Universität Bremen

Lazy prozedurale Weltgenerierung

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.) in Informatik

Eingereicht von

Maarten Behn

maarten.behn@uni-bremen.de

Gutachter*innen

Name 1
Name 2
Name 3

FB3

Uni Bremen

SoSe 2026

11. January 2026

1 Einleitung

1.1 Ziel der Arbeit

Ziel dieser Arbeit ist der Verglich und Entwicklung von Algorithm bzw. Systemen die partielle neu errechnung prozeduraler Welten erlauben, wenn sich generations Regeln ändern.

ToDo:

Ist Lazy überhaupt das richtige Wort für das was ich mache?

1.2 Aufbau der Arbeit

2 Stand der Technik

2.1 Prozedurale Generation

2.1.1 Noise & Zufälligkeit

- üblichster Weg
- Welt wird mit Zufälligkeit errechnet.
- Es gibt keine Regeln sonder die Regeln ergeben sich aus implizit aus dem Code
- Deterministisch nach Seeds

2.1.1.1 Nachteile

Es ist teils sehr schwierig sicherzustellen das komplexe Regeln eingehalten werden.

-> Fehlerhafte generation Möglich

2.1.2 Example based Model Synthesis

Eine spannende Arbeit im Bereich „constraint based generation“ ist Paul Merrels „Example-based Model Synthesis“. [1]

Die Idee des Algorithmus ist aus einem kleinen Input-Datensatz eine größere Struktur zu erzeugen die lokal den gleichen Regeln folgt.

Dabei besteht der Input-Datensatz aus einer Gitter Struktur, wo jede Celle im Gitter ein Wert zu geordnet wird. Nun wird eine Liste aller Nachbar Kombinationen erstellt die im Input Input-Datensatz vorkommen. Diese Liste beschreibt die Regeln nachdem ein neues Modell erzeugt wird.

Generations Algorithmus:

1. Dabei dem Gitter des neuen Modells in jeder Zelle alle Werte zugeordnet.
2. Wähle die Zelle mit den wenigsten Werten aus und entferne alle bis auf einen.
3. Entferne alle Werte aus den Nachbar Zellen für die eine Regel nicht mehr erfüllt ist.
4. Wiederhole 3. für jede Nachbar Zelle bei der Werte entfernt wurden.
5. Wiederhole 2. bis alle Zellen nur noch einen Wert enthalten.

2.1.3 Graph Grammars

2.1.4 Graph based Model Synthesis

ToDo:

Graph based Model Systemeis

2.1.5 KI basierte prozedurale Generation

Neuronale Netzwerke können genutzt werden um ähnliche Inhalte zu einem Trainings Datensatz zu erzeugen. In Arbeiten wie [2], [3] und [4] werden Neu-

ronale Modelle genutzt um Levels aus bekannten 2D Spielen zu reproduzieren. Dabei wird in [2] wird ein Level generations Model gegen ein Spieler Argent Modell trainiert, welches bewertet ob ein Level spielbar ist. Der Ansatz ist meistens fähig spielbare Levels zu generieren, wobei es vereinzelt zu generation Artefakten kommen kann. [2, S. 7]

Jedoch behandelt die Ansätze simple 2D Spiele, für die es viel vorhandene Level zu trainieren gibt. [2, S. 2]

ToDo:

Ist simple das richtige Wort?

In dem Ansatz wir außerdem gezeigt, dass Messbare Level Eigenschaften, wie die Menge der benötigten Sprünge genutzt werden können um „spannendere“ Level zu generieren. [2, S. 7]

[3]

- Analyse von der Möglichen validen Level eines 2D Mario spiels um unterschiedliche spannende Level zu erzeugen.

2.1.5.1 Terrain Diffusion

[5]

- Weiter Entwicklung von Noise Funktionen für Gelände generation.
- Es KI Modelle sind teilweise schneller als komplexe Noise Layer Systeme.
- Haben Geografische Erddaten als Input Datensatz verwendet.

2.1.5.2 Mit LLM Reden

[6]

- Aktuelle LLMs können sind fähig text basierte Darstellungen von Spiel Welten zu generieren und verstehen.
- Jedoch können sie nicht direkt mit Level Daten oder Tile Maps umgehen. Bzw dann generieren sie keine validen Ergebnisse

2.1.5.3 Umsetzung von generations Regeln in KI basierten Ansätzen

Bei der prozeduralen generation mit KI Systemen werden generations Regeln meist nicht explizit definiert. Sie ergeben sich aus dem Trainings-Datensatz und den Bewertungs Methoden der Modells in „Adversarialen“ Ansätzen.

