FH Aachen

University of Applied Sciences

Fachbereich 05: Elektrotechnik und Informationstechnik

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Bachelorarbeit zum Thema:

Entwicklung einer prozeduralen Low-Poly-Asset-Bibliothek mit Blender Geometry Nodes

Zur Erlangung des Grades: Bachelor of Science (B. Sc.)

Vorgelegt von:
Joshua Battenfeld
Leydelstraße 3, 52064, Aachen
info@joshuabattenfeld.com
Matrikelnummer: 3580943

Erstprüfer: Prof. Dr.-Ing. Frank Hartung
Zweitprüfer: René Heß, M. A.

Abgabedatum: 26.11.2025
Studiengang: Media and Communications for Digital Business (MCD) B. Sc.

Kurzfassung

Die Erstellung von 3D-Modellen stellt in der Spieleentwicklung einen erheblichen Produktionsaufwand dar. Prozedurale Generierung verspricht hier Entlastung, indem sie die Modellierung über parametrisierbare, teil-automatisierte Systeme unterstützt.

Moderne 3D-Software bietet zunehmend Schnittstellen zur prozeduralen Inhaltserstellung. Dazu zählt auch das populäre Open-Source-Programm Blender, das seit 2021 mit den Geometry Nodes eine nodebasierte Oberfläche zur nicht-destruktiven, parametrischen Generierung von 3D-Inhalten bereitstellt.

In dieser Arbeit werden die Herausforderungen, Potenziale und Limitationen der Entwicklung und des Einsatzes einer prozeduralen Low-Poly-Asset-Bibliothek mit Blender Geometry Nodes untersucht.

Hierzu wird ein als Blender Add-on implementiertes Toolkit entwickelt, das aus modularen Geometry Node Setups besteht und die effiziente Erstellung stilisierter Low-Poly-Welten in einem mittelalterlichen Setting ermöglicht. Anschließend wird das System im Rahmen einer Nutzerevaluation getestet und sowohl der prozedurale Ansatz als auch die konkrete Implementierung kritisch diskutiert.

Die Arbeit schließt mit der Erkenntnis, dass prozedurale Asset-Bibliotheken in der richtigen Umsetzung und Game-Engine naher Implementation einen echten Mehrwert liefern können. Insgesamt verdeutlicht die Arbeit das bislang unterschätzte Potenzial zugänglicher prozeduraler Asset-Bibliotheken, insbesondere für Indie-Entwickler.

Abstract

The creation of 3D models represents a considerable production effort in game development. Procedural generation promises to ease this burden by supporting modeling via parameterizable, semi-automated systems.

Modern 3D software increasingly offers interfaces for procedural content creation. This includes the popular open-source program Blender, which since 2021 has provided a node-based interface for non-destructive, parametric generation of 3D content with its Geometry Nodes.

This thesis examines the challenges, potential, and limitations of developing and using a procedural low-poly asset library with Blender Geometry Nodes.

To this end, a toolkit implemented as a Blender add-on is developed, consisting of modular Geometry Node setups that enable the efficient creation of stylized low-poly worlds in a medieval setting. The system is then tested in a user evaluation, and both the procedural approach and the concrete implementation are critically discussed.

The thesis concludes with the finding that procedural asset libraries can deliver added value when implemented correctly and closely integrated with the game engine. Overall, the thesis highlights the previously underestimated potential of accessible procedural asset libraries, especially for indie developers.

Fachhochschule Aachen

Fachbereich 05 – Elektrotechnik und Informationstechnik

Studiengang: Media and Communications for Digital Business (MCD) B.Sc.

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Ort, Datum	Joshua Battenfeld

Inhaltsverzeichnis

1. Einleitung

Diese Arbeit fokussiert sich auf die Erforschung der Blender Geometry Nodes als System zur Erstellung einer prozeduralen Asset-Bibliothek für die Spieleentwicklung.

Im Rahmen dieser Arbeit wurde eine prozedurale Asset-Bibliothek für Blender entwickelt, die als Add-on realisiert ist und auf einer Reihe von Geometry Node Setups basiert. Das resultierende System, das Low-Poly-Tool-Kit (LPTK), ermöglicht die effiziente Erstellung stilisierter Low-Poly-Umgebungen in einem mittelalterlichen Setting.
Zentrales Ziel des Projekts war es, eine benutzerfreundliche und erweiterbare Oberfläche zu schaffen, die es Nutzern erlaubt, bereits mit grundlegenden 3D-Kenntnissen komplexe Szenen zu erstellen, zu modifizieren und in gängige Game-Engines zu exportieren.

Das LPTK wandelt einfache geometrische Formen oder Kurven in konsistente, optisch ansprechende 3D-Modelle um. Damit adressiert es eine zentrale Herausforderung der Spieleentwicklung: die Balance zwischen künstlerischer Qualität, Produktionsgeschwindigkeit und technischer Flexibilität.

Insbesondere in der Prototypen-Entwicklung werden häufig abstrakte Platzhaltermodelle verwendet, die zwar schnelle Iterationen ermöglichen, jedoch die visuelle Aussagekraft einschränken. Das LPTK setzt an dieser Stelle an, indem es die Effizienz von Greyboxing mit den gestalterischen Möglichkeiten prozeduraler Systeme verbindet. Somit können bereits in frühen Entwicklungsphasen visuell ansprechende Szenen erstellt werden, ohne den üblichen Mehraufwand klassischer Modellierung in Kauf nehmen zu müssen.

Darüber hinaus zielt das Toolkit darauf ab, die Einstiegshürde für Solo-Entwickler und kleine Teams zu reduzieren. Prozedurale Systeme übernehmen einen Teil der technischen Komplexität, sodass sich Entwickler oder Artists stärker auf die inhaltliche Gestaltung konzentrieren können, anstatt der technischen Umsetzung.

Ausgehend von der Forschungsfrage

„Welche Herausforderungen, Potenziale und Limitationen ergeben sich bei der Entwicklung einer prozeduralen Asset-Bibliothek auf Basis von Geometry Nodes und einer Add-on-basierten Interaktionsoberfläche?"

umfasst diese Arbeit drei zentrale Untersuchungsbereiche:

1. Das Potenzial prozeduraler Assets für eine effiziente und konsistente Spielweltgestaltung.
2. Die technische Umsetzung prozeduraler Assets mithilfe von Blender Geometry Nodes.
3. Die Integration der entwickelten Systeme in ein benutzerfreundliches Blender-Add-on.

2. Theoretischer Hintergrund

Bevor die konkrete Umsetzung der prozeduralen Asset-Bibliothek besprochen werden kann, müssen einige Grundlagen geklärt werden. In Kapitel 2 werden diese besprochen. Wobei zunächst das geplante Einsatzgebiet des LPTK, also (Indie)-Spieleentwicklung dann 3D-Modelle in diesem Kontext, der Low-Poly-Artstyle, Blender als Software und anschließend Prozedurale Ansätze, inklusive der Grundlagen von Blenders Geometry Nodes, besprochen werden.

2.1 Indie-Spielentwicklung

Indie-Spiele haben in den vergangenen Jahren erheblich an Bedeutung gewonnen und stellen den Großteil der jährlichen Spieleveröffentlichungen und im Jahr 2024 etwa die Hälfte des jährlichen Umsatzes durch Spielverkäufe über die Plattform Steam dar[1]. Unter „Indie" versteht man in der Regel Produktionen kleinerer Studios oder Einzelentwickler, die ohne die finanzielle und organisatorische Unterstützung großer Publisher realisiert werden. Typisch für dieses Segment sind niedrigere Budgets, kleinere Teams und ein hohes Maß an kreativer Freiheit.

Allerdings ist der Begriff „Indie" nicht eindeutig definiert. Manche Definitionen beziehen sich auf die Finanzierungsstruktur (keine Unterstützung durch Publisher), andere auf die Teamgröße oder die Unabhängigkeit in kreativen Entscheidungen. Entsprechend unterscheiden sich auch die zugrunde liegenden Statistiken zu Indie-Produktionen je nach Quelle und Erhebungsmethode. Während einige Studien ausschließlich die Finanzierungskriterien heranziehen, erfassen andere alle Produktionen außerhalb klassischer AAA-Studios. Der Begriff der AAA-, AA- und Indie-Studios ist hierbei aber immer fließend zu betrachten und nicht eindeutig greifbar, Weshalb Statistiken in diesem Bereich sich auch nicht immer auf die gleichen Spiele/Studios beziehen[2]. Unabhängig von der genauen Definition gilt: Indie-Spiele stellen ein zentrales Segment der Branche

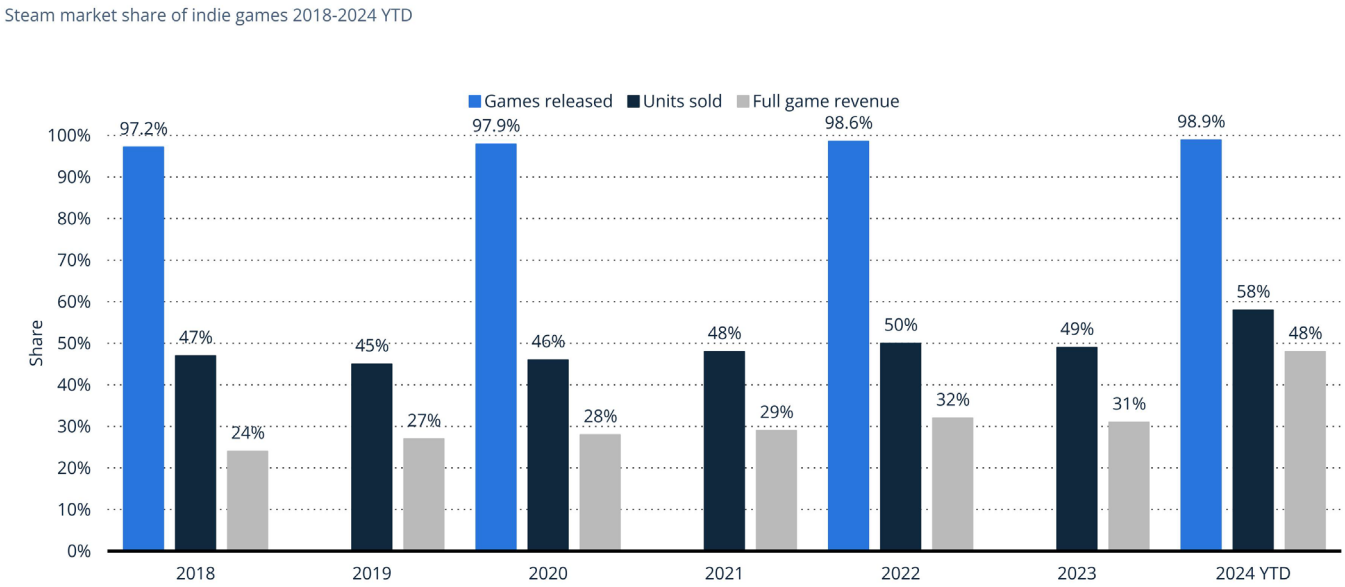


Abbildung 1: Marktanteil von auf Steam veröffentlichten Indie-Spielen von 2018 bis 2024 (Statista).

[1] [1].
[2] [2].

dar. 2024 waren zum Beispiel 98,9% aller Veröffentlichungen auf Steam Indie-Titel (Abbildung 1).

Aufgrund der meist kleinen Teamgrößen und fehlender Spezialisten entscheiden sich Indie-Entwickler deutlich häufiger für bestehende Softwarelösungen zur Spieleentwicklung, anstatt eigene technische Grundlagen wie Engines oder Frameworks zu entwickeln. Ein Blick auf entsprechende Branchenstatistiken zeigt, dass sich insbesondere kleinere Studios mit vergleichsweise geringen Verkaufszahlen (Abbildung 2) überproportional häufig für Unity als Game-Engine entscheiden.

Unity gilt damit im Indie-Bereich als besonders relevante Entwicklungsumgebung und prägt maßgeblich die Produktionsweise kleiner Teams.

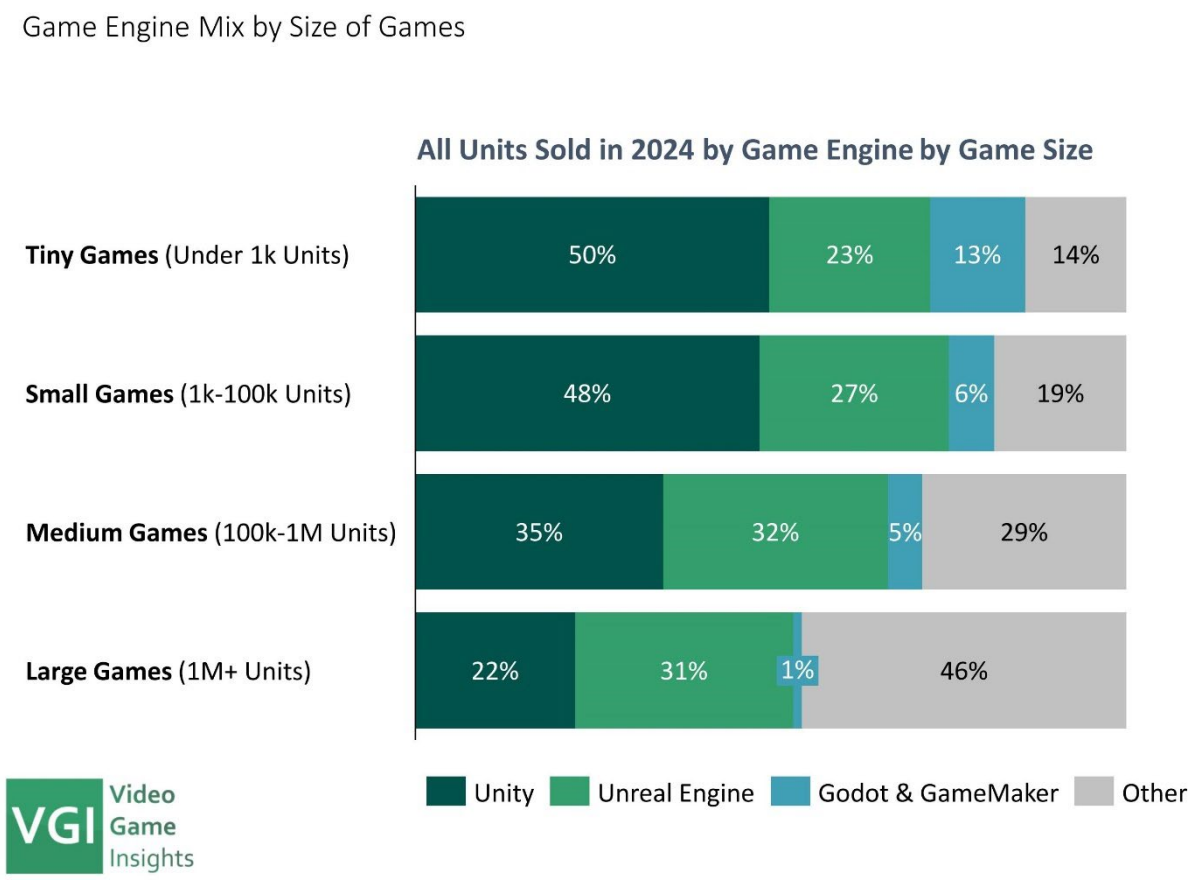


Abbildung 2: Game Engine Mix nach verkauften Einheiten [3].

2.2 3D-Modelle im Kontext der Spieleentwicklung

Im Kontext der Spieleentwicklung stellt die Erstellung visueller Inhalte in den meisten Fällen einen der größten Kostenfaktoren innerhalb der Produktionspipeline dar[3]. Seit Anfang der 2000er-Jahre dominieren 3D-Spiele den Markt, insbesondere bei den großen Produktionen im AA- und AAA-Segment.

Durch frei verfügbare Engines wie Unity, kostenlose Modellierungssoftware wie Blender und einen stetig wachsenden Asset-Markt können inzwischen jedoch auch Indie-Studios zunehmend 3D-Spiele realisieren. Abbildung 3 zeigt diesen Trend auf Basis der Wikipedia-Liste *„Indie games surpassing a million sales*", die für diese Arbeit in 2D- und 3D-Titel unterteilt wurde (Anhang A3).

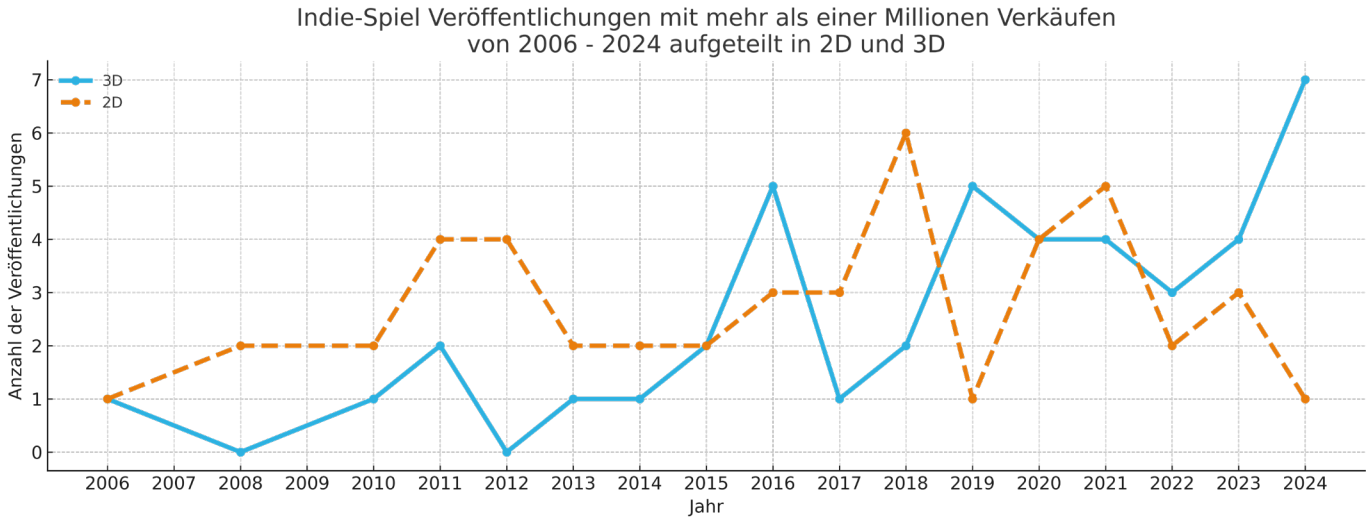


Abbildung 3: Entwicklung der veröffentlichten Indie-Spiele mit über einer Million Verkäufen von 2006 bis 2024, getrennt nach 2D- und 3D-Titeln. Die Darstellung zeigt die zunehmende Bedeutung von 3D-Produktionen im Indie-Sektor (eigene Darstellung).

Diese Entwicklung unterstreicht den wirtschaftlichen Stellenwert von 3D-Inhalten auch außerhalb des AAA-Bereichs. Modellierung, UV-Mapping, Texturierung, sowie die technische Aufbereitung für Echtzeit-Engines erfordern spezialisiertes Know-how und sind zeitintensiv.

Die Kosten von 3D-Modellen im, variieren je nach Genre und Art-Style stark von Spiel zu Spiel, stellen jedoch im Normalfall neben den Code und Game-Design einen der größten Kostenpunkte der 3D-Spieleentwicklung dar[4].

[3] [4, S. 1].
[4] [5, S. 783].

2.2.1 Polygonale Darstellung von 3D-Modellen

Es gibt verschiedene Methoden zur Repräsentation von dreidimensionalen Objekten. Im Kontext der Spieleentwicklung und Echtzeit-Computergrafik haben sich polygonale Modelle als Standard etabliert[5].

Polygonale Modelle bestehen aus sogenannten Meshes, also Polygonnetzen, welche die Oberfläche eines Objekts approximieren.

Ein solches Mesh setzt sich aus folgenden Grundelementen zusammen, welche in Abbildung 3 visualisiert sind:

- Eckpunkten (Vertices), rot dargestellt
    a. Punkte im dreidimensionalen Raum
- Kanten (Edges), grün dargestellt
    a. Verbindungen zwischen zwei Vertices
- Flächen (Faces), blau dargestellt
    a. geschlossene Flächen, die durch drei oder mehr Vertices gebildet werden

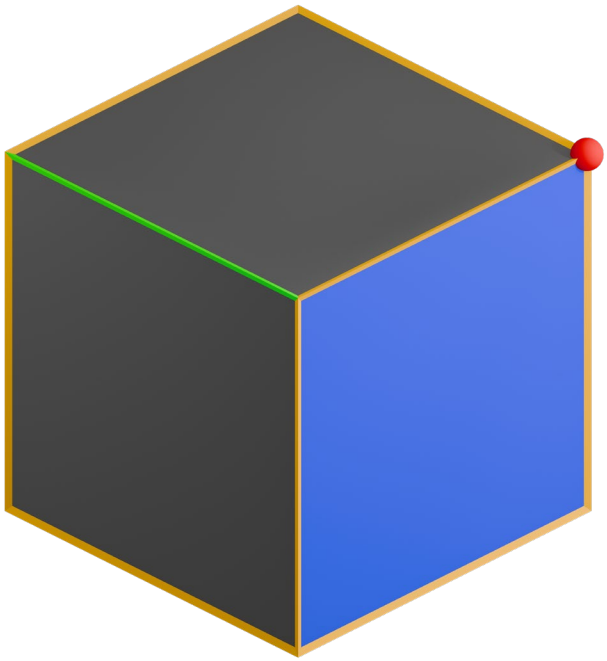


Abbildung 4: Polygonaler Würfel mit visualisiertem Vertex, Edge und Face (eigene Darstellung).

In modernen Produktionspipelines werden polygonale Modelle um weitere Komponenten wie Texturen, UV-Maps, Materialdefinitionen sowie Rigging- und Animationsdaten ergänzt.

Diese kombinieren sich zu vollständigen 3D-Assets, die in Game-Engines wie Unity oder Unreal Engine importiert und in Echtzeit dargestellt werden können. Im Rahmen der Arbeit beziehe ich mich im Kontext von 3D-Modellen oder 3D-Assets grundsätzlich auf polygonale Modelle.

Im folgenden Kapitel wird die Beschaffung solcher Modelle innerhalb der Spieleproduktion erläutert.

[5] [6, S. 1].

2.2.2 Beschaffung von 3D-Modellen

Grundsätzlich gibt es zur Erstellung der benötigten 3D-Assets verschiedene Methoden, mit verschiedenen Vor- und Nachteilen.

Im Folgenden werden die wichtigsten Ansätze kurz beschrieben:

- Händische Modellierung

Die klassische, manuelle Erstellung von Modellen in 3D-Software wie Blender, Maya oder 3ds Max. Sie bietet maximale Kontrolle über Form, Stil und technische Umsetzung, ist jedoch zeitaufwendig und entsprechend kostenintensiv[6].
Innerhalb dieser Kategorie existieren zahlreiche Unterformen, welche für spezielle Modellierung oder basierend auf Präferenz des Artists gewählt werden. Darunter beispielsweise:

  - Polygonale Modellierung

Modelle werden durch Manipulation einzelner Polygone aufgebaut, meist durch manuelle Extrusion, Skalierung und Verschiebung von Faces, Edges und Vertices.

  - Digitales Sculpting

Eine freiere, skulpturähnliche Methode, bei der Formen aus einer Basisgeometrie mithilfe von Werkzeugen zum Schieben, ziehen, glätten, greifen etc. erstellt werden. Wird häufig für organische Objekte wie menschliche Körper verwendet.

- KI-gestützte Modellgenerierung

KI-basierte Verfahren nutzen Machine-Learning-Modelle zur Erzeugung von Geometrie oder Texturen. Aktuelle Text-to-3D- und Image-to-3D-Ansätze wie Meshy, Rodin oder das Open-Source-Projekt Hunyuan-3D erzielen bereits beeindruckende Ergebnisse, weisen aber nach wie vor deutliche Schwächen in Bereichen wie Topologie[7], Retopologie und UV-Mapping auf und sind somit schwer in professionelle Workflows zu integrieren[8].
Entsprechend spielen KI-generierte Modelle derzeit noch eine untergeordnete Rolle in professionellen Workflows, werden aber zunehmend als unterstützende Werkzeuge eingesetzt.

- Prozedurale Modellierung

Beschreibt die regelbasierte, algorithmische Generierung von Geometrie.
Dieser Ansatz steht im Fokus dieser Arbeit und wird in Kapitel 2.5 ausführlich behandelt.

---

[6] [5, S. 783].
[7] Topologie bezieht sich bei Polygonalen 3D-Modellen auf die explizite Anordnung der Geometrie. Grundsätzlich ist hierbei das Ziel mit möglichst geringer Polygonanzahl einen möglichst hohen Detailgrad zu erzielen, also die verwendete Geometrie möglichst effizient zu nutzen.
[8] [5, S. 801].

- Photogrammmetrie:

Bezieht sich im 3D-Kontext auf die Übertragung von Objekten der physischen Welt in die digitale mithilfe von Bildern oder Scans und Photogrammetry Software. Diese Technik ist besonders relevant für die Erzeugung realistischer Assets.

- Hybride Workflows:

Innerhalb moderner Produktionspipeline werden häufig verschiedene Verfahren miteinander kombiniert. Beispielsweise durch die Generierung von Basismodellen mithilfe von KI oder prozedurale Systeme und anschließende Händische Überarbeitung[9] .

Während es keine genauen Zahlen bzgl. der Nutzung dieser Methoden in der Videospiel-Industrie gibt, dominiert in der Praxis laut verschiedenen Quellen und aus eigener Branchenerfahrung weiterhin die manuelle Modellierung, da sie maximale kreative Kontrolle und unmittelbares Feedback erlaubt[10].

Wichtig zu erwähnen sind auch Asset-Packs, die zwar keine Form der Erstellung, aber dennoch eine zentrale Möglichkeit zur Beschaffung von 3D-Modellen in der Spieleentwicklung darstellen.
Gerade kleinere Produktionen, die über keine oder wenige dedizierte Artists verfügen, greifen häufig auf Sammlungen vorgefertigter Assets zurück, die thematisch und stilistisch aufeinander abgestimmt sind.
Diese Vorgehensweise spart Zeit und Kosten, reduziert jedoch die gestalterische Freiheit und Individualität der Projekte. Das Vermischen verschiedener Asset-Packs (Kitbashing) kann dabei ebenfalls schnell zu stilistischen Inkonsistenzen führen.

Es existieren zahlreiche Wege, 3D-Modelle zu erstellen oder zu beschaffen. Unabhängig von der gewählten Methode können sich die resultierenden Modelle in Stil, Detailgrad und technischer Umsetzung stark voneinander unterscheiden. Im folgenden Kapitel wird die Low-Poly-Ästhetik behandelt, die einen spezifischen, stark stilisierten Ansatz der 3D-Modellierung beschreibt.

[9] [7, S. 120].
[10] [7, S. 118].

2.3 Low-Poly Artstyle

Unter Low-Poly versteht man im Kern die Verwendung von 3D-Modellen mit geringer Polygonanzahl. Dabei lassen sich jedoch zwei unterschiedliche Bedeutungen unterscheiden:

1. Technisches Low-Poly
   In der 3D-Grafik werden Modelle häufig in vereinfachter Form eingesetzt, um Rechenleistung zu sparen und eine flüssige Darstellung zu gewährleisten. Beispielsweise bei der Verwendung sogenannter Level of Detail (LOD)-Modelle[11], bei denen mit zunehmender Entfernung zum Betrachter ein Objekt durch eine weniger detaillierte Version ersetzt wird. Solche Low-Poly-Modelle entstehen also aus Gründen der Optimierung und dienen primär der Performance-Steigerung.

2. Stilistisches Low-Poly (Artstyle)
   Davon abzugrenzen ist der bewusste Einsatz von Low-Poly-Formen als künstlerische Stilrichtung. Dieser Ansatz hat seine Wurzeln zwar in den technischen Limitierungen der 1990er-Jahre, wurde aber in den späten 2010er-Jahren bewusst als ästhetische Entscheidung in verschiedenen Medien wieder aufgegriffen[12] und durch Spiele wie „Superhot"(2016), „Poly Bridge"(2016) oder „Besiege"(2015) im Mainstream verbreitet.

   Anders als beim technischen Low-Poly steht hier nicht die Optimierung, sondern die Stilisierung im Vordergrund. Low-Poly wurde aufgrund der technisch bedingten Vergangenheit häufig als minderwertig angesehen ist aber heutzutage mehr als etabliert in der Szene.

Diese Arbeit bezieht sich mit dem Low-Poly-Begriff auf den Low-Poly-Artstyle und nicht auf den primär technischen bedingten Begriff.

Innerhalb des Low-Poly-Artstyles haben sich verschiedene visuelle Ausprägungen etabliert, die sich im Grad der Abstraktion, im Umgang mit Farben sowie in der Detailtiefe unterscheiden. Eine der populärsten Stilrichtungen ist der von Synty Studios geprägte Low-Poly-Look. Dieser Stil zeichnet sich durch folgende Merkmale aus:

Flat Shading ohne ausgeprägte Licht- und Materialeffekte, klare, gesättigte Farben, minimale oder vollständig fehlende Texturen, häufig einfache Farbflächen, eine cartoonartige, stilisierte Formsprache, reduzierte, aber liebevoll gestaltete Details, die trotz geringer Polygonanzahl eine hohe Lesbarkeit gewährleisten

Synty Studios prägt diesen Stil seit vielen Jahren maßgeblich und bietet umfangreiche Low-Poly-Asset-Pakete in zahlreichen Themenbereichen an. Diese erfreuen sich insbesondere bei Indie-Entwicklern großer Beliebtheit und gehören im Unity Asset Store regelmäßig zu den meistverkauften Paketen. Im folgenden Kapitel prüfen wir diese Annahmen.

[11] [7, S. 123].
[12] [8, S. 1].

Abbildung 5: Mit LPTK erstelltes Terrain (eigene Darstellung).

2.3.1 Gründe für Low-Poly im LPTK

Für die Erforschung der Blender Geometry Nodes bietet sich der Low-Poly-Artstyle aus mehreren Gründen an. Ein zentraler Aspekt ist die starke in 2.1 beschriebene Relevanz von Unity im Indie-Segment. Wie zuvor dargestellt, greifen viele kleine Studios und unabhängige Entwickler auf Unity zurück. Innerhalb des Unity Asset Store wiederum zählen Low-Poly-Assets seit Jahren zu den beliebtesten und meistverkauften Inhalten (Abbildung 4). Diese Beliebtheit unterstreicht, dass der Stil im Indie-Bereich weit verbreitet und akzeptiert ist.

Eine Auswertung (Anhang A4) der 100 meistverkauften Assets im Unity Asset Store (Abbildung 6) zeigt die Relevanz der Low-Poly-Assets.

1. Ring 1: Art-Assets bilden mit 46,0% die größte Kategorie und spiegeln die höchste Kaufbereitschaft für vorgefertigte Inhalte wider.
2. Ring 2: Innerhalb der Art-Kategorie stellen 3D-Modelle mit 39,1% die wichtigste Untergruppe dar.
3. Ring 3: Die entscheidende Erkenntnis liefert der äußere Ring: 72,2% dieser kommerziell erfolgreichen 3D-Assets setzen auf einen Low-Poly-Look.

