HAW HAMBURG

BACHELORTHESIS
Cedric Stolze

Prozedurale Generierung von dreidimensionalen Strukturen mit zellularen Automaten und Wave Function Collapse

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Cedric Stolze

Prozedurale Generierung von dreidimensionalen Strukturen mit zellularen Automaten und Wave Function Collapse

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Wirtschaftsinformatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 28. März 2022

Cedric Stolze

Thema der Arbeit

Prozedurale Generierung von dreidimensionalen Strukturen mit zellularen Automaten und Wave Function Collapse

Stichworte

Prozedurale Generierung, Generierung von Höhlen, Zellulare Automaten, Wave Function Collapse

Kurzzusammenfassung

Prozedurale Generierung hilft unter anderem bei der Erzeugung vielseitiger und abwechslungsreicher virtueller Welten. Die unterschiedlichen Verfahren haben oft einen eingeschränkten Anwendungsbereich und können abseits davon schnell an ihre Grenzen kommen. Die Arbeit behandelt eine prototypische Implementierung zur Generierung von 3D-Höhlenstrukturen und kombiniert hierfür zellulare Automaten mit dem Wave Function Collapse. Die Umsetzung ermöglicht eine umfangreiche lokale und globale Modellierung und zeigt, wie die Kombination der verwendeten Verfahren die jeweiligen Stärken hervorhebt.

Cedric Stolze

Title of Thesis

Procedural generation in three dimensional space with cellular automata and wave function collapse

Keywords

procedural generation, cave generation, cellular automata, wave function collapse

Abstract

Procedural generation helps to create diverse and varied virtual worlds. The various methods often have a limited scope and can otherwise quickly reach their limits. This work proposes a prototype implementation for the generation of 3D cave structures that combines cellular automata with wave function collapse. The implementation allows for extensive local and global modeling and shows how the combination of the used methods highlights the strengths of each method.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

CA Zellulare Automaten.

GUI Graphical User Interface.

MC Marching Cube.

MVC Model-View-Controller.

PCG Prozedurale Content Generierung.

WFC Wave Function Collapse.

1 Einleitung

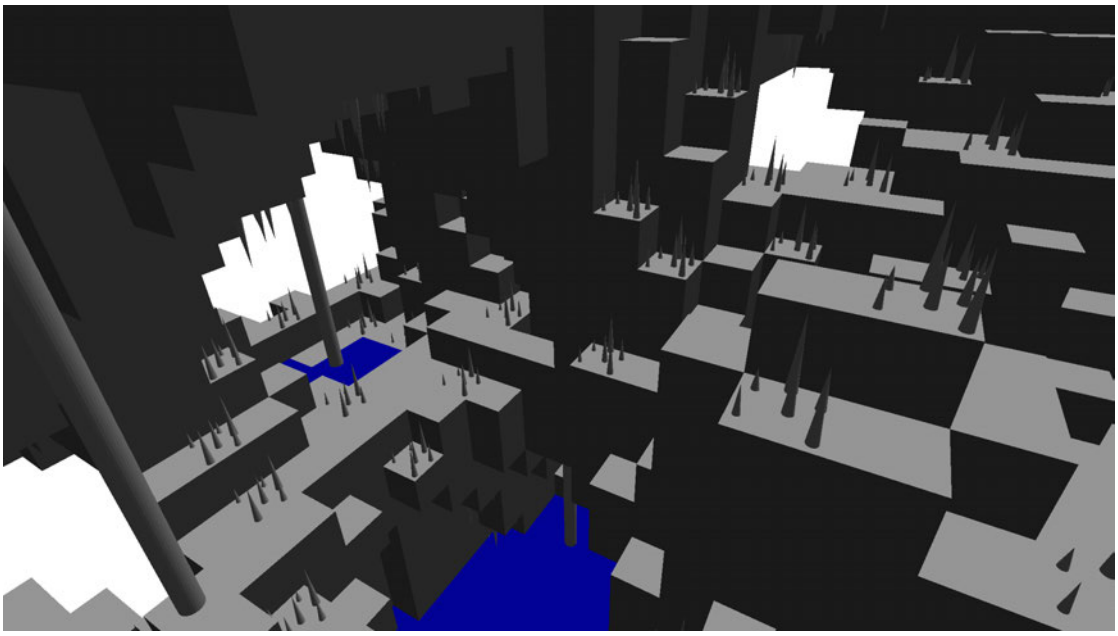


Abbildung 1.1: Ziel der Arbeit sind zellulare Höhlenstrukturen, welche mithilfe des WFC Algorithmus durch komplexe Objekte erweitert werden (eigene Darstellung)

1.1 Motivation

Der Aufwand für die Entwicklung von glaubhaften virtuellen Welten im Bereich der Computergrafik und insbesondere in Computerspielen erhöht sich stark mit dem Umfang und dem Detailgrad. Es sind zunehmend mehr und höher aufgelöste Modelle gefordert. Dies durch aufwendige Handarbeit umzusetzen kostet viele Ressourcen und ist für Entwicklerstudios häufig ein wirtschaftlich kritischer Faktor. Der Game Designer von *The Sims* und *Sim City* Will Wright [11] beschreibt diesen Umstand als *The Mountain of Content*

Problem [3]. Der Bereich der Prozedurale Content Generierung (PCG) beschäftigt sich mit dem Ansatz, genau hierfür Lösungen zu finden und den Aufwand für die Designer zu minimieren [2]. Es sollen virtuelle Modelle und Welten generiert werden, ohne dass diese einzeln per Hand modelliert werden müssen. Hierfür werden Verfahren verwendet, welche durch bestimmte Parameter und Bedingungen glaubhafte Inhalte erzeugen können. Diese sollten sich im Optimalfall in der Qualität nicht von den handgebauten Modellen unterscheiden können oder diese sogar durch eine höhere Vielfalt und Abwechslung übertreffen. Sie sollten vor allem reproduzierbar sein, denn prozedurale Generierungsverfahren haben meist ein deterministisches Verhalten.
Zellulare Automaten (CA) sind eines dieser Verfahren und gehören zu den ältesten und simpelsten Algorithmen in der prozeduralen Generierung [5]. Ein CA besteht aus einem Zellennetz, in der jede Zelle auf Basis seiner Nachbarzustände sterben oder geboren werden kann. Durch dieses simple Konzept können komplexe Systeme und Strukturen modelliert werden. Auf der anderen Seite stehen aktuellere Verfahren wie der Wave Function Collapse (WFC). Dieser basiert auf dem Konzept von Nachbarschaftsbedingungen und versucht aus einem beliebigen Eingabemodell lokale Muster zu extrahieren und dies durch Kollabieren von Zustandspositionen auf eine theoretisch unbegrenzte Ausgabe abzubilden [20].
Prozedurale Generierung kann im Bereich der Videospiele vor allem seine Stärken bei der Generierung von Landschaften zeigen. Hier ist oft die Größe, aber auch die Qualität der Landschaften entscheidend. Vor allem Höhlenstrukturen bieten ein großes Potential für eine algorithmische Generierung, da in Videospielen hier häufig das Entdecken und Erforschen neuer Tunnelsysteme im Vordergrund steht [29]. Beide genannten Verfahren sind in der Lage dies umzusetzen. Gerade zellulare Automaten können hier ihr Potential durch ihre simple Umsetzung und den organischen Strukturen zeigen, jedoch nur bis zu einer bestimmten Komplexität.

1.2 Problemstellung

Prozedurale Generierungsverfahren können in bestimmten Anwendungsfällen ihr volles Potential zeigen. Wenn die Anwendung jedoch komplexer wird und die Anforderungen sich vergrößern, werden Schwächen und Limitierungen der Verfahren deutlich.
Zellulare Automaten bieten global betrachtet eine enorme Vielfalt in der Erzeugung von Strukturen. Jedoch sind gewünschte Resultate schwer vorhersehbar und die Manipulation von lokalen Details im Nachhinein nicht möglich, da Anpassungen den gesamten

zellularen Automaten betreffen. Dies schränkt die Möglichkeiten zum Modellieren stark ein und ist in vielen Fällen für eine gesteuerte prozedurale Generierung unbrauchbar.
Der Wave Function Collapse hat hingegen andere Probleme. Zum einen garantiert das Verfahren im Gegensatz zum CA nicht, dass es überhaupt jemals zu einem Ergebnis kommt. Je nach Modell kann es vorkommen, dass Konflikte unvermeidbar sind und der Algorithmus nicht terminiert [28]. Zum anderen kann die Laufzeit sehr unzuverlässig sein, da im schlechtesten Fall der Algorithmus durch zuviele Kombinationen zu viele Iterationen benötigt.

1.3 Ziele und Vorgehen

Das Ziel der Arbeit ist es zu zeigen, wie sich zwei grundsätzlich verschiedene prozedurale Generierungsverfahren so kombinieren lassen, dass sie ihre Nachteile ausgleichen und die jeweiligen Stärken betonen. Dies soll an einer prototypischen Umsetzung durch den Einsatz von zellularen Automaten und dem Wave Function Collapse gezeigt werden.
Hierfür wird eine Software entwickelt, welche durch eine simple Benutzerschnittstelle mit Parametern und einem grafischen Editor ermöglicht, beide Verfahren losgelöst und in Kombination anzuwenden. Der Fokus liegt hier auf der Generierung von dreidimensionalen Höhlenstrukturen. Hierzu gehört die grundlegende Struktur, aber auch kleinere Details, welche die Struktur füllen und somit einem praktischen Anwendungsfall näher kommen. Die Arbeit legt hierbei keinen Wert auf ansprechende Grafik, sondern primär auf die prozeduralen Generierungsverfahren und wie diese sich kombinieren lassen.

2 Grundlagen

Im Folgendem werden einige grundlegende Konzepte der prozeduralen Generierung beschrieben und Techniken erläutert, auf welchen diese Arbeit aufbaut. Es soll einen groben theoretischen Einblick in die Konzepte gegeben und deren Relevanz und Anwendung in der Computergrafik verdeutlicht werden.

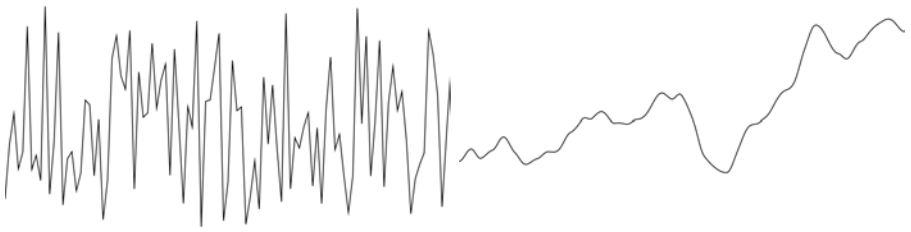
2.1 Prozedurale Generierung

Prozedurale Generierung oder auch Prozedurale Content Generierung (PCG) umfasst die Erzeugung von digitalen Inhalten einer Software, welche ohne manuellen Einfluss des Menschen realisiert wird. Darunter zählt unter anderem die Generierung von Charaktermodellen, Musik, Texturen oder auch ganzer Städte [8]. Hierfür werden Algorithmen und mathematische Modelle genutzt, um deterministisch Ergebnisse zu garantieren [47].
Einen großen Anwendungsnutzen findet die prozedurale Generierung im Bereich der Videospiele. Digitale Inhalte hierfür zu produzieren erfordert einen enormen Aufwand von Ressourcen. Entwicklerstudios nutzen PCG dementsprechend um Kosten, Zeit und Speicher zu sparen oder Welten zu erschaffen, welche für jeden Spieler einzigartig sind. Ohne PCG wäre diese Herangehensweise schwer umsetzbar, da jeder Inhalt aufwendig per Hand modelliert werden müsste [8].
Eines der ersten Videospiele, welches sich Algorithmen zur Generierung von Inhalten zu nutzen macht, ist das 1980 erschienene *Rogue* [21]. Hier werden komplexe Dungeons mit dessen interagierbaren Objekten wie Truhen oder Gegnern prozedural generiert. Die Motivation hierfür liegt vor allem in dem damals knappen Speicherplatz begründet [11].
Ein aktuelleres Beispiel für den Einsatz stellt die *Borderlands* Reihe dar. Hier wird PCG genutzt, um eine möglichst große Masse an vielfältigen Spieleinhalten zu ermöglichen [45]. Das Resultat sind Millionen von unterschiedlichen Waffen und Ausrüstungsgegenstände, welche den Sammler- und Wiederspielwert erhöhen soll. Mit stärkerer Hardware und der tieferen Forschung im Bereich PCG, stellen Speicherplatzlimiterungen immer

mehr eine Nebensächlichkeit dar und die erzeugte Vielfalt der Verfahren rückte mehr in den Fokus [45]. Einige Spiele wie *No Mans's Sky* (2016) machen PCG zu dem zentralen Spielerlebnis, indem es ganze Galaxien, Planeten und deren Fauna und Flora prozedural generiert [41]. Ohne diese Vorgehensweise wäre eine solche Umsetzung in diesem Umfang nicht möglich und verdeutlicht die Relevanz von PCG für unter anderem Videospiele.

