node* lokalisiert und ausgeklappt. Es wird ein neuer *Seed Parameter* erstellt und mit dem *Seed Input* der *Spatial Noise Node* verbunden (Abbildung 91). Als Wert für den *Seed Parameter* kann eine beliebige Zahl gewählt werden, um eine andere zufällige Verteilung der Vegetation zu erzeugen (Abbildung 92).

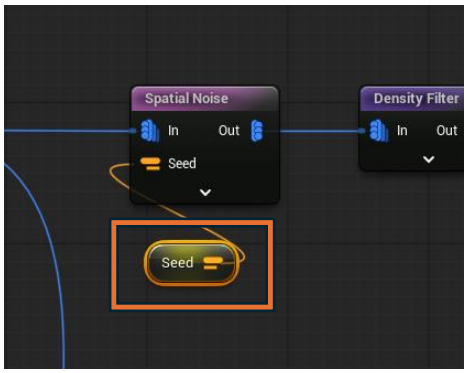


Abb. 93 Seed Parameter in Spatial Noise

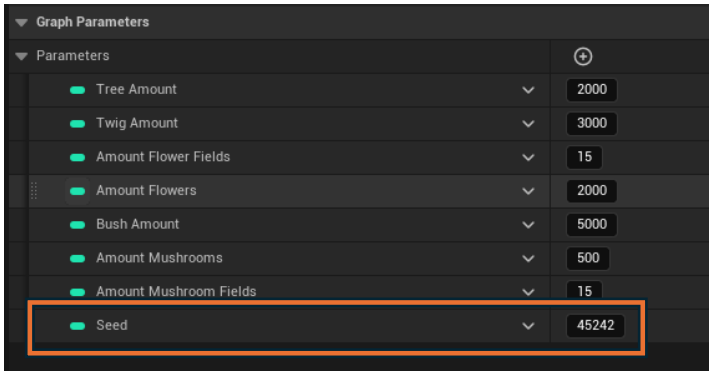


Abb. 92 Seed Parameter

Im Level kann der *PCG Graph* über den *Outliner* ausgewählt werden (Abbildung 94). Anschließend wird im *Details Panel* das zugehörige *PCG Component* geöffnet. Unter *Parameter Overrides* sind alle zuvor definierten Parameter einsehbar. Möchte man ein alternatives Waldlayout erzeugen, kann der *Seed Parameter* aktiviert und ein neuer Wert eingegeben werden (Abbildung 95), wodurch die Verteilung der Vegetation zufällig verändert wird. Diese Anpassung wirkt ausschließlich auf die jeweilige *PCG*-Instanz im aktuellen Level und verändert das ursprüngliche *PCG* nicht. Dadurch ist es beispielsweise möglich, in einem Level nur 100 Bäume zu spawnen, während im ursprünglichen Level weiterhin die gewünschte Anzahl von 2000 Bäumen erhalten bleibt.

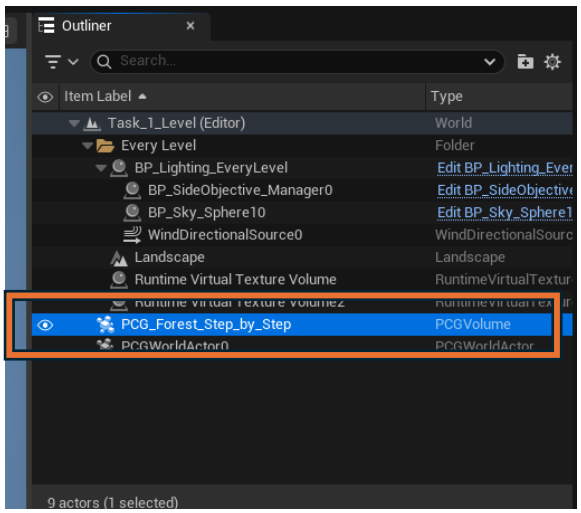


Abb. 94 PCG Forest im Outliner

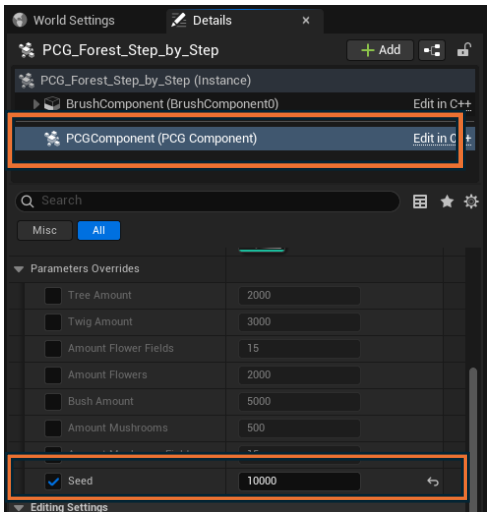


Abb. 95 PCG Parameter Override

8.2. Foliage Tool

8.2.1. Pilze

Für die Pilzfelder müssen zunächst aus den beiden Pilz *Static Mesh*es neue *Static Mesh Foliages* erstellt werden. Anschließend erfolgt das Platzieren der Pilze nach demselben Prinzip wie bei den Blumenfeldern, nur dass die Pilze diesmal ausschließlich innerhalb des Waldes und nicht außerhalb positioniert werden.

8.2.2. Bäume

Um die neuen Zypressenbäume in das Level zu platzieren, muss zunächst ein neues *Static Mesh Foliage* für die Zypressen erstellt werden. Anschließend werden alle Parameter wie bei den bereits vorhandenen Bäumen eingestellt. Um die Anzahl an Bäumen zu verdoppeln, wird der Wert der *Density/1Kuu* entsprechend verdoppelt. Danach werden mit dem *Erase Tool* und dem *Paint Tool* alle Bäume gleichmäßig im Level verteilt, bis alle Anforderungen erfüllt sind.

8.2.3. Büsche

Bei den Büschen wird dasselbe Prinzip angewendet. Die *Fern*- und *Sprössling-Static Mesh*es werden zunächst in *Static Mesh Foliages* umgewandelt. Anschließend werden die Büsche mithilfe des *Erase*- und *Paint*-Tools neu platziert, bis alle Anforderungen vollständig erfüllt sind.

8.2.4. Dorf

Um das Dorf zu platzieren, wird zunächst das Town Level im *Content Browser* ausgewählt und in das bestehende Level gezogen. Anschließend wird die Position des *Asset*s auf die Koordinaten 0, 0, 0 gesetzt. Im nächsten Schritt müssen alle vorhandenen *Foliage Meshes* innerhalb des Dorfbereichs mithilfe des *Erase Tools* entfernt werden. Danach werden die fehlenden *Foliage Meshes* mit dem *Paint Tool* neu platziert, bis alle Anforderungen wieder erfüllt sind.

8.2.5. Neues Level

Um ein neues Level mit Wald mithilfe des *Foliage Tools* auszustatten, wird zunächst das Tool erneut geöffnet. Im *Content Browser* können anschließend alle *Static Foliage Meshes* gesucht werden, die zuvor voreingestellt wurden (Abbildung 96). Dabei ist darauf zu achten, dass man sich tatsächlich im entsprechenden Content-Ordner befindet, da ansonsten *Meshes*, die sich in anderen Ordnern befinden, nicht angezeigt und verwendet werden können.