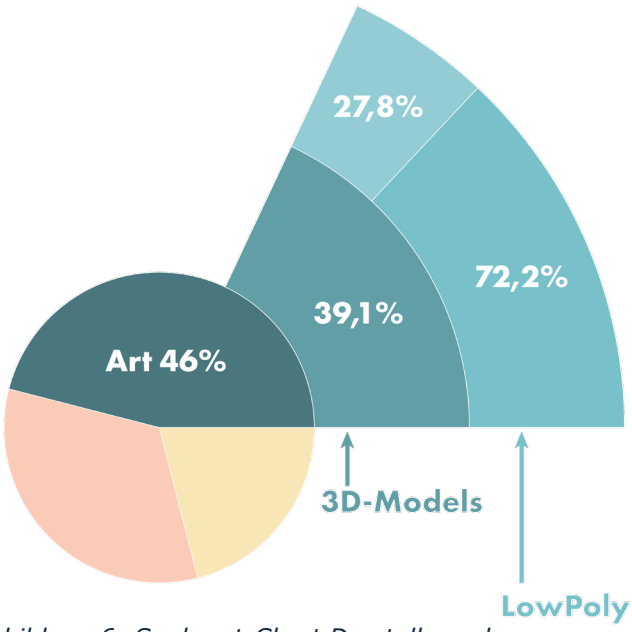


Abbildung 6: Sunburst-Chart Darstellung der " Top 100 paid Assets", 30.09.2025 (eigene Darstellung) Quelle der Daten in A4.

Darüber hinaus liegt der Schwerpunkt im Low-Poly-Stil stärker auf der Geometrie der Modelle, während aufwendige Materialien und komplexe PBR-Texturen in den Hintergrund treten. Dies macht den Ansatz besonders geeignet für die Erforschung und Umsetzung prozeduraler Modellierung in Blender Geometry Nodes. Gleichzeitig reduziert sich dadurch die Fehleranfälligkeit beim Export in externe Game Engines. Komplexe Shader-Setups, UV-Mapping oder Materialkombinationen, die häufig zu Problemen führen können, spielen im Low-Poly-Kontext durch die simplen Materialien eine deutlich geringere Rolle.

Schließlich fließt in die Entscheidung für Low-Poly für das LPTK auch meine jahrelange Erfahrung im Bereich der Low-Poly-Modellierung ein. Die Vertrautheit mit den typischen Anforderungen und Workflows ermöglicht es, ein praxisnahes Werkzeug zu entwickeln, das sich gezielt an den Bedürfnissen von Indie-Entwicklern orientiert. Zur Erstellung solcher Modelle haben sich, wie in Kapitel 2.2.2 beschrieben, verschiedene Softwarelösungen etabliert.
Eine im Indie-Kontext besonders bedeutende ist Blender, auf die im folgenden Kapitel näher eingegangen wird.

2.4 Blender im Indie-Spielentwicklungs Kontext

Blender wurde 1998 veröffentlicht und ist ein generalisiertes 3D-Softwarepaket, welches eine Vielzahl an Funktionen für unterschiedliche Branchen und Anwendungsfelder bietet. Besonders seit der Umstellung zur Open-Source Lizenz im Jahr 2002 wächst Blender stetig und hat besonders in den letzten Jahren erheblich an Relevanz im 3D-Bereich gewonnen (Abbildung 7).

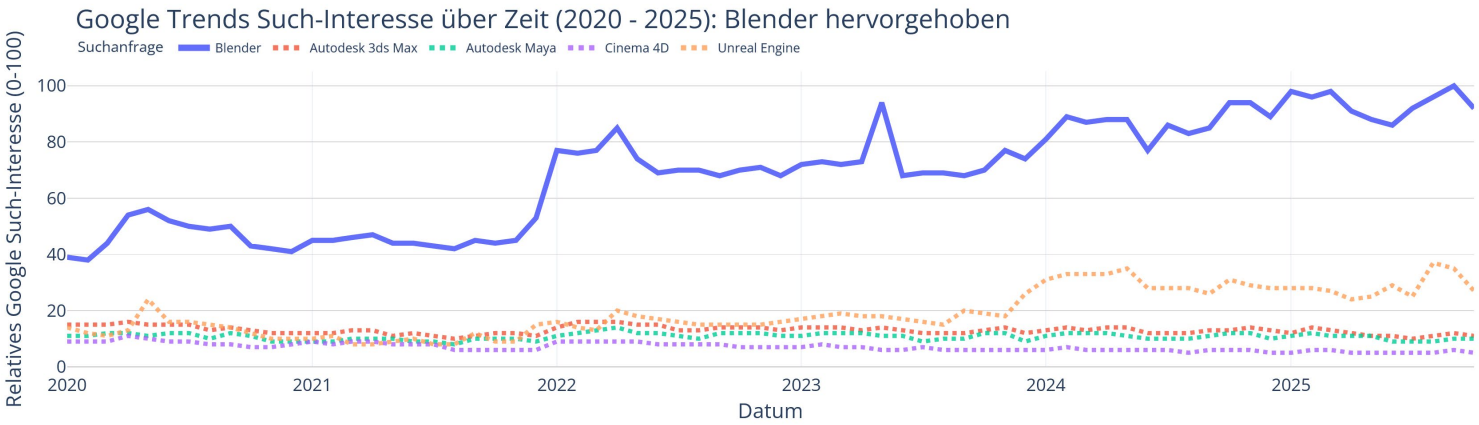


Abbildung 7: Google Trends Such-Interesse Populärer 3D-Programme, Blender Hervorgehoben. Datenquelle: Google Trends, Suchbegriffe im Zeitraum 01.01.2020 – 24.10.2025 (eigene Darstellung).

Dieser Zuwachs lässt sich sowohl durch den breiten Funktionsumfang als auch durch die niedrige Einstiegshürde und den freien Zugang erklären.

Blender bietet Oberflächen zur Modellierung, Animation, Texturierung, Rigging, UV-Mapping etc. und deckt die meisten Bedürfnisse an eine moderne 3D-Pipeline in einem Tool ab. Besonders im Indie-Segment und in kleineren Studios hat sich Blender als zentrales Werkzeug etabliert[13] .
Die Kombination aus Kostenfreiheit, einer aktiven Entwickler-Community und Integration moderner Werkzeuge, wie der Geometry Nodes macht es zu einer attraktiven Alternative zu kommerziellen Lösungen.

Gerade für Low- oder No Budget Produktionen ist Blender die einzige Möglichkeit und spart enorme Kosten. Der Einsatz vom weitverbreiteten Modellierungs- und Animations-Standard „Autodesk Maya" verursacht beispielsweise jährliche Lizenzkosten von 2119 € pro Nutzer[14] und ist für kleine Teams ohne Budget nicht möglich.
Was Blender durch seine Open-Source-Natur im direkten Kundensupport gegenüber kommerzieller Alternativen fehlt, kompensiert es durch seine sehr aktive und offene Community. Es gibt zu beinahe jeder Frage eine Antwort oder ein passendes Tutorial, wodurch die Einstiegshürde sowohl finanziell als auch im Nutzungskontext deutlich niedriger ist als bei den kostenpflichtigen Alternativen.
Auch für Forschungsprojekte, bei denen Flexibilität, Anpassbarkeit und Transparenz im Vordergrund stehen, bietet Blender durch seine offene Architektur klare Vorteile und lässt sich vergleichsweise einfach erweitern, beispielsweise durch selbst erstellte Python-Skripte oder Addons.

[13] [9].
[14] [10].

2.4.1 Blender Add-ons

Blender bietet im „Scripting"-Tab die Möglichkeit, mithilfe der Blender Python API (bpy)[15] Python-Skripte direkt im Editor auszuführen oder eigene Erweiterungen zu entwickeln. Diese Skripte werden in Form von sogenannten Modulen erstellt und können bestimmte Funktionen oder komplexe Abläufe automatisieren.

Ein einfaches Beispiel für ein Skript, welches alle Objekte in der aktuellen Szene verschiebt[16] :

```
1. import bpy # importiert das Blender Python API Modul
2.
3. scene = bpy.context.scene # setzt aktuelle Szene in Blender
4. for obj in scene.objects: # Schleife durch alle Objekte in der Szene
5.     obj.location.x += 1.0 # Verschiebung aller Objekte um eine Einheit entlang der X-Achse
```

Add-ons bauen auf dieser Funktionalität auf. Sie erlauben es mehrere Skripte zu einer strukturierten Erweiterung zusammenzufassen und ermöglichen eine direkte Integration in Blenders Benutzeroberfläche. Dadurch können Entwickler und Technical Artists den Funktionsumfang von Blender gezielt erweitern und an spezielle Workflows anpassen.

Im Kontext professioneller Workflows sowie spezialisierter Anwendungen stellen Add-ons ein zentrales Werkzeug zur Erweiterung der Funktionalität von Blender dar. Community-erstellte Add-ons tragen neben direkten Quellcode-Beiträgen wesentlich zur kontinuierlichen Weiterentwicklung der Software bei[17]. Besonders relevante Open-Source-Add-ons werden mitunter direkt in die Standarddistribution von Blender integriert und als native Erweiterungen bereitgestellt, prominente Beispiele sind Add-ons wie LoopTools[18] oder der NodeWrangler[19].

Neben Open-Source-Lösungen existiert ein breites Spektrum kommerzieller Add-ons, die spezifische Anwendungsprobleme lösen und über Drittanbieter vertrieben werden. Der größte Marktplatz für Blender-Erweiterungen ist Superhive (ehemals Blender Market), über den eine Vielzahl sowohl kommerzieller als auch frei verfügbarer Add-ons angeboten wird.

Die Installation von Add-ons ist sehr einfach und kann über das interne Erweiterungs-Panel von Blender erfolgen, was primär für Open-Source-Add-ons vorgesehen ist, oder alternativ manuell durch das Einfügen der entsprechenden Dateien der lokalen Festplatte. Diese Flexibilität erlaubt es Anwendern, die Softwareumgebung gezielt an spezifische Anforderungen anzupassen.

---

[15] [11].
[16] [12].
[17] [13].
[18] LoopTools, fügt mehrere Modellierungswerkzeuge hinzu: https://extensions.blender.org/add-ons/looptools/?utm_source=blender-4.5.3-lts.
[19] [14].

2.5 Procedural Content Generation

Procedural content generation (PCG) in Videospielen beschreibt die algorithmische Generierung von Spielinhalten (Game-Assets) mit limitiertem oder indirekten Nutzerinput[20].

PCG ist in der Videospiel-Entwicklung weit verbreitet und bezieht sich auf verschiedenste Arten von Inhalten. Beispiele für PCG reichen von prozeduralen Shadern und Materialsystemen, über die algorithmische Erzeugung von Meshes und Landschaften bis hin zu kompletten Spielwelten. Darüber hinaus können beispielsweise auch Musik, Animationen oder Partikelsysteme durch prozedurale Verfahren erzeugt werden[21].

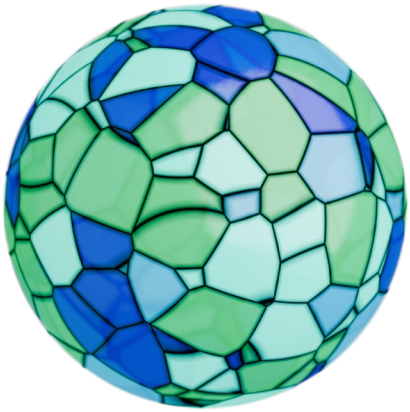


Abbildung 8, Prozeduraler Shader für Vornoi-basierte Glasmalerei (eigene Darstellung).

Seine Ursprünge hat PCG Anfang der 1980er Jahre. Spiele wie „Rogue" (1980) und „Elite" (1984) werden in diesem Kontext häufig als Vorreiter genannt[22]. Damals war PCG und vor allem die prozedurale Levelgenerierung für openworld-artige Spiele als eine Art Kompressionstechnik unersetzlich[23] . Zur damaligen Zeit war es unmöglich, große Mengen an vordefinierten Daten dauerhaft zu speichern. So wären die Entwickler von Elite nicht in der Lage gewesen acht spielbare Galaxien mit jeweils 256 vordefinierten Planeten auf der originalen „BBC Micro"-Diskette speichern können.

Seither hat sich PCG zu einem zentralen Bestandteil moderner Spieleentwicklung entwickelt und findet sich in beinahe allen aktuellen Titeln wieder, wobei die genaue Implementation und Nutzungsweisen sich komplett voneinander unterscheiden können.

So nutzt „*The Elder Scrolls IV: Oblivion*" (2006) prozedurale Systeme, um die Spielwelt mit Basis-Vegetation zu füllen, welche im Anschluss manuell von Artists bearbeitet wird. „Minecraft" (2009) hingegen generiert seine Spielwelten vollständig prozedural. Borderlands (2009) wiederum verwendet prozedurale Generierung um 17 Millionen verschieden Waffentypen mit unterschiedlichen Eigenschaften zu erzeugen.

Im Zuge dieser Arbeit steht die prozedurale Modellierung im Vordergrund, welche als eine zentrale Unterkategorie von PCG zu verstehen ist und sich auf die algorithmische Generierung und Manipulation von Geometrien bezieht.

[20] [15, S. 14].
[21] [16, S. 62].
[22] [15, S. 4].
[23] [17, S. 502].

2.5.1 Prozedurale Modellierung

Wie in 2.2.2 angesprochen ist die prozedurale Modellierung ein wichtiger Ansatz zur Geometrieerzeugung im Kontext der modernen Spieleentwicklung[24]. Im Gegensatz zur manuellen Modellierung beschreibt sie die Generierung und Manipulation von 3D-Geometrie auf Basis von definierten Regeln, Algorithmen und Parametern. Sie erlaubt es, komplexe Strukturen wie Gebäude, Vegetation oder ganze Landschaften effizient und reproduzierbar zu erzeugen[25].

Der Begriff an sich wird für eine Vielzahl unterschiedlicher Methoden und Systeme verwendet und bezieht sich dabei ebenso auf einfache prozedurale Modifikationen von handmodellierten Basisgeometrien als auch die automatische Erzeugung hoch komplexer Terrains.

Die Prozedurale Modellierung hat sich in den letzten Jahren stetig weiterentwickelt und es wurden verschiedenen Ansätze zur Generation verschiedener Objektetypen entwickelt. Diese werden dabei aber durch ihre parametrische und non-destruktive Natur vereint.

Historisch betrachtet, gibt es verschieden Wegweisende Ansätze. Fundamental sind dabei beispielsweise die 1968 von Aristid Lindenmayer eingeführten L-Systeme, welche zur Erforschung pflanzlicher Wachstumsprozesse entwickelt wurden und durch iterative Anwendung einfacher Regeln komplexe Strukturen erschaffen können[26].

In diesem Kontext ebenfalls häufig erwähnt, sind die Shape Grammars, welche häufig zur Erzeugung räumlicher Geometrien, bspw. im Architektur-Kontext, verwendet werden[27].

Während frühe Implementation überwiegend textuell, skript-oder codebasiert waren[28], haben aktuelle Tools den Fokus zunehmend auf visuelle und oder nodebasierte Workflows verschoben. Diese ermöglichen es, prozedurale Systeme interaktiv, modular und zugänglich zu gestalten, stehen in Form von Blender Geometry Nodes im Fokus dieser Arbeit und werden 2.5.5 genauer eingeführt.

Zunächst werden die Vor- und Nachteile des prozeduralen Ansatzes besprochen.

[24] [18].
[25] [18].
[26] [19, S. 1].
[27] [20, S. 615].
[28] [21].

2.5.2 Vor- und Nachteile prozeduraler Systeme

Prozedurale Verfahren bieten gegenüber der klassischen, manuellen Erstellung von Assets eine Reihe signifikanter Vorteile.

Sie ermöglichen eine effiziente und skalierbare Generierung großer Mengen an Inhalten die, wie in 2.2 beschrieben, einen zentralen Kostenfaktor der Spieleentwicklung darstellen. Ein einmal aufgesetztes System kann theoretisch unendlich viele Varianten eines 3D-Modells, beispielsweise eines Levels oder Baumes, erzeugen[29].

Darüber hinaus bieten prozedurale Systeme non-destruktive Workflows, bei denen Änderungen an Parametern jederzeit vorgenommen werden können, ohne die zugrunde liegende Struktur dauerhaft zu verändern. Dadurch lassen sich Varianten schnell erzeugen und Anpassungen effizient durchführen.

Trotz dieser Stärken stehen prozedurale Systeme vor verschiedenen Herausforderungen.

Die Entwicklung eines funktionierenden Regelwerks ist komplex und erfordert eine sorgfältige Definition der Generierungslogik, um konsistente und ästhetisch überzeugende Ergebnisse zu erzielen. Zur definiton ist eine Kombination aus künstlerischer und technischer Kompetenz erforderlich[30], wodurch eine große Einstiegshürde entsteht.

Die manuelle Erstellung eines einzelnen Assets ist im Regelfall schneller als das Aufsetzen eines komplexen Systems, welches dieses Asset automatisch generieren könnte. Auch wenn es theoretisch möglich ist, sollte nicht jedes Asset mit einem prozeduralen System erzeugt werden. Das *„Ten Thousands Bowls of Oatmeal Problem*" wird in diesem Kontext häufig genannt und soll zeigen, dass die unendliche Variation eines uninteressanten Assets, das Objekt nicht interessanter macht[31].

Darüber hinaus neigen PCG-Systeme dazu, wiedererkennbare Muster zu erzeugen welche von Spielern erkannt werden können. Ebenso schränken sie die künstlerische Kontrolle ein, da spezifische Änderungen in den meisten System nicht leicht zu definieren sind.

Trotz dieser Einschränkungen gilt prozedurale Modellierung heute als zentrale Technologie für skalierbare, wiederverwendbare und effizient produzierte Assets. Moderne Node-basierte Systeme wie Houdini oder Blender Geometry Nodes bieten inzwischen Möglichkeiten, diese Verfahren intuitiv zu gestalten und gezielt mit manuellem Design-Input zu kombinieren. Dieser hybride Ansatz, aus algorithmischer Generierung und künstlerischer Kontrolle, wird in der Forschung als „mixed authorship" bezeichnet und im folgenden Kapitel näher betrachtet.

[29] [18].
[30] [17, S. 513].
[31] [22, S. 3].

2.5.3 Automatic Generation versus Mixed Authorship

Für Spiele mit nahezu unendlichen Open-Worlds, wie beispielsweise das bereits erwähnte Minecraft (siehe 2.5), ist prozedurale Generierung unumgänglich. Diese benötigen ein Runtime PCG-System. Runtime-Systeme generieren Inhalte dynamisch auf dem Gerät des Nutzers bevor und/oder während der Spieler das Programm ausführt und die Welt erkundet. Diese Implementation funktioniert also ohne nachträglichen Design-Input, muss autonom spielbare Welten erzeugen und wird in der Literatur als „Automatic generation" bezeichnet[32] .

In der Realität benötigen allerdings nur wenige Spiele diese vollständige Prozeduralität zur Laufzeit. Die meisten Titel setzen auf vordefinierte Levels, welche größtenteils händisch von Designern und Artists entworfen und umgesetzt werden.
Aber auch hier können prozedurale „Design-Time"-Systeme[33] auf eine spannende Weise eingesetzt werden. Hierbei geben Designer oder Spieler gezielten Input, welcher durch die prozedurale Logik umgewandelt wird. In der wissenschaftlichen Literatur, beispielsweise in *Procedural Content Generation in Games* (2017), wird dieses Paradigma als „mixed authorship" definiert[34].

Dieses Prinzip wird in Abbildung 9 beispielhaft visualisiert. Die Basis ist der nicht eingefärbte Bereich des Terrains, der mithilfe der exponierten Parameter des Systems erzeugt wurde. Die grün überlegten Meshes sind händisch vom Designer hinzugefügte Geometrien, welche dem prozeduralen Base-Mesh als Union-Boolean-Operation hinzugefügt werden. Die rot überlegten Meshes sind ebenfalls manuell erstellte Geometrien, welche dem prozeduralen Terrain (samt der Union-Meshes) als Difference-Boolean abgezogen werden. So kann der Designer aktiv auf das prozedurale Terrain aufbauen.

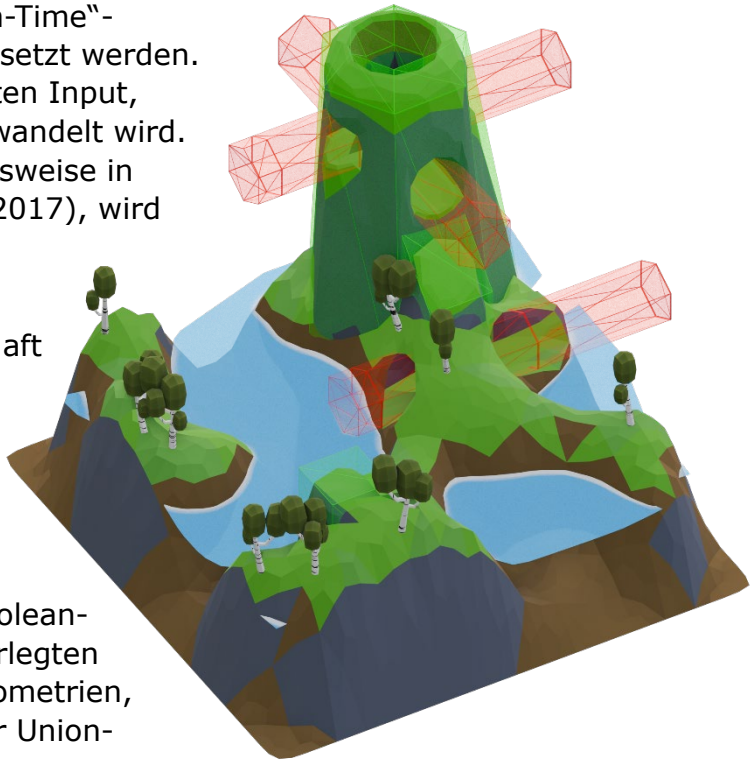


Abbildung 9: Beispielhafte Darstellung des ‚ProceduralTerrain' Systems des LPTK mit visualisierten Boolean-Meshes (eigene Darstellung).

Die in 2.5.2 besprochenen Nachteile von begrenzter Kontrolle, inkonsistenter Qualität und repetitiven Mustern werden hierbei durch die Möglichkeit zur manuellen Editierung von Designern und Artists mit minimalem Aufwand umgangen.

---

[32] [15, S. 10].
[33] Gegenstück zu Runtime-Systemen. Sie werden während der Level-Erstellung genutzt, die resultierenden Assets sind zur Laufzeit jedoch statisch.
[34] [15, S. 10].

2.5.4 Moderne Node-Based-Tools

Für die prozedurale Modellierung haben sich in den letzten Jahren zunehmend visuelle, Node-basierte Systeme etabliert. Diese ermöglichen es, komplexe Abläufe nicht ausschließlich über Code, sondern über visuell verbundene Funktionsblöcke abzubilden. Dadurch können auch Artists und insbesondere Technical Artists ohne tiefgehende Programmierkenntnisse prozedurale Systeme erstellen, verstehen, anpassen und erweitern.
Da die meisten prozeduralen Systeme visuelle Ergebnisse erzeugen, sind technische und künstlerische Aspekte eng miteinander verknüpft[35] . Dies hat zur starken Etablierung Node-basierter Workflows geführt. Vergleichbar mit Shader-Graph-Systemen, die ebenfalls von einer visuellen Darstellung komplexer Zusammenhänge profitieren.

Node-basierte Systeme bieten eine Reihe von Vorteilen gegenüber klassischen Skript- oder Code-basierten Lösungen. Der Aufbau aus einzelnen, modularen Funktionsknoten ermöglicht non-lineares Arbeiten, einfache Wiederverwendung von Teilen eines Setups und eine hohe Transparenz im Entstehungsprozess. Dadurch lassen sich selbst komplexe Beziehungen zwischen Eingabeparametern und Ausgaben visuell nachvollziehen.

2.5.4.1 Houdini als Industriestandard

Das bekannteste und am weitesten entwickelte System in diesem Bereich ist Houdini von SideFX.
Houdini gilt in der VFX- und Game-Industrie als absoluter Industriestandard für prozedurale Modellierung und ist in den meisten großen AAA-Studios im Einsatz. Das gesamte Programm basiert auf einem Node-Graph-Prinzip, das alle Bereiche von Geometrieerzeugung über Partikelsimulationen bis hin zu Materialsystemen miteinander verbindet.
Ein zentrales Merkmal ist die Houdini Engine, welche die direkte Integration prozeduraler Assets in Game-Engines wie Unreal und Unity ermöglicht[36]. Dadurch können Artists prozedurale Assets außerhalb von Houdini kontrollieren, Parameter anpassen und Änderungen direkt in der Engine sichtbar machen.

2.5.4.2 Spezialisierte Lösungen

Neben Houdini existieren auch einige industrierelevante spezialisierte Anwendungen, die auf bestimmte Bereiche des PCG-Kosmos fokussiert sind. Beispiele sind World Machine[37] für Terrain-Generierung oder Material Maker[38] zur prozeduralen Material-Generierung. Diese Tools sind zwar leistungsfähig, aber stark auf ihren jeweiligen Anwendungsbereich limitiert. Generalisten wie Houdini oder Blender Geometry Nodes bieten eine deutlich höhere Anpassbarkeit und sind zur Erstellung einer prozeduralen Asset-Library mit Fokus auf Geometrie-Generierung quasi unumgänglich.

---

[35] [17, S. 513].
[36] https://media.sidefx.com/uploads/products/engine/hengine_games_2023.pdf
[37] https://www.world-machine.com/
[38] https://www.materialmaker.org/

2.5.5 Blender Geometry Nodes

Mit der Veröffentlichung von Blender 2.92 (2021) wurden die Geometry Nodes als Node-basiertes, prozedurales und non-destruktives System zur Erstellung und Manipulation von Geometrien eingeführt. Sie erweitern das bestehende Modifier-Konzept von Blender um eine visuelle Programmierebene, in welche Geometrie über einen Node Graph beschrieben werden kann[39].

Einzelne Node Setups können als Modifier Objekten hinzugefügt werden. Die Objektgeometrie durchläuft dabei den Modifier-Stack von oben nach unten. Jeder Geometry-Node-Tree kann in diesem Stack wie ein einzelner Modifier auftreten und erhält über die Group Input Node seine Eingabedaten. Innerhalb des Node-Trees definieren verschiedene Operation Nodes (z. B. Set Position, Extrude Mesh, Distribute Points on Faces) die eigentlichen prozeduralen Schritte. Über die Group Output Node wird die modifizierte Geometrie anschließend zurück an den Modifier-Stack übergeben.

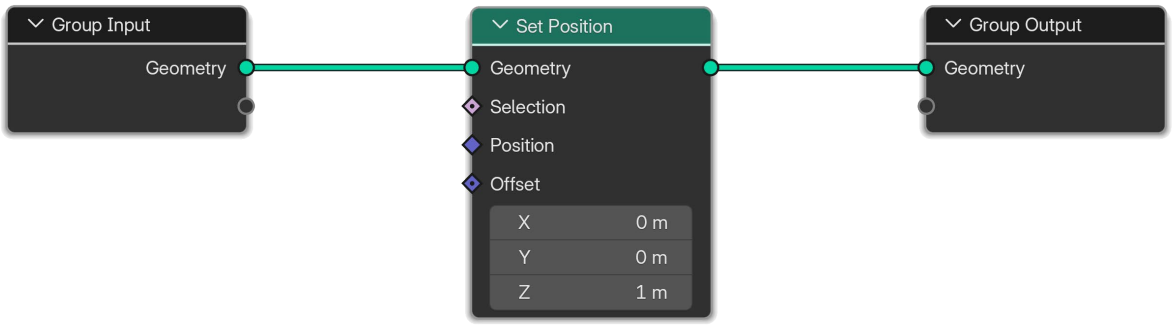


Abbildung 10: Beispielhafter Node Tree (eigene Darstellung).

Abbildung 10 zeigt beispielhaft, wie die Geometrie des Würfels, auf welchen der Geometry Nodes Modifier angewendet wurde aus der Group Input Node in das Socket[40] der Set Position Node gezogen wird. Über diese wird jeder Punkt der Geometrie um einen Meter entlang der Z-Achse verschoben und anschließend über die Group Output Node wieder in den Modifier Stack übergeben. Abbildung 11 visualisiert diese Veränderung. Der graue Würfel stellt die Geometrie vor der Set Position Node dar, der orangenen zeigt den Würfel nach der Operation.

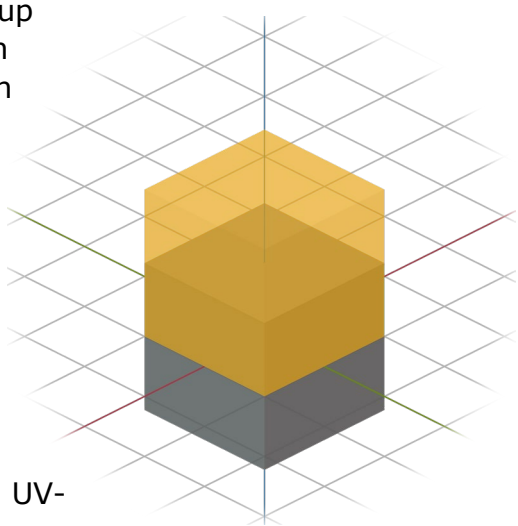


Abbildung 11: Visualisierung des Effekts der in Abbildung 10 gezeigten Set Position Node auf einem Würfel. Grauer Würfel vor, oranger nach der Set Position Operation (eigene Darstellung).

Die in die Group Input Node eingespeiste Geometrie umfasst dabei mehr als nur die reinen Positionsdaten der einzelnen Vertices. Sie stellt ein Datenpaket dar, welches sämtliche Attribute des Objekts wie Materialzuweisungen, UV-Koordinaten, Normalen oder benutzerspezifische Daten enthält. Diese Attributebene ist die Basis der prozeduralen Logik.

[39] Innerhalb dieser Arbeit wird während der konkreten Beschreibung der Geometry Nodes eine Vielzahl an englischen Fachbegriffen verwendet. Um den Lesefluss zu erhalten, wurde auf eine kursive Markierung dieser Begriffe bewusst verzichtet.
[40] Sockets sind die In- und Outputs einer Node.

2.5.5.1 Das Attribut-Konzept

Um Geometry Nodes genauer zu verstehen, ist das Konzept der Attribute grundlegend.

Innerhalb der Geometry Nodes sind Attribute ein generischer Begriff zur Beschreibung eines pro Element gespeicherten Daten-Blocks[41]. Attribute sind die Basis der prozeduralen Manipulation. Sie ermöglichen es Daten gezielt zu lesen, zu modifizieren und neu zu schreiben.