2.2 Perlin Noise

Perlin Noise ist ein weitverbreitetes Werkzeug in der prozeduralen Generierung und wurde 1985 von Ken Perlin entwickelt [36] und durch Perlin selbst 2002 verbessert [37]. Es beschreibt eine Rauschfunktion, welche eine weiche Kurve von Verlaufswerten erzeugt. Anders als rein zufällige Werte, sind dicht beieinanderliegende Punkte der Kurve sehr ähnlich zueinander (Abb. 2.1). Dies erlaubt weiche Übergänge zwischen zwei Punkten, welches unter anderem für die Generierung von Landschaften gut geeignet ist. Perlin Noise kann theoretisch für alle Dimensionalitäten angewendet werden. In der weiteren Erläuterung wird sich auf die zweite Dimension bezogen.



(a) zufällige Werte (b) Perlin Noise

Abbildung 2.1: Vergleich zwischen zufälligen Werten und Perlin Noise anhand eines Graphen (Darstellung in Anlehnung [43])

Über eine zweidimensionale Fläche wird ein Gitter M der Größe $n \times n$ gespannt. An jedem Eckpunkt $v_{i,j}$ mit $0 \leq i, j \leq n$ der Gitterzellen werden pseudozufällige Vektoren G zugeordnet, welche als Verlaufsvektoren angesehen werden. Die Rauschfunktion wird in die Richtung des Vektors einen positiven Verlauf haben. Für einen beliebigen Punkt (x, y) innerhalb der Fläche werden die Distanzvektoren D zu allen Eckpunkten $v_{i,j}$ der zugehörigen Zelle in M ermittelt. Mit diesen werden die Skalarprodukte zwischen D

und den Verlaufsvektoren der Ecken G berechnet durch $\vec{d_{i,j}} \cdot \vec{g_{i,j}}$ mit $\vec{d_{i,j}} \in D, \vec{g_{i,j}} \in G$. Die 4 resultierenden Vektoren werden nun mithilfe der normierten Ausgangskoordinaten interpoliert und geben einen Verlaufswert hierfür aus. Dieses Verfahren wird mit allen Punkten in der Fläche wiederholt [32].
Die großen Vorteile von Perlin Noise und anderen Rauschfunktionen liegt zum einen in ihrer entkoppelten Anwendung, da Sie keinen vordefinierten Modellen unterliegen und multidimensional und flexibel angewendet werden können. Zum anderen auch in ihrem kompakten Aufbau und der konstanten Laufzeit, weshalb Perlin Noise für prozedurale Generierung vor allem in der Echtzeitanwendung weite Verbreitung findet [24].

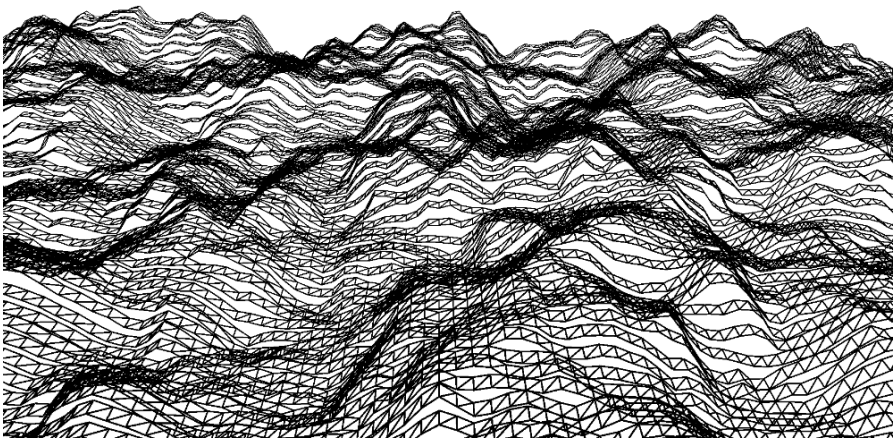


Abbildung 2.2: Durch Perlin Noise generierte Höhenkarte (Darstellung in Anlehnung [43])

Diese Eigenschaften werden in der prozeduralen Generierung häufig zur Landschaftsgenerierung benutzt [12]. Im simpelsten Fall wird ein 2D-Netz ausgebreitet und für jede Zelle ein Perlin Noise zugeordnet. Der resultierende Wert liegt in einem Werteintervall, meist zwischen -1 und 1. Ein Wert von 0 kann in dem Fall als Meeresspiegel der Landschaft interpretiert werden. Werte darüber formen Hügel, Berge und Täler. Werte unter 0 dementsprechend das Terrain vom Meeresgrund. Somit bildet das erzeugte Wertegitter eine Höhenkarte ab (Abb. 2.2). Je nach Parameter von Perlin Noise können Formen verzerrt und verstärkt werden, sowie verschiedene Rauschfrequenzen überlagert werden [32].

2.3 L-Systeme

Eine weitere Technik, die in der prozeduralen Generierung verwendet wird, sind die L-Systeme oder auch Lindenmayer-Systeme, welche der Biologe und Botaniker Aristid Lindenmayer 1968 entwickelte [26]. Sie sind definiert als formale Grammatik, mit welcher fraktale Modelle und Formen erzeugt werden können. Lindenmayer nutzte sie, um charakteristische Wachstumsmuster von Pflanzen zu beschreiben. Heute finden L-Systeme unter anderem ihre Anwendung in der prozeduralen Generierung von Städten [38][30] oder ähnlichen Infrastrukturen und Modellen.
Ein L-System ist definiert durch ein Tripel $L = (V, w, P)$ wobei V das Alphabet aller genutzten Zeichen und Konstanten ist. Das Axiom w ist eine nicht leere Zeichenkette mit $w \in V^+$ und definiert den initialen Zustand des Systems. P ist eine Menge an Produktionen mit $P \subset V \times V^*$ [39]. Eine Produktion $(\alpha \to \chi) \in P$ ist eine Ersetzungsregel, welche ein ausgehendes Zeichen α auf ein resultierendes Wort χ abbildet. Es wird angenommen, dass es für jedes Zeichen $\alpha \in V$ eine Produktion gibt. Ansonsten ist die Identitätsabbildung $\alpha \to \alpha$ implizit Teil von P. Des Weiteren gilt ein L-System nur dann als deterministisch, wenn jedes Zeichen genau auf ein Wort abgebildet werden kann [38]. Die bisher erläuterte Definition beschreibt ein L-System, welches kontextfrei ist. Im Gegensatz dazu steht die Variation der kontektsensitiven L-Systeme. Hier beziehen sich die Produtkionsregeln nicht nur auf das einzelne Zeichen, sondern auch auf die davor und danach auftauchenden Zeichen innerhalb der Zeichenkette.

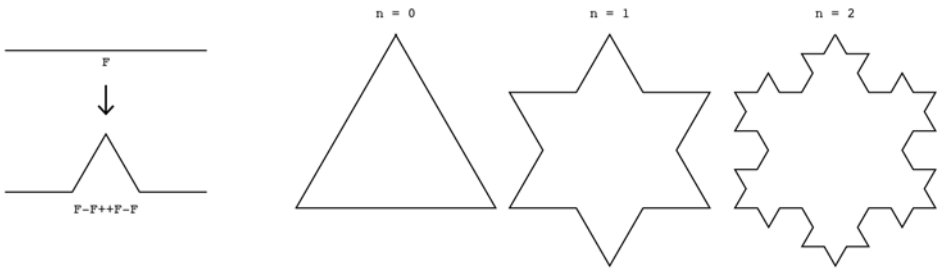


Abbildung 2.3: Durch ein L-System generiertes Fraktal (Darstellung in Anlehnung [16])

Ein simples Beispiel [16] für die Anwendung sind fraktale Strukturen wie beispielsweise eine Schneeflocke (Abb. 2.3). Hierfür wird das Alphabet $V = \{F, +, -\}$ mit folgenden Bedeutungen genutzt:

- F := zeichne eine Linie der Länge l
- $+$:= nach rechts drehen
- $-$:= nach links drehen

Mithilfe des Axioms $w = F++F++F$ und der Produktion ($F \rightarrow F-F++F-F$) kann rekursiv eine fraktale Kurve erzeugt werden, welche einer Schneeflocke ähnelt. In diesem Beispiel liegt die Rotation immer bei 60°. Die Länge der gezeichneten Linie entspricht nach n Iterationen $l_n = l_0 \times (\frac{1}{3})^n$, wobei l_0 die initiale Linienlänge ist.

2.4 Zellulare Automaten

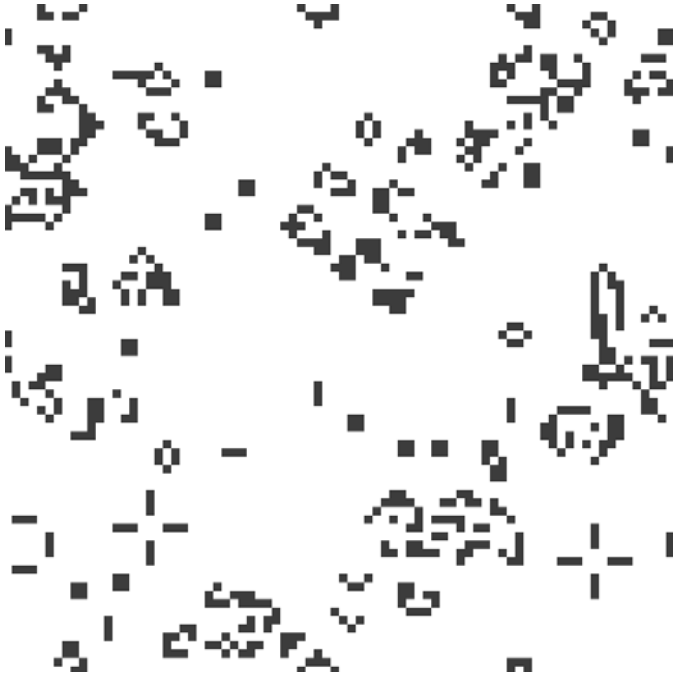


Abbildung 2.4: Conway's Game of Life (eigene Darstellung)

Ein zellularer Automat beschreibt ein iteratives Zellensystem, welches Zyklen von Zuständen durch Nachbarschaftsbedingungen darstellt. Erstmals wurden zellulare Automaten

1940 von Stanisław Ulam vorgestellt [49] und wurden später von John von Neumann erweitert. Zu der Zeit forscht von Neumann an der Idee der selbst reproduzierbaren Maschinen [43].

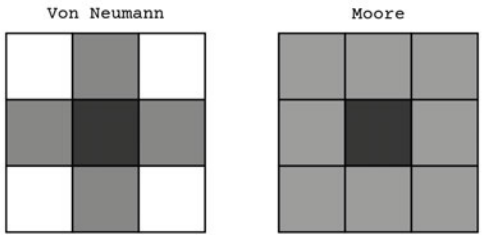


Abbildung 2.5: CA Nachbarschaften (Darstellung in Anlehnung [34])

Ein zellularer Automaten C ist definiert durch ein Quadrupel $C = (R, N, Q, \delta)$ [40]. Der Raum R stellt die Menge aller Zellen dar und kann in verschiedensten Dimensionen abgebildet werden. Die weitverbreitetste ist jedoch der zweidimensionale Raum R^2. Zu dem wird eine Nachbarschaft N definiert. Die Nachbarschaft legt fest, welche anliegenden Zellen einer Zelle in die Zustandsveränderung mit einbezogen werden. Darunter zählt die von Neumann Nachbarschaft $N_{neumann}$ und die Moore Nachbarschaft N_{moore} [34] - dargestellt in Abbildung 2.5.