ToDo:

Sollte ich KI sagen oder genauer sein?

Nachhaltig sicherzustellen, dass alle erwünschten Generationsregeln eingehalten werden, bleibt ein Bereich in dem Aktiv geforscht wird.

2.1.5.4 Bewertung

KI basierte Ansätze zur prozeduralen Generation sind ein sich aktiv entwickelnder Bereich indem

- Kann nur reproduzieren was schon vorhanden ist bzw benötigt größere Mengen an Input Daten.

-> Schränk Kreativität ein, obwohl man auch argumentieren kann, dass meist wiedererkennbare Strukturen generiert werden sollen, sodass die nicht wirklich ein Problem ist. Und durch geschickte Kombination bekannter Erdstrukturen kann viel neues erzeugt werden.

- Schwer weitere Regeln direkt einzubauen.
- Grundsätzlich ist es schwer dafür zu sorgen das Neuronale-Netzwerke nur valide Lösungen erzeugen, gerade wenn die Regeln komplex werden.

2.2 Lazy neu Errechnung

Dieser Teil betrachtet in wie weit die in Ansätze beschreiben in Abschnitt 2.1 Möglichkeiten zur Lazy Neuerrechnung beschreiben in Abschnitt 1.1 bieten.

ToDo:

Allgemein nach Arbeiten suchen.

2.2.1 Noise und Zufall

- Regeln ergeben sich aus Code also würde Regeln änder das der Code geändert wird.
- Individual Lösungen benötigt
- Teilweise werden generierte Ergebnisse aus alten Versionen in neuen weiter verwendet, aber sie entsprechen dann nicht den Regeln.
- Grundsätzlich benötigt man ein System welches die generations Regeln genauer modelliert um verallgemeinerbare Aussagen treffen zu können.

2.2.2 Lazy Model Systhesis

- Meine Erste Idee
- Man generiert ein valide Welt mit dem mit Model Systhesis
- Nun ändern sich die Regeln
- Man geht über alle Felder welche nicht valide Regeln haben.
- Nun müsste man in der Idee von Model-Systhesis wieder andere Werte zu diesen Feldern hinzufügen
- Aber welche? Alle?
- Das Hauptproblem: Die neu Hinzugefügten Werte sind erstmal auch nicht valide. Zu den Nachbarn müssten auch Werte hinzugefügt werden, damit sie valide sind.
- Das Problem gilt dann aber auch wieder für die Nachbarn. Und so weiter
- Wenn man einfach für alle Nachbarn alle Werte hinzufügt, hat man wieder das Komplett unentschiedene Feld normalen Model-Systhesis Algorithmus das bietet kein Vorteil.

- Also müsste man ein Menge an Feldern finden den man jeweils wieder Werte pro Feld hinzufügt, wo alle Regel eingehalten werden, damit mann dann wieder Model-Systhesis laufen lassen kann.
- Diese Menge müsste möglichst minimal sein.
- Vorallem sollte die Laufzeit-komplexität zum errechnen dieser Menge kleiner sein als Model-Systhesis selbst.
- Die einzige Idee die mir eingefallen ist um dies zu lösen:
  - Eine Breiten suche über ein Graph aller möglichen Werte die wir hinzufügen können. Um die erste valide Menge zu finden.
  - Das ist schrecklich was laufzeit und auch space Komplexität angeht.
- Der Vorteil von Model-Systhesis ist das in jedem Schritt alle Kombinationen an Werten eine valide Lösung ist.
- Aber diese Menge zu suchen ist sehr schwer.

2.2.3 KI basierte prozedurale Generation

- Schon alleine mehrere Ergebnisse die gleich aussehen zu errechnen ist schwer.

ToDo:

Aber vielleicht gibt es ja Arbeiten in die Richtung -> Nach Papern suchen

3 Theoretische Grundlagen

3.1 Träges errechnen (Lazy computation)

Träges errechnen auf englisch Lazy computation beschreibt die Idee in Computerprogrammen nur die Daten erst dann zu errechnen wenn sie benötigt werden.