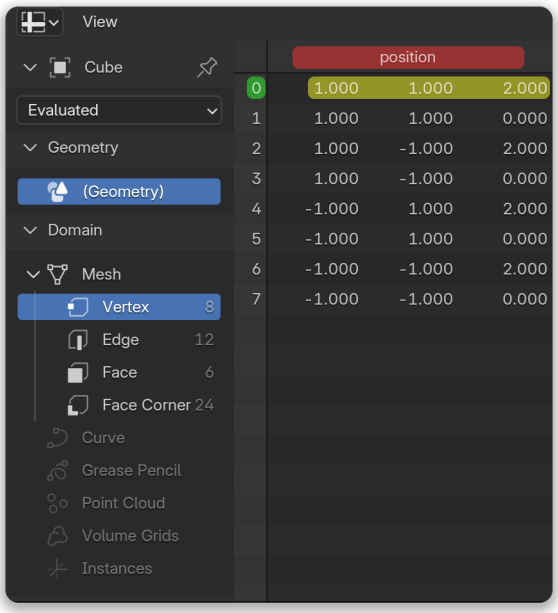


Abbildung 12: Spreadsheet-Übersicht der Vertex Domain eines Würfels (eigene Darstellung).

Jedes Attribut wird dabei durch vier Komponenten definiert:

1. Name, eindeutige Bezeichnung
2. Domain
3. Datentyp, Art der gespeicherten Werte
4. Wert, konkreter Wert

Einen Überblick über die verschiedenen Attribute und deren Komponenten kann man sich im Spreadsheet machen (Abbildung 12). Betrachtet man beispielsweise den verschobenen Würfel. Er besteht aus acht Vertices. Die Set Position Node arbeitet direkt mit dem positions-Attribut der einzelnen Vertices. Die beispielhafte Gliederung des Positions-Attributs des Vertex mit Index 0 (grün markiert) sieht nach der Verschiebung wie folgt aus:

- Name: position (rot markiert)
- Domain: Point (Vertex) (blau markiert)
- Datentyp: Vektor (3D-Vektor) (Implizit durch Wert)
- Wert: (1, 1, 2) (X-, Y- und Z-Position) (gelb markiert)

Verschieden Arten von Geometrie verfügen je nach Domain über verschiedene Standardattribute. So verfügen Faces beispielsweise über das sharp-face-Attribut, welches als Boolean gespeichert wird und determiniert, ob ein Face smooth oder sharp dargestellt werden soll. Points über Positionen, wie die Vertices des Beispielwürfels.

In Blender 4.5 stehen verschieden Datentypen zur Verfügung, welche innerhalb des Node Trees über verschieden Farben visuell kodiert werden (siehe Abbildung 13) und welche in ihrer Komplexität stark variieren. Von der Einfachheit eines Booleans bis hin zur komplexen 4x4 Matrix.

Zentral zum Verständnis dieser Arbeit ist die Unterscheidung folgender Datentypen:

Abbildung 13: Übersicht der für das LPTK relevanten Datentypen (eigene Darstellung).

[41] [23].

2.5.5.2 Das Feld-Konzept (Fields)

Die prozedurale Arbeitsweise von Geometry Nodes wird maßgeblich durch das Feld-Konzept (Fields) ermöglicht. Im Gegensatz zu klassischen Attributwerten, die als statische Werte pro Geometrieelement gespeichert werden, stellen Fields Funktionen dar, die einen Wert in Abhängigkeit eines Kontextes generieren[42]. Ein Feld repräsentiert somit eine dynamische Berechnung, die für jedes Geometrieelement (wie einen Vertex, eine Edge oder eine Instanz) ausgeführt wird, wenn der Node-Tree verarbeitet wird.

Die Formen der Sockets zeigen hierbei an, welche Sockets Fields und welche regulären Daten sind.

- Kreis: zeigt an, dass ein einzelner Wert erwartet wird, ein Feld kann nicht verbunden werden.
- Diamant: zeigt an, dass ein Feld erwartet wird, ein einzelner Wert kann aber angenommen werden.
- Diamant mit Punkt: Zeigt an, dass ein Socket, welches ein Feld annehmen kann, momentan einen einzelnen Wert annimmt.

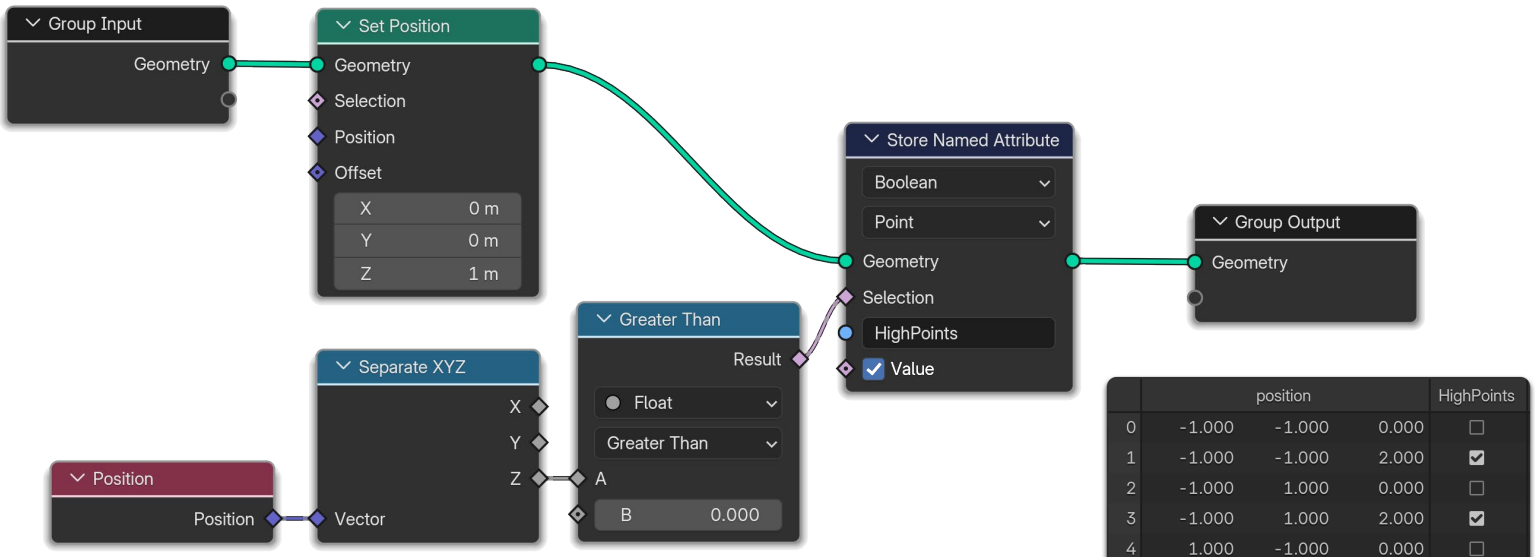


Abbildung 14: Field-basierte Attributzuweisung (eigene Darstellung).

Abbildung 13 zeigt eine dynamische Attributzuweisung des ‚HighPoints' Boolean Attributs auf der „Point"-Domain. Die Position-Node liefert dabei für jeden Vertex der Geometrie einen Wert, dieser wird in diesem Beispiel durch eine Separat XYZ-Node auf den Z-Wert reduziert. Die Greater Than Node bestimmt folgend, einen Schwellwert über welchem Z-Wert (0.000) eine ‚HighPoints' Zuweisung stattfindet.

Wird das Objekt verändert, beispielsweise indem die Vertices verschoben werden, wird die Attributzuweisung neu evaluiert.

	position			HighPoints
0	-1.000	-1.000	0.000	☐
1	-1.000	-1.000	2.000	☑
2	-1.000	1.000	0.000	☐
3	-1.000	1.000	2.000	☑
4	1.000	-1.000	0.000	☐
5	1.000	-1.000	2.000	☑
6	1.000	1.000	0.000	☐
7	1.000	1.000	2.000	☑

Abbildung 15: Darstellung der Vertex Domain des Spreadsheets nach der 'HighPoints' Zuweisung (eigene Darstellung).

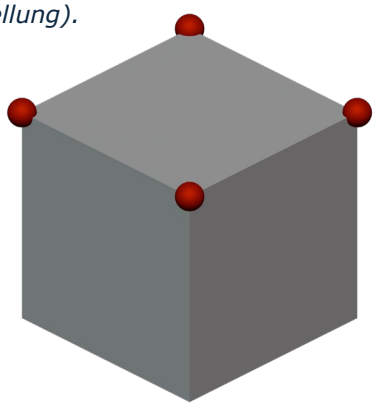


Abbildung 16: Hervorhebung der Vertices mit zugewiesenem 'HighPoints'-Wert durch Rote Kugeln (eigene Darstellung).

[42] https://docs.blender.org/manual/en/latest/modeling/geometry_nodes/fields.html

2.5.5.3 Entwicklung der Geometry Nodes und Arbeitsumgebung

Auch wenn die Grundkonzepte über die meisten Updates hinweg konstant bleiben, befinden sich Blenders Geometry Nodes weiterhin in aktiver Entwicklung. Mit nahezu jeder neuen Blender-Version werden zusätzliche Nodes eingeführt, bestehende überarbeitet oder deren Funktionsumfang erweitert. Bereits während der Bearbeitung des Praxisprojekts (Mai – Juli 2025), wurden mit der Veröffentlichung von Blender 4.5 LTS[43] zahlreiche neue und verbesserte Nodes hinzugefügt, während einige ältere als deprecated markiert wurden.

Geometry Nodes schließen damit die Lücke zwischen klassischer Modellierung und prozeduralen Systemen und bieten eine zunehmend leistungsfähige und frei zugängliche Alternative, die insbesondere für Artists und kleinere Studios, bei welchen Blender ohnehin im Einsatz ist, attraktive Möglichkeiten eröffnet.

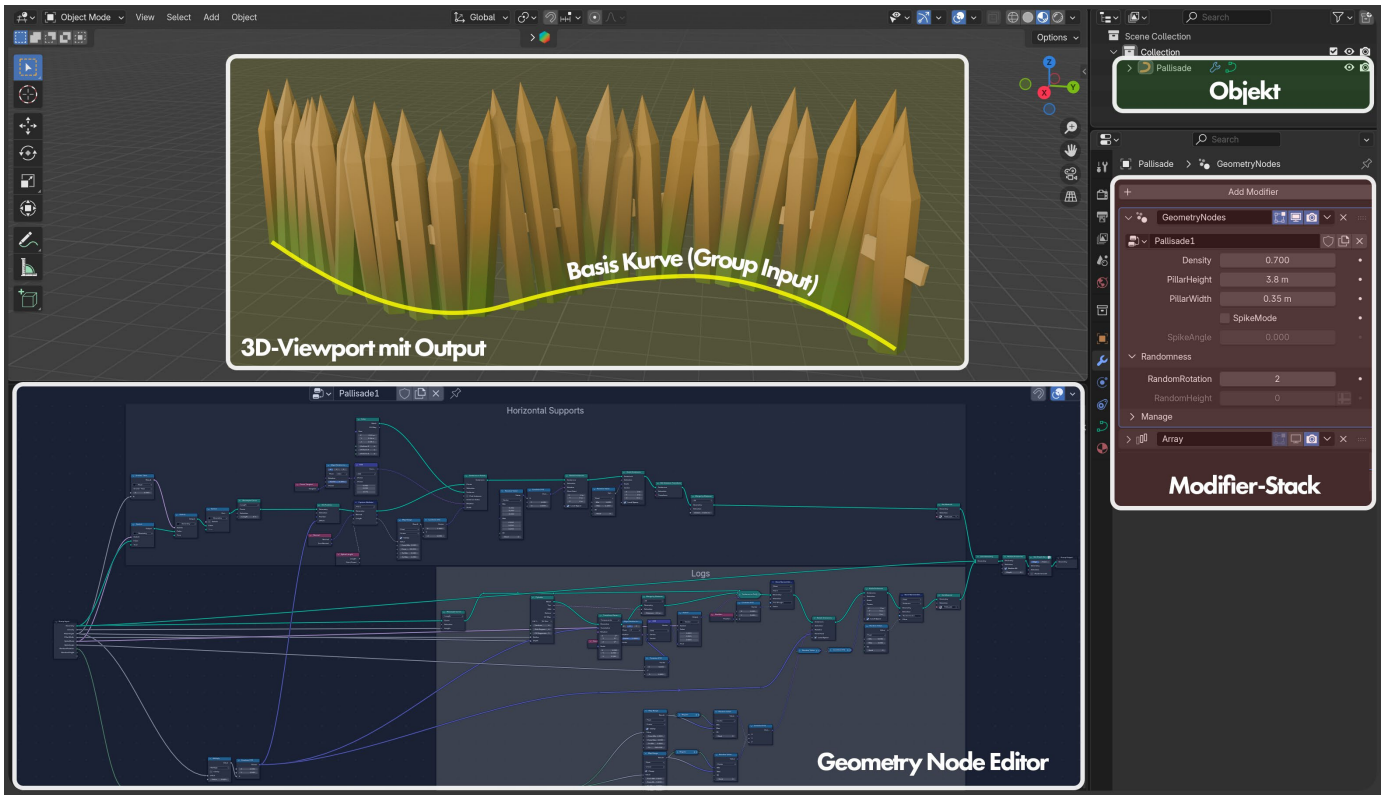


Abbildung 17: Geometry Nodes Oberfläche in Blender 4.5 am Beispiel des „Palisade1"-Node Trees (eigene Darstellung).

Die Darstellung (Abbildung 17) visualisiert, wie der Geometry Node Editor (zentral unten) prozedurale Logik für das „Palisade"-Objekt (oben rechts) definiert und das Ergebnis über die Group Output Node in den Modifier-Stack (unten rechts) übergibt, um die finale Geometrie im 3D-Viewport (zentral oben) aus der Basis-Kurve zu erzeugen.

[43] https://www.blender.org/download/releases/4-5/

3. Methodik

Die theoretischen Grundlagen bilden damit das fachliche Fundament dieser Arbeit. Auf dieser Basis widmet sich das folgende Kapitel der Methodik sowie der konkreten Zielsetzung des Low-Poly-Tool-Kits (LPTK).

Es beschreibt, wie aus den zuvor erläuterten Anforderungen der Spieleentwicklung, den Eigenschaften polygonaler 3D-Modelle und den Potenzialen prozeduraler Verfahren ein spezifischer Ansatz für die Entwicklung einer prozeduralen Asset-Bibliothek abgeleitet wurde.

3.1 Anforderungen an die entwickelte Asset-Bibliothek

Die zentrale Anforderung an das LPTK lässt sich in einem Satz definieren.

Entwickler mit minimaler 3D-Erfahrung sollen mit Hilfe des LPTKs in der Lage sein, mittelalterliche Low-Poly-Welten nach ihren Vorstellungen zu erstellen, diese jederzeit in einem non-destruktiven Workflow anzupassen und anschließend mit geringem Aufwand in die Game-Engine ihrer Wahl zu importieren.

Ausgehend von diesem Leitgedanken sowie den in der Literatur von Shaker et al. (2017, S. 6) beschriebenen *„Desirable Properties of a PCG Solution*" ergeben sich die folgenden spezifischen Anforderungen an das System:

1. **Benutzerfreundlichkeit**, eine einfache Bedienung, die auch Hobby- und Solo-Entwicklern den Zugang ermöglicht.

2. **Kontrollierbarkeit**, ein ausgewogenes Verhältnis zwischen intuitiver Nutzung und Parametern für Feinjustierungen.

3. **Optische Konsistenz,** die generierten Assets sollen einen einheitlichen Low-Poly-Look haben und sich an Synty-Qualität orientieren.

4. **Effizienz**, eine deutliche Beschleunigung des Workflows im Vergleich zur herkömmlichen Modellierung.

5. **Flexibilität**, non-destruktive Anpassungsmöglichkeiten sowie Erweiterbarkeit durch Dritte mittels zusätzlicher Node Setups.

6. **Kompatibilität**, einfache Exportierbarkeit der erstellten Modelle in gängige Game-Engines wie Unity oder Unreal.

Zusätzlich zu den Anforderungen an das prozedurale System soll auch die zugrunde liegende Software-Architektur des LPTK selbst unkompliziert aufgebaut und einfach zu erweitern sein.
Um das LPTK zu entwickeln, müssen zunächst die entsprechenden Werkzeuge ausgewählt werden. Diese Auswahl wird im folgenden Kapitel besprochen.

3.2 Auswahl der Werkzeuge

3.2.1 Blender und Geometry Nodes als prozedurale Basis

Zu Beginn des Projekts fiel die Wahl auf Blender Geometry Nodes. Diese Entscheidung beruhte darauf, dass meine bisherige 3D-Erfahrung auf Blender basierte und der Einsatz eines vertrauten Werkzeugs es mir erlaubte, mich unmittelbar auf die Erforschung der Geometry Nodes zu konzentrieren, anstatt mich in ein anderes Programm einzuarbeiten.

Houdini war mir als Software zwar bekannt, ich konnte aufgrund unzureichender Erfahrung mit prozeduraler Modellierung aber keine klaren Kriterien benennen, welche für oder gegen Blender sprechen würden.

Blender bietet, wie bereits in 2.4 beschrieben, zudem Eigenschaften, die es insbesondere im Indie-Kontext attraktiv machen: Es ist Open-Source, kostenfrei verfügbar, stark generalisiert und in der Indie-Community sehr verbreitet (Abbildung 18).

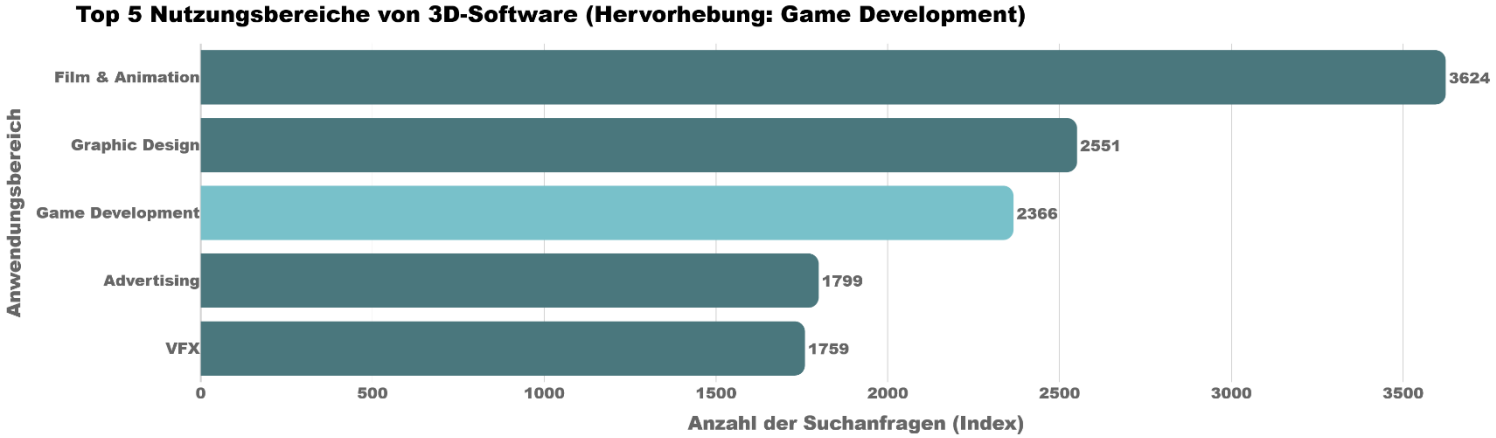


Abbildung 18: What kind of work do you do with Blender? (Datenquelle: 2024 Blender User Survey)[44] (eigene Darstellung).

Houdini hingegen ist, wie in 2.5.4 beschrieben, etablierter Standard für professionelle PCG-Projekte. Die Nutzung von Houdini erfordert tiefgehendes technisches Verständnis, wodurch es sich fast ausschließlich an Spezialisten richtet und deutlich weniger in der Indie-Szene verbreitet ist. Hinzu kommt, dass kommerzielle Projekte für die Verwendung von Houdini eine kostenpflichtige Lizenz benötigen, was jedoch häufig nicht in das Budget kleinerer Indie-Produktionen passt.

3.2.2 Add-on statt Blenders integrierter Asset-Library

Seit Blender 3.0 gibt es in Blender ein integriertes Asset-Library-System[45], welches ermöglicht Objekte, Materialien, Posen oder auch Geometry-Node-Groups zentral zu speichern und mithilfe einer einfachen Drag-and-Drop-Oberfläche über mehrere Projekte hinweg zu nutzen.

[44] [9].
[45] [24].

Dieses Library-System ist mittlerweile der Standard für kleinere Asset-Packs und besonders für klassische Assets wie statische 3D-Modelle, Materialien oder HDRIs[46] eine unkomplizierte Lösung.

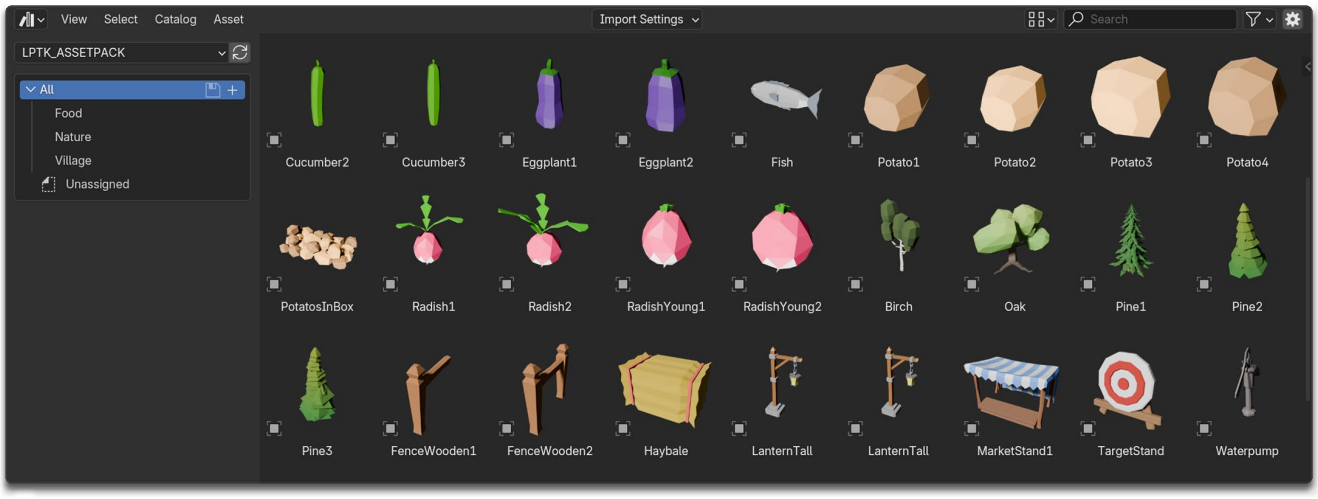


Abbildung 19: Asset Browser UI der LPTK Asset-Library (eigene Darstellung).

Die Vision des LPTK geht jedoch über die reine Wiederverwendung klassischer Assets hinaus. Das System soll nicht bloß das Platzieren von Inhalten ermöglichen, sondern den Nutzer gezielt durch den gesamten Erstellungsprozess bis hin zum Export führen.

Eine Implementierung des LPTK als eigenes Add-on bedeutet zwar einen erheblichen Mehraufwand, bietet jedoch entscheidende Vorteile und macht den Unterschied zwischen einem professionell nutzbaren Tool für Dritte und einer internen Library aus.

Vorteile eines eigenen Add-ons:

1. Mehr Kontrolle über den User-Flow während des gesamten Prozesses.
2. Spezifische Einfügungslogik für unterschiedliche Node Setups (z. B. mesh- oder curve-basierte Operationen).
3. Übersichtlichere Tooltips und ein konsistentes Interface.
4. Integration der Export-Funktionalitäten, ohne das Interface zu fragmentieren.

Gerade für weniger erfahrene Nutzer ist die integrierte Asset-Library in Blender sowohl in der Benutzung als auch in der Installation kompliziert und nicht intuitiv. Durch ein eigenständiges Add-on lässt sich der Workflow klarer strukturieren, wodurch das Tool insgesamt zugänglicher und effektiver wird.

[46] HDRIs stehen im 3D-Kontext für „*High Dynamic Range Environment Textures*"

4. Umsetzung

Das vierte Kapitel behandelt die praktische Umsetzung des beschriebenen Konzepts. Ziel ist es, zu zeigen, wie prozedurale Low-Poly-Assets mithilfe von Blender Geometry Nodes und der Blender Python API zu einem funktionalen Werkzeugsystem, dem LPTK, zusammengeführt werden können. [47]

Die Umsetzung gliedert sich in zwei Unterkapitel:

4.1 Geometry Node Trees:
Dieser Abschnitt widmet sich der Konzeption, Struktur und technischen Umsetzung der prozeduralen Systeme innerhalb von Blender. Anhand verschiedener Node Setups werden exemplarisch zentrale Konzepte vorgestellt und erläutert.

4.2 Add-on-Entwicklung:
Aufbauend auf den prozeduralen Systemen beschreibt dieser Abschnitt die Erweiterung von Blender um eine benutzerfreundliche Oberfläche und Automatisierungslogik. Mithilfe der Blender Python API wird das LPTK als Add-on implementiert, um die erstellten Node-Systeme zugänglich, modular und effizient nutzbar zu machen.

4.1 Entwicklung der Geometry Node Trees

In diesem Kapitel wird die Konzeption und Umsetzung prozeduraler Low-Poly-Assets mit Blender Geometry Nodes behandelt. Ziel ist es, die grundlegenden Prinzipien der Systemarchitektur verschiedener Systeme des LPTK zu erläutern und zu zeigen, wie modulare, nicht-destruktive Workflows für die Asset-Erstellung umgesetzt werden können.

Es existieren verschiedene Ansätze, prozedurale Generierung in Produktionspipelines zu integrieren. In vielen Fällen werden solche Systeme als Zwischenschritt oder Ausgangspunkt genutzt, um wiederkehrende Arbeitsschritte zu automatisieren. Dieser Ansatz eignet sich vor allem für größere Teams mit spezialisierten Tools oder Engine-basierten Pipelines.

Das LPTK verfolgt einen alternativen Ansatz. Es richtet sich gezielt an kleinere Teams oder Einzelentwickler, die in Blender arbeiten und einen direkten, intuitiven Zugang zu prozeduraler Modellierung suchen. Entsprechend liegt der Fokus weniger auf komplexer Pipeline-Integration, sondern auf Bedienbarkeit, Modularität und Stabilität.
Zentral ist dabei die Idee der Non-Destruktivität, jede Veränderung bleibt reversibel und parameterbasiert steuerbar. Die Arbeit mit den LPTK-Systemen soll sich soll sich in erprobte Arbeitsabläufe einpassen, vergleichbar mit einem integrierten Level-Editor, der die Arbeit in Blender erleichtert.

Darüber hinaus wurde bei der Entwicklung der einzelnen Node Trees auf eine klare, erweiterbare Struktur geachtet, sodass sowohl eigene Anpassungen als auch spätere Erweiterungen durch erfahrene Nutzer problemlos möglich sind.

[47] Selbstbenannte Konzepte werden dabei in einfachen Anführungszeichen gekennzeichnet.

Im Folgenden werden zunächst grundlegenden Konzepte anhand eines einfachen Beispiel-Assets erläutert. Anschließend wird die Parametrisierung zur Steuerung der Systeme vorgestellt, bevor vier ausgewählte, konzeptionell unterschiedliche Implementierungen des LPTK detailliert beschrieben werden.

4.1.1 Erste Experimente

Das Praxisprojekt zu dieser Arbeit stellt meinen ersten tiefergehenden Berührungspunkt mit Blenders Geometry Nodes und der prozeduralen Modellierung im Allgemeinen dar. Um ein grundlegendes Verständnis für die Funktionsweise, Möglichkeiten und Limitationen zu entwickeln, habe ich mich zunächst primär mithilfe von Online-Tutorials sowie der Analyse bestehender Systeme beschäftigt.

In dieser explorativen Anfangsphase entstanden verschiedene Node-Trees, von denen einige als technische Grundlage späterer Systeme dienten. Eines der ersten Systeme, welches in abgewandelter Form in das LPTK integriert wurde ist das ‚Funky-Tree'-System.

Das System basiert auf einem Tutorial[48] und verarbeitet freihand gezeichnete Splines (Kurven) zu stilisierten Low-Poly-Bäume.

Es ist dabei nicht vollprozedural, sondern instanziiert vormodellierte Äste und Laub auf der gemalten Kurve und fügt dieser einen konfigurierbaren Radius hinzu.

Die gemalte Kurve wird im ersten Schritt durch eine Resample Curve Node neu abgetastet und anschließend in drei parallel ausgeführten Node-Gruppen verarbeitet, deren Ausgaben danach wieder zusammengefügt werden.

Die ‚TopLeaf'-Gruppe führt eine endpoint selection aus und instanziiert auf dem obersten Punkt der Kurve das ‚TopLeaf'-Model.

Die ‚BranchesAndLeafs'-Gruppe instanziiert entlang der resample curve in die ‚Branches'-Collection, welche zwei Zweig-Varianten enthält.

Währenddessen erzeugt die ‚Trunk'-Gruppe mithilfe der Set Curve Radius Node ein Mesh aus der neu abgetasteten Kurve.

Anschließend werden die separat erzeugten Geometrien mit der Join Geometry Node zu einem Objekt zusammengeführt. Abbildung 20 auf der folgenden Seite zeigt den Aufbau des Node-Trees und visualisiert die Ergebnisse der einzelnen Gruppen.

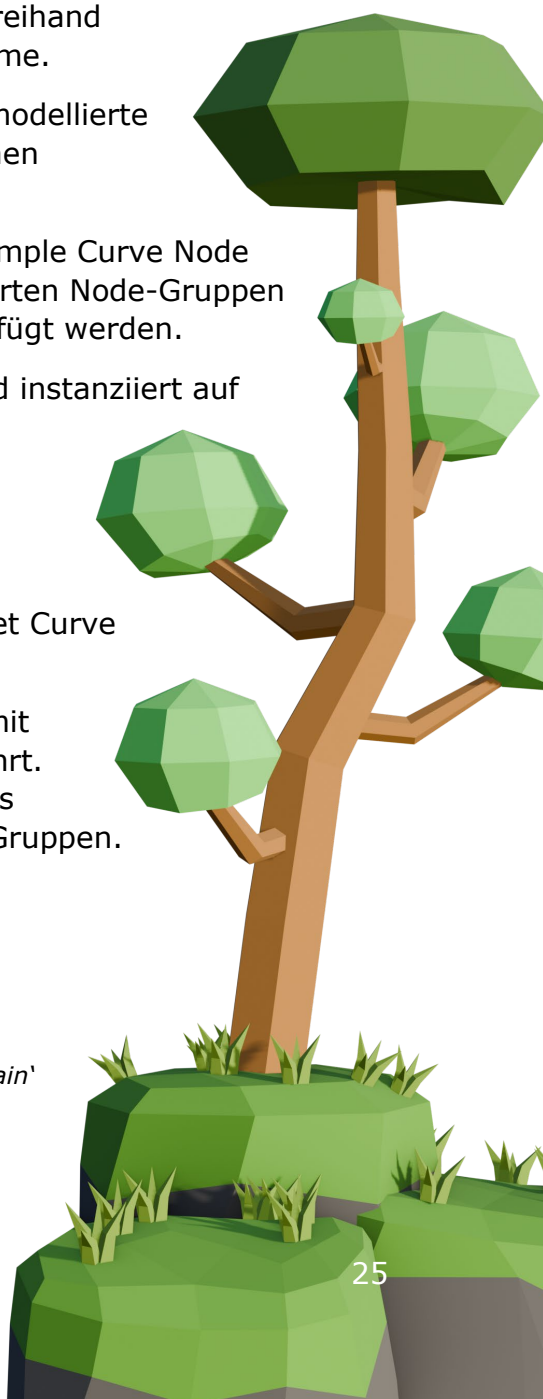


Abbildung 20:Rendering eines ‚FunkyTrees' auf einem ‚MeshTerrain' (eigene Darstellung).