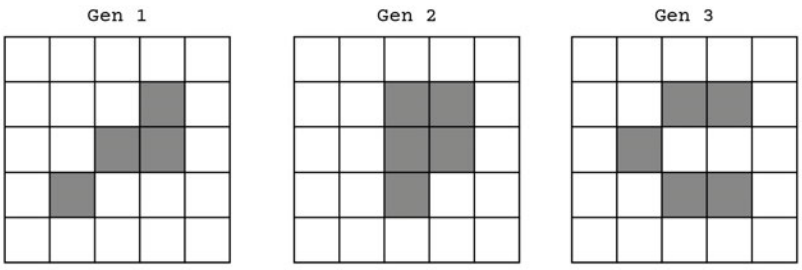


Abbildung 2.6: Generationswechsel innerhalb Game of Life (eigene Darstellung)

Jede Zelle besitzt eine Menge Q an Zuständen, welche in den meisten Fällen eine binäre Zustandsmenge beschreibt. In dem Fall kann eine Zelle leben oder tot sein. Auf Basis von R, N und Q wird zudem noch eine Funktion $\delta_N : Q \to Q$ definiert, welche den Übergang

von einem Zustand einer Zelle in einen anderen Zustand definiert. Ausgehend von einem Zustandraum R_t, wobei dieser die initialen Zustände jeder Zelle $c \in R$ des Raumes zum Zeitpunkt t darstellt, wird auf jede Zelle die Funktion δ_N angewendet. Anhand der Nachbarzellen $b \in c_N$ einer Zelle und deren Zustände wird der Zellenzustand von c geändert. Somit wird R_t auf R_{t+1} abgebildet und stellt somit einen iterativen Lebenszyklus über mehrere Generation einer Zellenmenge dar (Abb. 2.6).
Ein bekanntes Beispiel für zellulare Automaten ist *Conway's Game of Life* [5]. Es beschreibt ein simples Modell für Lebenszyklen und Populationsausbreitung von atomaren Zellen. Die Regeln basieren auf der Moore Nachbarschaft und sind wie folgt definiert:

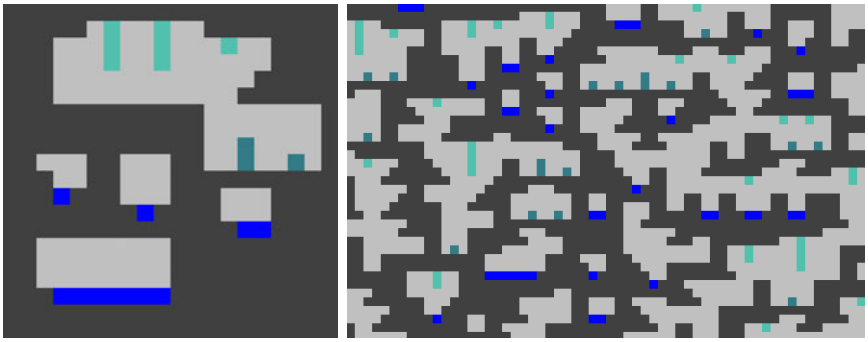
- eine lebende Zelle stirbt genau dann, wenn sie weniger als 2 oder mehr als 3 lebende Nachbarn besitzt
- eine tote Zelle wird genau dann geboren, wenn sie 3 lebende Nachbarn besitzt

Über die Generationen hinweg entstehen so vielfältige Formen. Einige Zellenanordungen brechen sofort zusammen, andere erreichen einen zyklischen und stabilen Zustand (Abb. 2.4). Im dreidimensionalen Raum lassen sich ähnliche Strukturen beobachten [4].

2.5 Wave Function Collapse

Der von Maxim Gumin entwickelte Wave Function Collapse [14] ist innerhalb der prozeduralen Generierung ein Algorithmus, welcher zur Textur-Synthese gehört. Das Ziel ist es, aus einer kleinen Eingabe wie einem Bild oder einem 3D-Modell, ein ähnliches Resultat auf eine theoretisch unbegrenzte Größe zu skalieren. Dabei sollen alle lokalen Merkmale wie Proportionen und Verhältnisse der Eingabe beibehalten werden und konsistent zueinander bleiben. Dies wird erreicht, indem die Eingabe in kleinere Teilmengen aufgespalten wird. Diese extrahierten Teilmengen werden auch Muster oder Prototypen [28] genannt. Zwischen diesen werden anschließend gültige Kombinationen von Anordungen durch Bedingungen festgelegt - den Adjacency Contraints. Deshalb gehört der Wave Function Collapse zu den bedingungslösenden Algorithmen [20][9].

Im Wesentlichen besteht der Algorithmus aus 2 Teilprozessen. Dem Verarbeiten vom Eingabeobjekt und dem darauf aufbauenden iterativen Kollabieren zur Ausgabe. Im Folgendem soll dies anhand einer zweidimensionalen Datenstruktur als Eingabe näher erläutert werden, welches beispielweise ein Bild sein kann (Abb. 2.7).



(a) Eingabe (b) Ausgabe

Abbildung 2.7: Höhlenstruktur generiert durch WFC (eigene Darstellung)

Der erste Teilprozess verarbeitet ein zweidimensionales Eingabebild α. Es wird iterativ eine Menge P an Mustern aus der Eingabe mittels eines Kernels der Größe $k \times k$ Pixeln ausgeschnitten. Die somit extrahierten Muster $m \subset \alpha$ stellen somit alle expliziten lokalen Teilmengen der Eingabe dar. P kann künstlich durch spiegeln, drehen oder ähnlichen Transformationen der Muster erweitert werden, um im späteren Schritt eine höhere Varianz in der Ausgabe zu erhalten [9]. Um nun mögliche Kombinationen der Muster zu definieren, muss es eine Funktion δ geben, welche die Nachbarschaftgültigkeit zweier Muster anhand einer Ausrichtung d zueinander und einer definierten Nachbarschaftbedingung β evaluieren kann. β kann beispielweise die Übereinstimmung von Pixelwerten sein. Die Funktion würde eine Nachbarschaft als gültig evaluieren, wenn beispielsweise alle überlappenden Pixel in der resultierenden Anordnung durch d von beiden Mustern gleich sind. Nun wird für jedes Muster m_x und für jede Ausrichtung eine Menge $V_{x,d}$ an gültigen Nachbarmustern durch die Funktion β und der Kombination durch alle anderen Muster m_y berechnet. Somit wird genau definiert, welche Paare (m_x, m_y) mit $m_x, m_y \in P$ zueinander eine gültige Nachbarschaftbedingung in einer bestimmten Ausrichtung haben. Der zweite Teilprozess stellt den Kern des Wave Funtion Collapse dar. Hier wird der Dimensionalität der Ausgabe entsprechend eine Matrix W definiert, welche namensgebend für den Algorithmus als *Wave* bezeichnet wird [14]. Jedes Element $p \in W$ stellt eine Superposition dar, die im entferntesten an den Zustandsbegriff der Quantenmechanik

angelehnt ist. Alle Positionen in W sind zu Beginn in einem maximal undefinierten Zustand, bei welchem sie jedes vorher berechnete Muster aus P gleichzeitig darstellen. Das Ziel ist es, diesen undefinierten Zustandsraum aller Positionen auf einen konkreten Zustand zu kollabieren. Wie undefiniert eine Zustandsmenge einer Position ist, wird durch die Entropie e beschrieben [6].

$$e = -\sum h_m \log(h_m) \tag{2.1}$$

Wobei h_m mit $0 \leq h_m \leq 1$ die Gewichtung der Muster beschreibt, welche die relative Häufigkeit innerhalb der Eingabe α darstellt. Der Wave Function Collapse ist bestrebt, alle Entropien e_p zu minimieren und präferiert zum Kollabieren die Position mit der niedrigsten Entropie. Die Position mit der niedrigsten Entropie hat die geringste Wahrscheinlicheit einen Konflikt auszulösen. Dies liegt daran, dass eine niedrige Entropie eine verringerte Anzahl an verbleibenden Zuständen der Position impliziert. Da auch die relativen Häufigkeiten der Muster mit einfließen, erhalten Positionen mit häufigen Musterfrequenzen eine niedrigere Entropie, was ebenso zur Konfliktvermeidung beiträgt.
W wird iterativ kollabiert, wobei jede Iteration aus 3 Schritten besteht [20]:

- Identifizieren der Position mit der niedrigsten Entropie (*observe*)
- Kollabieren der identifizierten Position(*collapse*)
- Propagieren der Änderungen durch das Kollabieren(*propagate*)

Im ersten Schritt wird die Position p identifiziert, welche die niedrigste Entropie aller Positionen in W hat. Für den Fall, dass hier mehrere Positionen infrage kommen, wird aus diesen eine zufällige Position gewählt. Die identifizierte Position wird anschließend auf einen definierten Zustand kollabiert. Sei Z_p die möglichen Zustände oder Muster, welche für p noch infrage kommen. Dann wird p in den Zustand $m \in Z_p$ kollabiert, welcher die höchste Wahrscheinlichkeit χ hat [42].

$$\chi(m) = \frac{h_m}{H_p} \tag{2.2}$$

H_p stellt die Summe der relativen Wahrscheinlichkeiten aller verbleibenden Zustände Z_p einer Position p dar. Somit kann die Position in den Zustand kollabiert werden, welcher relativ zu α die höchste Frequenz aufweist.

Wenn die identifizierte Position kollabiert ist, wird im dritten Schritt der geänderte Zustand auf W propagiert. Hierfür werden für alle in jeder Richtung d anliegenden Positionen p_d von p verglichen, ob deren mögliche Zustände durch die Änderungen von p noch gültig sind. Wenn ein Zustand der Nachbarposition p_d keine gültige Nachbarschaft mit einem der Zustände aus Z_p hat - also nicht mindestens ein gültiger Nachbarzustand existiert - wird dieser Zustand von p_d entfernt und die Entropie sinkt.

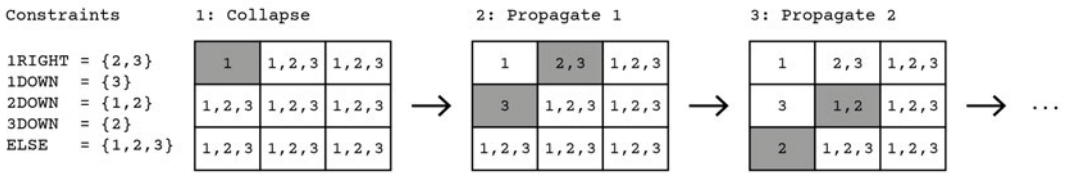


Abbildung 2.8: WFC propagieren (Darstellung in Anlehnung [42])

Falls dieser Fall eintritt, wird auch die Änderung von p_d propagiert. Das Propagieren wird so lange durchgeführt, bis keine Änderungen mehr auftreten (Abb. 2.8). In dem Fall, dass alle Zustände einer Position durch eine Änderung entfernt werden und die Position eine Entropie von 0 hat, gilt dies als Konflikt und ist ungültig. Hier werden alle Änderungen seit dem auslösenden Kollabieren zurückgesetzt [22].

3 Konzepte und Techniken

In diesem Kapitel soll genau erläutert werden, welche Techniken für die Umsetzung der Projektziele zum Einsatz kommen und wie bestehende Konzepte hierfür angepasst und erweitert werden. Alle beschriebenen Konzepte bauen auf den grundlegenden Ideen und Algorithmen aus Kapitel 2 auf.