Es reduziert Lag-spikes gerade beim starten eines neuen Prozesses, da Anstatt alle nutzbaren Daten nur die, die gerade angefragt benötigt werden errechnet werden.

Funktionale Programmiersprachen wie Haskell findet dieses Konzept vie Anwendung und erlaubt für unendliche Datenstrukturen.

- Graph grammers
- Rust features
- Stabiele Listen
- L-Systems
- CSG
- Geometry Nodes

3.2 Abhängigkeiten

3.2.1 Prozedurale Generation aus der Design Sicht

- Rekursiv
- Abhängigkeiten werden schnell komplex
- Dependency Injektion

Die Entscheidung ob in einer prozeduralen Welt ein Objekt existiert, bzw wo es plaziert werden soll, hängt davon ab wie der Rest der Welt generiert wird.

Zum Beispiel wenn man einen Wald mit einem Netz an Wegen generieren möchte, könnte es mehrere Abhängigkeiten geben.

- Ein Baum sollte in Mindestabstand zu jedem anderem Baum haben.
- Ein Baum sollte ein Mindestabstand zu jedem Weg haben.
- Zwei Wege sollten nicht direkt nebeneinander laufen.

Nun stellt sich die Frage wie man dies effizient generieren kann.

Der üblichste Weg ist einen Algorithmus zu finden der iterativ die Bestandteile entscheidet. Der Algorithmus ist dabei so gewählt das er für jedes Bestandteil nur Ergebnisse erzeugt, die alle Abhängigkeiten einhalten.

An dem Beispiel würde als ersten Schritt ein Netz an Wegen generiert werden. Hierbei würde ein Algorithmus gewählt werden der intrinsisch alle benötigten Beschränkung einhält.

Dafür würde sich z.B. Passions Disk Sampling für die Position der Knoten anbieten, um einen Mindestabstand zwischen den Knoten sicherzustellen.

Nun könnten alle Knoten mit einem Abstand in einem bestimmten Bereich mit einem Weg verbunden werden. So könnten unrealistisch kurzen oder langen Wegen vorgebeugt werden.

Jedoch kann es Beschränkungen geben, die nicht intrinsisch von einem generations System eingehalten werden.

z.B. um Vorzubeugen, dass zwei Wege direkt nebeneinander parallel laufen, müssten im nach hinein unerwünschte Wege raus gefiltert werden.

Wurde nun so ein valides Weg Netz gefunden, kann dieses genutzt werden um Bäume nur an Stellen zu plazieren, wo kein Weg ist.

Dieses System zu generation von Welten ist sehr weit verbreitet. Dabei werden die Algorithmen und parameter so gewählt, das meist ein realistisches Ergebnisse erzeugt wird.

Jedoch da die Kern-Beschränkungen meist nur indirekt zugesichert werden, kann es zu Fehlerhaften Welten kommen.

ToDo:

Beispiel broken Minecraft Seeds

ToDo:

Doppelung mit state of art

3.3 Systeme, die alle Abhängigkeiten sicherstellen.

3.3.1 Linear Programming

ToDo:

Erklärung

3.3.2 Model Synthesis

ToDo:

Erklärung

3.3.3 Problem

- Zu schlechte Laufzeit Komplexität
- Zu schlechte Speicher Komplexität
- Können nur auch endlichen Mengen an Parametern arbeiten
  - Bei Model Synthesis wird diese durch Das Gitter und die Endliche Menge an Werten sichergestellt.

Man könnte jeden möglichen Zustand von jedem möglichen Objekt als ein Knoten in einem Graph modellieren. Wenn man nun eine Kante für jede Abhängigkeit zwischen zwei möglichen Zuständen definiert

4 Meine Arbeit

4.1 Kern Idee

Die Idee ist das die gesamte zu generierende Welt als Graph an Operationen definiert. Operationen errechnen aus eine Menge an Input Werten in eine Menge an Output Werten.

Beispiel für Operationen sind:

- Addition
- Erzeugung einer Kugel aus Position und Größe
- Union zweier Volumen
- etc

Operationen nutzen die Ergebnisse von anderen Operationen und bilden so ein Graph an Abhängigkeiten.

Bei der Errechnung die Ergebnisse der Operations Knoten zwischen gespeichert.