[48] [25].

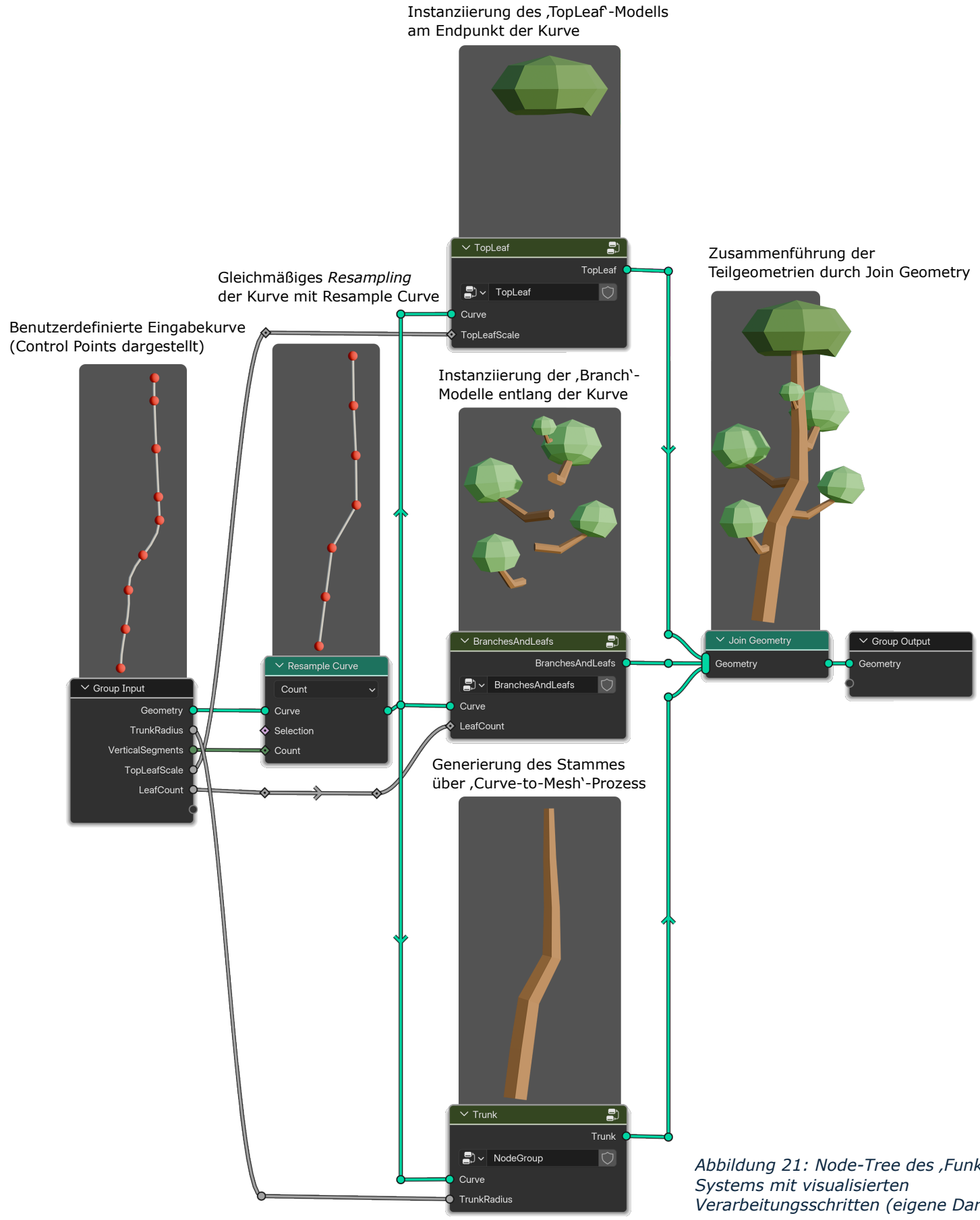


Abbildung 21: Node-Tree des ‚FunkyTree'-Systems mit visualisierten Verarbeitungsschritten (eigene Darstellung).

Um die prozeduralen Parameter des Systems zu steuern kann der Nutzer entweder die Form der Kurve im 3D-Viewport anpassen oder die exponierten Parameter des Systems konfigurieren. Die Parametrisierung von Geometry Nodes wird im nächsten Kapitel anhand des ‚FunkyTree'-Systems besprochen.

4.1.2 Parametrisierung anhand des ‚FunkyTree'-Systems

Wie bereits in 3.1 beschrieben, stellt die Kontrollierbarkeit prozeduraler Systeme eine zentrale Herausforderung dar. Einerseits sollen Nutzer in der Lage sein das Objekt oder die Geometrie genau nach ihren Vorstellungen anpassen zu können, ohne in den Geometry Node Editor einsteigen zu müssen, andererseits soll das System den Nutzer auch nicht mit zu kleinteiligen Konfigurationsmöglichkeiten erschlagen.

Die meisten Parameter im Node-Tree lassen sich über Sockets durch die Group Input Node in den jeweiligen Geometry Nodes Modifier, wie in 2.5.5 beschrieben, exponieren. Damit lassen sich interne Werte zugänglich machen, ohne in den Node Tree navigieren zu müssen. Beispielsweise kann der Parameter ‚TrunkRadius' aus der internen ‚Trunk'-Gruppe herausgelöst und direkt in den Group Input Node verschoben werden (siehe Abbildung 22).

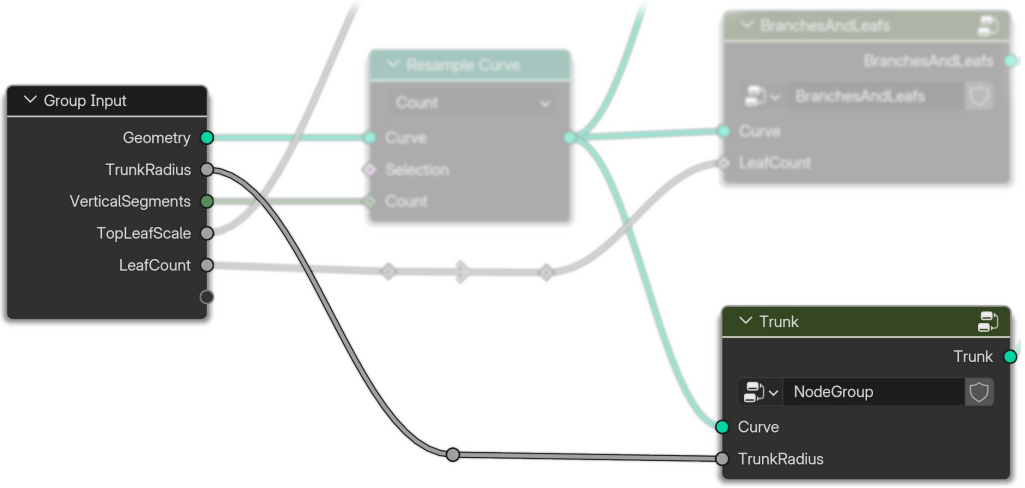


Abbildung 22: Ausschnitt vom ‚FunkyTree'-Setup mit Fokus auf der Group Input Node und der ‚Trunk'-Gruppe (eigene Darstellung).

Wird ein Parameter in den Group Input gezogen, erscheint dieser automatisch im Group Sockets Panel des Node-Trees und wird dadurch konfigurierbar.

Innerhalb des Group Sockets gibt es in Blender die Möglichkeit, die exponierten Parameter für optimale Nutzung zu spezifizieren. Es können Eingabedatentyp definiert, visuelle Gruppen (Panels) erstellt, Wertebereiche begrenzt, passende Standardwerte gesetzt und hilfreiche Tooltips eingefügt werden.

In Abbildung 23 ist der ‚TrunkRadius'-Parameter, des ‚FunkyTree' Group Sockets ausgewählt. Die Konfigurationsmöglichkeiten für diesen Input sind sichtbar. Der „Type" zeigt den Datentyp des ‚TrunkRadius', die Description ist der Tooltip, welcher beim Hovern über den Parameter im Modifier angezeigt wird. Der Subtype bestimmt die Darstellung im Modifier (Distance = Angabe des Wertes in Metern). Der Default

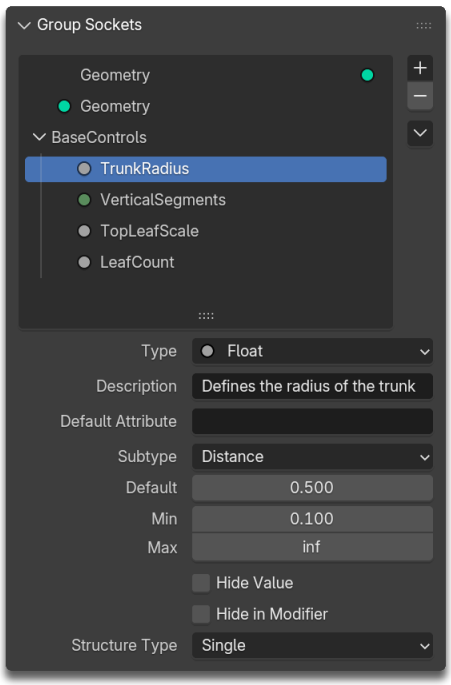


Abbildung 23: Group Sockets der "FunkyTree" Group Input-Node, einseh- und konfigurierbar im Node-Backend (eigene Darstellung).

Parameter bestimmt den Standardwert (0,5 Meter) und die Min- und Max-Felder definieren den Wertebereich, welchen der User im Modifier definieren kann.

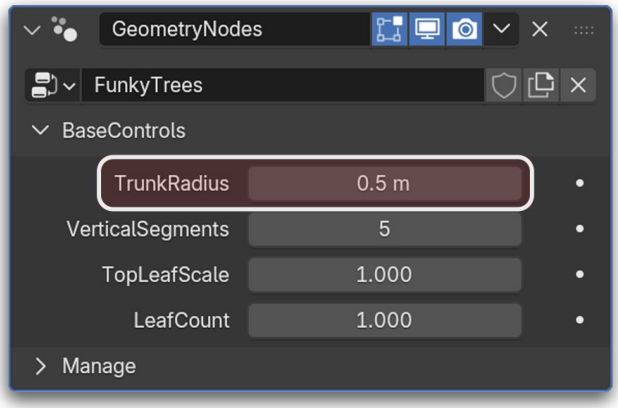


Abbildung 24: Geometry Nodes Modifier des ‚FunkyTree'-Systems (eigene Darstellung).

Abbildung 24 stellt die konfigurierten Group Sockets als kontrollierbare Parameter im Geometry Nodes Modifier an und ist somit das Frontend der in Abbildung 23 gezeigten Konfiguration.

Es existieren unterschiedliche Herangehensweisen zur Parametrisierung. Während einige Geometry Nodes Entwickler beinahe jeden Parameter exponieren, reduzieren andere die Bedienung bewusst auf weniger Kernparameter.

Da sich das LPTK explizit an Anwender ohne tiefgehende Kenntnisse zur prozeduralen Modellierung richtet, wurde stets eine Balance zwischen Kontrolle und Verständlichkeit bei der Architektur der Systeme angestrebt.

Die ausgewählten Parameter sind zu diesem Zweck mit praxistauglich Werten vorbelegt und auf sinnvolle Wertebereiche begrenzt. Eine strukturierte Gruppierung innerhalb der Parameter-Panels, hilfreiche Beschreibungen und Tooltips erleichtern die Nutzung.

Die Parametrisierung stellt somit einen wesentlichen Faktor für die Nutzbarkeit und Erweiterbarkeit der gesamten prozeduralen Asset-Bibliothek dar.

Während der Entwicklung sämtlicher LPTK-Assets wurde daher konsequent versucht, eine ausgewogene Balance zwischen kreativer Kontrolle und Bedienbarkeit zu erreichen.

Nachdem anhand des ‚FunkyTree'-Systems die grundlegenden Prinzipien zur Strukturierung, Modularisierung und Parametrisierung prozeduraler Node Setups vorgestellt wurden, folgt nun die schrittweise Erweiterung dieser Ansätze auf komplexere Anwendungskontexte.

Als nächstes wird ein kurvenbasiertes Verfahren zur Pfadgenerierung anhand des ‚StonePath'-Assets erläutert. Analog zum ‚FunkyTree', werden auch hier Objekte anhand einer Nutzerdefinierten Kurve instanziiert, jedoch wurde das System um einige anwendungsspezifische Funktionalitäten erweitert.

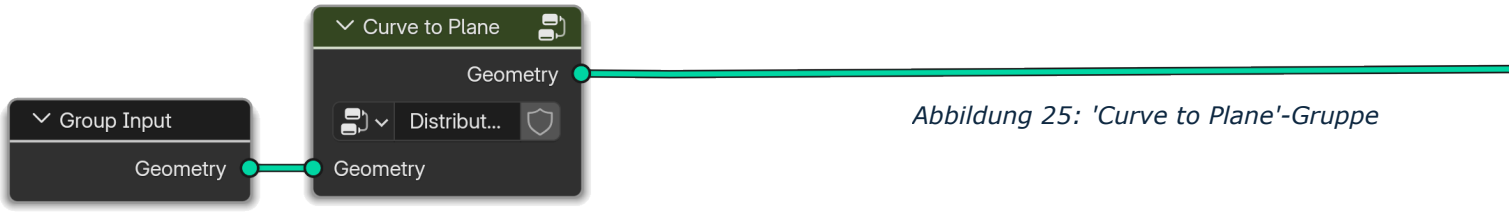


Abbildung 25: 'Curve to Plane'-Gruppe

4.1.3 Kurvenbasierte Pfadgenerieung

Für Assets wie Pfade bietet sich eine dynamische Generierung besonders gut an. Zwar existieren statische Baukastensysteme, welche die individuelle Zusammensetzung einzelner, vordefinierter Pfadelemente erlauben, ein prozeduraler Ansatz, der sich automatisch an unterschiedliche Untergründe oder Terrains anpasst und einem nutzerdefinierten Pfad folgt, stellt jedoch die deutlich elegantere und unkompliziertere Lösung dar.

In diesem Kapitel wird die prozedurale Pfadgenerierung anhand des ‚StonePath'-Systems erläutert, welches mithilfe von nutzerdefinierten Kurven dynamische Pfade erzeugen kann, die sich jedem Untergrund anpassen.
Das System ist vergleichsweise simpel, erlaubt es aber, Kurven auf Oberflächen zu zeichnen, entlang derer in einer definierbaren Breite Steine platziert werden, sodass ein natürlicher Pfad entsteht.

Zur Erstellung der Kurven wird, wie bereits beim ‚FunkyTree'-System, das integrierte Freehand Spline-Werkzeug von Blender verwendet. Dieses ermöglicht es, Kurven freihand zu zeichnen und direkt auf bestehende Objekte zu projizieren.
Nach dem Zeichnen der Basiskurve kann der Nutzer den Pfad mithilfe verschiedener exponierter Parameter non-destruktiv anpassen.

4.1.3.1 ‚Curve to Plane'

Um Steine entlang eines Pfades zu platzieren, bietet sich die Instanziierung von Objekten auf einer Fläche an.
Im Gegensatz zum ‚FunkyTree'-System, bei dem Äste direkt entlang der vom Nutzer gezeichneten Kurve instanziiert werden, benötigt der ‚StonePath' eine Fläche mit definierbarer Breite, auf der die Steine verteilt werden können.

Dafür wird die gezeichnete Kurve im ersten Schritt in eine Plane umgewandelt, die als Basis für die Instanzen dient.
Dieser Prozess erfolgt in der Node-Gruppe ‚Curve to Plane'.

Zunächst wird die Kurve entlang ihrer Normalen um die Hälfte des nutzerdefinierten Width-Parameters verschoben, welcher die Breite des späteren Pfades bestimmt.
Anschließend wird sie mithilfe der Curve to Mesh und Extrude Mesh Nodes in ein Mesh konvertiert und extrudiert, um eine ebene Fläche zu erzeugen. Durch die Verschiebung um die halbe Breite verläuft die Mittellinie dieser Fläche exakt entlang der ursprünglichen Kurve.
So entsteht eine flexible, parameterbasierte Grundlage, auf der die Steine später gleichmäßig verteilt werden können.

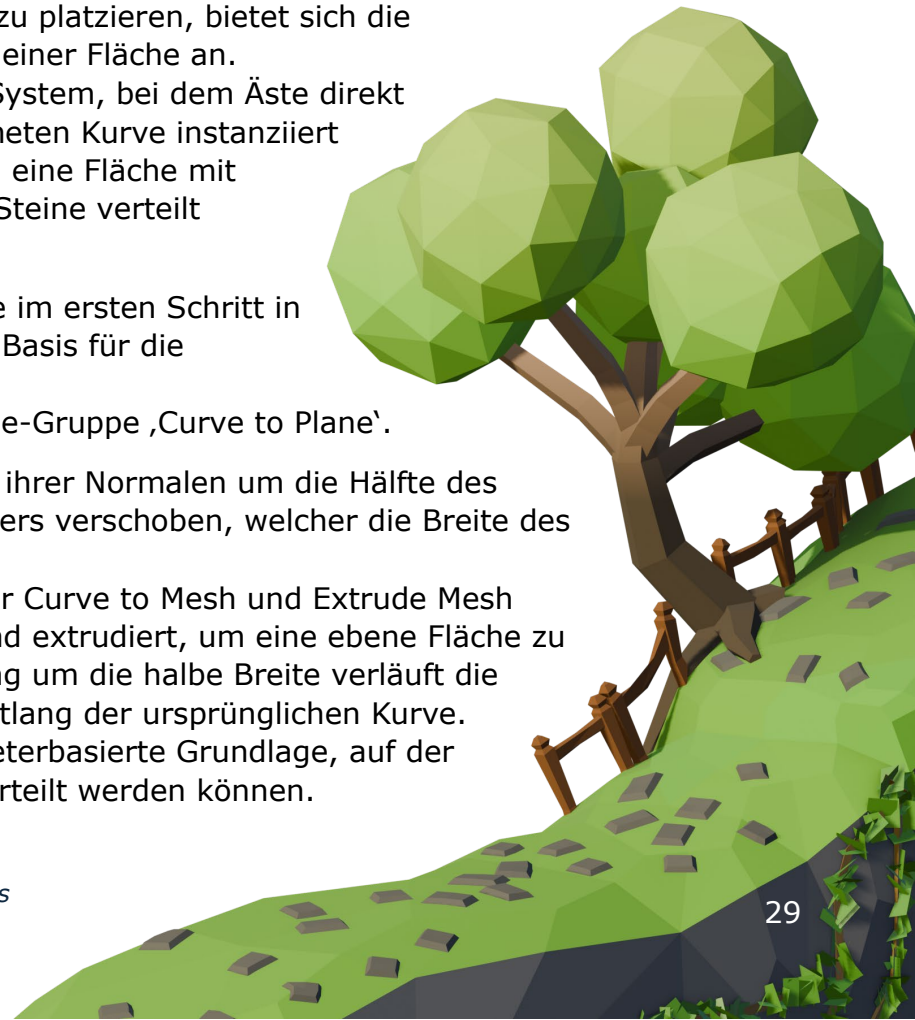


Abbildung 26: Darstellung des ‚StonePath'-Systems auf einem ‚MeshTerrain' (eigene Darstellung).

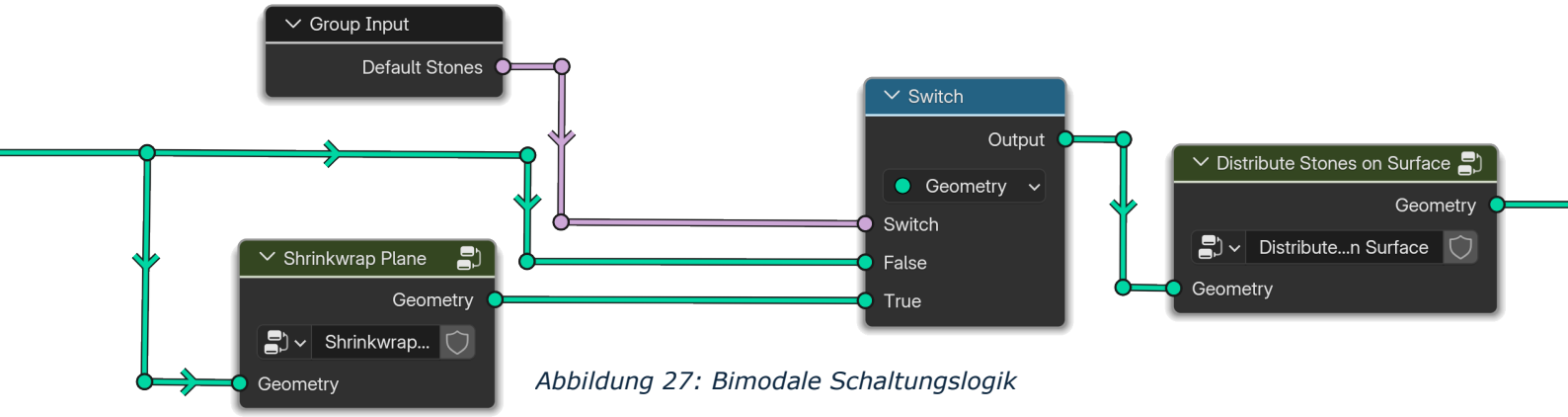


Abbildung 27: Bimodale Schaltungslogik

4.1.3.2 Instanziierung und Projektion mit ‚Stones on Surface'

Nachdem die Kurve in eine Plane überführt wurde, wandert diese optional durch die ‚Shrinkwrap Plane'-Gruppe und anschließend in die ‚Distribute Stones on Surface'-Gruppe. Diese bimodale Logik wurde implementiert, um sowohl nutzerdefinierte Assets verwenden zu können, als auch eine vollprozedurale Alternative zu bieten.

Modus 1: Interne Prozedurale Generierung (maximale Konformität)

Für Projekte, welche keine eigenen Steinkollektionen benötigen (repräsentiert durch den False-Pfad der Switch Node), wird die Stein-Geometrie prozedural erzeugt, wodurch eine optimale Projektion erfolgen kann:

1. Punktverteilung: Die Plane wird mittels der Distribute Points on Faces Node in eine Punktwolke überführt. Diese Punkte können dabei anhand exponierter Parameter vom Nutzer in der Dichte konfiguriert werden und stellen die Punkte zur Instanziierung der einzelnen Steine dar.
2. Instanz-Geometrie: Anstelle eines externen Assets wird ein einfaches 3x3-Grid (eine Plane mit neun Vertices) auf die Punkte instanziiert.
3. Projektionslogik: Die Vertices dieses einfachen Grids dienen als individuelle Projektionspunkte. Mithilfe der Geometry Proximity Node werden die nächstgelegenen Positionsdaten des Zielobjekts für jeden Vertex aller Grids ermittelt. Durch die nachfolgende Set Position Node werden die Vertices der Grids auf diese ermittelten Positionen verschoben.

Dieser Mechanismus projiziert die Geometrie jeder einzelnen Instanz auf die Oberfläche, wodurch sich die Steine dynamisch der Krümmung, Neigung und Höhe des Objekts, bspw. eines Terrains, anpassen und somit eine höchstmögliche Terrain-Konformität gewährleisten.

Modus 2: Externe Asset-Kollektion (Künstlerische Kontrolle)

Will der Nutzer eine externe Stein-Kollektion als Instanzobjekt verwenden (repräsentiert durch den True-Pfad der Switch Node), wird die Plane an sich zunächst als Ganzes auf das Ziel-Terrain projiziert (gewrappt).

1. Technischer Mechanismus: Dies geschieht durch die ‚Shrinkwrap Plane'-Gruppe, welche jeden Vertex des Pfades, mithilfe der Geometry Proximity Node auf das nächstliegende Face der Projizierungs-Geometrie projiziert.
2. Resultat: Anschließend wandert die per Shrinkwrap-Projektion angepasste Plane ebenfalls in die Distribute Points on Faces Node, wobei in diesem Fall die erzeugte Punktwolke und die daraus resultierende Instanziierung (in

der folgenden Gruppe) grob der Terrain-Oberfläche folgen. Da die Instanzen selbst jedoch ihre ursprüngliche Geometrie beibehalten, limitiert dieser Modus die Detailgenauigkeit der Terrain-Konformität zugunsten der Verwendung komplexer, manuell erstellter Assets[49].

4.1.3.3 ‚Material Manager'

Die korrekt platzierten Geometrien werden anschließend in die ‚Material Manager'-Gruppe übergeben.

Hier erfolgt die Materialzuweisung der einzelnen Steine. Die Logik verwendet eine Random Value Node, um einen zufälligen Wert für jede Instanz zu generieren. Dieser Index-Wert wird anschließend genutzt, um über die Set Material Index Node jeder Instanz eines von drei verschiedenen Materialien zuzuweisen. Die Farbe der einzelnen Materialien kann hierbei über exponierte Parameter im Modifier frei konfiguriert werden.

4.1.3.4 ‚Default Stone Extrusion and Deformation'

Dieser abschließende Verarbeitungsschritt wird nur auf die intern prozedural erzeugten Steine (Modus 1/False-Pfad) angewandt und dient der Erzeugung von geometrischer Tiefe und der Brechung der Uniformität.

Extrusion und Skalierung: Die auf das Terrain projizierten, aber noch flachen Stein-Planes werden mittels der Extrude Mesh Node extrudiert, um ihnen eine Höhe zu verleihen. Die neue, obere Fläche wird anschließend skaliert, um die Kanten optisch zu brechen und die Erscheinung eines abgerundeten Steins zu erzeugen. Die beiden Parameter können hierbei ebenfalls im Modifier frei konfiguriert werden.

Zufällige Verformung: Die final extrudierte Geometrie wird einer prozeduralen Deformation unterzogen. Hierfür wird eine Noise Texture Node mit einer Set Position Node kombiniert. Die Stärke der Verformung wird dabei von einer Random Value Node bestimmt, welche durch definierbare Min- und Max-Werte einer Map Range Node gesteuert werden kann.

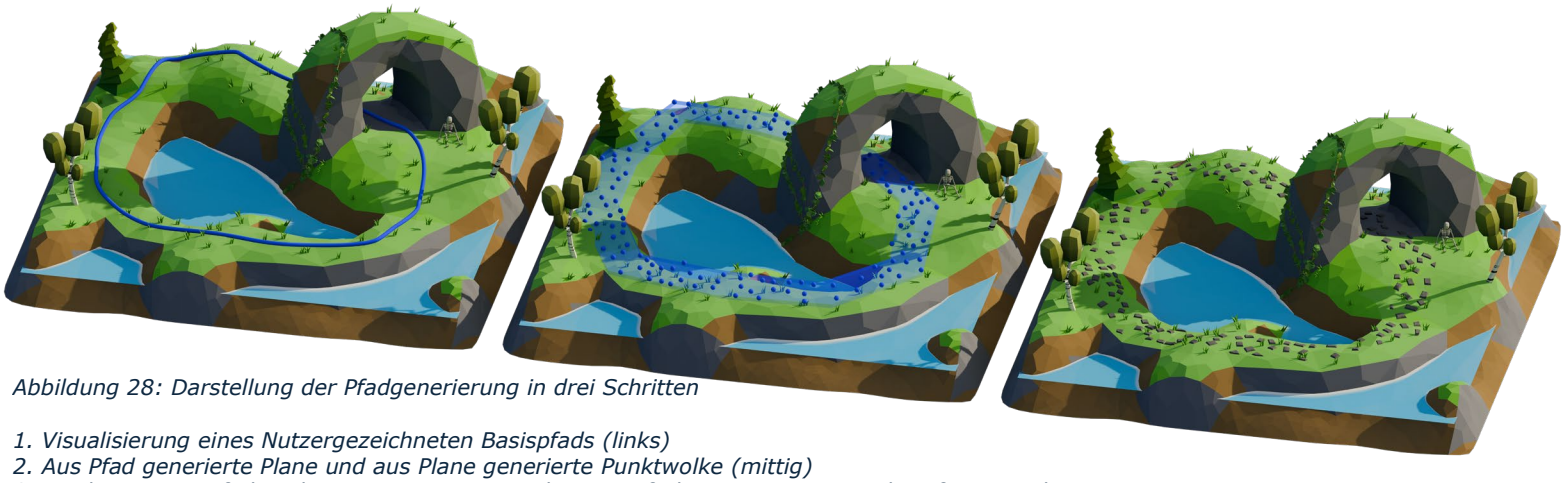


Abbildung 28: Darstellung der Pfadgenerierung in drei Schritten

1. Visualisierung eines Nutzergezeichneten Basispfads (links)
2. Aus Pfad generierte Plane und aus Plane generierte Punktwolke (mittig)
3. Finales Terrain Pfad nach Instanziierung, Projektion, Einfärbung, Extrusion und Verformung der Grids (rechts)

[49] Hierbei ist zu erwähnen, dass das System noch einen zusätzlichen Modus anbietet, welcher die Geometrie der eingespeisten Objekte analysiert, die Vertices der nach unten gerichteten Faces selektiert und diese auf das Terrain shrinkwrappt. Bei einfachen Assets kann dieser Ansatz funktionieren, bei komplexeren Stein-Assets kann dies jedoch zur starken Verzerrung der Steingeometrie führen.

Abbildung 29: Vereinfachte vertikale Darstellung des ‚ProceduralTerrain'-Setups (eigene Darstellung).

4.1.4 ‚ProceduralTerrain'

Das ‚ProceduralTerrain'-System ist das komplexeste System des gesamten Projekts und wurde von Beginn bis zum Abschluss des Praxisprojekts kontinuierlich erweitert und verfeinert.

(Voll-)prozedurale Terrain- und Level-Systeme sind, wie in 2.5 beschrieben, eine der häufigsten Formen von PCG-Integration in der Spieleentwicklung. Für einen Design-Time-bezogenen, Mixed-Authorship (in 2.5.3 behandelt) Workflow aber weniger geeignet. In diesem Kontext sind Kontrollier- und Erweiterbarkeit weitaus wichtiger als die Möglichkeit zur unendlichen Generierung.

Das ‚ProceduralTerrain' unterscheidet sich deshalb fundamental von den vollprozeduralen Implementationen eines Terrain-Systems. Es liefert eine prozedurale Basisgeometrie, die als Ausgangspunkt oder Inspiration dient, aber vollständig verände- und konfigurierbar ist.

Das System ist ein umfangreicher Node Tree mit über 50 konfigurierbaren Parametern. Auf diese Parameter und die grundlegende Architektur des Systems werde ich in diesem Kapitel eingehen. Die Logik des Systems ist größtenteils in Reihe geschaltet und iteriert die Geometrie Schritt für Schritt.