3.1 Anforderungen

Die nachfolgend beschriebenen Anforderungen an die prototypische Umsetzung sollen den Umfang und Fokus der Arbeit verdeutlichen. Alle Anforderungen sollen messbar sein, um den Erfolg und die Umsetzung evaluieren zu können. Dabei wird zwischen rein funktionalen und nicht funktionalen Anforderungen unterschieden. Im Sinne der Software Entwicklung können folgende Punkte ebenfalls als *User Stories* formuliert werden. Für den Zweck der Übersichtlichkeit werden sie jedoch als Stichpunkte aufgelistet.

3.1.1 Funktionale Anforderungen

Die funktionalen Anforderungen definieren die Kernfunktionen und Features der Umsetzung. Darunter zählen grundsätzliche Verhaltensanforderungen der Algorithmen, aber auch mögliche Benutzerinteraktionen mit der Software.

- es sollen dreidimensionale Höhlenstrukturen generiert werden können, welche eine starke Konnektivität aufweisen und durchgängig sind
- die grundsätzliche Höhlenstruktur wird durch zellulare Automaten realisiert und durch WFC mit Detailobjekten ergänzt
- der WFC soll die grundsätzliche Höhlenstruktur nicht modifizieren

- beide prozedurale Generierungverfahren sollen auch einzeln in der Software verwendet werden können
- die Benutzerschnittstelle soll durch Parametereinstellungen und einem einfachen 3D-Editor auf die Höhlenstrukturen Einfluss nehmen können
- es soll zusätzlich eine simple zellenbasierte 3D-Modellbearbeitung bereitgestellt werden

3.1.2 Nicht funktionale Anforderungen

Zu den nicht funktionalen Anforderungen gehören Randbedingungen und Qualitätsmerkmale an der Umsetzung der Software. Dazu gehört beispielsweise, wie stabil das System läuft, wie gut es getestet ist und wie effizient mit Ressourcen umgegangen wird.

- alle verwendeten Algorithmen sollen sich deterministisch verhalten und generierte Modelle durch einen Seed reproduzierbar sein können
- die Generierung der Strukturen durch beide Verfahren sollte bis zu einer Gittergröße von 15^3 Zellen unter einer Berechnungszeit von maximal 10 Sekunden bleiben (CPU basiert und ohne Rendern)
- die Softwarearchitektur sollte modular und untereinander schwach gekoppelt sein
- innerhalb der Kernkomponenten sollten alle essenziellen Berechnungen durch Unit Tests abgedeckt sein

3.2 Erweiterung der zellularen Automaten

Ein Zellulare Automaten (CA) bietet großes Potential in dem simplen Aufbau, welches durch wenige Bedingungen und Regeldefinitionen unterschiedlichste Ergebnisse erzielen kann und auch schnell zu berechnen ist [19]. Aus diesen Gründen eignen sich zellulare Automaten sehr gut für die prozedurale Generierung. Jedoch ist es trotz ihres deterministischen Verhaltens schwer vorhersehbar, welches Ergebnis der Automat erzeugen wird. Die globale Generierung ist durch das genaue Definieren der Nachbarschaftregeln noch gut kalkulierbar - beispielweise wenn eine Höhlenstruktur generiert werden soll. Eine

lokale Beeinflussung ist hingegen schwer möglich. Genaue Wege und Pfade oder präzise Start und Endpunkte von Höhlen können vorweg nicht definiert werden. Die Arbeit beschäftigt sich unter anderem mit beschriebener Limitierung und wird im Folgendem erläutert.

3.2.1 Zellulare Automaten in der dritten Dimension

Für diese Arbeit wird vor allem der dreidimensionale zellulare Automat behandelt. Dieser ist eine weitere Abstraktion der mehr verbreiteten ein- und zweidimensionalen zellularen Automaten und unterliegt derselben grundlegenden Definition [40]. Wie in Kapitel 2.4 beschrieben, ist ein zellularer Automat C definiert durch ein Quadrupel $C = (R, N, Q, \delta)$. Der größte Unterschied liegt in der räumlichen Dimensionalität, welcher bei einem dreidimensionalen Automaten R_3 entspricht und den daraus resultierenden Nachbarschaftsbedingungen N.
Im Gegensatz zum zweidimensionalen Automaten werden neben der X- und Y-Achse auch die benachbarten Zellen in Richtung der Z-Achse mit einbezogen. Trotzdem können die bewährten Nachbarschaftsdefinitionen - Moore und von Neumann - in einer adaptierten Form (Abb. 3.1) genutzt werden [13].

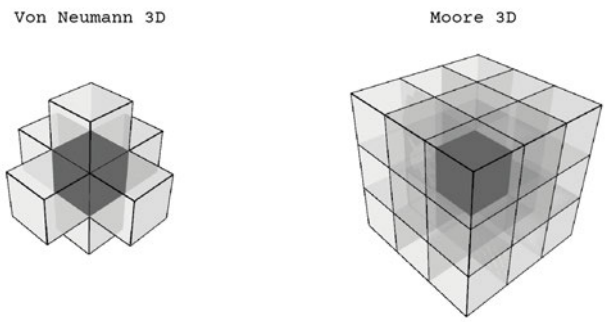


Abbildung 3.1: CA Nachbarschaften 3D (Darstellung in Anlehnung [13])

Um eine Nachbarschaft zu bestimmen, werden alle Zellen als Nachbarn in Betracht gezogen, welche sich in einem 3x3x3 Zellengitter befinden. Hier befindet sich die Zelle - auf welche sich die Nachbarschaft bezieht - im Zentrum. Um eine Moore-Nachbarschaft im dreidimensionalen Raum abzubilden, gelten alle Zellen in dem beschriebenen Gitter als Nachbarn, außer die im Zentrum liegende. Damit erhöht sich die Moore-Nachbarschaft

gegenüber der zweidimensionalen Abwandlung von 8 auf 26 Nachbarzellen. Die von Neumann Nachbarschaft zählt in dem Gitter alle Zellen als Nachbarn, welche direkt auf den Achsen der zentralen Zelle liegen. Hier erhöht sich Anzahl der möglichen Nachbarn von 4 auf 6 gegenüber der zweidimensionalen Form (Abb. 3.1). Verwendete Nachbarschaftregeln werden durch die potentiell erhöhte Anzahl an Nachbarzellen deutlich komplexer, aber auch vielseitiger.

3.2.2 Parametrisierung von zellularen Automaten

Die Parametrisierung von zellularen Automaten wird in zwei Abstufungen unterteilt. Zum einen die globale Steuerbarkeit des Automaten, welche sich auf das grobe Verhalten der gesamtheitlichen Ebene bezieht. Also in welcher Form, Ausprägung und Verhältnissen eine Struktur erzeugt wird. Zum anderen die lokale Steuerbarkeit, die auf Details innerhalb der Struktur selbst Einfluss nimmt. Dazu gehören genaue Positionierungen, Verläufe und punktuelle Anpassungen.
Die globale Parametrisierung kann durch bestimmte Faktoren in der Definition des Automaten selbst beeinflusst werden. Der initiale Zustand der Zellen nimmt großen Einfluss auf die spätere Form der Zellenstruktur. Es kann hier beispielsweise definiert werden, an welcher Position sich die initial lebenden Zellen im Raum befinden. Eine gleichmäßige Verteilung führt zu grundsätzlich anderen Ergebnissen, als wenn diese in Clustern angeordnet sind. Ebenso nimmt die Dichte an lebenden Zellen der Startgeneration großen Einfluss auf den Verlauf der folgenden Zellengeneration. Der größten globalen Parameter sind jedoch die Nachbarschafts- und Regeldefinitionen. Sie sagen aus, welche Nachbarzellen einbezogen werden und anhand dessen, welche Zustände daraus resultieren. Die Nachbarschaftsregeln eines zellularen Automaten können notiert werden als:

$$\delta_N(x) = \begin{cases} q_1, & \text{wenn } f_1(x) \text{ zutrifft} \\ ... & \\ q_n, & \text{wenn } f_n(x) \text{ zutrifft} \end{cases} \tag{3.1}$$

Wobei q einen Zustand aus der Menge Q beschreibt und x die Anzahl an lebendigen Nachbarzellen darstellt. $f(x)$ definiert eine Nachbarschaftsregel und sagt aus, bei welcher Anzahl an Nachbarzellen ein bestimmter Zustand zutrifft und kann mit boolschen Operatoren ausgedrückt werden. Am Beispiel des *Game of Life* Automaten wäre die Bedingung

für den Übergang von einer toten in eine lebendige Zelle, wenn die Zelle genau 3 Nachbarn hat. Eine solche Bedingung kann ausgedrückt werden als $f_{born}(x) : x = 3$. Die zweite Bedingung für das Sterben einer Zelle ist, wenn eine Zelle weniger als 2 oder mehr als 3 Nachbarzellen hat. Die Funktion kann ausgedrückt werden als $f_{die}(x) : (x < 2) \wedge (x > 2)$. Dementsprechend können Bedingungen für einen Zustandswechsel nicht nur einzelne Vergleichsoperatoren haben. Sie können auch verschachtelte Bedingungen beinhalten und damit als Binärbaum ausgedrückt werden.

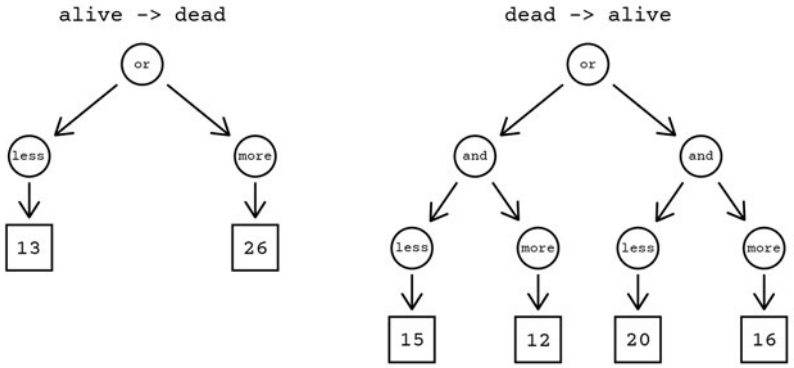


Abbildung 3.2: Eine Zustandsregel eines zellularen Automaten als Binärbaum dargestellt (eigene Darstellung)

Für die lokale Manipulierung eines zellularen Automaten gibt es keine Parameter oder Regelvariationen, welche punktuelle und gezielte Änderungen im Zustandsraum erlauben. Ein zellularer Automat ist ein konsistentes und einheitliches System, in welchem für jede Zelle dieselben Bedingungen gelten. Jedoch ist jede Zelle abhängig von ihren Nachbarzellen und somit kontextabhängig. Diese Eigenschaft kann dafür ausgenutzt werden, eine teilweise partielle Kontrolle zu erhalten. Dies wird erreicht, indem eine Teilmenge der Zellen fixiert wird. Damit ist gemeint, dass fixierte Zellen als Nachbarn gelten und somit andere Zellen beeinflussen können, jedoch ihren eigenen Zustand während eines Generationswechsels nicht ändern. Dementsprechend kann ein gewünschter Zustandsraum vordefiniert werden, indem Teilbereiche fixiert werden und nur außerhalb dessen der zellulare Automat Generationsübergänge durchführt. Die Zellen außerhalb des fixierten Bereiches passen sich den Bereichen nach den Regeln des zellularen Automaten an, da die fixierten Zellen noch zum Kontext zählen.