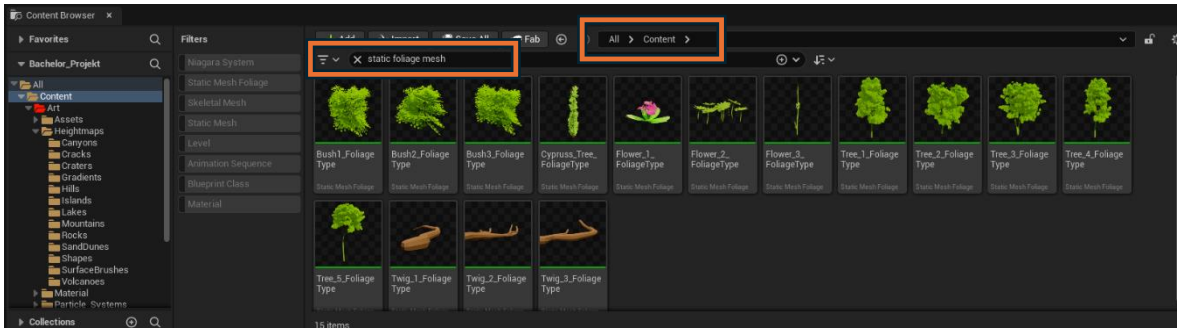


Abb. 96 Static Foliage Mesh im Content Browser

Anschließend können alle *Static Foliage Meshes* ausgewählt und in den *+Drop Foliage Here*-Bereich gezogen werden (Abbildung 97, 98). Danach verläuft der weitere Ablauf identisch mit der zuvor beschriebenen Levelgestaltung. Da die neue *Foliage* nur im aktuellen Level gemalt wird, bleibt das zuvor erstellte Level unverändert.

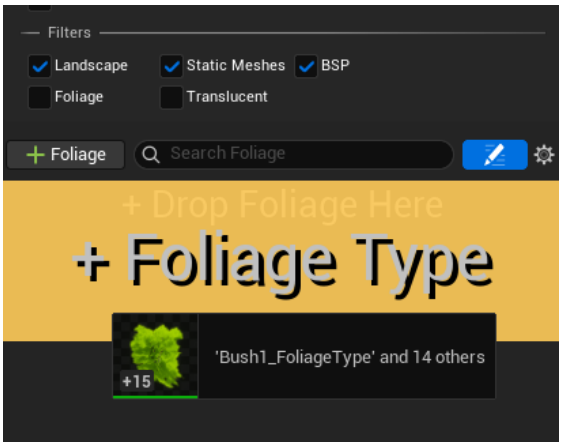


Abb. 97 Foliage Tool Drag and Drop

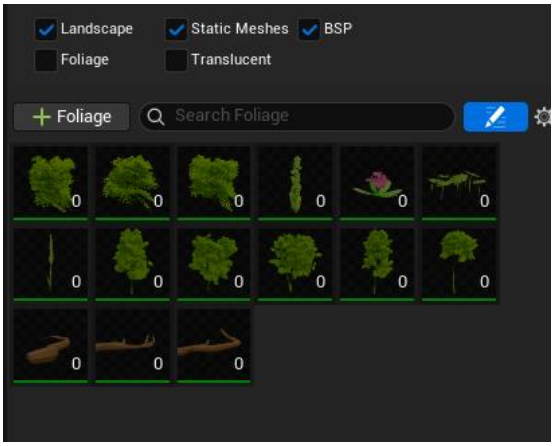


Abb. 98 Alle Foliage Meshes

9. Durchführung und experimentelle Evaluation

9.1. Ablauf

Das Experiment wurde im Rahmen eines Mixed-Methods-Designs durchgeführt und umfasste einen leitfadenbasierten Versuch sowie eine ergänzende Umfrage mit quantitativen und qualitativen Anteilen. Ziel war es, die Effizienz und Benutzerfreundlichkeit bei der Levelerstellung mit der *Unreal Engine* zu untersuchen.

Für die Rekrutierung wurden gezielt Personen aus dem Bereich der Spieleentwicklung angesprochen, darunter Studierende, selbstständig tätige Entwicklerinnen und Entwickler sowie Fachkräfte aus der Branche. Als Einschlusskriterien galten eine aktive Beschäftigung mit Spieleentwicklung oder ein laufendes Studium im Bereich Game Development sowie grundlegende Kenntnisse im Umgang mit der *Unreal Engine*. Personen ohne entsprechende Vorerfahrung wurden von der Teilnahme ausgeschlossen. Insgesamt wurden zehn potenzielle Teilnehmende über Discord kontaktiert, von denen vier ihre Teilnahme zusagten. Zwei der Teilnehmenden waren selbstständig in der Spieleentwicklung tätig, eine Person studierte Spielentwicklung und eine Person hatte bereits ein kommerziell veröffentlichtes Spiel veröffentlicht.

Das Experiment fand am 16. Oktober 2025 um 17 Uhr statt und dauerte etwa fünf Stunden. Die Durchführung erfolgte mit der *Unreal Engine* in der Version 5.6. Vorab erhielten alle Teilnehmenden per E-Mail das vorbereitete Projekt sowie eine schriftliche Anleitung zum Öffnen und zur Einrichtung der Testumgebung. Zusätzlich wurden die in Kapitel 6 und 7 beschriebenen Leitfäden zur Nutzung der *Foliage*- und *PCG*-Systeme bereitgestellt, die während der Aufgabenbearbeitung als Orientierung dienten.

Zu Beginn der Sitzung trafen sich alle Teilnehmenden in einem gemeinsamen Discord-Call, teilten ihren Bildschirm und füllten den ersten Teil des Fragebogens (siehe Anhang) aus. Anschließend erhielten sie Zeit, sich mit den Leitfäden vertraut zu machen, bevor sie individuell mit den Aufgaben begannen. Alle Aufgaben sind in Kapitel 5.8 beschrieben. Die Bearbeitung erfolgte asynchron, da die Zeitmessung aufgabenspezifisch durchgeführt wurde. Die Dauer jedes Aufgabenabschnitts wurde mit individuellen Stoppuhren erfasst. Nach Abschluss einer Aufgabe informierte die jeweilige Person die Versuchsleitung, welche die Erfüllung der Aufgabenanforderungen überprüfte und die Zeit stoppte. Wurden einzelne Kriterien nicht erfüllt, wurde die Aufgabe erneut bearbeitet, wobei die Zeitmessung fortgesetzt wurde, bis alle Anforderungen erfüllt waren.

Während der Durchführung wurde auf eine ruhige Arbeitsatmosphäre geachtet. Nach Abschluss aller Aufgaben füllten die Teilnehmenden den zweiten Teil des Fragebogens aus, in dem sie ihre subjektive Einschätzung zu Effizienz und Benutzerfreundlichkeit dokumentierten. Anschließend durfte der Teilnehmer den Discord Call verlassen.

9.2. Einordnung der Teilnehmer

Das Experiment wurde mit insgesamt vier Teilnehmenden durchgeführt. Die Kommunikation erfolgte über *Discord*, da es den Teilnehmenden ermöglichte, an ihren eigenen, vertrauten Arbeitsplätzen zu arbeiten. Alle verfügten über eine vergleichbar leistungsfähige Hardware, sodass technische Unterschiede keinen relevanten Einfluss auf die Durchführung oder die Messergebnisse hatten.