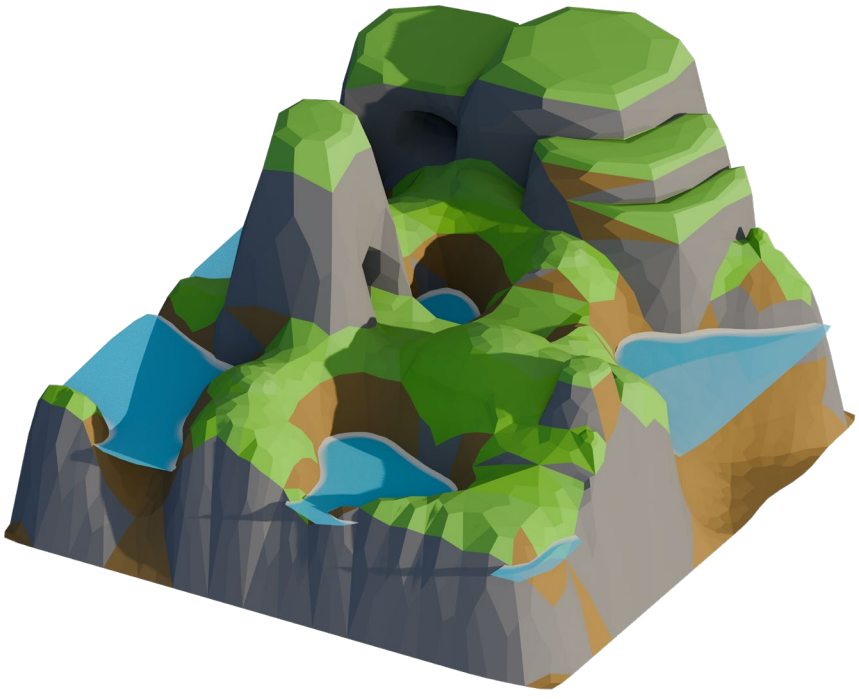


Abbildung 30: Beispiel Rendering eines ‚ProceduralTerrain' (eigene Darstellung).

4.1.4.1 Basis-Mesh & Booleans

Die erste Node-Gruppe erzeugt das Basis-Mesh des Terrains. Ausgangspunkt ist eine einfache Plane, auf die der Geometry Nodes Modifier angewendet wird. Diese wird zunächst unterteilt (subdivided) und anschließend mithilfe mehrerer kombinierter Noise Texture Nodes verformt. Über eine Set Position Node werden die Z-Koordinaten der einzelnen Vertices anhand der noise-basierten Werte angepasst, wodurch eine prozedural generierte Terrainoberfläche entsteht.

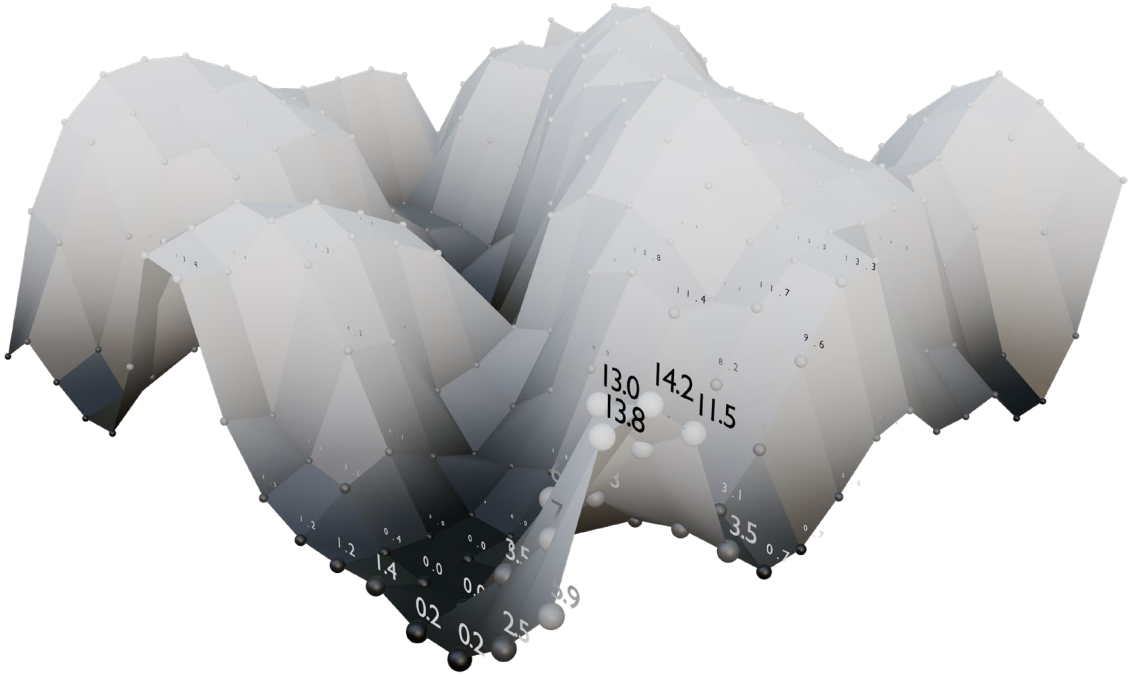


Abbildung 31: ‚ProceduralTerrain' Basis-Mesh mit visualisierten Vertices und deren Z-Positionen (eigene Darstellung).

Für die Verformung stehen drei unterschiedlich konfigurierte Noise-Maps zur Verfügung, welche über die Presets ‚Default', ‚Hills' und ‚Plateau' abgerufen werden können.

Das Terrain durchläuft anschließend zwei Mesh Boolean Nodes: zunächst einen Union Boolean, danach einen Difference Boolean. Beide verwenden nutzerdefinierte Collections als Input, wodurch zusätzliche Geometrien manuell auf das Terrain addiert oder daraus subtrahiert werden können.

4.1.4.2 ‚Merge & Triangulation'

Anschließend wird die Geometrie in die ‚Merge & Triangulation'-Gruppe geführt. Hier werden Vertices, welche einen definierten Distanzschwellwert unterschreiten, zusammengeführt (merged). Dieser Schritt ist essenziell, um die durch Noise erzeugte mit der vom Nutzer eingefügten Geometrie zu verschmelzen und unnatürliche Übergänge zu vermeiden.

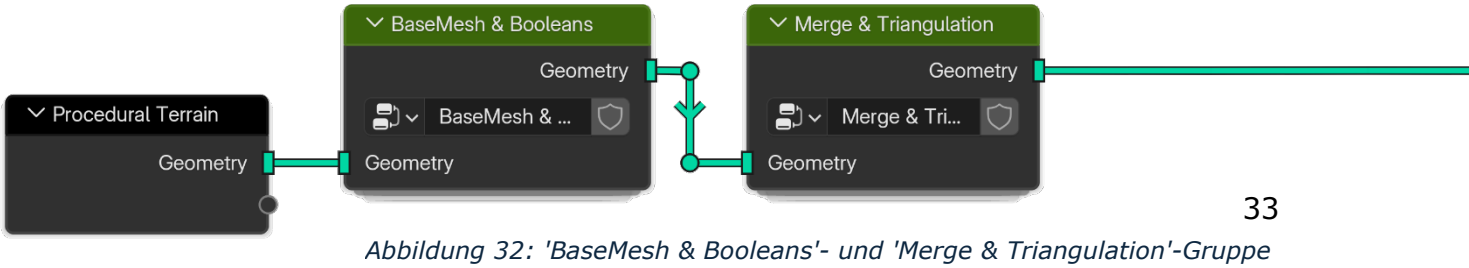


Abbildung 32: 'BaseMesh & Booleans'- und 'Merge & Triangulation'-Gruppe

4.1.4.3 ‚Material Manager‘

Nachdem durch die ersten beiden Node-Gruppen das grundlegende Mesh erzeugt wurde, wird die Geometrie aufgetrennt und parallel an zwei weitere Gruppen übergeben.

Der ‚Material Manager‘ übernimmt dabei die Aufgabe, einzelnen Flächen (Faces) automatisch verschiedene Materialien zuzuweisen. Die Selektion erfolgt zunächst parametrisch mithilfe mehrerer miteinander verknüpfter Systeme, kann jedoch bei Bedarf auch manuell angepasst werden.

Standardmäßig umfasst das System vier Materialien: ‚Stone‘, ‚Dirt‘, ‚Grass‘ und ‚TopGrass‘. Die jeweiligen Basisfarben dieser Materialien können direkt im Modifier-Panel angepasst werden, wodurch der Nutzer sofortiges visuelles Feedback erhält.

Im ersten Schritt der parametrischen Selektion wird für jede Fläche das Skalarprodukt zwischen ihrer Normalrichtung und der globalen Z-Achse berechnet. Dadurch erhält jedes Face einen numerischen Wert im Bereich von -1 (vollständig nach unten gerichtet) bis 1 (vollständig nach oben gerichtet), der beschreibt, wie stark seine Ausrichtung mit der globalen Z-Richtung übereinstimmt. Nutzer können dadurch mithilfe der Compare Node Neigungswinkel-Schwellwerte konfigurieren, um den verschiedenen Faces entsprechende Materialien zu zuzuweisen.

So lässt sich das Verhältnis zwischen ‚Stone‘, ‚Dirt‘, ‚Grass‘ und ‚TopGrass‘ feinjustieren und an verschiedene Geländetypen anpassen. Darüber hinaus bietet das System mehrere optionale Zusatzfunktionen:

- Ein höhenbasierter ‚Stone Threshold‘, der die Materialverteilung an die relative Höhe des Meshes koppelt und so die gezielte Definition bergiger Regionen in entsprechenden Höhen präziser abbildet.
- Eine ‚Overhang Detection‘, die mithilfe von Raycasts entlang der positiven Z-Achse erkennt, ob eine Fläche überdeckt ist und die Zuweisung eines Overhang-Materials erzwingt.

Diese Kombination aus geometrischer Analyse und benutzerdefinierbaren Parametern ermöglicht eine präzise, visuell stimmige und zugleich prozedurale Materialverteilung.

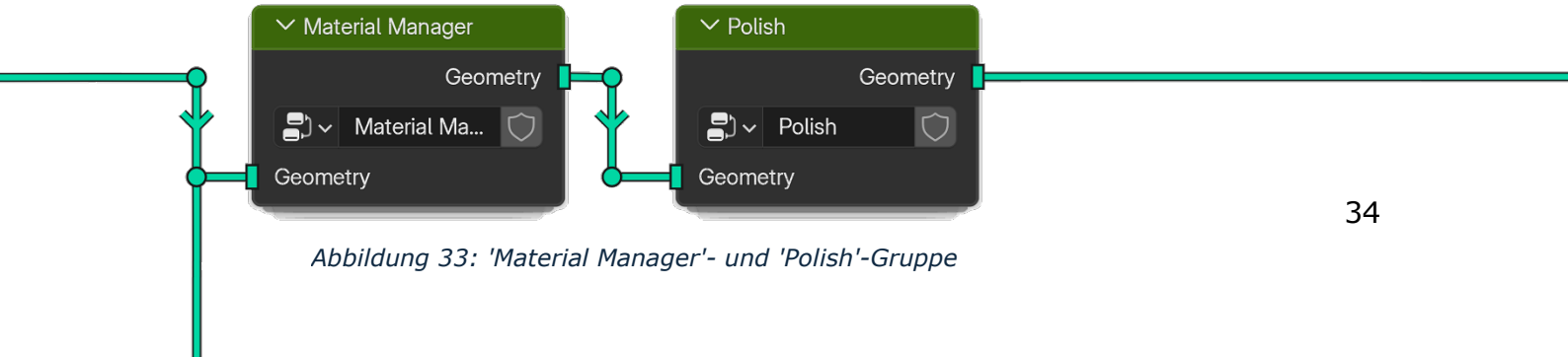


Abbildung 33: 'Material Manager'- und 'Polish'-Gruppe

Abbildung 34: 'Water Generation'-Gruppe

4.1.4.4 ‚Water Generation'

Parallel zur Geometrie im ‚Material Manager' wird die Basisgeometrie auch in das ‚Water Generation'-System übergeben. Die einfachste Möglichkeit, Wasser in ein Terrain zu integrieren, besteht darin, eine Plane mit einem Wassermaterial über die gesamte Fläche des Terrains zu legen.
Dieser Ansatz ist jedoch sehr limitiert. Er erlaubt lediglich einen globalen Wasserspiegel und verhindert die gezielte Deaktivierung einzelner Wasserbereiche, wodurch es unmöglich ist, eine Schlucht oder Höhle unterhalb des globalen Wasserspiegels korrekt darzustellen.

Um diese Einschränkung zu umgehen, wurde ein gruppenbasierter Ansatz entwickelt, welcher eine flexiblere Steuerung und die gezielte Entfernung der einzelnen Wasseroberflächen anhand ihrer Gruppen ermöglicht.

Bei der Umsetzung traten mehrere Komplikationen auf, wodurch das System komplex und rechenintensiv wurde. Eine der zentralen Herausforderungen bestand darin, die relevante Geometrie zu selektieren, um die Punkte einzelner Gewässer präzise zu gruppieren und anschließend als zusammenhängende Meshes zu verbinden.

Zur Generierung der Wasseroberflächen werden zunächst die äußeren Kanten der Basisgeometrie selektiert und entlang der Z-Achse nach oben verschoben, um eine geschlossene Hülle um die Geometrie zu erzeugen. Anschließend wird diese Geometrie samt der Hülle mit einer Distribute Points on Faces Node in eine Punktwolke umgewandelt. Alle Punkte, welche nicht auf Höhe (Z-Position) des nutzerdefinierten Wasserspiegels plus dem definierbaren Schwellwert des WaterLevel-Parameters liegen, werden gelöscht.

Im nächsten Schritt werden die übrigen Punkte auf eine einheitliche Z-Position verschoben, wodurch eine saubere, horizontale Verteilung auf Wasserspiegelhöhe entsteht (dargestellt als rote Punkte in Abbildung 35).
Aus dieser bereinigten Punktwolke wird mit der Points to Volume Node ein Volumen erzeugt (blaue Volumen in Abbildung 35). Dieser Schritt ist essenziell, durch ihn ist es möglich, nebeneinanderliegende Punkte zu einem geschlossenen Volumen zusammenzufassen. Das resultierende Volumen wird anschließend in ein Mesh konvertiert.

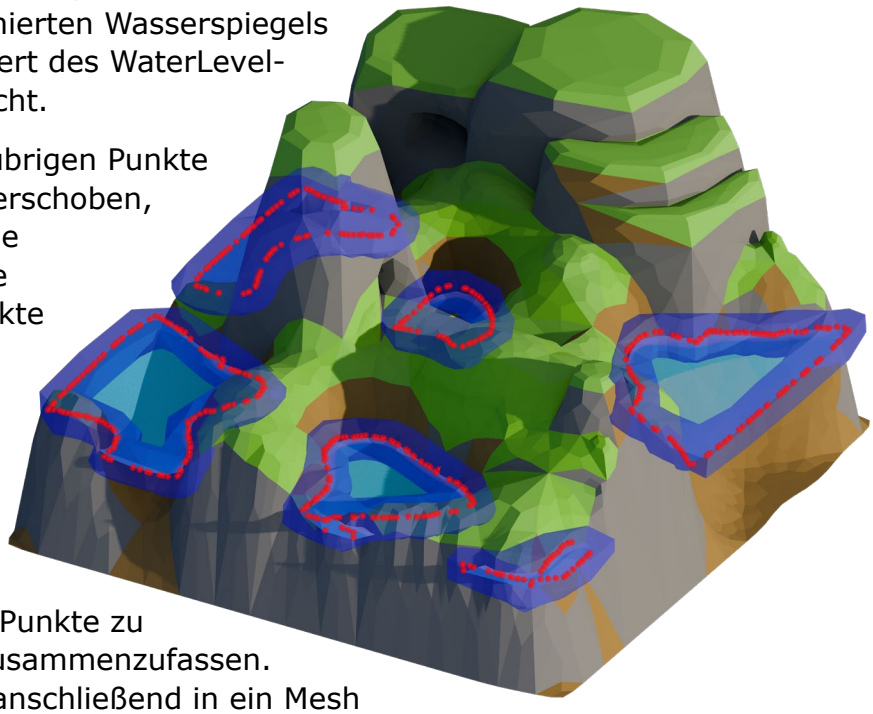


Abbildung 35: ‚ProceduralTerrain' mit visualisierten Punktwolken (rot) und hervorgehobenen Wasser-Volumen (blau) (eigene Darstellung).

Diese separaten Meshes lassen sich mithilfe der Mesh Island Node gruppieren. Über den Radiusparameter der Point to Volume Node lässt sich steuern, welche Punkte als zusammenhängend interpretiert werden.
Ist der Radius jedoch zu groß, beziehungsweise der Abstand zweier eigentlich getrennter Gewässer kleiner als der Abstand zu dem nächsten Punkt eines anderen Gewässers, kann es vorkommen, dass optisch getrennte Wasserflächen fälschlicherweise zu einer gemeinsamen Gruppe verschmolzen werden.

Nach der Gruppierung werden die einzelnen Mesh Islands in eine Repeat Zone geführt, welche in ihrer Funktionsweise einer for-loop ähnelt.
Innerhalb dieser Zone können die einzelnen Gruppen separat weiterverarbeitet werden.
Die Meshes werden erneut mithilfe der Mesh to Points Node in Punktwolken umgewandelt, auf eine identische Z-Position gebracht, abstands-basiert zusammengeführt (merged) und schließlich mit einer Convex Hull Node zu flachen, geschlossenen Wasserflächen zusammengeführt.

Was auf den ersten Blick trivial wirkt, eine distanz-basierte Gruppierung von Vertices, war in der Praxis erstaunlich herausfordernd.
Ab einem gewissen Grad an Komplexität existieren kaum noch Tutorials oder dokumentierte Workflows zu Geometry-Nodes.
Es gibt nur wenige wirklich erfahrene Anwender, und noch weniger von ihnen erstellen didaktisch aufbereitete Inhalte.

Nach ausgiebigen Versuchen mit der Geometry Proximity Node, um eine stabile Lösung für die Distanz-basierten Gruppierungen zu finden, konnten keinen konsistenten Ergebnis erzielt werden.
Der hier gewählte Volumen-basierte Ansatz in Kombination mit der Mesh Island Node war zwar nicht der direkteste, für mich persönlich jedoch der einfachste und verlässlichste Weg, das gewünschte Verhalten umzusetzen.
Der Ansatz hat letztlich gut funktioniert, auch wenn er in der Umsetzung deutlich komplexer und rechenintensiver war, als ich ursprünglich geplant hatte.

Rückblickend war das System zwar funktional und technisch interessant, aber für den eigentlichen Zweck der Operation überentwickelt.

Im weiteren Projektverlauf entstand eine wesentlich einfachere und elegantere, Boolean-basierte Lösung:
Dabei wird eine leicht herunterskalierte Plane auf die Höhe des definierten Wasserspiegels gesetzt und das Terrain anschließend über einen Difference Boolean davon abgezogen.
Das Ergebnis sind dieselben Wasserflächen, welche sich ebenfalls durch verschiedene Mesh Island Indices ansprechen lassen, jedoch mit einem Bruchteil der Komplexität. Das ursprüngliche System besteht aus 46 Nodes und eine Ausführungszeit von ~24ms, das neue besteht aus 8 Nodes und hat bei derselben Ausgangsgeometrie eine Ausführungszeit von ~6ms. Damit bleibt der erste Ansatz ein interessantes Experiment, zeigt aber, wie schnell sich prozedurale Systeme in ihrer eigenen Komplexität verlieren können. Das Beispiel unterstreicht die Bedeutung funktionaler Effizenz gegenüber technischer Finesse, wie wichtig es ist, architektonische Entscheidungen kontinuierlich zu hinterfragen und Vereinfachungen bewusst anzustreben.

4.1.4.5 ‚Polish‘

Nach der Materialzuweisung wird die Geometrie an die ‚Polish‘-Gruppe übergeben. Diese fasst mehrere optionale Funktionen zusammen, die das Terrain visuell verfeinern und zusätzliche Konfigurationsmöglichkeiten bereitstellen.

‚Extrude Grass‘:
Erlaubt die Extrusion von Flächen mit ‚Grass‘- und ‚TopGrass‘-Material ab einer definierbaren Mindesthöhe, um gezielt geometrische Tiefe zu erzeugen.

‚Snow‘:
Optional aktivierbares Feature, das ein Schnee-Material auf ausgewählte Flächen aufträgt.
Parameter wie Mindesthöhe, maximaler Neigungswinkel und optionale Extrusion des Schnees sind konfigurierbar.
Die erzeugten Schnee- und Gras-Extrusionen können anschließend Subdivided werden, um sie organisch in die bestehende Geometrie zu integrieren.

‚Scattering‘:
Ermöglicht die Platzierung von Objektkollektionen auf unterschiedlichen Materialien direkt im ‚ProceduralTerrain‘-System.
Nutzer können Kollektion, Dichte, Skalierung und einen zufälligen Skalierungsparameter individuell anpassen.

Nach Abschluss dieser Verarbeitungsschritte werden die Terrain-Geometrie und die parallel erzeugte Wasser-Geometrie über eine Join Geometry Node kombiniert.
Zum Abschluss wird automatisch eine UV-Map generiert, welche für die folgenden Systeme, insbesondere das ‚ScatterMeshes‘-System und die ‚ScatterCurves‘, benötigt wird.

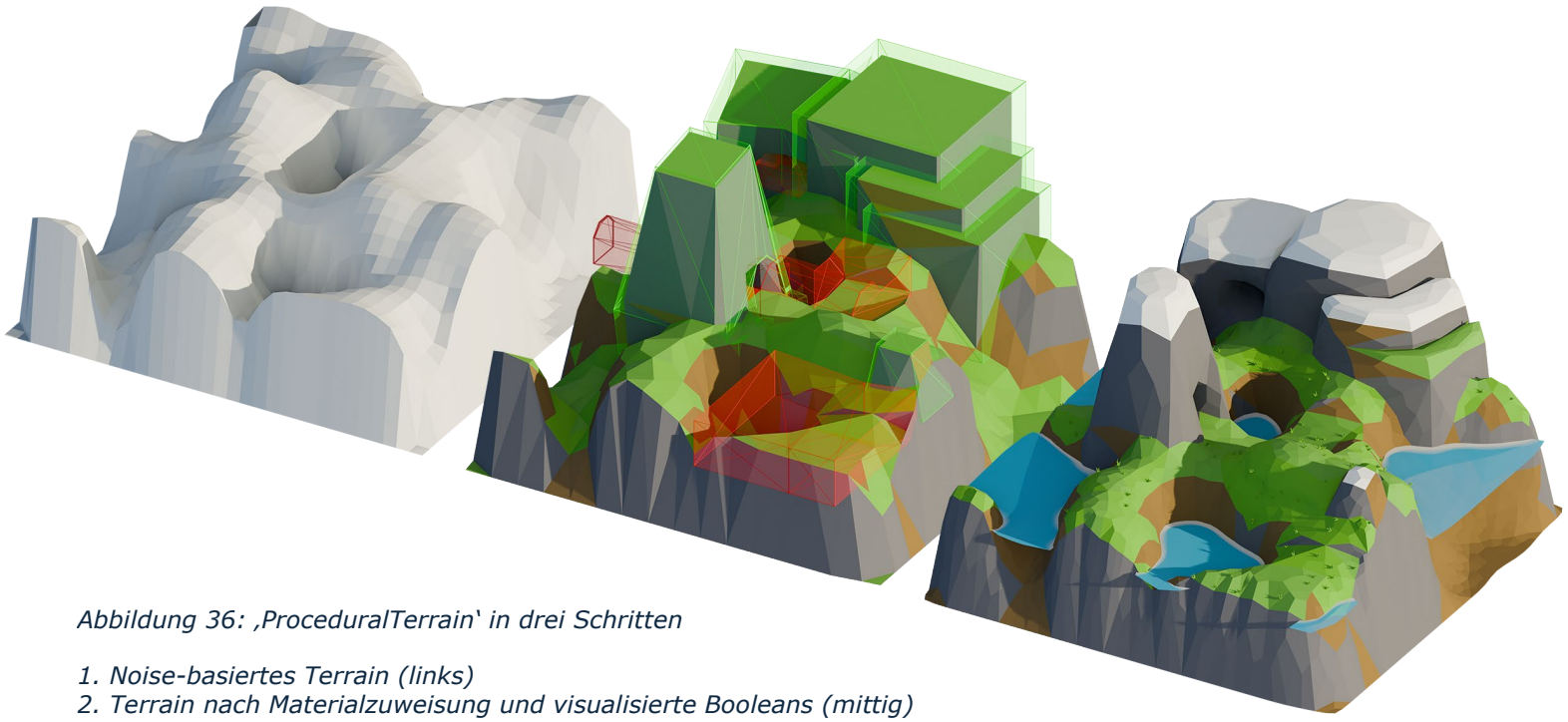


Abbildung 36: ‚ProceduralTerrain‘ in drei Schritten

1. Noise-basiertes Terrain (links)
2. Terrain nach Materialzuweisung und visualisierte Booleans (mittig)
3. Finales Terrain nach Wassergeneration und durchlaufen der ‚Polish‘-Gruppe (rechts) (eigene Darstellung).

4.1.5 Erweiterung zum ‚MeshTerrain‘

Das anfänglich Entwickelte und umfangreich beschriebene ‚ProceduralTerrain‘-System stellt die ursprüngliche Terrain-Implementierung des LPTK dar. Durch das Prinzip der noise-basierten Basisgeometrie-Generierung in Kombination mit den nutzerdefinierten Boolean Operationen bietet es große Freiheit bei der Gestaltung.

Während einer Testphase zeigte sich jedoch, dass die Noise-basierte Basisgeometrie zwar interessant und für explorative Workflows geeignet ist, jedoch stört, wenn man ein Terrain nach einer konkreten Vorgabe realisieren und somit volle Kontrolle behalten möchte. Dabei wurde die noise-basierte Basisgeometrie häufig auf Höhe Null gesetzt, um eine freiere Gestaltung ausschließlich mittels der Boolean Geometrien zu erzielen.

Auf Grundlage dieser Erkenntnis wurde das ‚MeshTerrain‘-System entwickelt. Das ‚MeshTerrain‘ funktioniert im Kern wie das Procedural Terrain und durchläuft alle im vorherigen Kapitel vorgestellten Operationsgruppen, mit dem Unterschied, dass die Basisgeometrie nicht durch Noise definiert wird, sondern das System direkt auf die Geometrie des Objekts auf, welches es zugewiesen wird, verwendet. So kann das ‚MeshTerrain‘ im Gegensatz zum ‚ProceduralTerrain‘ nicht bloß als eigenständiges System- sondern vielmehr als eine Art Post-Processing-Layer für jede Art von Objekt genutzt werden.

Abbildung 37: Mesh-Terrain auf Basis zweier einfacher Box-Geometrien (eigene Darstellung).

Durch die Entkopplung der prozeduralen Logik ergeben sich interessante und flexible Möglichkeiten zur Erstellung und Modifizierung von Objekten.

Die Basisgeometrie kann so durch alle erdenklichen Methoden erzeugt und prozedural überarbeitet werden. Beispielsweise können nutzerdefinierte Height Maps oder bestehende Terrain-Assets als Basisgeometrie genutzt werden.

Ebenso können verschieden manuelle Modellierungstechniken angewandt werden. Abbildung 37 zeigt, wie ein sehr simples Objekt (Wireframe orange visualisiert) interessante Geometrien erzeugen kann.

Besonders interessant ist dabei der Ansatz eine sculpting-basierte Basisgeometrie (Abbildung 38) mit dem System zu kombinieren. So kann der Nutzer ähnlich wie bei den kurven basierten Systemen malend die Geometrie des Terrains beeinflussen.

Die Kombination verschiedener Modellierungsansätze ist mit prozeduralen Workflows besonders interessant. Ein UV-basierter Ansatz zum Scattering wird im folgenden Kapitel besprochen.

Abbildung 38: ‚MeshTerrain‘ auf durch sculpting definierter Basisgeometrie (eigene Darstellung).

4.1.6 Scattering-Systeme

Scattering-Systeme gehören zu den am häufigsten eingesetzten prozeduralen Workflows zur effizienten Verteilung großer Mengen wiederholter Elemente wie Vegetation, Steinen, Pilzen oder kleineren Requisiten. Dabei werden Objektinstanzen automatisiert oder semiautomatisiert im Raum platziert und über Parameter wie Skalierung, Rotation oder Dichte regel- und/oder zufallsbasiert variiert.

Für das LPTK eignet sich ein malbasierter, interaktiver Workflow besonders gut. Ein solcher Ansatz kombiniert eine hohe gestalterische Freiheit (‚User Authority') mit einer direkten, non-destruktiven und intuitiven Bedienung, sodass der Arbeitsprozess dem traditionellen Level-Painting ähnelt.

4.1.6.1 Herausforderungen eines Weight-Paint-basierten Ansatzes

Ein gängiger Ansatz zur Instanzverteilung mithilfe von Geometry Nodes nutzt eine Kombination aus Distribute Points on Faces und Instance on Points Nodes, wobei die Punktdichte über ein Weight-Attribut gesteuert wird.
Während der Umsetzung eines solchen Workflows zeigten sich im prozeduralen Low-Poly-Kontext jedoch zwei wesentliche Einschränkungen:

1. Weight-Painting: funktioniert nicht auf unrealisierter Geometrie
   Geometry Nodes erzeugen „virtuelle", d. h. nicht realisierte Geometrie. Auf dieser kann kein Weight-Painting erfolgen, ohne dass der Modifier destruktiv angewendet wird, was dem non-destruktiven Grundprinzip des LPTK widerspricht.

2. Weight-Painting: ist an Vertex-Dichte gekoppelt
   Die Auflösung des Weight-Paintings hängt direkt von der Anzahl der verfügbaren Vertices ab. Da Low-Poly-Assets bewusst mit geringer Polygonanzahl modelliert werden, ist eine präzise räumliche Maskierung kaum möglich (siehe Abbildung 39). Dies führt entweder zu grober Verteilung oder zwingt zu unnötig hoher Topologie, was wiederum dem Low-Poly-Design widerspricht.

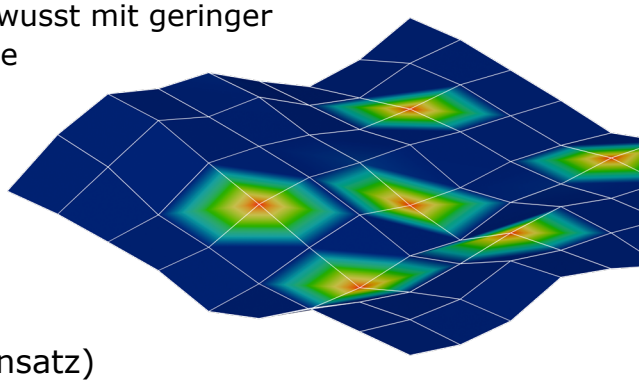


Abbildung 39: Visualisierung von Weightpainting auf niedrig aufgelöster Plane. Die Roten Regionen zeigen Vertices mit Weight 1.0, die blauen mit weight 0.0. Aufgrund der niedrigen Auflösung wirkt sich der Weight-Paint auf die umliegenden Faces aus (eigene Darstellung).