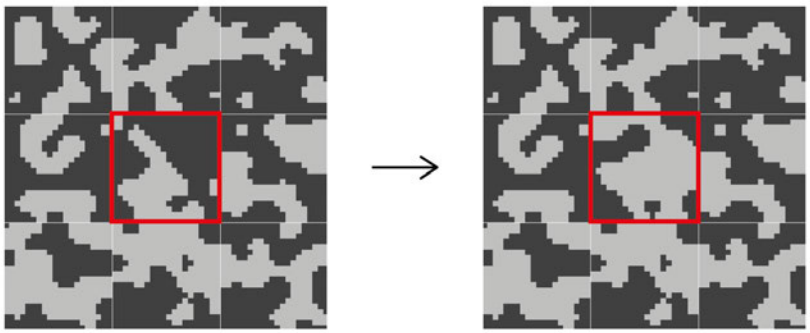


Abbildung 3.3: Lokale Manipulation von zellularen Automaten durch fixieren von Zellen (eigene Darstellung)

Die Umsetzung macht sich dessen zu nutzen, indem ein Zustandsraum R in n gleich große Teilbereiche $T \subset R$ zerlegt wird. Hierbei ist jeder Teilbereich T ein eigenständiger zellularer Automat, welcher jedoch im Kontext zu den direkt anliegenden Nachbarn steht. Jede Zelle $c \in T$, welche am Rand des Teilbereiches T liegen, haben die wiederum am Rand positionierten Zellen des anliegenden Teilbereiches T' als Nachbarn (wenn diese in die Nachbarschaftsdefinition fallen). Hier werden die impliziten Nachbarzellen $c' \in T'$ im Kontext von T als fixierte Zellen behandelt. Sie nehmen also Einfluss auf die am Rand liegenden Zellen von T, jedoch werden diese im Zuge des Generationswechsels nicht beeinflusst. Dadurch stellt der Zustandsraum R ein Netz an einzelnen zellularen Automaten dar, die jedoch gegenseitig in Bezug zueinander stehen und ein konsistentes System erzeugen. Eine lokale Modifizierung kann nun in dem Sinn durchgeführt werden, indem einzelne Teilbereiche angesteuert werden können, sich diese aber im Zuge des Kontextes ändern. Der Kontext selbst bleibt unberührt - zu sehen in Abbildung 3.3 außerhalb des markierten Bereiches.

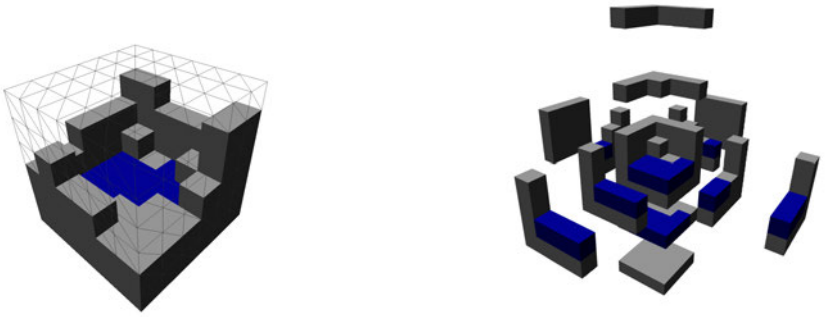
3.3 Kombination mit Wave Function Collapse

Das Ziel der Arbeit ist es zu zeigen, dass zellulare Automaten in Kombination mit anderen Algorithmen der prozeduralen Generierung zu einem vielseitigen Werkzeug werden. Im Folgendem wird die Kombination mit dem Wave Function Collapse erläutert, welche sich im Speziellen auf die Generierung von Höhlensystemen konzentriert.
Der zellulare Automat soll hierbei für die Generierung der eigentlichen Höhlenstruktur

verwendet werden, wo hingegen der Wave Function Collapse für die Generierung von Details zuständig ist. Dies können beliebige Objekte wie zum Beispiel Stalagmiten, Wasser oder Truhen sein. Dabei soll die eigentliche Höhlenstruktur nicht vom WFC manipuliert werden, sondern die Struktur durch die Objekte ergänzen. Die Kombination mit zellularen Automaten soll ebenfalls einen großen Nachteil des Wave Function Collapse ausgleichen. Mit der Komplexität der Modelle wächst auch die Wahrscheinlichkeit für Konflikte im WFC Algorithmus [28]. Die generierte Struktur des Automaten dient hier stets als Absicherung.

3.3.1 Generierung von Prototypen

Der grundsätzliche Algorithmus - welcher im Kapitel 2.5 erläutert wird - bleibt unverändert. Es werden aus dem zugrunde liegenden Eingabemodell iterativ gleich große Muster extrahiert. Die Muster werden ab hier als Prototypen bezeichnet und haben die Dimension des Kernels, welcher die Größe der Prototypen definiert und während der Iteration systematisch wie ein Zeiger durch das Modell geführt wird.



(a) Eingabemodell (b) extrahierter Prototyp

Abbildung 3.4: Beispiel eines extrahierten Prototypen aus einem Eingabemodell (eigene Darstellung)

Falls ein Teil des Kernels über die Modellgrenzen hinaus zeigt, wird die entgegengesetzte Seite des Modells mit einbezogen. Somit wird das Modell künstlich mit sich selbst erweitert und Prototypen erzeugt, die Beziehungen und Muster auch an den Modellgrenzen abbilden [42].

Ein Prototyp hat genau 6 Seiten, welche ab hier als Sockel bezeichnet werden. Es gibt je Prototyp 4 horizontale und 2 vertikale Sockel. Zusätzlich werden von jedem Prototypen um die Y-Achse rotierte Kopien extrahiert, um die Varianz an Prototypen zu erweitern. Im Laufe der Extrahierung kann es vorkommen, dass dieselben Prototypen mehrmals in dem Eingabemodell existieren. Identische Prototypen werden nur genau einmal gespeichert. Es wird aber die Häufigkeit für die spätere Berechnung der Gewichtungen als Frequenz hinterlegt [6].

3.3.2 Definierung der Nachbarschaftsbedingungen

Um die Datenstruktur an die des CA anzupassen, erwartet der WFC als Eingabemodell ein dreidimensionales Gitter an Zellen. Anders als bei den Zellen der Automaten, welche aus Zellen für Höhlentunnel und Höhlenwände bestehen, können die WFC Modelle weitere Zellentypen beinhalten, welche später zu der Höhlenstruktur ergänzt werden sollen.

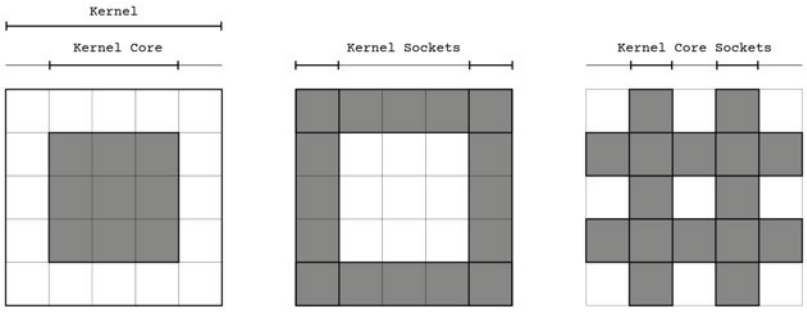


Abbildung 3.5: Darstellung eines Kernels, dem Kernelkern und der Sockel (eigene Darstellung)

Zwei generierte Prototypen gelten als gültige Nachbarn genau dann, wenn alle Zellen beider entgegengesetzter Sockel übereinstimmen. Sei nach der Definition in Kapitel 2.5 p_1 und p_2 zwei Prototypen mit $p_1, p_2 \in P$. Es soll geprüft werden, ob p_2 ein gültiger Nachbar von p_1 in Richtung d ist. Dafür werden ihre zugehörigen Sockel s_1 und s_2 verglichen. s_1 ist der Sockel von p_1 in Richtung d, wobei s_2 der Sockel von p_2 in entgegengesetzter Richtung d' ist. s_1 bezieht sich auf die Teilmenge eines Prototyps, welche die Grenze oder Außenschale in Richtung d beschreibt und in einer Tiefe von $k = 0$ liegen. Für s_2

gibt es zwei mögliche Definitionen. s_2 kann ähnlich wie s_1 die Teilmenge in Richtung d' mit einer Tiefe von k beschreiben. In dem Fall reicht ein Vergleich von s_1 und s_2 für die Bestimmung einer gültigen Nachbarschaft und würde dazu führen, dass die Prototypen ohne Überlappung konsistent zueinander sind. Wenn eine Überlappung der Prototypen gewollt ist, muss s_2 eine Sockeltiefe von $k + 1$ haben. Nach dieser Definition hat jeder Prototyp zwei Sockelmengen für jede Richtung $d \in D$. Ein Mal die äußeren Sockel in einer Modelltiefe von k und ein Mal die inneren Sockel in einer Tiefe von $k + 1$. p_1 und p_2 gelten dann nur als gültige Nachbarn, wenn der äußere Sockel von p_1 mit dem inneren Sockel von p_2 und umgedreht übereinstimmt [42] - dargestellt in Abbildung 3.6.

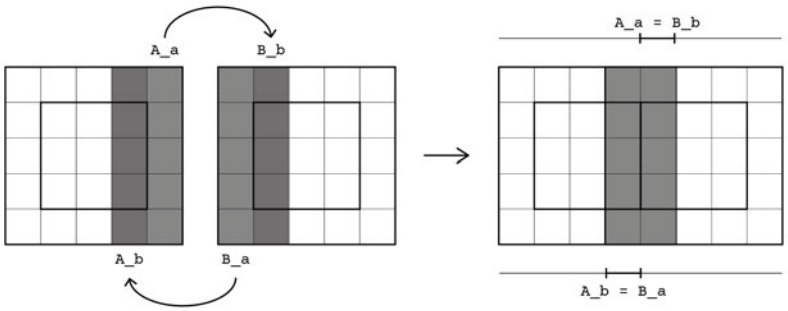


Abbildung 3.6: Adjacency Contraints zweier überlappender Prototypen (Darstellung in Anlehnung [42])

3.3.3 Vorverarbeitung im Kontext des CA

Der grundsätzliche Algorithmus bleibt in Kombination mit zellularen Automaten derselbe. Es wird eine Menge P_α an Prototypen aus einem Eingabemodell α wie zuvor beschrieben extrahiert. Zusätzlich dazu wird mit demselben Verfahren eine Menge P_β mit dem vom zellularen Automaten generierte Modell β erzeugt. Beide Mengen werden anschließend zu einer gemeinsamen Menge P vereint, sodass $P_\alpha, P_\beta \subset P$ gilt. Zwischen den Prototypen werden anschließend valide Nachbarschaften markiert. Wenn die kombinierte Menge P an Prototypen nun mittels WFC kollabiert wird, würden auf der einen Seite beide Modelle teilweise in dem Ergebnis vorkommen und konsistent zueinander sein. Die ursprüngliche Struktur vom CA würde so jedoch verändert werden.

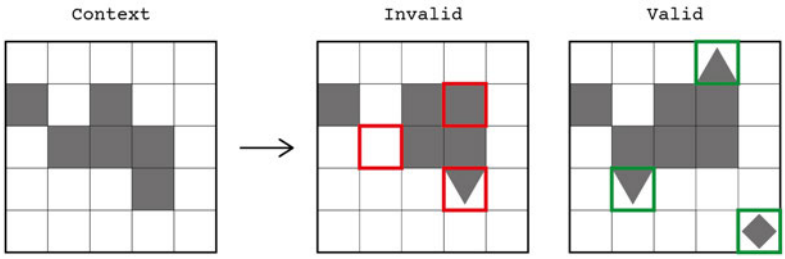


Abbildung 3.7: Darstellung der kontextabhängigen Gültigkeit von Zellen eines Prototyps (eigene Darstellung)

Um die vom zellularen Automaten generierte Struktur β während des Wave Function Collapse beizubehalten, werden alle Zustandsposition der Welle W in einem Vorverarbeitungsschritt initial im Kontext von β kollabiert. Das später kollabierte Modell von W wird dieselben Größenverhältnisse haben wie β. Im behandelten Fallbeispiel der Höhlengenerierung, gibt es zwei mögliche Zustände aller Zellen in β. Eine Zelle $c \in \alpha$ kann eine Höhlenwand oder einen Tunnel beschreiben, wobei eine Höhlenzelle im folgenden als q_h und eine Tunnelzelle als q_t bezeichnet wird. Beide Zustände gehören zu der Zustandsmenge Q_β, wobei dies eine Teilmenge der Menge aller Zustände Q ist. Alle anderen möglichen Zellenzustände die durch α dazukommen, werden definiert mit $Q_\alpha = Q \setminus Q_\beta$. Ein Prototyp einer Position $p \in W$ besteht ebenfalls aus einem Gitter an Zellen mit der Dimensionalität des Kernels aus dem Extrahierungsschritt. Dementsprechend müssen alle Zellen C_p einer Position p die Zellen C_β in derselben Position von der vom zellularen Automaten generierten Struktur β repräsentieren. Um die Struktur zu erhalten, werden bestimmte Bedingungen für jeden möglichen Prototyp in Abhängigkeit zur Position definiert. Für jede Zelle $c_p \in C_p$ und der relativ zugehörigen Zelle $c_\beta \in C_\beta$ muss gelten:

- wenn $c_\beta = q_h$, dann muss gelten: $c_p = q_h$
- wenn $c_\beta = q_t$, dann muss gelten: $c_p \neq q_h$

Falls eines dieser Bedingungen für einen Prototyp P in einer Position p nicht zutrifft, dann wird dieser aus der Zustandsmenge Z_p entfernt - die Entropie sinkt. Dies wird mit jeder Position in W wiederholt und somit sichergestellt, dass alle im Kontext des zellularen Automaten ungültigen Prototypen aus allen Superpositionen rausgefiltert sind. Dies kann dazu führen, dass kein Prototyp aus P_α übrig bleibt, da alle für diese Position ungültig sind. Jedoch ist garantiert, dass für jede Position mindestens ein Prototyp gültig

bleibt, da jeder Prototyp aus P_β alle Zellenbedingungen erfüllt. Schließlich sind es Zellenmuster, welche direkt aus β extrahiert sind. Ein Konflikt im Laufe des anschließenden Wave Function Collapse ist somit ausgeschlossen. Die Änderungen werden im Zuge des Vorverarbeitungsschritt auf alle anderen Positionen propagiert. Das Resultat behält alle Zellen bei, welche als Höhlenzellen gelten und den ergänzten Zellen aus Q_α.