Aus den im Vorfeld erhobenen Fragebogen ging hervor, dass alle Teilnehmenden bereits Erfahrung mit der *Unreal Engine* gesammelt hatten (vgl. Tabelle 6), diese jedoch überwiegend im akademischen Kontext oder nur gelegentlich nutzten (vgl. Tabelle 7).

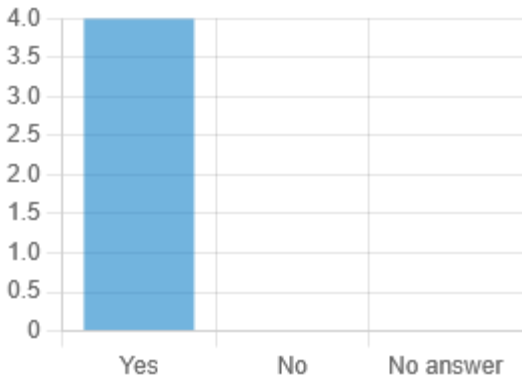


Tabelle 5 "Have you used Unreal Engine before?"

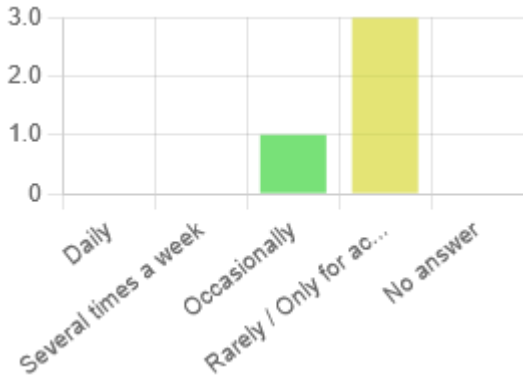


Tabelle 6 "How frequently do you work with the engine?"

Die Gruppe setzte sich aus Personen mit unterschiedlichen Schwerpunkten zusammen, darunter sowohl Programmiererinnen und Programmierer als auch Artists, die jeweils unterschiedliche Stärken im Umgang mit spezifischen Tools aufwiesen. Auffällig war, dass keine der testenden Personen über nennenswerte Vorerfahrung im Bereich der *Procedural Content Generation (PCG)* verfügte (vgl. Tabelle 8), während mehrere Teilnehmende bereits mit dem *Foliage Tool* gearbeitet hatten und dort als fortgeschrittener einzustufen waren (vgl. Tabelle 9).

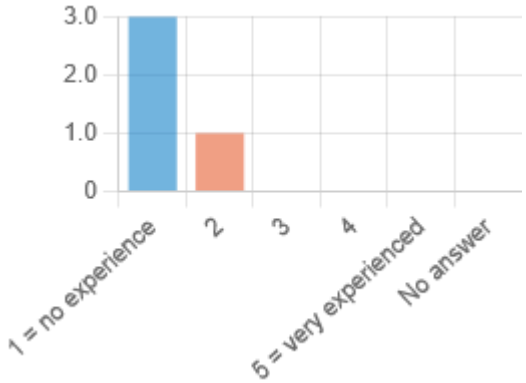


Tabelle 7 "How familiar are you with the following topics (PCG Graph)?"

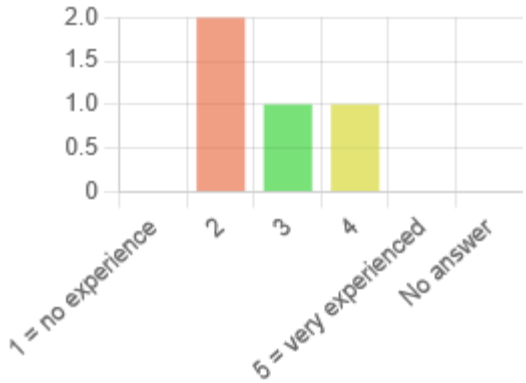


Tabelle 8 "How familiar are you with the following topics (Foliage Tool)?"

9.3. Rohdaten

Die folgende Tabelle zeigt die von den Teilnehmenden erfassten Bearbeitungszeiten für jede Aufgabe, angegeben in Minuten.

	Person 1	Person 2	Person 3	Person 4
Level with Foliage	16.49	37.56	32.05	46.56
Changes in Foliage	7.14	17.27	19.49	23.21
New Level with Foliage	17.23	18.43	19.55	23.50
Level with PCG	50.22	61.46	67.06	72.32
Changes in PCG	15.14	26.19	19.49	22.01
New Level with PCG	1.40	1.42	1.48	1.57

Tabelle 9 Time spent on Level Creation in Minutes

9.4. Auswertung der Tabelle

In den folgenden Diagrammen sind die gemessenen Zeiten der einzelnen Teilnehmenden für die jeweiligen Arbeitsschritte in Minuten dargestellt.

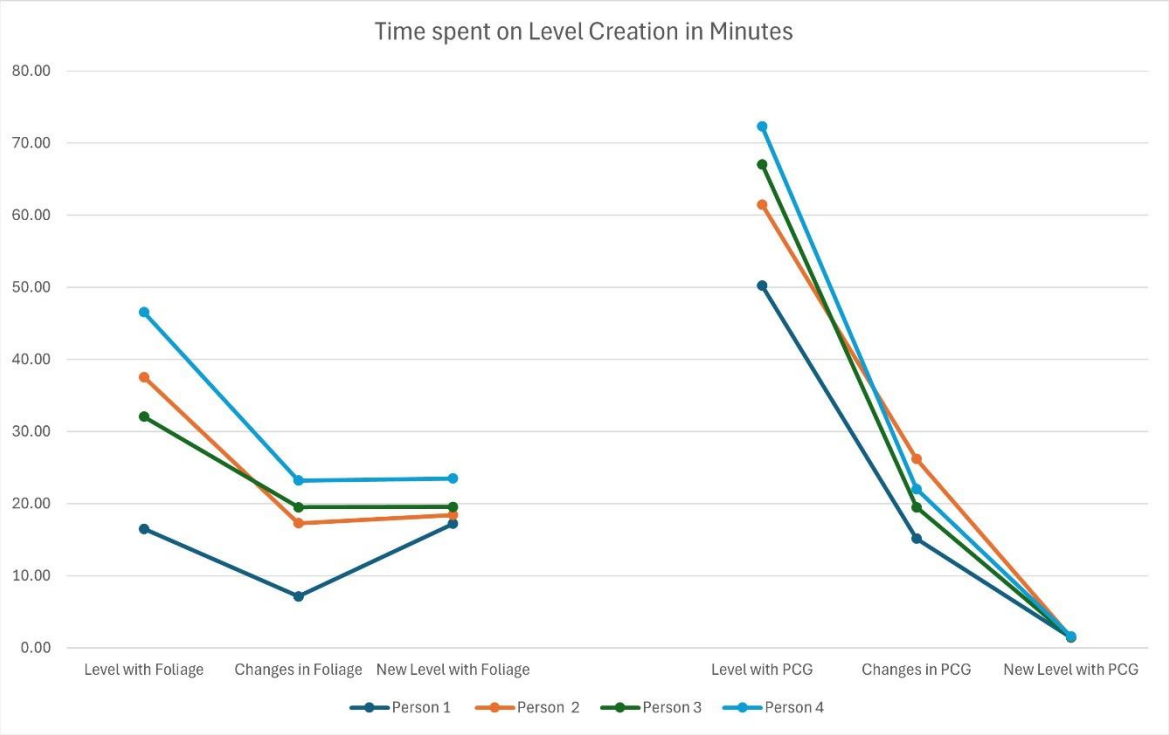


Tabelle 10 Linien Graph aller Zeiten in Minuten

Dabei zeigt sich, dass die erstmalige Levelerstellung mit dem *Foliage Tool* deutlich schneller durchgeführt werden konnte als mit dem *PCG Graph*. Die durchschnittliche Bearbeitungszeit für die initiale Levelerstellung mit dem *Foliage Tool* betrug 33,17

Minuten, während die durchschnittliche Zeit für die Erstellung eines vergleichbaren Levels mit *PCG* bei 62,77 Minuten lag.