Wenn sich nun der Abhängigkeits-Graph minimal ändert entspricht das ein Ergebnisse potenziell nicht mehr dem Abhängigkeits-Graph. Dies verstehe ich in dem Rest der Arbeit als eine invalide Lösung.

Jedoch muss durch die zwischen Speicherung nicht der ganze Graph neu errechnet werden, sondern nur der Teil der nicht mehr valide ist.

Die Kern Leistung meiner Arbeit ist die Entwicklung einer effizienten Implementierung für dieses Modell.

4.2 Framework

Die Generation wird in drei Stufen aufgeteilt.

1. „Composer“ Ein Graphischer Editor indem ein Nutzer eine Generations Regeln und Abhängigkeiten einstellen kann.

Hier für habe ich Node Graph System genutzt.

ToDo:

Warum?
- Mann kann leicht dafür sorgen dass die Konfiguartion valid ist.
- Abhängigkeiten leichter erkennbar

2. „Template“ Eine Datenstruktur die alle Abhängigkeiten darstellt.
3. „Collapser“ Stellt die aktuelle Welt da.

Nimmt ein neues „Template“ und ändert alles was nötig ist damit die Welt dem neuen „Template“ entspricht.

ToDo:

Sollte ich die Stufen nochmal umbenennen? Gerade collapser ergibt nicht so viel sinn.

4.2.1 Composer

4.2.2 Template

Das Template besteht aus zwei Graphen.

Template $:= (G_{\text{val}}, G_{\text{dep}})$

$G_{\text{val}} := (V_{\text{val}}, E_{\text{val}})$ ist ein Graph der die zu generierende Welt als Mathematische Formel beschreibt. Es gilt folgende Datentypen:

- Zahlen
- Positionen
- Volumen (als CSG implementiert)
- Eine Menge Positionen die nach einer Regel definiert sind

ToDo:

Ist Datentypen wirklich das beste Wort oder lieber Wert?

Für Positions Mengen habe ich Gitter und pseudo zufällige Verteilung implementiert.

Datentypen sind meist durch andere Datentypen definiert z.B. ist eine 3D Position durch 3 Zahlen definiert.

Ein $v_{\text{val}} \in V_{\text{val}}$ hat folgende Eigenschaften:

$\delta^-(v_{\text{val}})$:= Werte die für die Errechnung genutzt werden.

$\delta^+(v_{\text{val}})$:= Werte indem es für die Errechnung genutzt wird.

Daher spannt ein

ToDo:

In related Work zeigen dass es ok ist ein Graph so zu deffienieren

Der zweite Bestandteil des Templates ist der Abhängigkeits-Graph $G_{\text{dep}} := (V_{\text{dep}}, E_{\text{dep}})$.

Einer Untermenge der Werte $v_{\text{val}} \in V_{\text{val}}$ ist ein Knoten $v_{\text{dep}} \in V_{\text{dep}}$ zugeordnet, beschrieben durch die Funktion $\text{val}(v_{\text{dep}})$.

Bei der generation Welt wird nicht nur das finale Ergebnisse sondern auch die Werte der v_{dep} gespeichert. So kann bei einer Regeländerung weithin valide Werte wiederverwendet werden. Dies macht die „lazyness“ meines Ansatzes aus.

Bei der Entscheidung wie vielen Werten ein v_{dep} zu geordnet wird muss zwischen dem „Overhead“ und der Zeitersparniss durch wiederverwendung des Wertes abgewägt werden.

Im meiner Implementierung speichere ich alle Positions Mengen Werte zwischen.

Die Knoten im Abhängigkeits-Graph speichern von welchen andern Knoten sie abhängen $\delta^-(v_{\text{dep}})$, also alle Knoten in G_{dep} dessen Werte zur Errechnung genutzt werden.

ToDo:

Wie sage ich das es nur die nächsten sind?

ToDo:

Grafik von Datentypen und Abhängigkeits-Graph

4.2.3 Collapser

Der Collapser enthält einen Graphen der den errechneten Template entspricht. Dazu eine Queue an Aufträgen der noch zu erzeugenden und errechnenden Knoten. Diese Listen sind nach Level der Knoten sortiert. Der Collapser führt die Aufträge der Queue iterativ aus bis es keine Aufträge mehr gibt.