4 Implementierung

Das folgende Kapitel erläutert die praktische Implementierung der Konzepte im Rahmen der in Sektion 3.1 definierten Anforderungen. Beschrieben werden genutzte Technologien, die grundlegende Architektur und Muster, als auch die genaue Umsetzung der Software.

4.1 Verwendete Technologien

Das Projekt ist auf Basis von Java 15 geschrieben. Die grafische Benutzerschnittstelle ist mit Swing umgesetzt, welches eine Sammlung an Bibliotheken für die Umsetzung einer GUI bereitstellt und Teil der Java Runtime ist. Für das Rendern von 3D-Grafiken wird *jMonkey* 3.4 verwendet, welche eine für Java optimierte 3D-Engine ist. *jMonkey* ist eine Abstraktionsschnittstelle zu OpenGL und basiert auf Szenengraphen[1].
Das Softwareprojekt ist Teil eines übergeordneten Projekts von Prof. Dr. Philipp Jenke, welches mithilfe von Studenten stetig weiter entwickelt wird. Es stellt Implementierungen und Werkzeuge für den Themenbereich der Computergrafik bereit, welche von Datenstrukturen und Algorithmen bis hin zur prozeduralen Generierung reichen. Die in dem Kapitel behandelte Umsetzung nutzt hiervon einige der grundsätzlichen Abstraktionen, welche vor allem für die dreidimensionalen Ansichten genutzt werden.

[1] Offizielle *jMonkey* Dokumentation, https://wiki.jmonkeyengine.org/docs/3.4/documentation.html (Zugriffsdatum: 2022-02-05)

4.2 Software Architektur

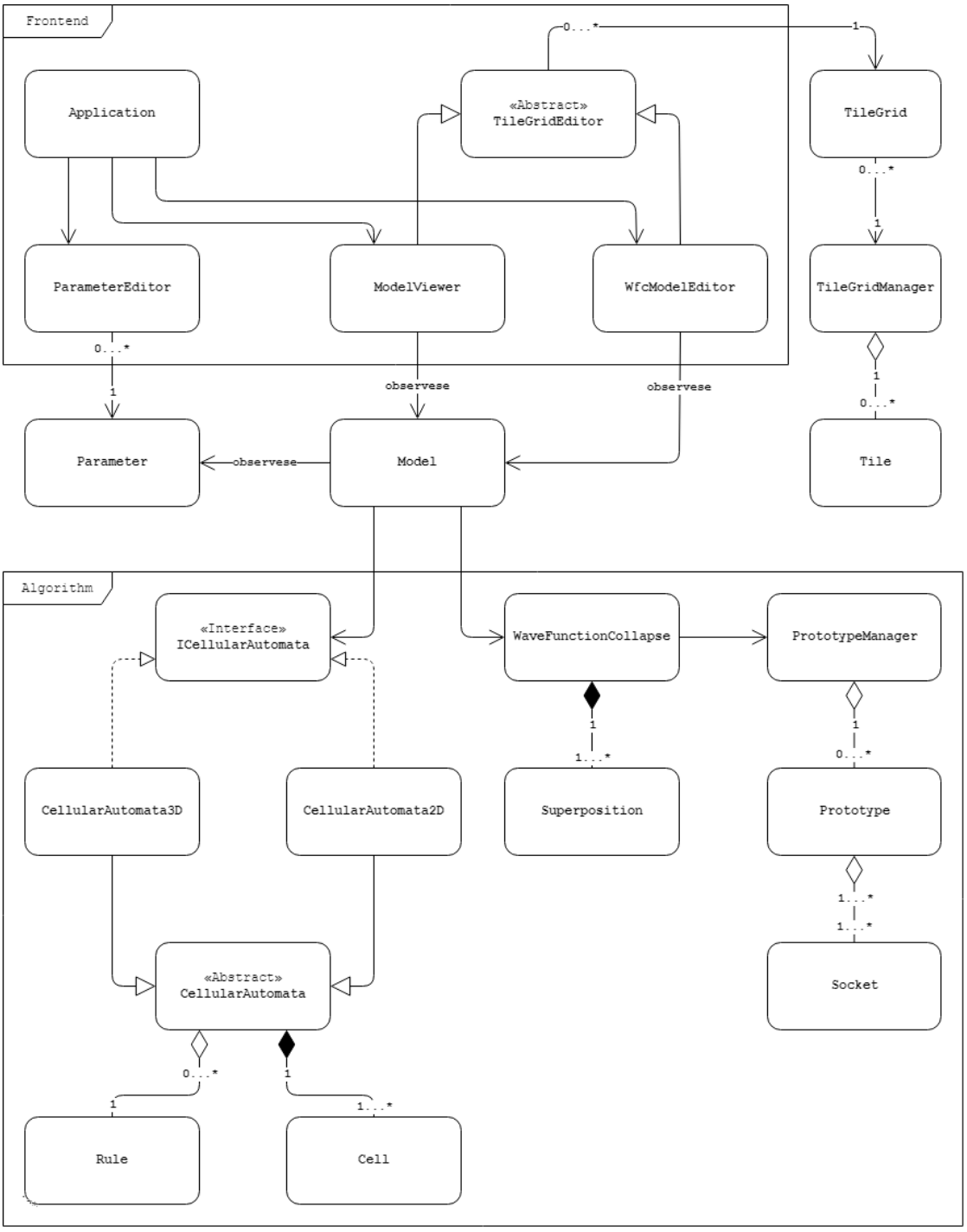


Abbildung 4.1: Architktur und Aufbau der Software (eigene Darstellung)

4.2.1 Komponenten

Für die Umsetzung des Projekts sind alle Komponenten zum einen der Benutzerschnittstelle zugeordnet, welche die Visualisierung und Steuerung übernimmt und zum anderen zu den im Hintergrund laufenden Logikprozessen und Datenstrukturen. Letztere stellen die Kernkomponenten dar, welche wiederum folgendermaßen aufgeteilt sind:

- Parameter
- Modelle
- Verarbeitungen

Parameterkomponenten definieren voreingestellte Werte, legen Bedingungen und Vorgaben fest und beinhaltet keine Logik. Hierunter fallen alle einstellbaren Werte, welche für die prozedurale Generierung relevant sind. Ein Modell hingegen ist eine Datenstruktur, welche eine Datenform und einen Kontext auf Basis der Parameter beschreibt, wie zum Beispiel eine Matrix von Zellen. Das Modell kann anschließend durch Prozesse verarbeitet und manipuliert werden. Diese Verarbeitungen können Algorithmen der prozeduralen Generierung darstellen.
Die visualisierenden Komponenten stehen in Beziehung zu den verarbeiteten Modellen. Diese können die Modelle steuern und die Parameter anpassen. Dabei ist zwischen den einzelnen Komponenten eine schwache Kopplung dadurch gewährleistet, dass jede Komponente für sich steht und eine Abstraktion durch klare Schnittstellen nutzt. Die verringerte Kopplung sollte jedoch nicht die Kohäsion der Komponenten zu stark wachsen lassen.

4.2.2 Verwendete Entwurfsmuster

Ein Softwareprojekt umzusetzen erfordert je nach Anforderung den Einsatz etablierter struktureller Entscheidungen. Hierfür verwendet die Implementierung verschiedene Entwurfsmuster, welche die Architektur und das Verhalten der Software vereinheitlichen, sowie die Codequalität steigern [15].
Darunter fällt die in der vorherigen Sektion beschriebenen Beziehungen der Komponenten. Sie sind dem Architekturmuster des Model-View-Controller (MVC) [23] nachempfunden, wobei das Model die Modell- und Parameterkomponenten darstellt, der Controller die Verarbeitungs- und Steuerungskomponenten und die View die visualisierenden

Komponenten. Im MVC steht das Datenmodell im Mittelpunkt, auf welches die anderen beiden Teile Zugriff haben. Das Datenmodell hingegen kennt diese nicht und kann somit leicht ausgetauscht werden. Es entsteht eine lose Kopplung.
Aktionen und Ereignisse in den Teilbereichen haben indirekten oder direkten Einfluss auf die anderen Bereiche. Die steuernden Komponenten haben oft direkten Einfluss auf das Modell. Dieses hingegen wird verändert, welches indirekten Einfluss auf die visualisierende Komponente hat. Dieser indirekte Einfluss ist durch das Beobachter-Verhaltensmuster umgesetzt [1]. Hier werden ereignisbasierte Abhängigkeiten zwischen Komponenten definiert, bei welcher eine Komponente auf eine beobachtbare Komponente reagiert. Änderungen an der beobachteten Komponente werden an den Beobachter weitergegeben wird, ohne dass die beobachtete Komponente diese kennen muss. In der Implementierung beobachten die visualisierenden Komponenten die Modelle, wobei die Modelle von den Parametern abhängig sind und Änderungen registrieren. Somit entsteht eine hierarchische Struktur an Komponenten, welche sich gegenseitig indirekt beeinflussen können, jedoch nur an die Komponenten einer Ebene unter sich gekoppelt ist.
Ein weiteres verwendetes Architekturmuster ist das Pipe-Filter-Muster [10]. Dieses wird vor allem in der Implementierung der verarbeitenden Komponenten genutzt, indem das Datenmodell sequenziell durch mehrere transformierende Teilschritte geführt wird. Dadurch können verschiedene prozedurale Generierungsverfahren auf dasselbe Datenmodell dynamisch und nach Bedarf angewendet werden. Dies erfordert jedoch, dass alle angewendeten Transformatoren mit dem Datenmodell umgehen können und keine Vorbedingung von anderen Transformatoren erwartet werden. Alle verwendeten Filter müssen also für sich stehen, was wiederum die Kopplung der Komponenten weiter reduziert.

4.2.3 Ablauf

Der grobe Programmablauf beginnt mit der Initialisierung des Modells auf Basis der definierten Parameter. Dazu gehört die Größe der Zellenstruktur, der Seed und alle Voreinstellungen der genutzten PCG Algorithmen. Anschließend wird das Modell durch die einzelnen Verarbeitungsschritte transformiert. In der Implementierung wäre das der zellulare Automat und der Wave Function Collapse. Die resultierende Zellenstruktur wird daraufhin in ein Datenmodell umgewandelt, mit welcher die visualisierende Komponente umgehen kann.