Bei der anschließenden Anpassung der Level auf Grundlage neuer Anforderungen ergaben sich dagegen nur geringe Unterschiede. Hier lagen die Durchschnittszeiten bei 16,78 Minuten für das *Foliage Tool* und 20,71 Minuten für den *PCG Graph*.

Besonders deutlich traten die Unterschiede bei der Erstellung weiterer Level hervor. Während sich bei der erfahrensten Testperson die Bearbeitungszeit für ein neues Level mit dem *Foliage Tool* kaum veränderte (von 16,49 Minuten auf 17,23 Minuten), zeigte sich bei weniger erfahrenen Anwenderinnen und Anwendern eine deutliche Verbesserung ihrer Bearbeitungszeit. Diese Entwicklung lässt sich darauf zurückführen, dass die Teilnehmenden durch den bereitgestellten Leitfaden zunehmend mit den Funktionen und Arbeitsschritten des *Foliage Tools* vertraut wurden und dadurch effizienter arbeiteten.

Beim *PCG Graph* hingegen lagen die Zeiten aller Teilnehmenden sehr eng beieinander. Die durchschnittliche Zeit für die Erstellung eines weiteren Levels betrug hier lediglich 1,47 Minuten, was den Effizienzvorteil des prozeduralen Ansatzes bei wiederholter Anwendung deutlich hervorhebt.

9.5. Auswertung des Fragebogens

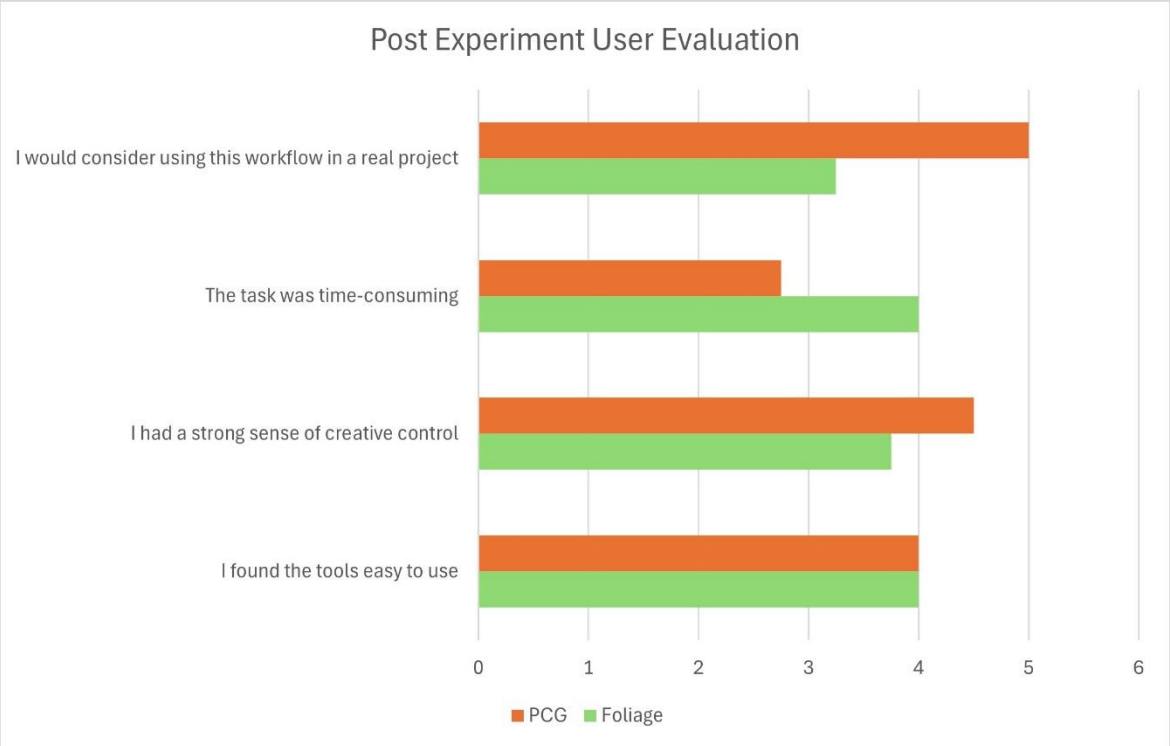


Tabelle 11 Balkendiagramm der Teilnehmer Antworten nach dem Experiment

Nach Abschluss der Aufgaben bewerteten die Teilnehmenden beide Workflows anhand eines kurzen Fragebogens auf einer Likert-Skala. Tabelle 12 zeigt den Durchschnittswert der Teilnehmenden für die Aussagen.

Die Ergebnisse deuten auf ein insgesamt positives Nutzererlebnis für beide Workflows hin, mit teilweise Präferenzen zugunsten des prozeduralen Ansatzes (*PCG*). Besonders bei der Frage, ob der Workflow in einem realen Projekt verwendet werden würde, erzielte *PCG* mit einem Durchschnittswert von rund 5,0 eine höhere Zustimmung als das herkömmliche *Foliage* Tool mit 3,25. Dies spricht dafür, dass die Teilnehmenden den *PCG*-Ansatz trotz des höheren initialen Aufwands als zukunftsfähiger und nachhaltiger einschätzen.

In Bezug auf den wahrgenommenen Zeitaufwand ergab sich ein umgekehrtes Muster. Das *Foliage* Tool wurde als etwas zeitintensiver empfunden (4,0) als der *PCG*-Workflow (2,75). Diese subjektive Wahrnehmung deckt sich nicht mit den Messdaten aus Abschnitt 9.2, wo *PCG* längere Initialzeiten und vergleichbare oder leicht höhere Anpassungszeiten zeigte. Offenbar wird der Zeitaufwand bei *PCG* subjektiv nicht als so belastend erlebt.

Hinsichtlich der kreativen Gestaltung schnitt *PCG* ebenfalls etwas besser ab mit 4.5 im Vergleich zu 3.75 beim *Foliage* Tool. Die Teilnehmende gaben an, dass die regelbasierte Steuerung der Graphen ihnen trotz Automatisierung ein gutes Gefühl von Kontrolle ermöglichte. Diese Einschätzung ist überraschend, da prozedurale Systeme häufig als weniger direkt steuerbar gelten, wie bereits in Kapitel 4 festgestellt wurde.