4.2.3.1 Formale Definition

Collapser $:= (G_{\text{col}}, Q)$

$G_{\text{col}} := (V_{\text{col}}, E_{\text{col}})$

Ein $v_{\text{col}} \in V_{\text{col}}$ hat folgende Eigenschaften:

$\delta^-(v_{\text{col}}) :=$ Collapser Knoten die zur Errechnung benötigt werden.

$t(v_{\text{col}}) :=$ Der Template Knoten der dem Collapser Knoten entspricht.

$\text{created}(v_{\text{col}}) :=$ Der Knoten der diesen Knoten erzeugt hat.

$\forall v_i \in \delta^-(v_{\text{col}}) \quad \text{data}(v_{\text{col}}, v_i) :=$
Der Wert des ahängigen Knoten der diesem Knoten zu geordnet ist.

Q ist die Queue in Aufträgen Jeder Auftrag $q \in Q$ hat ein Level $l(q)$. Dieses entspricht bei Erzeugungs-Aufträgen dem Level des zeugenden Knoten und bei Errechnungs-Aufträgen das Level des des zu errechnen Knoten.

$\text{pop}(Q) := \min_{q \in Q}(l(q))$

4.2.3.2 Erzeugungs Aufträge

Erzeugungs Aufträge enthalten den Index eines Knoten im Collapser Graph und den Index eines Erzeugungs-Eintrag in dessen Template-Knoten. Dieser Erzeugungs-Eintrag definiert entweder dass es genau n Knoten geben soll, oder die Zahl vom Datentypen abhängt.

Darauf wird die vorhandene Menge an Kindern mit der gewünschten Menge verglichen und bei Ungleichheit neue Kinder Knoten erzeugt / gelöscht.

ToDo:

Algorithmus

4.2.3.3 Wenn ein Knoten erzeugt wird

Wenn ein Knoten erzeugt werden die Knoten von dem er im Template abhängt im Collapser-Graph gesucht. Hierfür werden die vorerrechneten relativen Schritte verwendet die wie eine Weganweisung dienen um den Collapser-Graph von dem neu erzeugten Knoten zu sein Abhängigkeiten zu gelangen.

ToDo:

Algorithmus

4.2.3.4 Wenn ein Knoten gelöscht wird

Wenn ein Knoten gelöscht wird werden rekursiv alle Knoten gelöscht die von ihm abhängen.

ToDo:

Algorithmus

4.2.4 Errechnungs Aufträge

Errechnungs Aufträge errechnen den Wert von einem Knoten neu.

Dabei wird die der DAG der die Formel des Wert beschreibt rekursiv gelöst.

Wenn die Formel Hooks zu anderen Template Knoten enthält werden die entsprechenden Knoten in der Liste der Abhängigen Knoten des „Collapser“ Knoten gesucht und die Werte zu Errechnung der Formel genutzt.

Grundlegende Eigenschaft des Systems ist das es mehrere Knoten im Collapser für einen Knoten im Template geben kann. Und so einen für eine Hook eine Liste an Werten gefunden wird.

Daher sind alle Operationen zum errechnen der Formel als Operationen auf Listen geschrieben.

ToDo:

Beispiel

4.2.5 Abhängigkeits Kreise und Levels

Das Template kann Abhängigkeits Kreise enthalten. Um trotzdem eine valide Lösung errechnen zu können muss es für jeden Knoten v_{val} einen validen Null Wert geben.

So kann der Abhängigkeits Graph iterativ gelöst werden. Dazu werden pro Kreis im Abhängigkeits-Graph eine Kante als durchgeschnitten markiert $\delta^+_{\text{cut}}(v_{\text{dep}}) \subseteq \delta^+(v_{\text{dep}})$. Der Abhängigkeits-Graph ohne die durch geschnittenen Kanten $\delta^+_{\text{not cut}}(v_{\text{dep}}) := \delta^+(v_{\text{dep}}) \setminus \delta^+_{\text{cut}}(v_{\text{dep}})$ ist ein DAG. Also kann jedem Knoten ein Level $l(v_{\text{dep}})$ zu geordnet werden.

$$l(v_{\text{dep}}) > l(v_i) \quad \forall v_i \in \delta^+_{\text{cut not}}(v_{\text{dep}})$$

Die Knoten werden Level nach Level erzeugt und so sichergestellt das alle nicht geschnittenen Abhängigkeiten schon errechnet worden sind, wenn der Knoten selbst errechnet wird. Hat ein Knoten geschnittene Abhängigkeiten werden diese zu Errechnung genutzt wenn sie existiert. Andernfalls wird der Null Wert verwendet. Jeder Knoten der Null Werte für seine geschnittenen Abhängigkeiten nutzt wird nochmal errechnet, sobald alle Knoten einmal errechnet wurden. Dies wird so lange wiederholt bis keine Null Werte mehr verwendet worden sind.