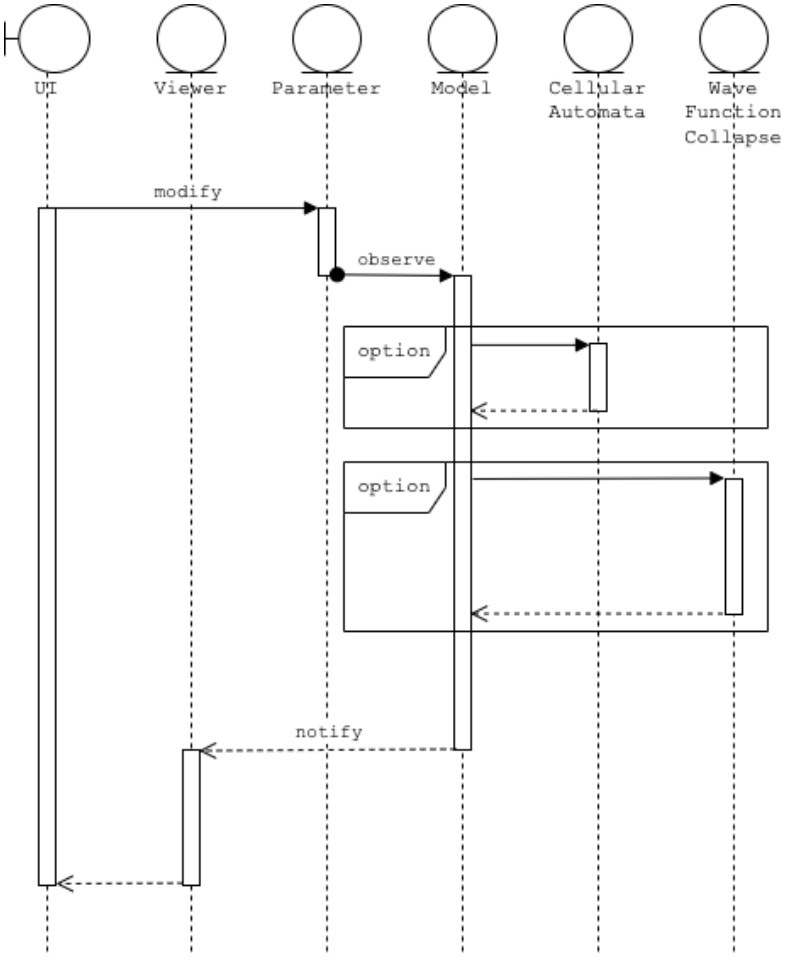


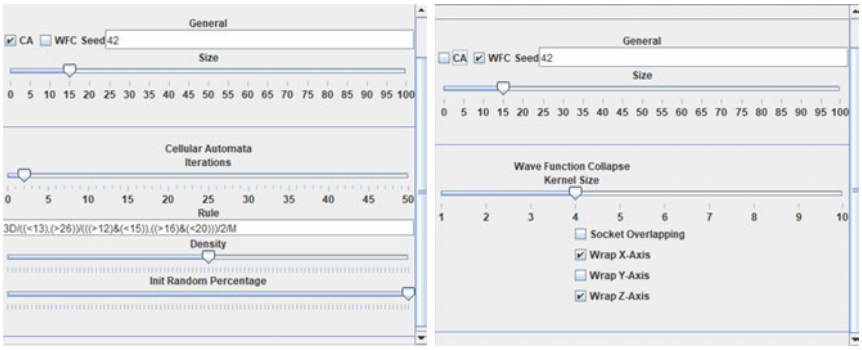
Abbildung 4.2: Grober Sequenzablauf der Software (eigene Darstellung)

4.3 Umsetzung des Editors

Die Benutzerschnittstelle bietet verschiedene Ansichten, um die Modelle zu visualisieren und zu manipulieren. Hierfür wird im Folgenden nur die oberflächliche Implementierung erläutert. Insgesamt werden drei Ansichten zur Verfügung gestellt:

- Parametereinstellungen
- Modellbetrachtung
- WFC Modellierung

In den Parametereinstellungen können globale Anpassungen an der Generierung vorgenommen werden. Darunter fällt die Größe des Zellengitters, ein vordefinierter Seed und die Auswahl der genutzten Algorithmen. Eine der Anforderungen ist es, dass alle Generierungsverfahren auch für sich alleine und in Kombination funktionieren sollen. Somit ist es möglich, die Auswahl dynamisch anzupassen. Alle relevanten Parameter für die einzelnen Algorithmen können hier ebenfalls eingestellt werden, wenn diese in die Generierung mit einbezogen sind (Abb. 4.3).

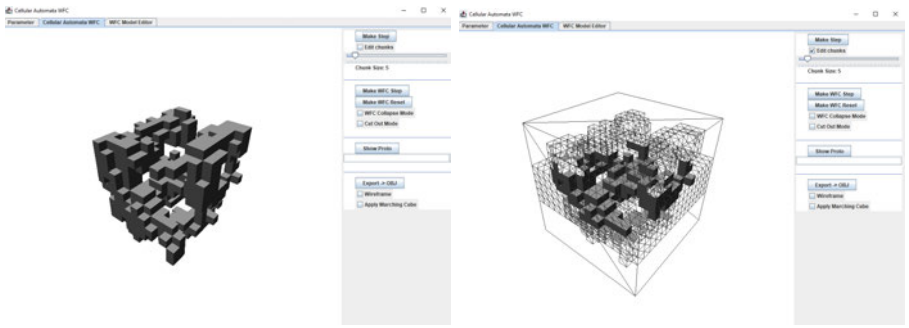


(a) CA Parameter (b) WFC Parameter

Abbildung 4.3: Übersicht der Parametereinstellungen (eigene Darstellung)

Die beiden Modellansichten sind in Ihrer Umsetzung sehr ähnlich, nutzen jedoch verschiedene Modelle als Basis, welche unterschiedliche Funktionalitäten voraussetzt. Sie erben von einer übergeordneten Klasse *TileGridEditor*, welche grundsätzliche Funktionen für das Bearbeiten eines 3D-Modells bereitstellt und verwaltet eine Matrix an Zellen. *TileGridEditor* übernimmt die gesamte Darstellung und Übersetzung der Zellen in Polygone, welche anschließend dargestellt werden können. Bei der Darstellung können beispielsweise

bestimmte Bereiche einer Achse des Zellengitters hervorgehoben werden, um die Handhabung zu vereinfachen. In der Implementierung werden hervorgehobene Bereiche als *scope* bezeichnet (Abb. 4.4b). Zu dem bietet es Möglichkeiten, einzelne Zellen und Bereiche in Abhängigkeit vom *scope* direkt mit dem Mauszeiger anzusprechen. Dies erlaubt weitere Möglichkeiten für die Bearbeitung und Manipulierung der Modelle. Die Übersetzung der Zellen in Polygone kann in der Umsetzung durch zwei verschiedene Implementierungen durchgeführt werden. Zum einen eine einfache Übersetzung, bei welcher jede Zelle durch den zum Zellenzustand passenden Modell ersetzt wird. Dies wird in der Implementierung durch eine *Tile* umgesetzt, welches ein Objekt mit einem Zellenzustand und einer Dreiecksgeometrie ist. Jede *Tile* hat dabei eine feste Größe in Form eines Kastens und wird initial durch einen *TileManager* geladen. Die möglichen *Tile* Objekte können durch das Ergänzen weiterer Dreiecksgeometrien erweitert werden. Die Übersetzung der Zellenwerte kann zum anderen auch durch den *Marching Cube* Algorithmus umgesetzt werden, welcher genauer im Anhang B beschrieben wird. Diese Übersetzung kann jedoch nur mit den binären Zustandswerten des zellularen Automaten umgehen.



(a) reguläre Ansicht (b) Ansicht mit Scope

Abbildung 4.4: Modellansichten (eigene Darstellung)

Innerhalb der Modellbetrachtung wird mithilfe des *TileGridEditor* das generierte Zellenmodell dargestellt. Die generierte Struktur des zellularen Automaten kann hier mithilfe der Maus an Teilbereichen manipuliert werden. Der WFC Modellierungsbereich hingegen bietet ein frei definierbares Zellengitter, welches durch den Mauszeiger mit *Tile* Objekten gefüllt werden kann. Die modellierten Strukturen, welche als Eingabe für den Wave Function Collapse dienen, können gespeichert werden. Hierfür wird die Matrix mit allen Zellenwerten und die dazugehörigen *Tile* Objekte als Schlüsselwertpaare in der Datei hinterlegt.

4.4 Aufbau des zellularen Automaten

Für die Umsetzung des zellularen Automaten nach dem in Sektion 3.2 beschriebenen Konzepts, wird eine abstrakte Klasse definiert, welche die grundsätzlichen Funktionen eines CA bereitstellt und mit bestimmten Eingabeparametern umgehen kann. Die Implementationen des zwei- und dreidimensionalen zellularen Automaten erben von der abstrakten Klasse. Hier liegt der einzige Unterschied beider Klassen in dem Zellennetz, welches beim 2D-Automaten eine zweidimensionale Matrix und beim 3D-Automaten dementsprechend eine dreidimensionale Matrix verwaltet. Die Matrix ist ein verschachteltes Array von Zellenobjekten. Grundsätzlich enthält ein Zellenobjekt ein Zustand als Ganzzahl. Dabei ist ein lebender Zustand als 1 und ein toter Zustand mit einer 0 defniert. Zusätzlich kann eine Zelle s Übergangszustände haben, welche die Anzahl der Übergangsgenerationen einer sterbenden Zelle definieren [46]. Dies wird umgesetzt, indem ein lebender Zustand nun als die Gesamtanzahl an Übergangszustände $s-1$ beschrieben wird. Wenn eine Zelle $s = 5$ Übergangszustände besitzt und in einer Generation t stirbt, dann wird der Zellenzustand s Generation dekrementiert, bis es in Generation $t + s$ auf den Zustand 0 fällt und somit tot ist. Während der Übergangsgenerationen gilt die Zelle noch als lebend. Der Zustand kann in dieser Zeit jedoch nicht verändert werden. Diese Eigenschaft der Zellen führt zu einer stark erhöhten Vielfalt des zellularen Automaten.

4.4.1 Globale Generierung

Für die globale Generierung des zellularen Automaten, welche das Verhalten der Zellen und damit die grundsätzliche Struktur festlegt, werden bestimmte Parameter an der Schnittstelle bereitgestellt:

- die Dimensionalität d des Automaten, wobei 2D und 3D unterstützt wird
- eine ganzzahlige Größe n, welche die Anzahl der Zellen des Automaten mit n^d festlegt
- einen Seed als 64 bit Ganzzahl, auf welchen der pseudozufällige Zahlengenerator von Java für die Initialisierung der Zellen zurückgreift

- einen ganzzahligen Wert, welcher die Dichte an lebenden Zellen während der Initialisierung der Zellenmatrix festlegt, wobei bei einem Wert von 100 alle Zellen initial leben und bei einem Wert von 0 alle tot sind
- die Regeldefinition, welche unter anderem die Nachbarschaft und die Nachbarschaftbedingungen definiert