Beide Tools wurden als einfach zu bedienen bewertet (4,0 für beide). Dies deutet darauf hin, dass die Einarbeitung in den *PCG*-Editor für Unreal-Nutzende keine signifikante Hürde darstellte.

Insgesamt lässt sich aus den Umfrageergebnissen ableiten, dass die Teilnehmenden den *PCG*-Workflow als moderne und praktikable Alternative zum traditionellen *Foliage* Tool betrachten. Trotz höherer Komplexität in der Einrichtung wurde der Ansatz als zukunftsorientiert, kontrollierbar und nutzungsfreundlicher von den Teilnehmenden wahrgenommen.

Auf die Frage, welche der beiden Arbeitsweisen als angenehmer empfunden wurde, gaben alle Teilnehmenden übereinstimmend an, dass sie die Arbeit mit *PCG* als insgesamt positiver wahrnahmen (vgl. Tabelle 13).

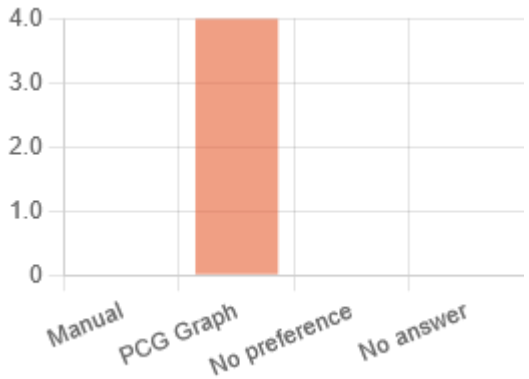


Tabelle 12 "Which workflow felt better to use overall?"

9.6. Limitationen

9.6.1. Externe Validität der Online-Umfrage

Die Rücklaufquote der Online-Umfrage unter *Indie*-Studios ist deutlich geringer als geplant. Viele kleine Studios waren nur schwer erreichbar, da auf deren Webauftritten häufig ein Impressum oder Kontaktdaten fehlen bzw. keine direkten Kontaktmöglichkeiten bestehen. Dadurch konnte nur eine kleine Stichprobe gewonnen werden, was die Aussagekraft der Ergebnisse einschränkt.

9.6.2. Begrenzte Stichprobengröße

Das Hauptexperiment wird mit lediglich vier Teilnehmenden durchgeführt. Ein Grund für die geringe Anzahl an Testpersonen liegt im hohen Aufwand pro Versuch. Aufgrund dieser kleinen Stichprobe können keine statistisch belastbaren Aussagen über die allgemeine Effizienz oder Präferenz zwischen den getesteten Workflows getroffen werden. Die Ergebnisse sind daher als explorativ zu verstehen und dienen vor allem der Hypothesenbildung für weiterführende Studien.

9.6.3. Projekt Setup

Da das Projekt-Setup individuell erstellt wird, existiert bislang keine unabhängige Replikation der Ergebnisse.

9.6.4. Begrenzte Datentiefe und Messgenauigkeit

Die Zeitmessung erfolgt manuell durch Start- und Stopp-Signale der Teilnehmenden. Kleinere Verzögerungen oder Unklarheiten beim Übergang zwischen Aufgabenphasen können die gemessenen Werte leicht verfälschen. Ebenso wird der qualitative Fragebogen nachträglich ausgefüllt, wodurch Erinnerungsverzerrungen (recall bias) möglich sind.

9.6.5. Lern- und Reihenfolgeeffekte

Da alle Teilnehmenden beide Workflows nacheinander durchlaufen, können Lerneffekte nicht vollständig ausgeschlossen werden. Erfahrungen aus der ersten Aufgabe könnten die Bearbeitungszeit und subjektive Bewertung der zweiten beeinflusst haben.

9.6.6. Remote-Durchführung über Discord

Die Experimente finden dezentral über Discord statt, wodurch Einflüsse der jeweiligen Hardware-Leistung, Netzwerklatenz und Arbeitsumgebung nicht kontrolliert werden können. Dies könnte sich auf die gemessenen Bearbeitungszeiten auswirken.

10. Fazit

10.1. Zusammenfassung der zentralen Erkenntnisse

Diese Arbeit untersucht den Unterschied zwischen traditioneller, manueller Levelgestaltung mittels des *Foliage*-Tools und prozeduraler Erzeugung mit *PCG-Graphen* in der *Unreal Engine*. Methodisch wurde ein Mixed-Methods-Ansatz gewählt, indem zwei Prototypen entwickelt wurden. Es wurde ein standardisierter Leitfaden bereitgestellt und mit vier Teilnehmenden ein Within-Subject-Experiment durchgeführt, in dessen Rahmen sowohl die benötigten Bearbeitungszeiten als auch subjektive Bewertungen erhoben wurden. Damit lassen sich quantitative Aussagen zur Effizienz sowie qualitative Aussagen zur Handhabbarkeit und Akzeptanz beider Workflows treffen.

Die quantitativen Messdaten zeigen ein klar differenziertes Bild. Die initiale Erstellung eines Levels mit dem *Foliage*-Tool verlief durchschnittlich deutlich schneller (Durchschnitt = 33,17 Minuten) als die erstmalige Erstellung mit dem *PCG*-Graph (Durchschnitt = 62,77 Minuten). Bei nachfolgenden Änderungen liegen die Zeiten beider Verfahren enger beieinander (Durchschnitt Änderungen *Foliage* = 16,78 min, *PCG* = 20,71 min). Besonders auffällig ist jedoch der Vorteil des *PCG*-Ansatzes bei der Wiederverwendung und beim Erstellen weiterer Level. Während die Erstellung eines neuen Levels mit manueller Arbeit durchschnittlich 19,68 Minuten beanspruchte, dauerte derselbe Arbeitsschritt mit einem einmal erstellten *PCG-Graphen* nur 1,47 Minuten. Diese Zahlen deuten darauf hin, dass *PCG-Graphen* einen signifikanten Vorteil bei Skalierung und Wiederverwendbarkeit bieten, obwohl ihr Initialaufwand höher ist.

Die subjektiven Bewertungen ergänzen die Messdaten. Die Teilnehmenden schätzten den *PCG*-Workflow hinsichtlich Zukunftsfähigkeit und kreativer Kontrolle tendenziell positiver ein und gaben an, den *PCG*-Ansatz in realen Projekten eher in Betracht zu ziehen als das reine *Foliage*-Vorgehen. Gleichzeitig wurden Hürden wie erforderliches Fachwissen, initialer Implementationsaufwand und potenziell aufwändigeres Debugging als relevante Einschränkungen genannt. Diese Einschätzungen stimmen mit den Umfrageergebnissen der begleitenden Umfrage unter *Indie*-Studios überein, die *PCG* zwar als potenziellen Zeit- und Qualitätsgewinn beschreibt, dessen Einsatz jedoch stark von vorhandener Expertise und Projektanforderungen abhängig sieht.

Zusammenfassend zeigt die Untersuchung aber, dass *PCG-Graphen* bei wiederholter Nutzung eine erhebliche Zeitersparnis ermöglichen und von Nutzenden positiv bewertet werden.