4.2.6 Vorrechnete Schritte zu Abhängigkeiten

Um zu vermeiden, dass im Collapser alle Knoten durchsucht werden müssen um die abhängigen Knoten zu finden, werden speichert jeder Knoten im Abhängigkeits-Graph die relativen Schritte zu den Knoten von den er abhängig ist.

Um nun die alle abhängigen Knoten im Collapser Graph zu finden können diese Relativen Schritte wie eine Wegbeschreibung genutzt werden.

ToDo:

Vielleicht ein Beispiel

4.2.6.1 Implementierung

In der Implementierung wird zwischen geschnittenen und nicht geschnittenen Abhängigkeiten unterschieden.

Die nicht geschnittenen Abhängigkeiten sind als ein Baum an Schritten implementiert. Schritte gehen entweder hoch in ein Knoten vom dem dieser abhängig ist, oder runter ein einen Knoten der von diesem abhängt.

Wenn ein neuer Knoten im Collapser erzeugt wird, wird der Baum genutzt um alle abhängigen Knoten im Collapser zu finden.

Geschnittene Abhängigkeiten werden als Weg an Schritten gespeichert und erst gesucht, wenn deren Wert benötigt wird, Geschnittene Abhängigkeiten haben ein höheres Level als der Knoten selbst und existieren zur Zeit des Knoten wahrscheinlich noch nicht.

4.3 Warum Rust

- Moderne Sprache die viel potential für performance Optimierung bietet.
  - Da Laufzeit und Speicher Komplexität zwei wesentliche Bestandteile der Bewertung der Arbeit sind, ist es Sinnvoll eine Sprache zu wählen die viel Kontrolle über diese Bestandteile bietet, damit sie zu vollem Maße optimiert werden können.
- Genaues Type system zur Modellierung von Datenstrukturen
- Gute unterstützung aller Benötigenden Funktionen
  - Type Save Programming

- Generic Funktions
- Safe multi threading
- Ui und Redering Libaries

4.4 Implementation

4.4.1 Stabiele Listen

Alle Graphen sind mit stabilen Listen implementiert.

Eine stabile Listen ist eine sich automatisch vergrößerndes Array. Wenn dass aktuelle Array voll ist wird ein größeres Array alloziert und die werte mit einer memcopy operation rüber kopiert. Dazu änderen sich die Indexe von Elementen in stabilen Listen nicht. Wenn ein Element entfernt wird, werden nicht die weiteren aufgeschoben um die Lücke zu schließen sondern in der Lücke wird der Index des nächsten besetzten Element notiert. Dazu wird pro Element eine Versionsnummer gespeichert. Die oberen 32 bit des Index entsprechend dieser Version. Bei einem Zugriff wird die Version des Index mit der Version des Elements an der Stelle des Index verglichen. So bleiben Indexe valide solange das Element in der Liste ist und es wird erkannt wenn man mit veralteten Indexen zugreift.

Stabiele Listen erlauben Graphen effizient als Listen darzustellen, indem in den Knoten die Indexe der anderen Knoten gespeichert werden zu den Kanten existieren. Stabiele listen sind dabei wesentlich schneller als HashMaps.

Mit der Versionierung der Indexe können invalide Kanten sicher erkannt werden.

4.4.2 Multi Threading

Der Collapser sowie die Sampeling Operationen werden in asynchronous Workern ausgeführt. Zur Kommunikation werden Channel verwendet.

ToDo:

Quelle

Der Composer läuft im Render Thread und errechnet bei jeder Änderung das aktuelle Template. Dieses Template wird über ein Channel zum Collapser gesendet. Wenn der Channel noch ein altes Template enthält werden die Änderungen Notizen des alten Template zum neuen hinzugefügt und das Template wird ersetzt.

Die errechnete CSG Darstellung der Welt wird mit einem weiteren Channel an die Sampler gesendet, welche dieses in ein Voxel DAG oder Mesh umrechnet und auf die GPU gelanden.