4.1.6.2 UV-basiertes Curve-Scattering (LPTK-Ansatz)

Zur Lösung dieser Limitierungen wurde im LPTK ein UV-basiertes Curve-Scattering-System entwickelt. Es nutzt die Curve Sculpting-Werkzeuge von Blender (ursprünglich für Hair-Workflows konzipiert) und kombiniert diese mit kurvenbasierter Instanziierung in Geometry Nodes.

Dabei können Kurven direkt auf einem Objekt platziert werden, deren Geometrie innerhalb der Geometry Nodes genutzt werden kann, um kurvenbasierte prozedurale Assets wie Bäume, aber auch Standardobjekte zu instanziieren.

Hauptvorteile des UV-basierten Curve-Scattering-Verfahrens:

1. Topologie unabhängige Präzision
   Die Maskierung der Scatter-Bereiche erfolgt im UV-Raum, statt über Vertex-Daten. Dadurch bleibt die Präzision vollständig erhalten auch bei Low-Poly-Modellen,

2. Kurven statt Punkte = volle Kontrolle für Kurven-basierte Assets
   Jede Instanz basiert auf einer editierbaren Kurve, die nachträglich mithilfe der Curve Sculpting Brushes transformiert, verlängert, gekrümmt oder gelöscht werden kann. Kurven-basierte Asset-Systeme wie der in 4.1.1 beschriebene ‚FunkyTree' erhalten somit direkt editierbare Kurven als Geometrie-Input, während statische Assets lediglich den Startpunkt der Kurve zur Instanziierung nutzen.

Einschränkungen und Anforderungen dieses Verfahrens:

1. UV-Maps sind erforderlich
   Das jeweilige Objekt muss UV-unwrapped sein, was einen zusätzlichen Setup-Schritt zur Automatisierung erfordert. Sobald Mesh-Geometrie hinzugefügt oder entfernt wird, muss diese neu unwrapped werden. Je nach Topologie kann dies Auswirkungen auf die Performance des Systems haben.

2. UV-Änderungen wirken sich auf Positionierung der Instanzen aus
   Werden Flächen hinzugefügt, entfernt oder verschoben werden UV-Maps neu unwrapped, wodurch es zur Positionsänderungen der Instanzen kommen kann. In der Praxis bleibt dies jedoch im Normalfall unkritisch, solange die Ausgangsgeometrie nicht grundlegend verändert wird.

Im LPTK sind zwei Scattering-Systeme implementiert.

ScatterMeshes

1. Fügt dem Zielobjekt ein „Empty-Hair-Object" hinzu
2. User kann eigene Mesh-Objekte oder Collections als Instanzquellen wählen
3. Platzierung erfolgt über Curve Sculpting Brushes
4. Parameter wie Skalierung und Rotation können im Modifier angepasst werden

ScatterCurves

1. Funktioniert analog zu ScatterMeshes
2. Instanziert jedoch Kurven basierte Assets (z. B. Baumsysteme ‚Birch', ‚Pine1' oder ‚FunkyTree')
3. Editierbare Kurven dienen direkt als Geometrier Input für prozedurale Generierung (nicht reduziert auf Startpunkt)
4. Erweiterbar um weitere Asset-Systeme

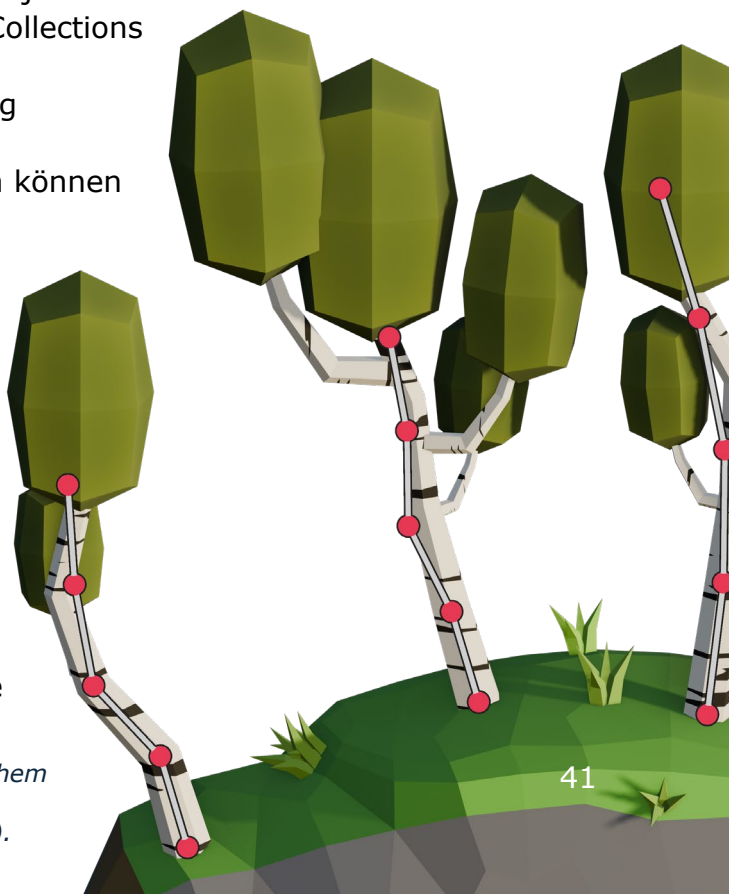


Abbildung 40: Darstellung eines ‚ScatterCuves'-Systems auf welchem drei "Haare" (Kurven) platziert wurden, welche mithilfe der Hair Sculpting Brushes angepasst werden können (eigene Darstellung).

4.2 Entwicklung des Add-ons in Python

Wie bereits in 2.4.1 beschrieben, bestehen Blender-Add-ons aus einem oder mehreren Python-Skripten, welche in einem ZIP-File zusammengefasst werden können. In diesem Abschnitt wird die Entwicklung des LPTK-Add-ons beschrieben. Während die Geometry Node Setups (Kapitel 4.1) den prozeduralen Kern des Systems bilden, vereint das Add-on diese in einer einheitlichen, zugänglichen Benutzeroberfläche und stellt die funktionale Verbindung zwischen Nutzerinteraktion und den zugrundeliegenden Node Setups her.

Das Ziel der Add-on-Entwicklung war es, eine klare, leicht verständliche und erweiterbare Struktur zu schaffen, sowohl im Frontend (Nutzeroberfläche) als auch im Backend (Code- und Datenstruktur). Dabei standen Verständlichkeit, Anpassbarkeit und Stabilität über der formalen Perfektion des Codes.
Das System soll es ermöglichen, dass spätere Erweiterungen, etwa durch neue Node Setups, oder zusätzliche Funktionen, mit minimalem Aufwand, umgesetzt werden können.

Die Entwicklung des Add-ons umfasst:

- Die Entwicklung der grundlegenden Struktur zum Einlesen bestehender Geometry Node Systeme und einfügen dieser in neue Szenen
- Die Entwicklung der Benutzeroberfläche
- Die Implementierung des Game-Engine Syncs
- Die Entwicklung des semi-automatischen Thumbnail-Renderes

Ein Großteil der Implementierung wurde manuell konzipiert, mithilfe KI-gestützter Prototypen entwickelt und anschließend auf das eigene Verständnis hin angepasst. Auf diese Weise konnten früh funktionierende Ergebnisse erzielt werden, die folgend so überarbeitet wurden, dass zu jedem Entwicklungsstand ein vollständiges Verständnis der Codebasis bestand. Dadurch wurde verhindert, dass unübersichtlicher oder schwer wartbarer Code entsteht.

Anstelle objektorientierter Muster wurde, wenn möglich, bewusst ein linearer, klar lesbarer Aufbau gewählt bspw. If/else-Strukturen statt komplexer Klassen. Dieser Ansatz erleichtert das Verständnis des Ablaufs und vereinfacht zukünftige Anpassungen und Ergänzungen ohne lange Einarbeitungszeit.

Zunächst wird das Einlesen der einzelnen Node Trees mit einem Skript behandelt.

4.2.1 Einlesen der Node Trees

Um die erstellten Geometry Node Setups im Add-on verfügbar zu machen, werden diese in einer festen Struktur organisiert. Jedes Setup wird in einer separaten .blend-Datei gespeichert. Dabei trägt sowohl die Datei als auch der darin enthaltene Node Tree denselben Namen. Alle entsprechenden Dateien befinden sich im Unterordner ‚/my_geonodes', der im Root-Verzeichnis des Add-ons abgelegt ist.

Zusätzlich enthält das Root-Verzeichnis die Datei ‚geo_nodes.json', welche sämtliche Setups beschreibt. Für jeden Node-Tree werden dort die folgenden Attribute hinterlegt:

Beispielhafte geo_nodes.json Struktur für das ‚MeshTerrain'-Asset im LTPK:

```
"MeshTerrain": {
        "filename": "MeshTerrain.blend",
        "node_type": "CUBE",
        "category": "Terrain",
        "thumbnail": "MeshTerrain.jpg"
    },
```

Das Add-on liest diese Datei beim Start ein und überführt die Informationen in ein Dictionary. Auf diese Weise lassen sich die Pfade zu den Node Trees sowie die zugehörigen Metadaten flexibel abrufen.

Das folgende Code-Snippet zeigt den Aufbau der Datenstruktur aus der .json-Datei:

```
geo_nodes[name] = {
        "filepath": os.path.join(geonodes_folder, data["filename"]),
        "node_type": data["node_type"],
        "category": data["category"],
        "thumbnail": data["thumbnail"]
            if os.path.isabs(data["thumbnail"])
            else os.path.join(thumbnails_folder, data["thumbnail"])
    }
```

Durch diese Vorgehensweise ist es möglich, neue Node Trees einfach durch Hinzufügen einer .blend-Datei im entsprechenden Verzeichnis, sowie eines Eintrags in ‚geo_nodes.json' in das Add-on zu integrieren, ohne dass Anpassungen im Quellcode erforderlich sind.

4.2.2 ‚Node-Types‘

Wie bereits in 4.2 beschrieben war ein ausschlaggebendes Argument gegen die built-in Asset-Library von Blender die fehlende Unterstützung für Kontextabhängige Operationen nach Einfügung eines Assets.

Im Asset-Library-Workflow können einzelne Assets über Drag-and-Drop direkt in die Szene geladen werden. Dies geschieht jedoch, ohne dass der Nutzer anschließend in einen spezifischen Arbeitsmodus bspw. Edit-, Shading- oder Sculpting-Mode überführt werden kann.

Um diese Funktionalität im Add-on bereitzustellen, erhält jeder Geometry Node Tree einen sogenannten Einfügungskontext. Dieser definiert, auf welcher Geometrie und mit welcher Initialkonfiguration der Node Tree in die Szene geladen wird. Nach der Einfügung werden kontextspezifische Instruktionsketten ausgeführt, die den Nutzer automatisch in die passende Blender-Umgebung bringen, um das jeweilige Asset unmittelbar weiterbearbeiten zu können.

Um diese Unterscheidung zu definieren, wurde das ‚node_type‘-Konzept entwickelt über welches zwischen verschiedene Einfügungskontexten der jeweiligen Node Trees unterschieden werden kann.

Zum jetztigen Zeitpunkt wird unterschieden zwischen: ‚CURVE_LOW‘, ‚PLANE‘, ‚TERRAIN2‘, ‚CUBE‘, ‚GATE‘, ‚CURVE‘ und ‚SCATTER‘, wobei einige Typen von mehreren Node Trees „genutzt“ werden und andere komplette special-case Lösungen sind.

Beispielhafte Instruktionen für ‚node_type‘: ‚CURVE_LOW‘:

```
 1. if self.node_type == "CURVE_LOW":
 2.                 bpy.ops.curve.primitive_bezier_curve_add(enter_editmode=True,
location=context.scene.cursor.location)
 3.                 obj = bpy.context.active_object
 4.                 obj.data.resolution_u = 3
 5.                 bpy.ops.curve.select_all(action='SELECT')
 6.                 bpy.ops.curve.delete(type='VERT')
 7.                 bpy.ops.wm.tool_set_by_id(name="builtin.draw")
 8.                 settings = context.scene.tool_settings.curve_paint_settings
 9.                 settings.depth_mode = 'SURFACE'
10.                 settings.use_stroke_endpoints = True
```

Wenn das ausgewählte Setup beispielsweise den Typ ‚CURVE_LOW‘ besitzt, wird zunächst eine Bézier-Kurve an der Position des 3D-Cursors erzeugt und der Nutzer automatisch in den Edit Mode versetzt. Anschließend werden die Standardpunkte der Kurve selektiert und entfernt, sodass ein leeres Kurvenobjekt als Ausgangspunkt entsteht.
Danach versetzt das Skript den Nutzer in den „Draw“-Mode, in dem neue Kurvensegmente freihändig gezeichnet werden können. Abschließend wird der „Surface“-Mode aktiviert, wodurch die Kontrollpunkte der Splines direkt auf vorhandene Oberflächen projiziert werden können, bspw. anderen LPTK-Setups. Zusätzlich wird die Option „use_stroke_endpoints = True“ gesetzt, sodass nur der erste Kontrollpunkt der Spline auf einer bestehenden Geometrie platziert wird.

Praktisch ist das bspw. für die Bäume, bei welchen nur die Wurzel auf den bestehenden Objekten platziert werden soll, der Stamm aber nicht.

4.2.3 ‚Node-Spawning‘

Nachdem die einzelnen Einsetzungs-Kontexte beispielhaft erklärt wurden, gehe ich nun auf die konkrete Einsetzungsimplementierung der Assets ein.

1. Laden der Node Trees aus den externen Dateien

Zum Importieren von Daten wie Objekten, Materialien oder Geometry Node Trees eignet sich die „Append“-Funktion. Diese kopiert Daten aus einer Blender-Datei in ein anderes, ohne dabei eine Beziehung zur originalen Ausgangsdatei herzustellen.

Sobald der Nutzer im LPTK-Panel (Kapitel 4.2.4) ein Asset auswählt, erhält der Operator den Dateipfad sowie den Namen des zu ladenden Node-Trees.
Über den Kontextmanager bpy.data.libraries.load() wird die entsprechende .blend-Datei geöffnet und geprüft, ob der gewünschte Node Tree enthalten ist:

```
1. def execute(self, context):
2. self.report({'INFO'}, f"Spawning {self.node_group_name}...")
3. try:
4. with bpy.data.libraries.load(self.filepath, link=False) as (data_from, data_to):
5. if self.node_group_name in data_from.node_groups:
6. data_to.node_groups.append(self.node_group_name)
7. else:
8. self.report({'ERROR'}, f"Node group '{self.node_group_name}' not found in {self.filepath}")
9. return {'CANCELLED'}
```

Dieser Schritt importiert ausschließlich den benötigten Node-Tree, unabhängig davon, welche weiteren Daten die .blend-Datei enthält.

2. Ausführung des jeweiligen Einfügungskontext (‚Node-Type‘)

Nach dem Import des ausgewählten Node Trees durchläuft das Skript die ‚NodeType‘-Prüfung und führt je nach ausgewähltem Asset eine der definierten If-Bedingungen aus. In diesen wird wie in 4.2.2 beschrieben immer ein Objekt, bspw. eine Kurve oder ein Würfel, in die Szene eingefügt und kontextspezifische Operationen vorgenommen.

```
10. if self.node_type == "CURVE_LOW": # Node-Types / Einsetzungskontext
11. elif self.node_type == "PLANE":
          …
16. elif self.node_type == "SCATTER":
          …
```

3. Zuweisung des Geometry Nodes Modifiers

Sobald das korrekte Ausgangsobjekt erzeugt und dem Kontext entsprechend vorbereitet wurde, wird dem Objekt ein Geometry Nodes Modifier hinzugefügt, welchem der importierte Node Tree zugewiesen wird.

```
18. obj = bpy.context.active_object # Auswahl des korrekten, im Node-Type erstellten Objekts
19. obj.name = self.node_group_name # Namenszuweisung des erstellten Objekts
20. modifier = obj.modifiers.new(name="GeometryNodes", type='NODES') # Zuweisung eines
GeometryNodes modifiers
21. if bpy.data.node_groups.get(self.node_group_name):
22. else:
23. self.report({'INFO'}, f"Successfully spawned {self.node_group_name}")
24. return {'FINISHED'}
```

Um mit diesen Funktionalitäten zu interagieren, wurde eine Nutzeroberfläche implementiert, dessen Design und Umsetzung im folgenden Kapitel erläutert werden.

4.2.4 Nutzeroberfläche

Die Oberfläche des Add-ons lässt sich dank der Blender Python API nahtlos in die bestehende Blender-UI integrieren. Entwickler haben dadurch Zugriff auf nahezu alle Bereiche der Software und können diese beliebig anpassen oder erweitern.

Blender bietet unterschiedliche Oberflächen, für verschiedene Workflows, welche sich individuell anordnen und konfigurieren lassen. Zentral sind hierbei die sogenannten Editor Types[50] wobei der „3D-Viewport" den Arbeitsbereich zur Navigation und Modellierung im dreidimensionalen Raum bildet. Zur Integration eigener Add-on-Oberflächen existieren keine klaren Richtlinien. Der letzte offizielle User Interface Design Guide wurde 2019 mit Blender 2.8 veröffentlicht[51]. Dennoch haben sich durch die stetige Entwicklung von Third Party Add-ons in verschiedenen Kategorien informelle Best Practices und Designkonventionen ergeben.

Um eine leicht zugängliche und übersichtliche Nutzeroberfläche zu gewährleisten, benötigt das LPTK Platz für Vorschaubilder, Knöpfe und Tooltips. Für Add-ons mit einem Asset fokussiertem Inhalt und vergleichbarem Funktionsumfang, welche nicht direkt auf bestehende Blender Funktionen aufbauen, ist die Sidebar[52], auch „N-Panel" genannt, ein idealer Ort. Hier können verschiedenste Tools übersichtlich angeordnet und bei Bedarf einzeln ein- und ausgeklappt werden.

Abbildung 41 zeigt die realisierte Oberfläche des LPTK. Sie lässt sich im 3D-Viewport über das N-Panel öffnen und ist in drei separat ein- und ausklappbare Unterbereiche gegliedert.

1. **‚Asset -Panel':**
   Hierüber können die einzelnen Node Setups in die Szene geladen werden. Dargestellt werden die Assets in einem vertikalen Layout mit Vorschaubild und zughörigem Knopf.
   Über Kategorien lassen sich unterschiedliche Asset-Gruppen ein- oder ausblenden. In Abbildung 41 ist die Kategorie ‚Plants' ausgewählt.
2. **‚Collection Exporter':**
   Darunter befindet sich die Oberfläche des ‚Collection Exporters', mit welchem die Assets kollektionsbasiert exportiert werden können (siehe 4.2.5)

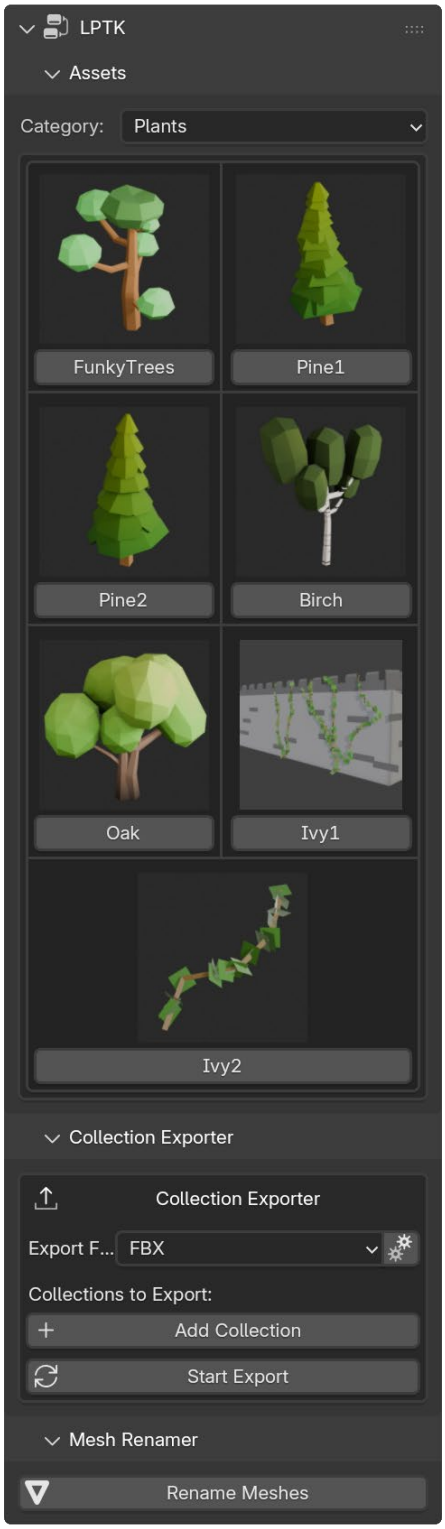


Abbildung 41: Nutzeroberfläche des LPTK (eigene Darstellung).

[50] Die verschiedenen Editoren können verschiedene Daten des Projekts anzeigen. So lassen sich bspw. im Timeline-Editor Keyframes einsehen und setzen oder Shader im Shader Editor erstellen.
[51] https://developer.blender.org/docs/release_notes/2.80/python_api/ui_design/
[52] https://docs.blender.org/manual/en/latest/interface/window_system/regions.html

3. ‚Toolbox‘:

Im Unteren Bereich befindet sich die ‚Toolbox‘, welche Platz für verschiedene Hilfreiche Funktionen bei der Arbeit mit dem LPTK bietet. Zum jetzigen Zeitpunkt findet sich an dieser Stelle der ‚Mesh Renamer‘, mit welchem automatisch Objekt- und Mesh-Namen angeglichen werden können.

4.2.4.1 Implementierung der Oberfläche anhand des Asset Panels

Die Umsetzung der Benutzeroberfläche erfolgt über die Klassenstruktur der Blender Python API und basiert hauptsächlich auf der vordefinierten Panel-Basisklasse[53]. Jedes Panel wird als eigene Klasse definiert, die von bpy.types.Panel erbt. Ein übergeordnetes Panel („LPTK“) fungiert dabei als Container, dem die drei Subpanels über ihre parent_id zugeordnet werden. Blender erkennt diese automatisch und rendert sie im N-Panel.

Ein vereinfachter Auszug zeigt den grundlegenden Aufbau:

```
1. class GEO_PT_panel(bpy.types.Panel):
2.     bl_label = "LPTK"
3.     bl_space_type = 'VIEW_3D'
4.     bl_region_type = 'UI'
5.     bl_category = "LPTK"
```

Jede Panel-Klasse enthält eine draw()-Methode, die beim Rendern der Oberfläche aufgerufen wird. Darin werden alle UI-Elemente definiert, also Knöpfe, Textfelder oder Dropdown-Menüs. Im Fall des Asset Panels liest die Methode automatisch alle verfügbaren Assets aus dem ‚geo_nodes‘-Dictionary (in 4.2.1 erklärt) und stellt sie im Interface dynamisch dar:

Das folgende vereinfachte Beispiel verdeutlicht das Prinzip:

```
1. for name, data in geo_nodes.items():
2.     layout.template_icon(icon_value=thumbnail[name])
3.     layout.operator("geo.spawn", text=name)
```

Die Schleife erzeugt für jedes gespeicherte Asset ein Vorschaubild und den zugehörigen Lade-Button. Ein Klick auf den Knopf ruft den Operator ‚geo.spawn‘ (wie in 4.2.3 gezeigt) auf, der das entsprechende Geometry Node Setup in die Szene lädt.

Das ‚Category‘-Menü, mit dem die angezeigten Assets gefiltert werden können, wird ebenfalls automatisch aus den in ‚geo_nodes‘ hinterlegten Metadaten abgeleitet:

```
1. bpy.types.Scene.geo_spawner_category = bpy.props.EnumProperty(
2.     items=get_categories,
3.     name="Category",
4.     description="Filter assets by category",
```

Auf diese Weise werden die erweiterbaren Aspekte der Oberfläche dynamisch erzeugt. Neue Assets erscheinen automatisch im Panel, sobald sie in der JSON-Datei registriert werden, ohne dass zusätzlicher Code angepasst werden muss. Neue Panels oder Operatoren lassen sich problemlos durch Ergänzung weiterer Klassen erzeugen, während der bestehende Code unverändert bleibt.

[53] https://docs.blender.org/api/current/bpy.types.Panel.html

4.2.5 Integration des Game-Enginge-Syncs

Damit 3D-Modelle in der Spieleentwicklung verwendet werden können, müssen diese aus der jeweiligen Modellierungsumgebung in die entsprechende Game-Engine übertragen werden.

Im Blender Geometry Nodes-Workflow bedeutet dies, dass die Modifier zunächst angewendet, das Objekt anschließend in ein Mesh konvertiert und in einem gängigen Format (z. B. .fbx) exportiert werden muss.
Dabei geht der non-destruktive Workflow verloren.

Wie in Kapitel 2.5.4 beschrieben, bietet der Industriestandard Houdini mit der eigenen Houdini Engine eine direkte Integration prozeduraler Systeme in Game Engines wie Unity und Unreal.
Diese enge Verknüpfung bietet einen unkomplizierten Workflow dar, da die prozedural generierten Assets non-destruktiv und in Echtzeit innerhalb der Game Engine angepasst werden können. Ein erneuter Export oder Import der Geometrie entfällt vollständig, was die Non-Destruktivität des Workflows erhält und Iterationszeiten sowie Fehleranfälligkeit deutlich reduziert.

Blender bietet keine native Integration der Geometry Nodes in gängige Game Engines, was damit zusammenhängt, dass dafür spezielle Schnittstellen entwickelt werden müssten, welche auf proprietären Systemen kommerzieller Engines aufbauen würden, was für eine Open-Source Projekt unpassend wäre.

Es existieren jedoch einige Third-Party-Ansätze, die versuchen, diese Lücke zu schließen. Zu den bekanntesten zählen Altermesh für die Unreal Engine sowie BEngine, welche sowohl Unity als auch Unreal unterstützt.
Beide Werkzeuge ermöglichen eine eingeschränkte Synchronisierung von Geometry Node-Systemen zwischen Blender und der jeweiligen Game Engine mit der Möglichkeit die in Blender exponierten Parameter direkt anzupassen, ohne erneuten Ex- und Import.
Da es sich hierbei jedoch um kleine, unabhängige und kommerzielle Projekte einzelner Entwickler handelt, ist ihre Langzeitstabilität stark von Updates der Engines und von Blender selbst abhängig.

Zum jetzigen Zeitpunkt scheint Altermesh nicht mehr unterstütz zu werden (letztes Update am 21. Mai 2024, ohne weitere Kommunikation seitens des Entwicklers auf dem offiziellen Discord-Server). Der Entwickler der BEngine hingegen engagiert sich noch aktiv mit der Entwicklung des Tools und geht auf Nutzerfeedback und spezifische Probleme ein.
Jede neue Version kann jedoch zu Komplikationen führen und das Risiko für eine Ende der Unterstützung seitens der Entwickler ist hoch. Außerdem führt das Aufbauen auf bestehende thrid-party Lösungen zu weiterem Installationsaufwand und ggf. mehr Kosten seitens der LPTK-Nutzer. Deshalb habe ich mich bewusst gegen die bestehende Lösung von Externen als Synchronisations-Tool entschieden. Dennoch war klar, dass für ein nutzbares Werkzeug ein non-destruktiver, einfacher und schneller Workflow unersetzlich ist, weshalb eine eigene Lösung, der ‚Collection Exporter' entwickelt wurde, welcher im nächsten Kapitel behandelt wird.

4.2.5.1 Collection Exporter

In Game-Engine-Umgebungen sind Assets typischerweise in Ordnerstrukturen organisiert, wobei die 3D-Modelle in spezifischen Unterordnern abgelegt werden. Standardmäßig werden die Modelle manuell über das Export-Menü in gängigen Formaten wie .fbx oder .gltf in den entsprechenden Ordnern gespeichert. Dieser manuelle Workflow ist jedoch zeitaufwendig, potenziell destruktiv und fehleranfällig, insbesondere bei komplexen Szenen mit vielen Objekten oder bei häufigen Iterationen während der Entwicklungsphase.

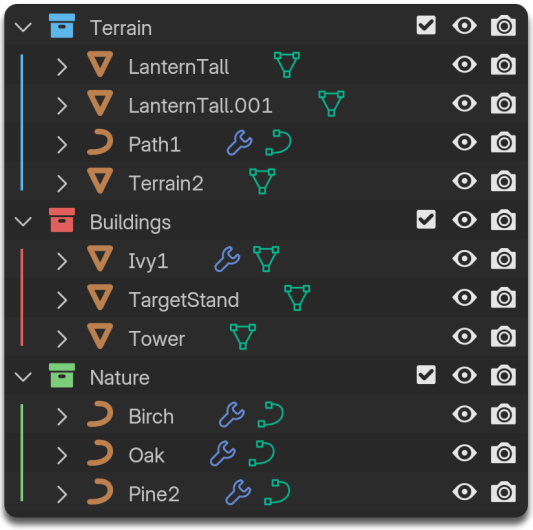


Abbildung 42: Beispielhafte Darstellung einer Kollektionsstruktur im Outliner (eigene Darstellung).

Kollektionen[54] sind ein Werkzeug zur Organisation in Blender. Sie funktionieren ähnlich wie Ordner und ermöglichen es, verschiedene Objekte logisch zu gruppieren, ohne diese in eine Transformationsbeziehung zu stellen (im Gegensatz zum Parenting). Diese Kollektionen sind die Basis der implementierten Export-Logik.

Abbildung 42 zeigt eine einfache Kollektionsstruktur, hierbei wurden drei Kollektionen angelegt und mit verschiedenen Objekten gefüllt, um diese logisch voneinander zu trennen.

Um einen möglichst einfachen und non-destruktiven Iterationsworkflow zu bieten, wurde der ‚Collection Exporter' entwickelt, welcher sich direkt im Add-on-Panel des LPTK befindet.

Über ihn können Kollektionen innerhalb der Blender-Szene einem beliebigen Pfad zugewiesen werden, wie in Abbildung 43 dargestellt.

Formatspezifische Exporteinstellungen können über das Zahnrad konfiguriert werden. Zum Anlegen neuer Kollektionen im Export-Workflow können diese über den „Add Collection"-Knopf hinzugefügt werden. Drückt der Nutzer den „Start Export"-Knopf, wird die zentrale Methode des Exporter-Skripts ausgeführt und die Objekte der zugehörigen Kollektionen in die entsprechenden Verzeichnisse exportiert.

Die Implementierung des ‚Collection-Exporters' wird im folgenden Kapitel besprochen.

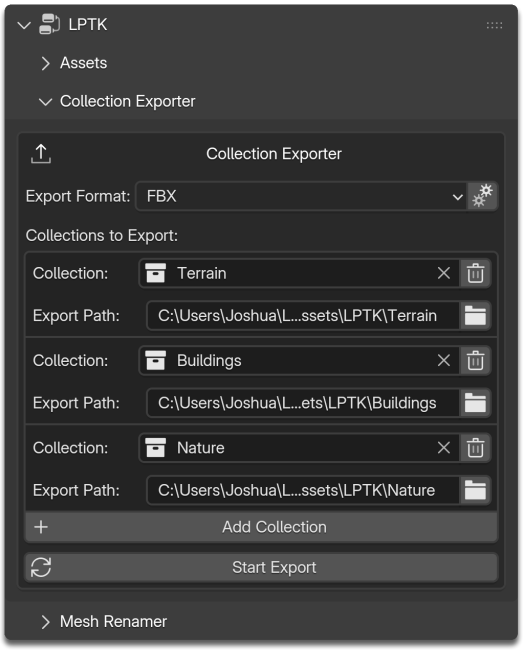


Abbildung 43: ‚Collection Exporter'-Panel innerhalb des LPTK Add-ons (eigene Darstellung).