Abschließend ist wichtig darauf hinzuweisen, dass diese Befunde durch die kleine Stichprobengröße von 4 begrenzt ist. Die Resultate liefern eine praxisnahe Momentaufnahme und Hypothesen für weitergehende Untersuchungen, erlauben aber keine generalisierbaren, statistisch robusten Schlussfolgerungen. Dennoch legt die Kombination aus Messdaten und subjektiver Bewertung nahe, dass sich *PCG-Graphen* vor

allem dann lohnen, wenn Wiederverwendung, Variantenbildung und Skalierbarkeit zentrale Anforderungen darstellen.

10.2. Praxisempfehlungen für Entwicklerteams

Ausgehend von den gewonnenen Erkenntnissen lässt sich ein sachliches Vorgehen für Entwicklerteams formulieren. Bei der Entscheidung für oder gegen den Einsatz prozeduraler Graphen sollte das Projekt maßgeblich berücksichtigt werden. Insbesondere Projekte mit großflächigem oder wiederkehrendem Content, mit mehreren Levelvarianten oder mit einem hohen Bedarf an schnellen Iterationen profitieren von einer anfänglichen Investition in *PCG-Graphen*, da sich der höhere Initialaufwand durch die deutlich verkürzte Erstellungszeit für weitere Level lohnt.

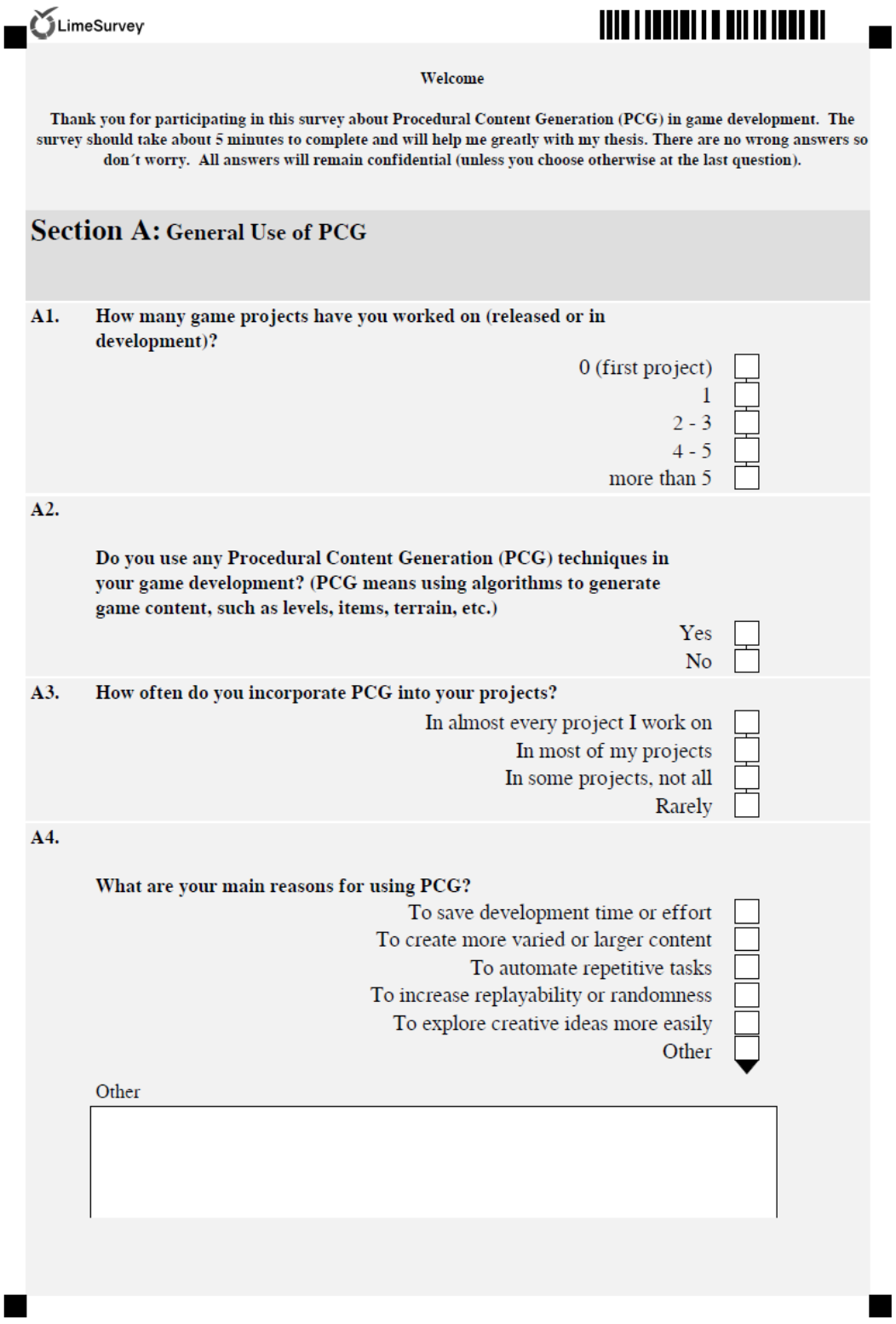
Empfohlen wird ein modularer und hybrider Arbeitsansatz. Dazu gehört die frühzeitige Entwicklung wiederverwendbarer Subgraphen sowie die Parametrisierung zentraler Einstellungen wie *Seed*, Dichte und Varianten, um später flexible Anpassungen zu ermöglichen. Prozedurale Verfahren sollten vornehmlich für großflächige und repetitive Inhalte genutzt werden, während in narrativ relevanten oder künstlerisch fein abgestimmten Bereichen gezielte manuelle Platzierung erfolgen sollte. Auf diese Weise lassen sich Effizienzgewinne erzielen, ohne die gestalterische Qualität der Levels zu beeinträchtigen.

Der in dieser Arbeit enthaltene Guide stellt einen geeigneten Einstieg dar, um die wichtigsten Funktionen von *PCG-Graphen* wie Subgraphstrukturen, Parametrisierung, Debugging-Werkzeuge und Wiederverwendungsstrategien praxisnah zu erlernen. Durch die im Guide dargestellten Beispiele und empfohlenen Arbeitsschritte lässt sich die anfängliche Lernkurve verkürzen und die Integration in die bestehenden Pipelines beschleunigen.

Zusammenfassend zeigt die Untersuchung, dass sich der Einsatz von *PCG-Graphen* für Entwicklerteams durchaus lohnen kann, sofern Wiederverwendbarkeit, Skalierbarkeit und schnelle Iterationen zentrale Anforderungen des Projekts sind. Eine wohlüberlegte Kombination aus modularer *PCG*-Entwicklung und gezielter manueller Nachbearbeitung erweist sich als zielorientiertester Weg, um die Effizienzvorteile prozeduraler Verfahren nutzbar zu machen, ohne die gestalterischen Ansprüche der Projekte zu gefährden.

11. Anhang

11.1. A Survey on Procedural Content Generation in Indie Game Development



LimeSurvey

Welcome

Thank you for participating in this survey about Procedural Content Generation (PCG) in game development. The survey should take about 5 minutes to complete and will help me greatly with my thesis. There are no wrong answers so don´t worry. All answers will remain confidential (unless you choose otherwise at the last question).