Wenn die neue Welt hochgenanden wurde wird der Render Thread über ein Channel über diese Änderung informiert.

4.4.3 Small Vectors

Für die Zwischenspeicherung der Werte werden Small Vectors verwendet. Diese haben die Eigenschaft, dass die ersten N Elemente direkt auf dem Stack alloziert werden. Erst wenn diese voll sind wird ein Array auf dem Stack alloziert.

Da alle Werte müssen als Liste behandelt werden, da ein Knoten immer von mehreren Knoten für ein Input abhängt. Jedoch enthält diese Listen meist doch nur ein Element. Small Vectoren erlauben für diese Fälle die Stack allozieren zu sparen und bieten bessere Cache Lokalität.

4.5 Zeigen dass es Funktioniert

- minimal änderungen führen zu minimaler arbeit.

4.6 Nutzungs Beispiele

- Welt editor
- Infitie World Generation
- LOD systems

4.7 Visulation

4.7.1 Direkt CSG Rendering

4.7.2 Voxel sampeling

ToDo:

Sollte ich auf die Implementation und Datenstruktur ein gehen? Warscheinlich nicht.

- Minimale Änderungen im CSG führen zu nur minimaler Neuerrechnung des Voxel DAGs

4.7.3 Meshing

- Minimale Änderungen im CSG führen zu nur minimaler Neuerrechnung des Meshes

4.8 Bewertung und Vergleich zu andere Systemen

4.8.1 Wie kann man überhaupt das System gut vergleichen?

4.8.1.1 Gleiche Regeln in beidem Systemen implementieren und dann Geschwindigkeit vergleichen

- Probleme
  - Ich habe nur meine Implementation.
  - Gibt es überhaupt Regeln für Welten die man in meinem System und WFC darstellen kann?
    - Warscheinlich wären die Regeln biased.

– Regeln von vorhanden Spielen nutzen?

4.8.1.2 Nutzerfreundlichkeit bewerten

Sollte in meinem Fall im Bereich von gesunden Menschenverstand sein und daher wird keine Studie benötigt.

5 Future Work

5.1 Nullwerte führen zu leeren Lösungen

Da der Null Wert per definition eine valider Wert ist kann es dazu kommen, dass sich ein Abhängigkeits-Kreis zu null als Lösung entwickelt, auch wenn es theoretisch andere Lösungen gebe. Um dies zu lösen müsste ein anderer Ansatz zur Lösung von Abhängigkeits-Kreisen genutzt werden.

Arbeiten in Richtung closely connected Components in Verbindung könnten hier eine Lösung sein.

5.2 Ein system nutzen was nicht auf CSG basiert um danach nicht ein Mesh oder Voxel model errechnen zu müssen?

Bibliografie

[1] P. Merrell, „Example-based model synthesis“, in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, in I3D '07. Seattle, Washington: Association for Computing Machinery, 2007, S. 105–112. doi: 10.1145/1230100.1230119.

[2] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, und S. Risi, „Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network“. [Online]. Verfügbar unter: https://arxiv.org/abs/1805.00728

[3] M. Bazzaz und S. Cooper, „Level Generation with Constrained Expressive Range“, in *Proceedings of the 20th International Conference on the Foundations of Digital Games*, in FDG '25.: Association for Computing Machinery, 2025. doi: 10.1145/3723498.3723845.

[4] O. Withington und L. Tokarchuk, „Compressing and Comparing the Generative Spaces of Procedural Content Generators“, in *2022 IEEE Conference on Games (CoG)*, 2022, S. 143–150. doi: 10.1109/CoG51982.2022.9893615.

[5] A. Goslin, „Terrain Diffusion: A Diffusion-Based Successor to Perlin Noise in Infinite, Real-Time Terrain Generation“. [Online]. Verfügbar unter: https://arxiv.org/abs/2512.08309

[6] J. Whitehead *u. a.*, „Conversational Interactions with Procedural Generators using Large Language Models“, in *Proceedings of the 20th International Conference on the Foundations of Digital Games*, in FDG '25.: Association for Computing Machinery, 2025. doi: 10.1145/3723498.3723788.

Idee:

Text

ToDo:

Text

TODO: Text

Frage:

Text

$$A = \pi \tag{1}$$