[54] https://docs.blender.org/manual/en/latest/scene_layout/collections/collections.html

4.2.5.2 Implementierung der Export-Logik

Die Export-Funktion iteriert über alle vom Nutzer definierten ‚Collection-Entries', verarbeitet deren Inhalt und exportiert die überarbeiteten Meshes in das gewünschte Zielformat. Der Workflow bleibt dabei vollständig non-destruktiv, da ausschließlich temporäre Objektkopien genutzt werden. Die folgenden Schritte fassen die grundlegende Funktionsweise zusammen:

1. Duplikation der Export-Objekte

Für jedes Objekt wird zunächst überprüft, ob es einen exportierbaren Typ besitzt („MESH", „CURVE" oder „FONT") Anschließend wird eine temporäre Kopie erzeugt. Dies stellt sicher, dass der Exportprozess die Ursprungsobjekte nicht verändert.

```
233. for obj in collection.objects:
234.  if obj.type in {'MESH', 'CURVE', 'FONT'}:
235.      dup = obj.copy()
236.      dup.data = obj.data.copy()
```

2. Konvertierung der Objekte in ein Mesh

Anschließend werden die zugelassenen Objekte in ein Mesh konvertiert, dabei werden die Geometry Nodes Modifier angewandt und die prozedural erzeugte Geometrie realisiert.

```
243. bpy.ops.object.convert(target='MESH')
```

3. ‚Vertex Color Baking Automation'

Falls vom Nutzer aktiviert, wird folgend ein automatisierter Bake-Prozess ausgeführt, der die Materialfarben in ein Vertex-Color-Attribut überträgt (wird im folgenden Kapitel besprochen).

4. Export an den spezifizierten Pfad

Danach wird das Objekt anhand seines Namens und dem in der ‚Export Collection' definierten Pfades exportiert, dabei werden die in den Exporteinstellungen festgelegten Parameter berücksichtigt (Forward Axis etc.).

5. Entfernung der Duplikate aus der Datei

Abschließend wird das temporäre Duplikat vollständig aus der Szene entfernt, sodass der Nutzer ohne Veränderung an seinem Projekt weiterarbeiten kann.

```
275. bpy.data.objects.remove(dup, do_unlink=True)
```

Das Ergebnis ist eine Reihe einzelner FBX-Dateien im definierten Zielordner, während die originale Blender-Datei unverändert bleibt. Nutzer können durch diesen Workflow mit einem Knopf ihre Game-Engine Umgebung mit ihrer Blender Szene synchronisieren.

4.2.5.3 ‚Vertex Color Baking Automation‘

Wie in Kapitel 2.3 gezeigt, verzichtet der Low-Poly-Artstyle zwar häufig auf komplexe Materialien, dennoch können einfache Farbverläufe oder leichte Variationen die visuelle Qualität deutlich erhöhen (siehe Abbildung 44). Solche Gradients werden in Blender typischerweise über Shader erzeugt, die jedoch von Game-Engines nicht direkt übernommen werden können. Um den Effekt zu übertragen, müssten die Materialien entweder in der Engine nachgebaut oder als Textur mit korrekt erstellten UV-Maps exportiert werden. Beides manuelle und destruktive Arbeitsschritte, die dem non-destruktiven LPTK-Workflow widersprechen.

Um einfache Materialeffekte dennoch automatisiert exportieren zu können, wurde innerhalb des ‚Collection Exporters‘ eine Vertex-Color-Baking Automation integriert. Vertex Colors werden direkt im Mesh gespeichert und können ohne zusätzliche Materialien von allen gängigen Game-Engines verwendet werden.

Wird Vertex Color Baking in den Exportoptionen aktiviert, durchläuft jedes exportierte Objekt nach der Mesh-Konvertierung den folgenden Ablauf:

1. Erstellen des Vertex-Color-Attributs

Auf der Face Corner Domain wird ein neues Farb-Attribut erzeugt. Existiert dieses bereits, wird es überschrieben.

2. Konfiguration der Bake-Einstellungen

Das Skript wechselt in die Cycles-Renderengine, aktiviert den Diffuse Bake und setzt „Vertex-Colors“ als Ziel. Direkte und indirekte Beleuchtung werden deaktiviert, sodass ausschließlich die Materialfarbe gebacken wird.

3. Bake-Prozess

Die Farbinformationen werden in das Vertex-Color-Attribut geschrieben.

4. Ersetzen des Materials

Abschließend wird das Ursprungsmaterial auf dem temporären Objekt durch einen einfachen Diffuse-Shader ersetzt, welcher das gebackene Vertex-Color-Attribut als Base Color verwendet. Dieses Setup kann von gängigen Game-Engines direkt interpretiert werden.

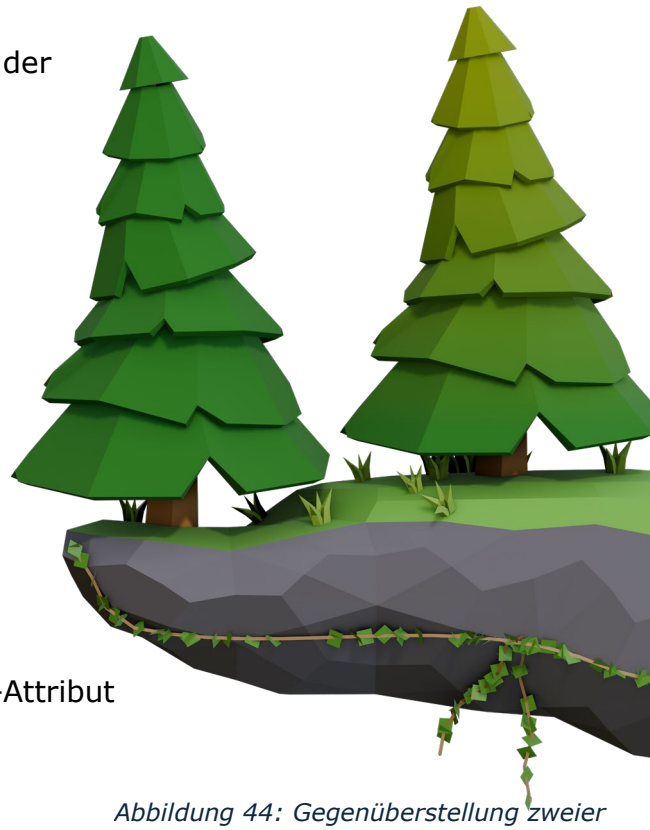


Abbildung 44: Gegenüberstellung zweier gleicher Tannen, links ohne Farbverlauf, rechts mit Farbverlauf (eigene Darstellung).

Dieses Verfahren ermöglicht einen vollständig automatisierten und non-destruktiven Export von einfachen Farbvariationen, ohne dass UV-Maps oder Texturen erstellt werden müssen. Gerade im Low-Poly-Kontext stellt dies eine schnelle Möglichkeit dar, Farbvariationen aus Blender in Game-Engines konsistent zu übertragen.

4.2.6 Entwicklung des ‚Thumbnail-Renderers‘

Wie in den vorherigen Kapiteln beschrieben, ist die einfache Erweiterbarkeit des LPTK ein zentrales Ziel. Die Kombination aus der klaren Ordner- und JSON-Struktur (siehe 4.2.1) und dem dynamisch generierten Asset-Panel (siehe 4.2.4) ermöglicht eine unkomplizierte Integration neuer Node Setups.
Damit diese neuen Assets nicht nur funktional, sondern auch visuell konsistent in der Benutzeroberfläche eingebunden werden, wurde ein separater, semi-automatischer ‚Thumbnail-Renderer‘ entwickelt. Dieser ermöglicht die Erzeugung einheitlicher Vorschaubilder für alle Node Setups.

Der Renderer ist vollständig in einer separaten Blender-Datei implementiert, die unter „thumbnailRenderer.blend“ im Root-Verzeichnis des Add-ons abgelegt ist. Diese Datei enthält eine vorkonfigurierte Szene, bestehend aus:

- einer Kamera mit fixer Perspektive,
- einer Beleuchtungssituation,
- sowie einer dafür vorgesehenen Kollektion namens ‚GeoNodes‘, in welche alle zu rendernden Assets platziert werden.

Neben dem 3D-Viewport befindet sich in der Datei ein geöffneter Text-Editor, der ein Python-Skript enthält. Dieses automatisiert den gesamten Rendervorgang. Beim Ausführen des Skripts werden alle Objekte der ‚GeoNodes‘-Kollektion nacheinander aktiviert, gerendert, im Thumbnail-Ordner mit ihrem Namen gespeichert und deaktiviert.

Die zentrale Funktion findet in diesem Code-Ausschnitt statt:

```
# Alle Objekte in der Kollektion werden für das Rendering deaktiviert
for obj in geo_collection.objects:
    obj.hide_render = True

for obj in geo_collection.objects:
    # Aktuelles Objekt wird aktiviert
    obj.hide_render = False

    # Pfad für die Ausgabe setzen.
    output_path = os.path.join(output_dir, f"{obj.name}.jpg")
    scene.render.filepath = output_path

    # Rendern und speichern.
    bpy.ops.render.render(write_still=True)
    print(f"Rendered and saved: {output_path}")

    # Aktuelles Objekt wird deaktiviert
    obj.hide_render = True
```

So kann das Thumbnail-Verzeichnis einfach und einheitlich aktualisiert werden, wenn sich Änderungen an bestehenden Setups ergeben oder neue hinzugefügt werden.

Da es sich um ein fortgeschritteneres Feature handelt, wurde dieses nicht direkt in der UI implementiert. Für Entwickler mit diesem Anspruch ist dies aber ein einfacher und zugänglicher Weg.

5. Empirische Evaluation

Nachdem die konkrete Umsetzung des LPTKs besprochen wurde, erforscht dieses Kapitel das Potenzial einer prozeduralen Low-Poly-Asset-Bibliothek im Kontext der Spieleentwicklung. Dabei wurde das LPTK im Rahmen einer Nutzerevaluation getestet. Ziel war es, Bedienbarkeit, Effizienz und Ergebnisqualität des Systems im Vergleich zu einem rein manuellen Workflow zu analysieren und zu bewerten, inwiefern das LPTK den Gestaltungsprozess erleichtert und qualitativ verbessert.

5.1 Aufbau und Methodik

Die Untersuchung wurde mit fünf Teilnehmern durchgeführt, die einen zielgruppenorientierten Querschnitt potenzieller Anwender abbilden sollten. Die Zusammensetzung war wie folgt:

- 2 Teilnehmer ohne Vorerfahrung in 3D-Modellierung oder Spieleentwicklung
- 2 Teilnehmer mit grundlegender Blender-Erfahrung und professioneller Erfahrung in der Spieleentwicklung
- 1 Teilnehmer mit professioneller Erfahrung in der Erstellung stilisierter Low-Poly-Assets in Blender

Jede Testperson erstellte in zwei Durchläufe dasselbe Level-Szenario, basierend auf einer vorgegebenen, händisch gezeichneten Referenzskizze (Anhang A7). Einmal mit Blender ohne Add-on und einmal mit Blender in Kombination mit dem entwickelten LPTK. Anschließend wurde das Level in beiden Szenarien in die Godot-Engine importiert.

Die Tests wurden einzeln durchgeführt und begleitet. Während die Einführungsphasen angeleitet wurden, erfolgte die Bearbeitung beider Szenarien selbstständig. Technische Rückfragen wurden in beiden Durchläufen beantwortet, ohne die inhaltliche Lösung vorzugeben. Die durchschnittliche Durchführungsdauer betrug circa 70 Minuten.

Evaluationsablauf:

Evaluationsschritt	**Dauer**
Einweisung in Blender	~10 min
Terrainerstellung ohne Add-on	~15 min
Export + Import nach Godot ohne Add-on	~5 min
Terrainerstellung mit LPTK	~15 min
Export + Import nach Godot mit LPTK	~5 min
Fragebogen	~5 min
Qualitatives Kurzinterview	~5 min

Der Fragebogen wurde mithilfe von Google Forms umgesetzt und bestand aus vier Abschnitten. Zunächst wurde die Vorerfahrung der Teilnehmer erhoben. Anschließend bewerteten die Tester den Workflow der Terrain-Erstellung einmal ohne und einmal mit dem LPTK. Darüber hinaus stand ein Freitextfeld zur Verfügung, in dem die Teilnehmer angeben konnten, welche Aspekte des LPTK ihnen besonders positiv oder negativ aufgefallen sind. Abschließend wurde ein auf die Vorerfahrung der Teilnehmer angepasstes Kurzinterview durchgeführt um die Nutzererfahrung vertieft zu besprechen.

5.2 Quantitative Ergebnisse

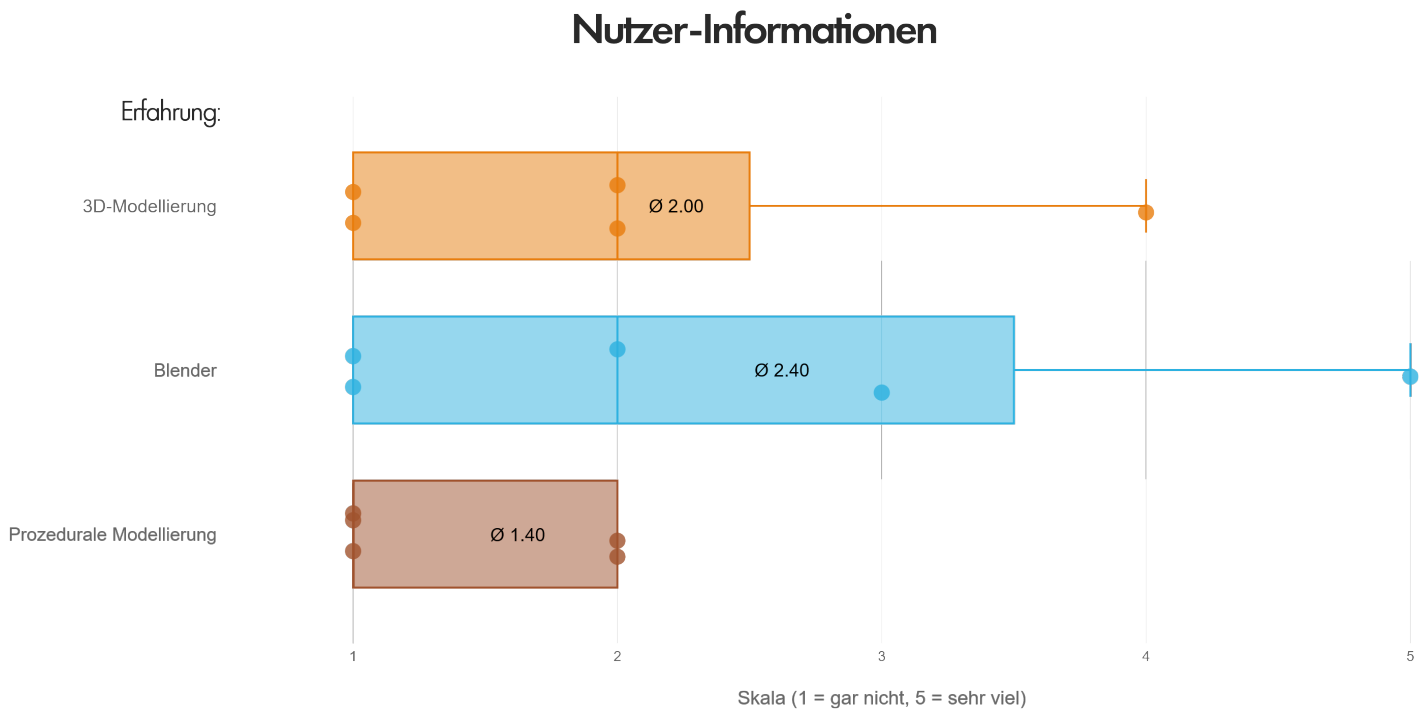


Abbildung 45: Nutzer-Evaluation, Selbsteinschätzung relevanter Vorerfahrung (eigene Darstellung).

Abbildung 45 zeigt die Selbsteinschätzung der Teilnehmer auf einer 5-stufigen Likert-Skala hinsichtlich ihrer Erfahrung mit 3D-Modellierung, Blender als Software und prozeduraler Modellierung. Die Stichprobe weist durchschnittlich niedrige bis moderate Erfahrungswerte auf, insbesondere im Bereich prozeduraler Modellierung (Ø 1,4), ist jedoch in anderen Bereichen individuell stark durchmischt. Die Blendererfahrung bildet bspw. Werte von 1 bis 5 ab. Diese Zusammensetzung entspricht der angestrebten Zielgruppe des LPTK und bildet eine geeignete Grundlage für die Bewertung der Nutzbarkeit des Systems.

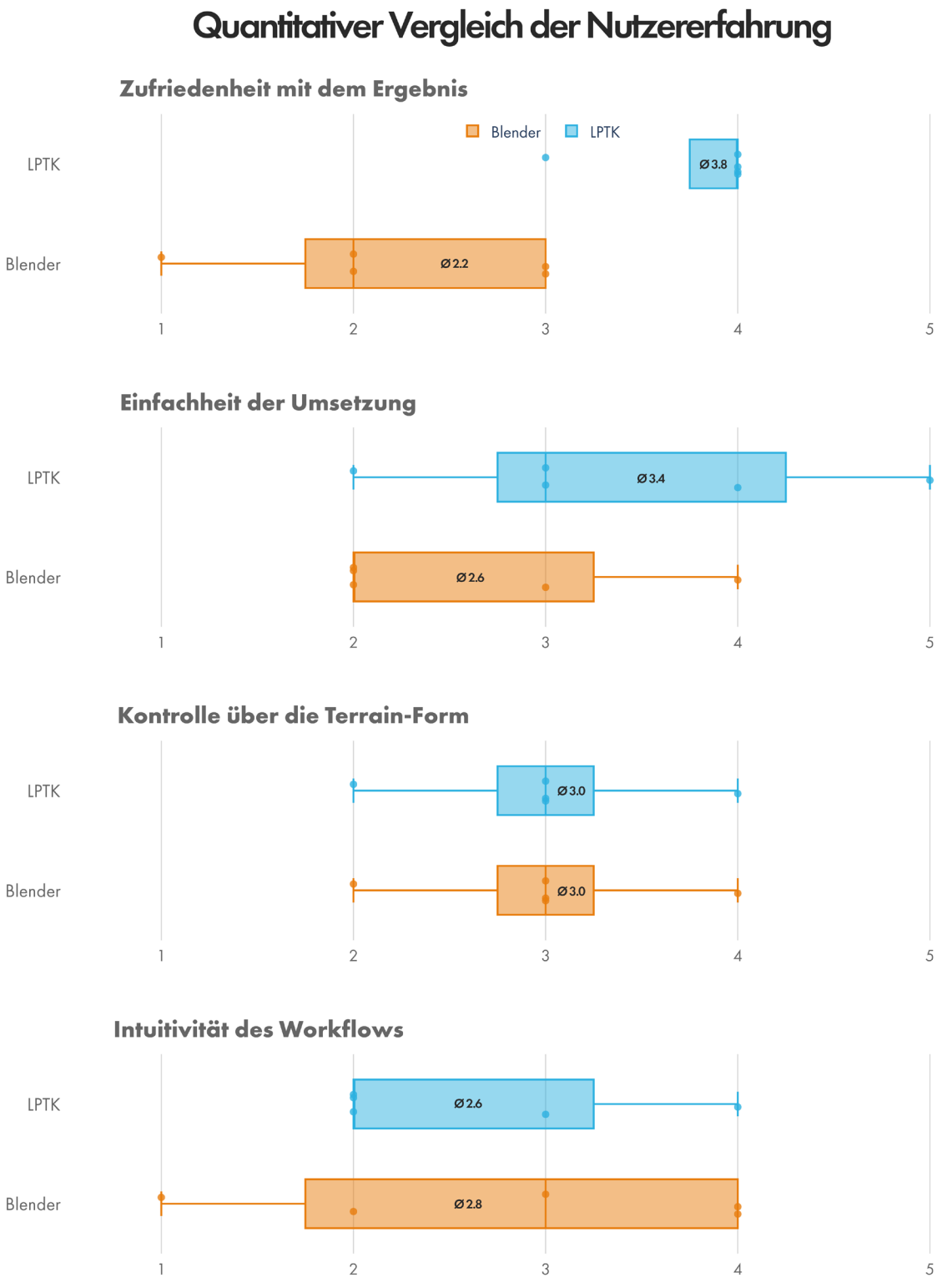


Abbildung 46: Quantitativer Vergleich der Nutzererfahrung des LPTK (eigene Darstellung).

Die quantitativen Ergebnisse, dargestellt in Abbildung 46, zeigen ein klares Muster. Die Einfachheit der Nutzung und speziell die Zufriedenheit mit den Ergebnissen bewerten die Tester mit dem LPTK deutlich höher. Während die Zufriedenheit mit Blender bei Ø 2,2 liegt, wurde sie mit dem LPTK mit Ø 3,8 bewertet. Die Einfachheit der Umsetzung bewerteten die Teilnehmer mit dem LPTK mit Ø 3,4, mit Blender hingegen nur mit Ø 2,6. Bei der gefühlten Kontrolle, welche die Tester über das Terrain hatten, liegen Blender und das LPTK mit Ø 3,0 gleich auf. Nur bei der Intuitivität des Workflows schneidet das LPTK mit Ø 2,6 minimal schlechter als Blender mit Ø 2,8 ab.

5.3 Qualitative Ergebnisse

Das qualitative Feedback aus dem Freitextfeld (Anhang A5) sowie den abschließenden Kurzinterviews (Anhang A6) liefert eine vertiefte Einsicht in die Nutzererfahrung mit dem LPTK.

Besonders positiv hervorgehoben wurden die kurvenbasierten Assets, die sich direkt in die Szene „malen" lassen. Das einfache Zeichnen von Bäumen, Pfaden und Efeu wurde von mehreren Teilnehmern ausdrücklich gelobt und als deutlich intuitiver und flexibler beschrieben als herkömmliche, statische, Asset-Workflows. Insbesondere die erfahreneren Entwickler betonten, dass dieses Interaktionsprinzip eine wesentlich natürlichere und effizientere Gestaltung ermöglicht.

Ein wiederkehrendes Thema war die sofortige visuelle Qualität der Ergebnisse. Testern gefiel, dass Formen „direkt gut aussehen", automatisch passende Materialien zugewiesen werden und das System damit bereits in nach wenigen Arbeitsschritten ästhetische und stimmige Resultate liefert, so konnten auch Tester ohne Modelliererfahrung überzeugende Ergebnisse erzielen (Abbildung 47 und 48, weitere Resultate im Anhang A2). Außerdem betonten die Tester, dass aufgrund der schnellen Ergebnisse, „die Arbeit mit dem LPTK mehr Spaß macht".

Abbildung 47: Zeigt das Ergebniss der Modellierung der Referenzskizze eines Test-Nutzers mit dem LPTK (eigene Darstellung).

Gleichzeitig zeigte das qualitative Feedback auch klare Verbesserungspotenziale am LPTK. Das ‚ProceduralTerrain'-System war zentral zur Modellierung der Szene und wurde von einigen Teilnehmern als „unintuitiv" und „kompliziert" beschrieben. Insbesondere der Boolean-basierte Workflow war für die Tester, die mit dem Konzept nicht vertraut waren, schwierig zu kontrollieren und es kam bei einigen zu Problemen und Unsicherheiten. Einige Nutzer bevorzugten daher das alternative ‚MeshTerrain', welches als wesentlich kontrollierbarer und vorhersehbarer wahrgenommen wurde.

Auffällig ist, dass viele der kritischen Punkte nicht auf das LPTK selbst, sondern auf Blender als Entwicklungsumgebung zurückgeführt wurden. Mehrere Tester gaben an, dass sie weniger durch das Toolkit, sondern vielmehr durch fehlendes Blender-Vorwissen, auf welchem das LPTK teilweise aufbaut, eingeschränkt wurden. Dies deutet darauf hin, dass das LPTK zwar einen niedrigschwelligen und benutzerfreundlichen Ansatz bietet, jedoch weiterhin an die Komplexität Blenders gebunden bleibt.

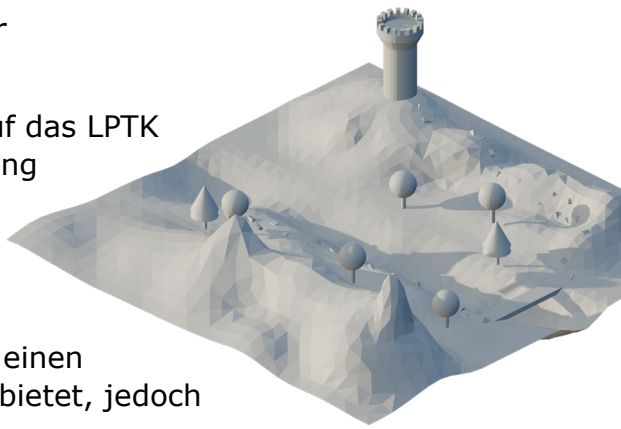


Abbildung 48:Zeigt das Ergebnis der Modellierung der Referenzskizze eines Test-Nutzers ohne das LPTK (eigene Darstellung).

Zusammenfassend bestätigen die qualitativen Rückmeldungen, dass das LPTK die kreative Arbeit deutlich erleichtert, ästhetisch hochwertige Ergebnisse ermöglicht und insbesondere durch seine

kurvenbasierten Interaktionswerkzeuge überzeugt. Gleichzeitig zeigen die Aussagen der Tester, an welchen Stellen eine Weiterentwicklung sinnvoll wäre.

6. Diskussion

Die Diskussion gliedert sich in drei Abschnitte, die gemeinsam darauf abzielen, die in Kapitel 1 formulierte Forschungsfrage unter Berücksichtigung der zentralen Untersuchungsbereiche zu beantworten.

Zunächst wird das LPTK als konkreter Entwicklungsansatz kritisch reflektiert, wobei Stärken, Schwächen und mögliche Erweiterungen des Systems herausgearbeitet werden.

Im Anschluss werden die technischen Möglichkeiten und Grenzen von Blender und den Geometry Nodes als Grundlage prozeduraler Assetgenerierung sowie deren Einbindung in eine Add-on-basierte Interaktionsoberfläche diskutiert.

Abschließend wird der Ansatz prozeduraler Assets im Kontext der Spieleentwicklung allgemein bewertet, um die gewonnenen Erkenntnisse in einen größeren fachlichen Zusammenhang einzuordnen.

6.1 LPTK als entwickelter Ansatz

Die Entwicklung des LPTK zeigt, dass der Ansatz einer benutzerorientierten prozeduralen Asset-Bibliothek grundsätzlich funktioniert und die prozedurale Arbeitsweise einen merkbaren Mehrwert liefern kann.

Der Post-Processing Mixed-Authorship-Ansatz, welchen viele der erstellten Assets verfolgten, hat sich als besondere Stärke des Systems herausgestellt. Systeme die eine Basisform, wie eine Kurve oder ein einfaches Mesh zu einem visuell komplexen und ansprechenden Ergebnis formen, haben sich in der Testerevaluation als intuitiv und wirkungsvoll herausgestellt.

Diese direkte, skizzenartige Arbeitsweise reduziert technische Komplexität spürbar und fördert einen kreativen, experimentellen Workflow, was genau dem Ziel des Projekts entspricht.

Das Toolkit zeigt außerdem, dass die Nutzung prozedurale Systeme bei der Erstellung kleiner bis mittelgroßer Low-Poly-Szenen deutlich zeit-effizienter ist. Feedback durch Tester und eigene Erprobung zeigen, dass der Look der generierten Assets konsistent ist. Das System macht sichtbar, dass prozedurale Methoden, wenn sie gut aufbereitet sind, auch für weniger erfahrene Nutzer einfach zugänglich gemachte werden können.

Gleichzeitig ist während der Entwicklung und Evaluation deutlich geworden, dass der Umfang des LPTK zu ambitioniert war. Der Anspruch war es, nur mit dem System komplexe Szenen vollständig abbilden zu können. Die Entwicklung einzelner Funktionen und Systeme hat jedoch sehr viel Zeit in Anspruch genommen. Das ursprüngliche in 4.1.4 thematisierte System zur Wassergruppierung ist bspw. aufgrund unzureichender Erfahrung und Funktionen innerhalb Blenders in eines der größten Projekte dieser Arbeit ausgeartet. Dadurch und durch andere Komplikationen wurde einige Node Setups, sowohl in ihrem Funktionsumfang als auch in ihrer Parametrisierung und Bedienlogik, nicht

vollständig umgesetzt. Aus dem ambitionierten quantitativen Anspruch und der mangelnden Zeit entstand ein Qualitätsgefälle zwischen einzelnen Systemen was beispielsweise dazu führte, dass einige Features des ‚ProceduralTerrains' nicht im ‚MeshTerrain' implementiert waren, was auch bei der Testerevaluation für Verwirrung sorgte.

Einige geplante Features wie Innenräume für das Tower- und Castle-Setup sowie ein mesh-basiertes Haussystem konnten nicht zufriedenstellend umgesetzt werden und wurden abgebrochen, obwohl begehbare Innenräume in vielen Spielkonzepten einen deutlichen Mehrwert bieten würden.

Trotz dieser Grenzen zeigt die in dieser Arbeit entwickelte Implementation des LPTK ein hohes Potenzial. Die Werkzeuge funktionieren, die Interaktion ist intuitiv, und die Ergebnisse sind konsistent reproduzierbar. Die Testerevaluation bestätigt, dass das Toolkit technische Hürden senkt und kreative Entscheidungen deutlich erleichtert. Damit liefert das LPTK nicht nur einen funktionsfähigen Prototypen, sondern auch wichtige Erkenntnisse darüber, wie eine prozedurale Asset-Bibliothek gestaltet sein kann, um zugänglich, erweiterbar und für die Spieleentwicklung einen tatsächlichen Mehrwert zu liefern.

6.2 Blender und Geometry Nodes als Basis des LPTK

Die Wahl von Blender und Geometry Nodes als technisches Fundament des LPTK hatte Vor- und Nachteile. Nachdem in 3.2.1 die Gründe genannt wurden, die vor dem Start des Projekts für Blender sprachen, wird in diesem Kapitel die Entscheidung nach Durchführung des Projekts diskutiert.

Wie bereits erwähnt, stellte die bereits vorhandene Erfahrung mit Blender einen entscheidenden Vorteil für die Wahl dar. Die Entwicklung der Systeme konnte nach kurzer Einarbeitung in die Grundkonzepte der Geometry Nodes beginnen, ohne dass viel Zeit in das Erlernen der grundlegenden Oberflächen einer alternativen Umgebung wie Houdini investiert werden musste.