Section A: General Use of PCG

A1. **How many game projects have you worked on (released or in development)?**

- ☐ 0 (first project)
- ☐ 1
- ☐ 2 - 3
- ☐ 4 - 5
- ☐ more than 5

A2.

Do you use any Procedural Content Generation (PCG) techniques in your game development? (PCG means using algorithms to generate game content, such as levels, items, terrain, etc.)

- ☐ Yes
- ☐ No

A3. **How often do you incorporate PCG into your projects?**

- ☐ In almost every project I work on
- ☐ In most of my projects
- ☐ In some projects, not all
- ☐ Rarely

A4.

What are your main reasons for using PCG?

- ☐ To save development time or effort
- ☐ To create more varied or larger content
- ☐ To automate repetitive tasks
- ☐ To increase replayability or randomness
- ☐ To explore creative ideas more easily
- ☐ Other

Other

LimeSurvey

A5.

What are your main reasons for not using PCG?

- [ ] My game doesn't need it / not relevant to my project
- [ ] It seems too complex or difficult to implement
- [ ] I lack knowledge or experience with PCG methods
- [ ] I don't have time to learn or set up PCG
- [ ] I prefer designing content manually
- [ ] Other

Other

A6. **Since you don´t use PCG graphs, which methods do you primarily use to build Levels?**

- [ ] Hand-crafted in-engine
- [ ] Using grid systems
- [ ] Other

Other

Section B: Type of PCG Content

B1. **What types of game content do you generate using PCG?**

- [ ] Level or dungeon layouts
- [ ] Terrain, landscapes, or environments
- [ ] Building or architecture placement
- [ ] Items, weapons, or loot
- [ ] Characters or enemies (appearance, stats, etc.)
- [ ] Quests or narrative elements (story, dialogue)
- [ ] Other

Other

LimeSurvey

B2. **What problems or challenges are you facing when it comes to generating PCG content? Please feel free to add a comment if there's anything else you'd like to highlight — your input will help me a lot.**

- [ ] Ensuring generated content is playable and error-free
- [ ] Balancing randomness with intentional design
- [ ] Providing enough variety and avoiding repetition
- [ ] Performance issues during generation or gameplay
- [ ] Limited algorithmic or technical expertise
- [ ] Meeting player expectations for a polished experience
- [ ] Other

Other

Section C: Tools and Engines

C1.

Which game engine do you primarily use for your projects?

- [ ] Unreal Engine
- [ ] Unity
- [ ] Godot
- [ ] GameMaker
- [ ] Other

Other

C2. **Have you used the PCG Graph feature in Unreal Engine (for procedural generation)**

- [ ] Yes
- [ ] Custom Blueprints or C++ in Unreal Engine (handmade PCG scripts)
- [ ] Other

Other

LimeSurvey

C3.

How would you rate the ease of use of Unreal's PCG Graph?

- 1 – Very difficult ☐
- 2 – Somewhat difficult ☐
- 3 – Neutral / Unsure ☐
- 4 – Somewhat easy ☐
- 5 – Very easy ☐

Section D: Effects of PCG on Development

D1.

Please indicate your agreement with the following statements about the impact of PCG on your development.

	1 = Strongly disagree	2	3	4	5 = Strongly agree
PCG has generally saved me development time.	☐	☐	☐	☐	☐
PCG has increased the variety of content in my	☐	☐	☐	☐	☐
PCG has enhanced my creative flexibility.	☐	☐	☐	☐	☐
PCG has made debugging or testing content	☐	☐	☐	☐	☐

Section E: Demographic Information

Team size: What best describes the size of your development team?

E1. **Team size: What best describes the size of your development team?**

- Solo developer (just me) ☐
- Small team (2–5 people) ☐
- Medium team (6–15 people) ☐
- Large team (16 or more people) ☐

E2.

Approximately how many years of game development experience do you have?

- Less than 1 year ☐
- 1–2 years ☐
- 3–5 years ☐
- 6–10 years ☐
- More than 10 years ☐

LimeSurvey

E3.

Genre focus: What is your primary game genre or focus?

- [ ] Action / Platformer
- [ ] Shooter (FPS / TPS)
- [ ] Adventure
- [ ] Role-playing (RPG)
- [ ] Simulation
- [ ] Strategy / Tower Defense
- [ ] Puzzle / Casual
- [ ] Dating Simulator

E4. **Would you like to stay anonymous in this survey?**

- [ ] Yes, I want to stay anonymous.
- [ ] No, I am comfortable providing my studios name in the comment

Thank you for completing my survey!

11.2. Participant Questionnaire - A Survey on Procedural Content Generation in Indie Game Development

LimeSurvey

Hi there! Let's start with a few quick questions about your background.

Section A: Pre-Experiment – Background & Experience

Before we begin, we'd like to learn a bit about your background and experience with Unreal Engine. Do this part before you take the experiment.

A1. Have you used Unreal Engine before?

- ☐ Yes
- ☐ No

A2. If yes, for how long have you been using Unreal Engine (approximately)?

- ☐ Less than 3 months
- ☐ 3–12 months
- ☐ 1–2 years
- ☐ More than 2 years

A3. How frequently do you work with the engine?

- ☐ Daily
- ☐ Several times a week
- ☐ Occasionally
- ☐ Rarely / Only for academic projects

A4. How familiar are you with the following topics?

	1 = no experience	2	3	4	5 = very experienced
Terrain Sculpting (Landscape Tool)	☐	☐	☐	☐	☐
Foliage Tool (vegetation placement)	☐	☐	☐	☐	☐
Blueprint or Editor Scripting	☐	☐	☐	☐	☐
PCG Graph (Procedural Content Graphs)	☐	☐	☐	☐	☐

Section B: Instructions

Please perform the experiment task now. When you are finished, click 'Next' to proceed.

B1.

LimeSurvey

Section C: Post-Experiment – Workflow Evaluation

Thank you for completing the tasks! Now, I would like to hear about your experience using the both workflows. Please reflect on how easy or difficult you found the tasks, how creative or satisfying the workflows felt, and your preferences.

C1. Manual Workflow Evaluation

Please rate the following statements:

	1 = strongly disagree	2	3	4	5 = strongly agree
I found the tools easy to use.	☐	☐	☐	☐	☐
I had a strong sense of creative control.	☐	☐	☐	☐	☐
The task was time-consuming.	☐	☐	☐	☐	☐
I would consider using this workflow in a real	☐	☐	☐	☐	☐

C2. PCG Graph Workflow Evaluation

Please rate the following statements about the PCG Graph Workflow:

	1 = strongly disagree	2	3	4	5 = strongly agree
I found the tools easy to use.	☐	☐	☐	☐	☐
I had a strong sense of creative control.	☐	☐	☐	☐	☐
The task was time-consuming.	☐	☐	☐	☐	☐
I would consider using this workflow in a real	☐	☐	☐	☐	☐

C3. Which workflow felt better to use overall?

- ☐ Manual
- ☐ PCG Graph
- ☐ No preference

C4. Which workflow felt more creative or satisfying to work with?

- ☐ Manual
- ☐ PCG Graph
- ☐ No preference

C5. Do you have any other feedback you would like to share?

LimeSurvey

Thanks you very much for your time and participation.

12. Quellenverzeichnis

Ahearn, L. (2008) *3D game environments: create professional 3D game worlds*. Burlington, MA: Focal Press/Elsevier.

Bortz, J. and Döring, N. (2006) *Forschungsmethoden und Evaluation: für Human- und Sozialwissenschaftler ; mit 87 Tabellen*. 4., überarb. Aufl., [Nachdr.]. Heidelberg: Springer-Medizin-Verl (Springer-Lehrbuch Bachelor, Master).

Burhöi, F.B. *et al.* (2017) 'Procedural Generation of a 3D Terrain Model Based on a Predefined Road Mesh', in. Available at: https://www.semanticscholar.org/paper/Procedural-Generation-of-a-3D-Terrain-Model-Based-a-Burh%C3%B6i-Gelotte/da45ff1a8a1eb27bf4ff1443120b3651fe0038d4#citing-papers (Accessed: 15 June 2025).

Cardamone, L. *et al.* (2011) 'Evolving Interesting Maps for a First Person Shooter', in C. Di Chio et al. (eds) *Applications of Evolutionary Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture Notes in Computer Science), pp. 63–72. Available at: https://doi.org/10.1007/978-3-642-20525-5_7.

Consalvo, M. and Paul, C.A. (2018) '"If you are feeling bold, ask for $3": Value Crafting and *Indie* Game Developers', *Transactions of the Digital Games Research Association*, 4(2). Available at: https://doi.org/10.26503/todigra.v4i2.89.

Cox, J. (2014) 'What Makes a Blockbuster Video Game? An Empirical Analysis of US Sales Data', *Managerial and Decision Economics*, 35(3), pp. 189–198. Available at: https://doi.org/10.1002/mde.2608.

Craveirinha, R. and Roque, L. (2015) 'Studying an Author-Oriented Approach to Procedural Content Generation through Participatory Design', in K. Chorianopoulos et al. (eds) *Entertainment Computing - ICEC 2015*. Cham: Springer International Publishing (Lecture Notes in Computer Science), pp. 383–390. Available at: https://doi.org/10.1007/978-3-319-24589-8_30.

Douglas Heaven (2016) *When infinity gets boring: What went wrong with No Man's Sky*, *New Scientist*. Available at: https://www.newscientist.com/article/2104873-when-infinity-gets-boring-what-went-wrong-with-no-mans-sky/ (Accessed: 16 June 2025).

Foliage Tool | Unreal Engine 4.27 Documentation | Epic Developer Community (2025) *Epic Games Developer*. Available at: https://dev.epicgames.com/documentation/en-us/unreal-engine/building-virtual-worlds-in-unreal-engine (Accessed: 25 June 2025).

Gervás, P. *et al.* (2005) 'Story plot generation based on CBR', *Knowledge-Based Systems*, 18, pp. 235–242. Available at: https://doi.org/10.1016/j.knosys.2004.10.011.

González-Duque, M. *et al.* (2020) 'Finding Game Levels with the Right Difficulty in a Few Trials through Intelligent Trial-and-Error'. arXiv. Available at: https://doi.org/10.48550/arXiv.2005.07677.

Khalifa, A. *et al.* (2019) 'Intentional computational *Level* design', in *Proceedings of the Genetic and Evolutionary Computation Conference*. *GECCO '19: Genetic and Evolutionary Computation Conference*, Prague Czech Republic: ACM, pp. 796–803. Available at: https://doi.org/10.1145/3321707.3321849.

Korn, O. *et al.* (2017) 'Procedural Content Generation for Game Props? A Study on the Effects on User Experience', *Computers in Entertainment*, 15(2), pp. 1–15. Available at: https://doi.org/10.1145/2974026.

Landscape Brushes in Unreal Engine | Unreal Engine 5.6 Documentation | Epic Developer Community (2025) *Epic Games Developer*. Available at: https://dev.epicgames.com/documentation/en-us/unreal-engine/landscape-brushes-in-unreal-engine (Accessed: 23 June 2025).

Lang, D.S. (2010) 'Empirische Forschungsmethoden'.

Level Designers Reveal Early Stages in 'Blocktober' Hashtag (2017). Available at: https://80.lv/articles/*Level*-designers-reveal-early-stages-in-blocktober-hashtag (Accessed: 25 June 2025).

Ma, C. *et al.* (2014) 'Game *Level* layout from design specification', *Computer Graphics Forum*, 33(2), pp. 95–104. Available at: https://doi.org/10.1111/cgf.12314.

Minecraft - Procedural Content Generation Wiki (2012). Available at: http://*PCG*.wikidot.com/*PCG*-games:*Minecraft* (Accessed: 15 June 2025).

Parberry, I. (2014) 'Designer Worlds: Procedural Generation of Infinite Terrain from Real-World Elevation Data', 3(1).

Pereira de Araujo, R. and Souto, V.T. (2017) 'Game Worlds and Creativity: The Challenges of Procedural Content Generation', in A. Marcus and W. Wang (eds) *Design, User Experience, and Usability: Designing Pleasurable Experiences*. Cham: Springer International Publishing, pp. 443–455. Available at: https://doi.org/10.1007/978-3-319-58637-3_35.

Phillips, A. *et al.* (2016) 'Feminism and procedural content generation: toward a collaborative politics of computational creativity', *Digital Creativity*, 27(1), pp. 82–97. Available at: https://doi.org/10.1080/14626268.2016.1147469.

Procedural Content Generation Overview | Unreal Engine 5.6 Documentation | Epic Developer Community (2025) *Epic Games Developer*. Available at: https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation-overview (Accessed: 27 June 2025).

Roden, T. and Parberry, I. (2004) 'From Artistry to Automation: A Structured Methodology for Procedural Content Creation', in M. Rauterberg (ed.) *Entertainment Computing – ICEC 2004*. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture Notes in Computer Science), pp. 151–156. Available at: https://doi.org/10.1007/978-3-540-28643-1_19.

Rodrigues, L., Bonidia, R. and Brancher, J. (2020) ‘Procedural versus human *Level* generation: Two sides of the same coin?’, *International Journal of Human-Computer Studies*, 141, p. 102465. Available at: https://doi.org/10.1016/j.ijhcs.2020.102465.

‘*Skyrim*: Radiant Quest System | Terminally Incoherent’ (2011), 16 December. Available at: https://www.terminally-incoherent.com/blog/2011/12/16/*Skyrim*-radiant-quest-system/index.html (Accessed: 20 June 2025).

Tait, E.R. and Nelson, I.L. (2022) ‘Nonscalability and generating digital outer space natures in No Man’s Sky’, *Environment and Planning E: Nature and Space*, 5(2), pp. 694–718. Available at: https://doi.org/10.1177/25148486211000746.

Wijaya, W. and Rahman, A. (2018) ‘Analisis Perbandingan Perlin Noise Dan Simplex Noise Untuk Penciptaan Permukaan Daratan Pada Pembuatan Game’, *Konferensi Nasional Sistem Informasi (KNSI) 2018* [Preprint]. Available at: https://jurnal.atmaluhur.ac.id/index.php/knsi2018/article/view/383 (Accessed: 15 June 2025).

Zheng, Y. (2024) ‘Brief Dreams: An Interactive Procedural Virtual World Builder for Imaginary Landscapes’.