Die grundlegenden Konzepte zur Funktionsweise von Geometry Nodes wirken anfangs komplex. Das Spreadsheet, Fields, Instanziierung und Selektion unterscheiden sich in vielerlei Hinsicht stark von manuellen Workflows und schrecken selbst erfahrene Blender-Nutzer anfangs ab. Sobald die grundlegenden Prinzipien jedoch verstanden sind, lassen sich Systeme flexibel erweitern, funktionelle Muster erkennen und in unterschiedlichen Kontexten wiederverwenden.

Durch die Kombination verschiedenster Nodes sind umfangreiche Systeme, die komplexe Probleme lösen, mit Geometry Nodes durchaus umsetzbar. Häufig wird jedoch selbst für die Erstellung simpler Systeme eine Vielzahl an kombinierten Nodes benötigt, was die Erstellung der Node Trees unnötig verkompliziert.

Was Blender als Basis rückblickend besonders interessant für Mixed-Authorship orientierte Systeme macht, sind die klassischen Modellierungswerkzeuge, die bereits sehr ausgereift sind und sich mit prozeduralen Systemen wie beispielsweise dem MeshTerrain optimal kombinieren lassen. Polygonale Modellierung, Sculpting oder Hair-Sculpting bieten eine starke Grundlage und sind optimal für experimentelle Systeme wie die in 4.1.6 beschriebenen

‚ScatterCurves' nutzbar. Diese Kombination aus traditioneller Modellierung und prozeduraler Generierung war für das LPTK ein großer Vorteil, da viele Ideen zur Interaktion mit dem System Ideen aus beiden Bereichen miteinander verknüpft.

Dem entgegen birgt die Wahl von Blender und insbesondere der Geometry Nodes zur Erstellung einer professionellen auch einige Risiken.
Wie bereits in 2.5.5. beschrieben, befinden sich die Geometry Nodes noch in voller Entwicklung und wurden in den letzten Jahren mehrmals fundamental verändert. Das LPTK wurde mit Blender 4.5 entwickelt, mit der Veröffentlichung von Blender 5.0 wurden die Geometry Nodes erneut in vielen Bereich überarbeitet, sodass einzelne Setups in Zukunft potenziell nicht mehr funktionieren oder angepasst werden müssen. Diese fehlende Stabilität erschwert es, langfristig nutzbare Systeme zu bauen und in einem professionellen Kontext einzusetzen.

Auch die Dokumentation ist nicht durchgehend zuverlässig. Grundlagen werden teilweise gut erklärt, aber komplexere Konzepte wie Repeat Zones oder fortgeschrittene Selektionslogiken werden nur sehr oberflächlich behandelt und nicht anhand passender Beispiele besprochen. In der Community gibt es zwar einzelne Creator, die komplexere Systeme vorstellen, doch im Vergleich zur klassischen Modellierung oder Shader-Entwicklung ist das verfügbare Lernmaterial für Geometry Nodes deutlich geringer, was dazu führt, dass bei spezifischen Problemen eigene Lösungen erarbeitet werden müssen. Viele Tutorials sind zudem, ähnlich wie die Dokumentation, schnell veraltet, da sich die Nodes ständig verändern, was die Fehlersuche oder Weiterentwicklung zeitaufwendig macht.

Ein weiterer limitierender Faktor zur Erstellung nutzerfreundlicher Systeme ist das Modifier-Stack eingebundene Interface der Geometry Nodes. Viele Parameter, welche die prozedurale Logik steuern können, nicht verfügbar gemacht werden. Beispielsweise lassen sich weder Kurven (Float/RGB) noch Colors Ramps exponieren. Die Konfiguration des Panels ist ebenfalls eingeschränkt. Zwar werden Parameter im Modifier je nach Nutzbarkeit visuell kodiert, es gibt aber keine Möglichkeit exponierte Parameter dynamisch zu generieren, was die kontextabhängige UI-Gestaltung erschwert, wodurch zwangsläufig Kompromisse bei Bedienbarkeit und Klarheit der Systeme entstehen.

Hinzu kommt die fehlende native Synchronisation in gängige Game Engines. Für einen Spieleentwicklungs-Workflow wäre eine native und direkte Anbindung, durch ein neues Dateiformat oder analog zur Implementation der Houdini Engine, ein großer Vorteil.

Insgesamt zeigen Blender und Geometry Nodes im Speziellen jedoch ein großes Potenzial, vor allem für kleine Teams oder Solo-Entwickler, die nach einer kostengünstigen und flexiblen Lösung suchen und denen kontinuierliche Weiterentwicklung wichtiger ist, als absolute Langzeitstabilität.

Insgesamt kann festegestellt werden dass, sich Blender für sehr umfangreiche, langfristig gepflegte Bibliotheken heute nur eingeschränkt empfehlen lässt. Für kleinere, experimentelle Systeme, wie das LPTK, ist Blender aber eine optimale

Basis, weil es schnelle Iterationen erlaubt, eine starke Modellierungsumgebung mitbringt und prozedurale Experimente sehr direkt unterstützt.

6.2.1 Blender Python API zur Add-on Entwicklung

Die Entwicklung des Add-ons mithilfe der Blender Python API (bpy) nahm im Vergleich zur Erstellung der prozeduralen Assets weniger Zeit in Anspruch, soll in diesem Kontext dennoch kurz reflektiert werden.

Die bpy ist ein mächtiges Werkzeug und für die professionelle Integration benutzerdefinierter Funktionen in Blender unumgänglich. Sie bietet im Kontext des LPTK eine Flexibilität, welche mit dem integrierten Asset-Library-System nicht erreicht worden wäre. So ermöglichten sie eine Vielzahl an Workflow-Optimierungen, wie die in 4.2.2 beschriebenen, kontextspezifischen ‚NodeTypes', wodurch die Nutzung des LPTK erleichtert wurde.

Gleichzeitig traten während der Entwicklung spezifischer Features einige Herausforderungen auf. So ist die in Kapitel 4.2 angesprochene Operatoren-Logik von Blender stark kontextabhängig. Dadurch kann die Ausführung der Instruktionen abweichend vom erwarteten Verhalten erfolgen, wenn sich der Nutzer in einem spezifischen Fenster oder UI-Element befindet. Diese Kontextsensitivität erschwert das Debugging erheblich und kann bereits bei geringfühgig unterschiedelicher Nutzung zu unerwarteten Fehlern führen.

Zuletzt leidet auch die Blender Python API, ähnlich wie die Geometry Nodes, stellenweise unter einer unzureichenden Dokumentation. Durch die schnelle Weiterentwicklung, Änderung von Konzepten und die Open-Source-Natur kommt es stellenweise zu undokumentierten Funktionen. Für die Integration spezifischer Features, wie der in Kapitel 4.2.5.3 beschriebenen ‚Vertex Color Baking Automation', existieren wenige Ressourcen, was die Entwicklungszeit deutlich verlängert.

6.3 Prozedurale Assets für die Spielentwicklung

Im Folgenden wird der prozedurale Ansatz im Kontext der Spieleentwicklung diskutiert, unabhängig von der spezifischen Entwicklungsumgebung. Im Fokus stehen dabei Systeme mit einem Mixed-Authorship-Ansatz[55], welche den Kern dieser Arbeit und des entwickelten LPTKs darstellen.

Die Entwicklung prozeduraler Systeme zur Asset-Generierung ist initial meist mit einem deutlich höheren Aufwand verbunden als die manuelle Erstellung einzelner Modelle. Ein einfacher Low-Poly-Baum kann beispielsweise innerhalb weniger Minuten manuell modelliert und texturiert werden, während die Entwicklung eines Systems, das vergleichbare Bäume automatisch generiert, wesentlich mehr Zeit beansprucht. Prozedurale Lösungen amortisieren sich daher vor allem dann,

[55] Die „Autorenschaft" des Nutzers ist innerhalb der im LPTK implementierten Systeme stärker gewichtet als in den in der Literatur beschriebenen Beispielen, weshalb der Begriff nur bedingt zutrifft. Die hier vorgestellten Systeme ließen sich präziser als ‚user-authoritative' bzw. ‚nutzer-autoritativ' beschreiben.

wenn ein Asset häufig eingesetzt wird oder signifikant von Eigenschaften wie Variation, Anpassbarkeit und Wiederverwendbarkeit profitiert.

Aus diesem Grund muss in der Spieleentwicklung kritisch abgewogen werden, ob ein Asset durch prozedurale Eigenschaften einen realen Mehrwert erhält. Ein prozeduraler Ansatz sollte niemals als Selbstzweck dienen. Die bloße technische Machbarkeit rechtfertigt nicht automatisch den Entwicklungsaufwand. Insbesondere Technical Artists und Entwickler neigen dazu, aus technischer Begeisterung komplexe Lösungen zu implementieren, ohne dass die Problemstellung diese Komplexität erfordert. Weder das Endergebnis noch der Workflow profitieren von einer theoretisch unendlichen Anzahl an Baumvarianten, wenn das Projekt faktisch nur wenige, klar definierte Modelle benötigt.

Bieten prozedurale Systeme jedoch einen funktionalen Vorteil, etwa durch die präzisere Abbildung einer kreativen Vision, erleichterte Anpassungen oder eine spürbare Beschleunigung des Workflows, entfalten sie ein erhebliches Potenzial.

Besonders der Bereich Mixed-Authorship-Asset-Packs, wie in dieser Arbeit erforscht, bietet hierbei vielversprechende Möglichkeiten. Hier greifen Skaleneffekte, die den hohen Initialaufwand der Entwicklung rechtfertigen. Da das prozedurale System nicht nur für ein einziges Projekt, sondern projektübergreifend von einer Vielzahl von Entwicklern genutzt werden kann, amortisiert sich die komplexe Entwicklung deutlich schneller als bei einer proprietären In-House-Lösung.

Durch die Prozeduralisierung werden zudem wesentliche Nachteile klassischer Asset-Packs gelöst. Der Nutzer muss seine Vision nicht mehr an die statischen Formen der vorhandenen Assets anpassen. Stattdessen ermöglichen es die prozeduralen Parameter, die Assets flexibel an die eigene kreative Vision anzugleichen.

Trotz dieses Potenzials stellen Mixed-Authorship-Systeme für 3D-Geometrie in gängigen Asset-Stores derzeit noch eine Nische dar. Während prozedurale Materialien in der Industrie bereits weit verbreitet sind, existieren kaum vergleichbare, zugängliche Lösungen für die Modellgenerierung. Die Entwicklung und Etablierung solcher Asset-Packs würde somit eine signifikante Lücke im aktuellen Marktangebot schließen und könnte einen echten Mehrwert bieten.

7. Fazit und Ausblick

Die intensive Auseinandersetzung mit der prozeduralen Modellierung sowie die praktische Ausarbeitung des LPTK haben gezeigt, dass im Mixed-Authorship-Ansatz ein großes Potenzial steckt. Die Ergebnisse machen deutlich, dass dieser Weg, verglichen mit statischen Asset-Packs und der manuellen Modellierung, einen echten Mehrwert bieten kann, sobald Flexibilität, Anpassbarkeit und Entwicklungsgeschwindigkeit gefragt sind.

Zwar existieren auf dem Markt bereits vereinzelte prozedurale Systeme für spezifische Aufgaben wie die Terrain-Generierung, umfassende und zugängliche Bibliotheken für Indie-Entwickler fehlen hingegen weitgehend. Die Arbeit zeigt, dass der Ansatz insgesamt noch unterschätzt wird und viele Möglichkeiten für effizientere Workflows bietet.

Gleichzeitig haben Evaluation und Diskussion des Ansatzes aber auch Herausforderungen aufgezeigt. Damit solche Werkzeuge ihren vollen Nutzen bei der potenziellen Zielgruppe entfalten können, müssen sie so nah wie möglich am Zielsystem, der Game-Engine, integriert sein. Das LPTK ist zum jetzigen Zeitpunkt am stärksten durch die Integration in Blender eingeschränkt. Trotz des entwickelten ‚Collection Exporters' und der ausgearbeiteten Nutzeroberfläche stellt dieser technische Zwischenschritt eine große Hürde dar.

Für das LPTK ist das Projekt mit dieser Arbeit dennoch nicht beendet. Geplant sind eine Migration auf Blender 5.0 sowie eine Aufarbeitung der einzelnen Systeme basierend auf dem erhaltenen Nutzerfeedback. Mein Ziel ist es, das Toolkit anschließend kostenlos zu veröffentlichen. Damit möchte ich Indie-Entwicklern eine konkrete Hilfe an die Hand geben und weiter auf das vielversprechende Thema der prozeduralen Modellierung aufmerksam machen.

Literatur

[1] Statista. "DIGITAL & TRENDS Indie gaming." Zugriff am: 30. September 2025. [Online.] Verfügbar: https://www.statista.com/study/188180/indie-gaming/

[2] Gamalytic. "Publisher class definition." Zugriff am: 29. September 2025. [Online.] Verfügbar: https://gamalytic.com/about

[3] Video Game Insights. "The Big Game Engine Report of 2025." Zugriff am: 30. September 2025. [Online.] Verfügbar: https://app.sensortower.com/vgi/assets/reports/The_Big_Game_Engines_Report_of_2025.pdf

[4] E. Folmer, "Component Based Game Development – A Solution to Escalating Costs and Expanding Deadlines?," in *Component-Based Software Engineering* (Lecture Notes in Computer Science 4608), D. Hutchison et al., Hg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, S. 66–73.

[5] C.-H. Kung und K.-C. Liang, *Exploring the Usability and Future Development of AI-Generated 3D Models in CAD Workflows and the Metaverse Based on 3D Model Standards*, 2025.

[6] GMU MPI Saarbrücken, *Geometric Modeling Based on Polygonal Meshes*.

[7] S. Münster *et al.,* "3D Modeling," in *Handbook of Digital 3D Reconstruction of Historical Architecture* (Synthesis Lectures on Engineers, Technology, & Society 28), S. Münster et al., Hg., Cham: Springer Nature Switzerland, 2024, S. 107–128.

[8] M. Gai und G. Wang, *Artistic Low Poly rendering for images* (32), 2016.

[9] D.-M. Lee. "Blender 3D as a Catalyst for Indie Game Development."

[10] Autodesk Inc. "Maya Preisübersicht." Zugriff am: 23. November 2025. [Online.] Verfügbar: https://www.autodesk.com/de/products/maya/overview

[11] Blender Authors. "Blender API Overview." Zugriff am: 29. September 2025. [Online.] Verfügbar: https://docs.blender.org/api/current/info_overview.html#

[12] Blender. "Add-on Tutorial." Zugriff am: 29. September 2025. [Online.] Verfügbar: https://docs.blender.org/manual/en/latest/advanced/scripting/addon_tutorial.html

[13] Nathaniel Rupsis, *How do I contribute? — Blender Conference 2024.* [Online]. Verfügbar unter: https://youtu.be/AAOyToizw_M?si=jvyK7aICk_JnFuoz&t=2288

[14] Blender. "Node Wrangler." Zugriff am: 16. Oktober 2025. [Online.] Verfügbar: https://docs.blender.org/manual/en/latest/addons/node/node_wrangler.html

[15] Noor Shaker, Julian Togelius, Mark J. Nelson, *Procedural Content Generation in Games*. Gewerbestrasse 11, 6330 Cham, Switzerland: Springer International Publishing Switzerland, 2017.

[16] J. Togelius *et al., Procedural Content Generation: Goals, Challenges and Actionable Steps*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013.

[17] Gillian Smith, *Procedural Content Generation An Overview.* [Online]. Verfügbar unter: https://www.gameaipro.com/GameAIPro2/GameAIPro2_Chapter40_Procedural_Content_Generation_An_Overview.pdf

[18] R. M. Smelik, T. Tutenel, R. Bidarra und B. Benes, *A Survey on Procedural Modelling for Virtual Worlds* (33), 2014.

[19] A. L. Przemyslaw Prusinkiewicz, *The Algorithmic Beauty of Plants*.

[20] P. Müller, P. Wonka, S. Haegler, A. Ulmer und L. van Gool, "Procedural modeling of buildings," *ACM Trans. Graph.*, Jg. 25, Nr. 3, S. 614–623, 2006.

[21] Lars Krecklau and Leif Kobbelt, *Interactive Modeling by Procedural High-Level Primitives*.

[22] Y. Rabii und M. Cook, "Why Oatmeal is Cheap: Kolmogorov Complexity and Procedural Generation," in *Proceedings of the 18th International Conference on the Foundations of Digital Games*, Lisbon Portugal, P. Lopes, F. Luz, A. Liapis und H. Engström, Hg., 04122023, S. 1–7, doi: 10.1145/3582437.3582484.

[23] Blender. "Attributes." Zugriff am: 15. November 2025. [Online.] Verfügbar: https://docs.blender.org/manual/en/latest/modeling/geometry_nodes/attributes_reference.html

[24] Blender. "Blender Asset-Library-System." Zugriff am: 13. Oktober 2025. [Online.] Verfügbar: https://docs.blender.org/manual/en/latest/files/asset_libraries/introduction.html#what-is-an-asset-library

[25] ALLTHEWORKS11. "Easy Geometry Nodes - Low-poly Stylized Trees BLENDER 3.0." Zugriff am: 14. Oktober 2025. [Online.] Verfügbar: https://www.youtube.com/watch?v=G4VEHi3dI6w&

Abbildungsverzeichnis

Bildquellen

Abbildung 1: Marktanteil von auf Steam veröffentlichten Indie-Spielen von 2018 bis 2024 (Statista).

Quelle: https://www.statista.com/statistics/1535485/steamsteam-annual-indie-game-share/

Abgerufen am: 30.09.2025

Abbildung 2: Game Engine Mix nach verkauften Einheiten (Video Game Insights).

Quelle: https://app.sensortower.com/vgi/assets/reports/VGI_Global_Indie_Games_Market_Report_2024.pdf

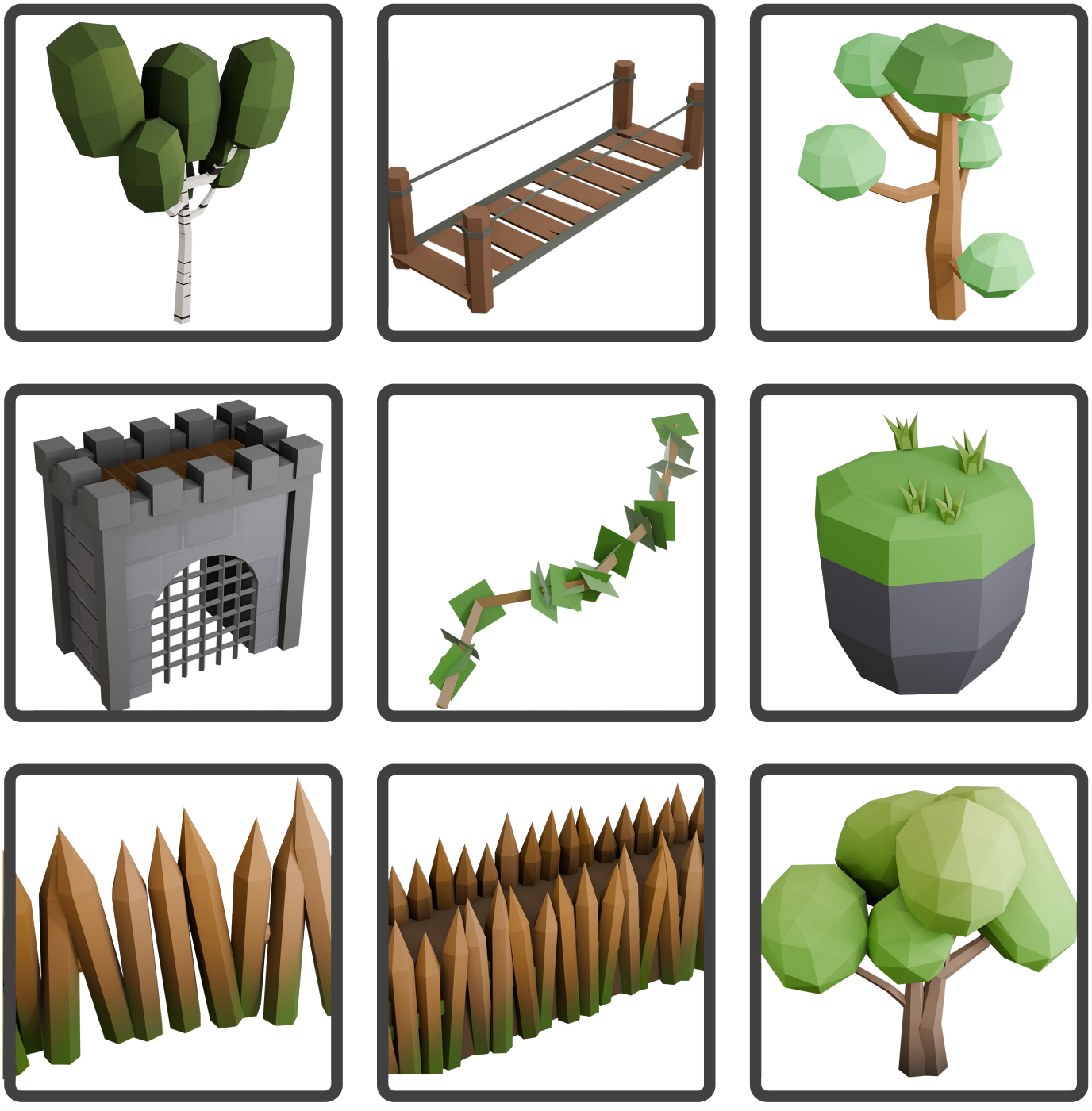
Abgerufen am: 30.09.2025

Alle weiteren Abbildungen sind eigene Darstellungen.

Anhang

A1 Übersicht über aller Thumbnails der verfügbaren Node Setups des LPTK

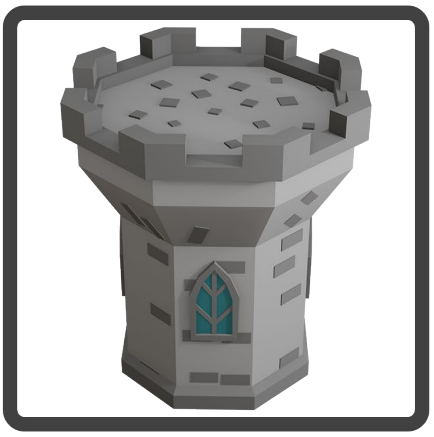
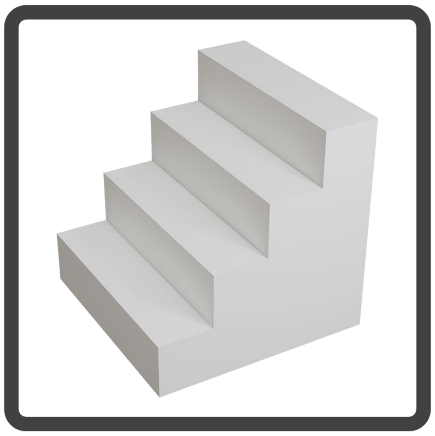
1/2

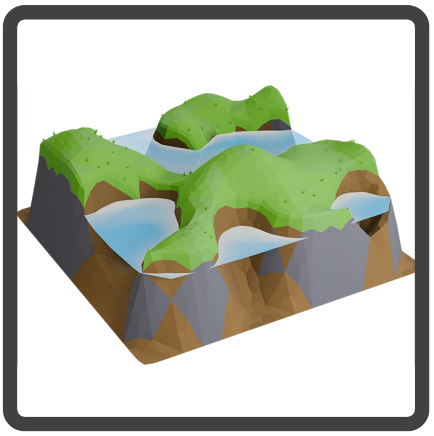


A1 Übersicht über aller Thumbnails der verfügbaren prozeduralen Assets des LPTK

2/2



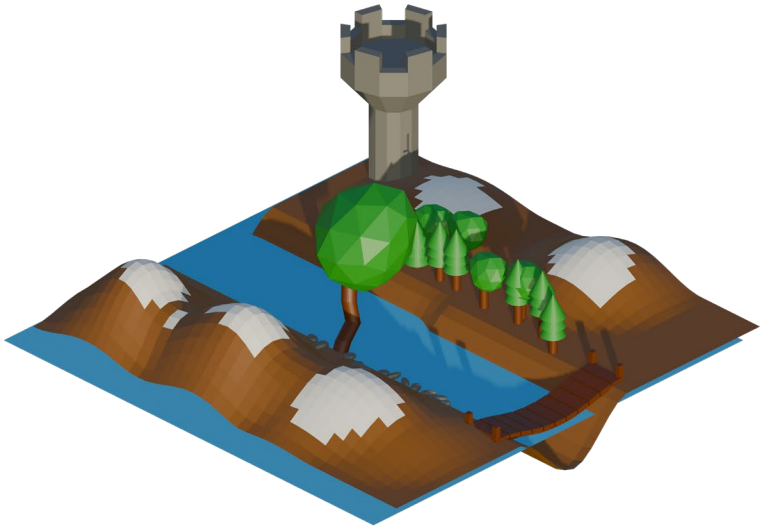


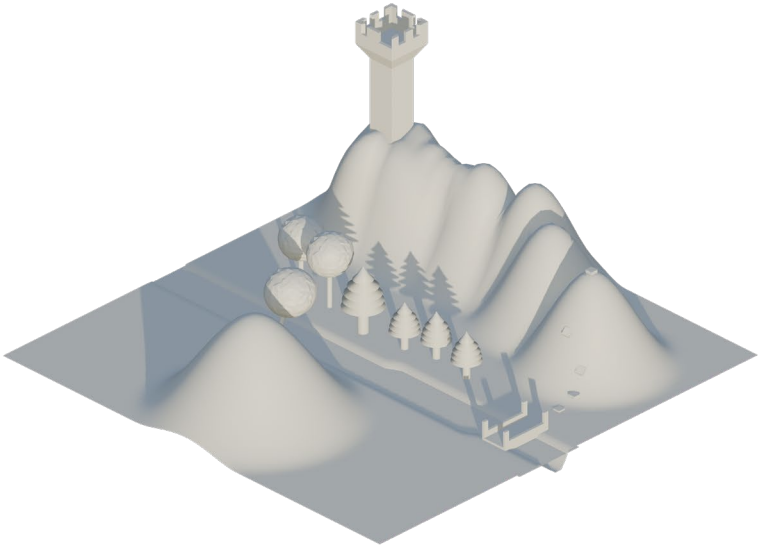

A2 Ergebnisse der Modellierung innerhalb der Nutzerevaluation

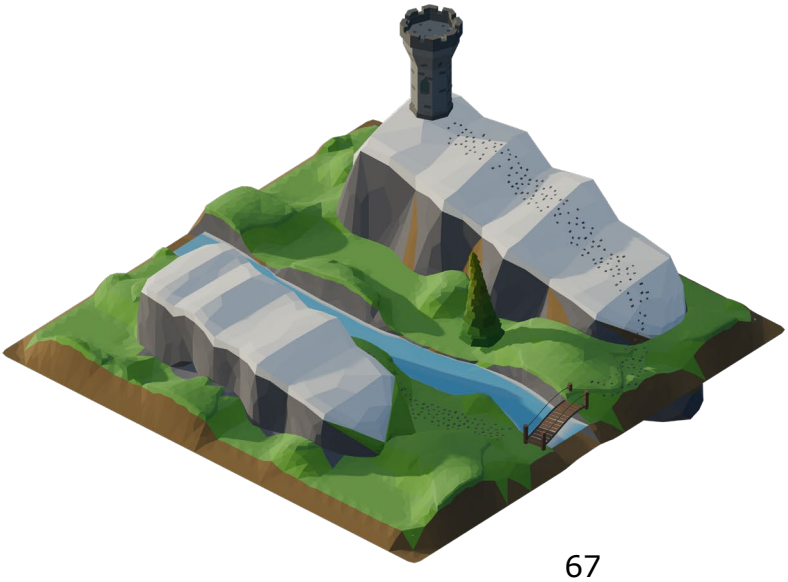
Renderings aller Ergebnisse der Nutzerevaluation.

Links nur Blender, Rechts mit LPTK

1/2





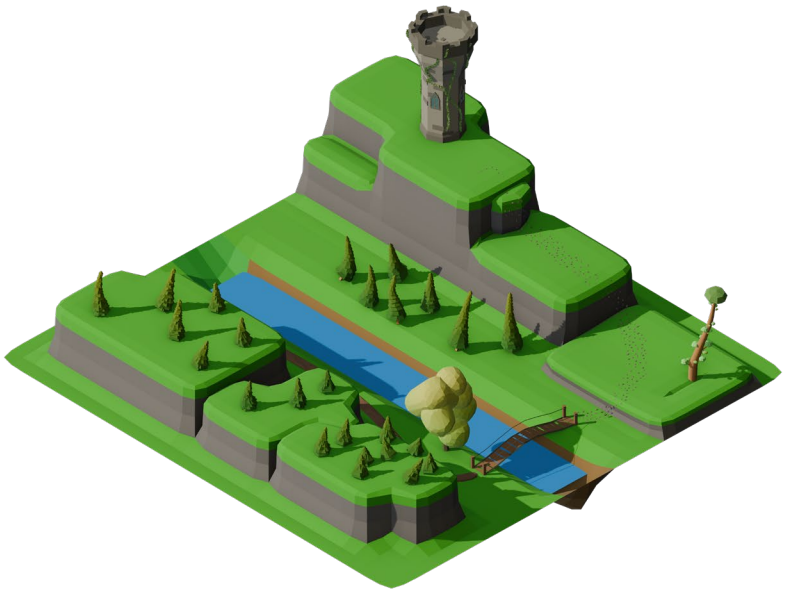


A2 Ergebnisse der Modellierung innerhalb der Nutzerevaluation

Renderings aller Ergebnisse der Nutzerevaluation.

Links nur Blender, Rechts mit LPTK

2/2



A3

A3 Kategorisierung von Indie-Spielen mit mehr als einer Millionen Verkäufe

Die Ausgangsquelle wurde zur Erstellung von Abbildung 3 händisch in Spiele mit 2D und 3D-Darstellungen unterteilt. Aufgrund des Umfangs der Analyse wurde diese nicht ausgedruckt, kann aber online abgerufen werden

Auswertung: https://joshuabattenfeld.com/LPTK/THESIS/A3

Als Ausgangsquelle diente: https://en.wikipedia.org/wiki/Indie_game

Abgerufen am: 18.11.2025

A4

A4 Kategorisierung der „Top 100 Paid Assets" des Unity Asset Stores

Zur Erstellung von Abbildung 6 wurde die analysiert und die einzelnen Assets händisch kategorisiert. Aufgrund des Umfangs der Analyse wurde diese nicht ausgedruckt, kann aber online abgerufen werden

Auswertung: https://joshuabattenfeld.com/LPTK/THESIS/A4

Als Ausgangsquelle diente: https://assetstore.unity.com/top-assets/top-paid

Abgerufen am: 30.09.2025

A5

A5 Ergebnisse der Nutzerevaluation, Google-Forms

Ergebnisse als csv: https://joshuabattenfeld.com/LPTK/THESIS/A5

A6

A6 Ergebnisse der Nutzerevaluation, Kurzinterviews

Stichpunktartige Zusammenfassung:
https://joshuabattenfeld.com/LPTK/THESIS/A6

A7

A7 Referenzskizze der Nutzerevaluation

Skizze: https://joshuabattenfeld.com/LPTK/THESIS/A7