Der Parameter für die Regeln besteht aus einer Zeichenkette, in welchem die einzelnen Teile der Regeldefinition durch ein Trennsymbole abgeschnitten sind. Eine Regel r hat die Form [46]:

$$r := \{D, \delta_0, \delta_1, s, N\} \tag{4.1}$$

Wobei wie oben beschrieben D die Dimensionalität angibt, s die Anzahl der Zellenzustände und N die in Kapitel 2.4 beschriebenen Nachbarschaften. In Kapitel 3.2 wird das Konzept einer Nachbarschaftregel δ erläutert. Im Falle der implementierten Regeldefinition gibt es eine Regel δ_0, welche die Bedingung für das Sterben einer Zelle angibt und eine Regel δ_1, die eine Bedingung für die Geburt einer Zelle definiert. Dabei kann eine Bedingung ein Vergleich mit der Nachbarschaftanzahl sein oder weitere Bedingungen beinhalten und ist umschlossen von Klammern. Eine Vergleichsbedingung unterstützt die Operatorsymbole $=, <, >$ und hat die Form (<Operator><Vergleichszahl>). Mehrere Bedingungen müssen durch die Operatorsymbole , & logisch verknüpft werden und haben die Form (<Bedingung> <Operator> ... <Bedingung>). Ein Beispiel für eine Regeldefinition nach dieser Form sähe für einen *Game of Life* ähnlichen Automaten wie folgt aus: 2D/((<2),(>3))/(=3)/2/M.
Es wird eine statische Methode bereitgestellt, welche eine Zeichenkette dieser Form validieren und einlesen kann, um es anschließend in ein Regelobjekt umzuwandeln. Hierfür wird die Zeichenkette durch das Trennsymbol aufgeteilt und verglichen, ob alle Teile die erwartete Form aufweisen. Das Einlesen der Zustandsbedingungen wird über ein Stapel gelöst, welche alle enthalten Zeichen somit vom letzten zum ersten Zeichen abarbeitet. Falls das Zeichen an oberster Stelle des Stapels eine schließende Klammer ist, wird ein Bedingungsobjekt initialisiert. Der Stapel wird solange abgearbeitet, bis eine öffnende Klammer eingelesen wird und somit das Ende der Bedingung markiert. Falls innerhalb der Klammern weitere Klammern vorkommen, wird die Bedingung durch weitere Bedingungen verschachtelt. Alle eingelesenen Zeichen zwischen den Klammern werden durch die erwarteten Werte in die Datenstruktur des Bedingungsobjektes gespeichert. Dieses enthält einen der beschriebenen Operatoren mit einem zugehörigen Vergleichswert oder einer Liste an Bedingungen. Während des Einlesens wird ein Zähler bei einer schließen-

den Klammer inkrementiert und bei Öffnenden dekrementiert. Wenn dieser Zähler nach dem Prozess auf 0 steht, liegt eine korrekte Klammerung vor. Falls dies nicht der Fall ist, ein unerwartetes Symbol vorkommt oder ein erwartetes Symbol fehlt, gilt die Bedingung als ungültig und ein Fehler wird geworfen. Die aufgebaute Datenstruktur für eine Regel entspricht wie in Abbildung 3.2 einem Binärbaum, in dem eingeklammerte Zeichen als Knoten behandelt werden und die konkreten Zahlen als Blätter.
Im konkreten Beispiel von Höhlen resultieren die generierten Strukturen in stark verzweigte Tunnelsysteme. Mehrere Tunnelsysteme müssen jedoch nicht zusammenhängend sein, wobei sich viele kleinere unabhängige Strukturen bilden können oder einzelne Zellen losgelöst vom Rest vorhanden sind. Dies kann zu einem unruhigen Rauschen innerhalb der Struktur führen. Um dem entgegenzuwirken, wird ein Nachverarbeitungsschritt angewendet, welcher die größte zusammenhängende Höhlenstruktur extrahiert und durch den *Flood Fill* Algorithmus umgesetzt ist (siehe Anhang A).

4.4.2 Lokale Generierung

Für eine lokale Manipulation eines zellularen Automaten wird das Konzept aus Sektion 3.2 implementiert, indem das Zellenobjekt eine weitere Objektvariable erhält. Ein boolscher Wert kann je Zelle gesetzt werden, welcher eine Fixierung innerhalb des Zellennetztes definiert. Wenn dieser gesetzt ist, zählt die Zelle weiterhin in anliegende Nachbarschaften hinein. Jedoch kann der eigene Zellenzustand während eines Generationswechsels nicht verändert werden.
Für die Anwendung wird eine Methode von den zellularen Automaten bereitgestellt, welche eine Matrix an Zustandswerten je Zelle als Argument annimmt. Die entsprechenden Zellen, welche mit einem positiven Zustandswert markiert sind, werden mit dem angegeben Wert fixiert. Bei einem negativen Wert kann der Zustand eine Zelle weiterhin verändert werden. Somit dient die übergebene Matrix als eine Maske von fixierten Zellen.

4.5 Einbindung von Wave Function Collapse

Der Wave Function Collapse wird nach dem Konzept aus Sektion 3.3 implementiert und besteht aus zwei Teilprozessen. Zum einen die Verarbeitung von zellenbasierten Modellen, zum anderen das Kollabieren der darauf basierenden Superpositionen.

4.5.1 Modellverarbeitung

Ziel der Modellverarbeitung ist es, eine Menge an Mustern aus dem Eingabemodell zu extrahieren. Die Muster werden in einem Prototypenobjekt gespeichert, welche zusätzlich für jede Seite die Sockel und eine Liste an gültigen Nachbarprototypen hinterlegt haben. Die gesamte Umsetzung für diesen Schritt ist in einer Verwaltungsklasse *PrototypeManager* gekappselt, welche eine Liste aller Prototypen und aller Sockel führt. In ihr ist ebenso die Form des Kernels hinterlegt, welche die Größe der Prototypen vorgibt. Dabei wird unterschieden zwischen einem Kernel der überlappend ist oder nicht [42]. Bei einem überlappenden Kernel werden pro Seite des Prototypen zwei verschiedene Sockel definiert. Zum einen den äußeren Sockel, welcher alle äußersten Zellen einer Seite beschreibt. Und zum anderen die Kernsockel, welche direkt in einer Zellentiefe unter dem entsprechenden äußeren Sockel liegen. Bei einer Kernelgröße von 5 hätte der gesamte Kernel eine Zellengröße von 5^3, der Kernelkern jedoch nur eine Größe von 3^3. Wenn der Kernel nicht überlappend ist, dann hätten beide die selbe Größe. Die Zellen im Kernelkern sind später die Zellen, welche nach dem Kollabieren in der Ausgabe übrig bleiben. Das genauere Konzept hinter der überlappenden Nachbarschaftsbedingung wird beschrieben in Sektion 3.3.
Während der Extraktion der Prototypen im *PrototypeManager* wird durch das Eingabemodell iteriert. Die Iterationsintervalle innerhalb des Zellennetzes kann in Schritten von einer Zelle durchgeführt werden, um eine möglichst große Menge an möglichen Prototypen zu extrahieren oder in Schritten, welche der Größe des Kernelkerns entsprechen. Dies kann sinnvoll sein, wenn nur die nötigsten Prototypen gebraucht werden. Es wird vorerst geprüft, ob von der vom Kernel extrahierten Zellenmenge bereits ein Prototyp vorhanden ist. Zwei Prototypen gelten als gleich genau dann, wenn jede Zelle an der selben Stelle identisch ist. Falls ein Prototyp mehrmals auftaucht, wird ein Zähler inkrementiert, welcher die Frequenz innerhalb des Eingabemodells darstellt. Um nicht immer jede Zelle mit jeder Zelle aller anderen Prototypen vergleichen zu müssen und um in späteren Schritten effizienter Prototypen zu speichern, wird jeder erzeugte Prototyp indexiert. Der Index ist in dem Fall ein Hashwert, der als Zeichenkette hinterlegt ist und abhängig von allen Kernelzellen des Prototyps generiert wird.
Die Hashfunktion ist eine sehr simple Implementierung[2], welche als Argument die numerischen Zustandswerte der Zellenmenge als Zeichenkette nimmt und über jedes Zeichen iteriert. Für den verwendeten Zweck ist sie ausreichend und hat keinen Anspruch auf

[2] Implementierung stammt von einer Antwort der Frage *Good Hash Function for Strings* auf Stackoverflow vom User *Mifeet*: https://stackoverflow.com/a/2624210 (Zugriffsdatum: 2022-02-03)

Algorithm 1 Hashfunktion für Zeichenketten

```
function HASH(X, α, p)
    hash ← α
    for all x ∈ X do
        hash ← hash × p + x
    end for
    return hash
end function
```

kryptografische Vollständigkeit. Der Hash wird berechnet aus einer Zeichenkette X, einer Primzahl α und einem Faktor p (Algorithmus 1). Mithilfe des Hashwertes der Prototypen können diese somit sehr schnell verglichen und effizient im *PrototypeManager* gespeichert werden. Für jeden Prototypen werden bei Bedarf zusätzlich rotierte Kopien um die Y-Achse angelegt. Kopien aus der Rotation der anderen Achsen werden nicht erzeugt, um die Ausrichtung der Zellen im Kontext einer Höhle zu erhalten.
Wie bereits beschrieben werden für jeden Prototypen und für jede der 6 Seiten die Sockel des Kernels und des Kernelkerns definiert. Ein Sockelobjekt besitzt hierbei zum einen eine Ausrichtung in die Vertikale, welches die Seiten auf der Y-Achse beschreibt. Zum anderen eine Ausrichtung der Horizontalen, also die Seiten auf der X- und Z-Achse. Während der Erzeugung der Sockel wird ähnlich wie bei den Prototypen versucht, transformierte Kopien der Sockel anzulegen. Dabei werden vertikale Sockel ähnlich den Prototypen um 90° rotiert. Horizontale Sockel werden nur an der vertikalen Achse gespiegelt. Alle Prototypen werden ebenso mit der beschriebenen Hashfunktion indexiert und gesammelt im *PrototypeManager* gespeichert. Jeder Sockel erhält zusätzlich weitere Kennzeichen, die an dem Indexwert hinterlegt werden. Dies kann der Rotationswert sein, ob der Sockel eine gespiegelte Kopie eines Sockels ist, symmetrisch ist oder in welcher Ausrichtung sich der Sockel befindet[3]. Eine solche Verschlagwortung erspart bei der späteren Berechnung der zueinander passenden Sockel weitere Komplexität. Zum Beispiel kann es während der Generierung durchaus vorkommen, dass genau derselbe Sockel bereits durch einen anderen Prototypen erzeugt ist. In dem Fall erhält der Prototyp die Referenz des bereits erzeugten Sockels. Weitere Transformationen werden somit überflüssig. Die Gleichheit wird einfach durch den Vergleich der Indizes gelöst, welche wie beschrieben durch die relevanten Eigenschaften des Sockels erzeugt werden.
Wenn alle Prototypen mit ihren entsprechenden Sockeln generiert sind, werden für jede Seite eines Prototyps eine Menge an passenden Nachbarprototypen angelegt. Jeder Sockel

[3]Der Ansatz für die Verschlagwortung von Sockeln stammt aus einem Videoblog von Martin Donald: https://www.youtube.com/watch?v=2SuvO4Gi7uY (Zugriffsdatum: 2022-01-12), 06:32-06:48

einer Seite des Prototyps wird hierfür mit jedem Sockel der entgegengesetzen Seite der anderen Prototypen verglichen. Falls die Sockelpaare passen, wird der Prototyp für diese Seite der gültigen Nachbarschaftsmenge hinzugefügt. Hierfür werden von den beiden entgegengesetzten Seiten die Sockelpaare verglichen. Das Sockelpaar besteht immer aus dem äußeren Sockel des einen Prototyps und dem Kernsockel des anderen. Zwei Sockel gelten als passend genau dann, wenn ihr Index derselbe ist. Zwei Prototypen gelten wiederum als passend, wenn beide beschriebenen Sockelpaare passen. Wenn alle Prototypen mit allen Prototypen - auch sich selbst - verglichen sind, besitzt jeder Prototyp für jede Seite eine Menge an Indizes der potentiell gültigen Nachbarn. Der Teilprozess ist somit abgeschlossen. Die indexierten Mengen im *PrototypeManager* können zu einem späteren Zeitpunkt dynamisch durch weitere Eingabemodelle erweitert werden.

4.5.2 Kollabierung der Superpositionen

Der eigentliche Kernprozess des Wave Function Collapse liegt in dem Kollabieren der Superpositionen. Hierfür wird eine Matrix an Superpositionen initialisiert, welche namensgebend für den Algorithmus als *wave* deklariert ist. Ein Superpositionsobjekt verwaltet eine Zustandsliste an boolschen Werten, in welchem jeder Wert einen zuvor generierten Prototypen im *PrototypeManager* repräsentiert. Wenn der entsprechende Prototyp für eine Superposition noch nicht ausgeschlossen ist, wird der Zustandswert für die Position auf wahr gesetzt. Zu Beginn werden alle Zustandswerte einer Superposition als möglich initialisiert, womit die Position als undefiniert gilt. Die jeweilige Entropie wird durch die hinterlegten Frequenzen der verbleibenden Prototypen und der Funktion 2.1 berechnet. Die grundsätzliche Implementierung des Algorithmus versucht so viele Iterationen über die *wave* durchzuführen, bis sie entweder komplett kollabiert ist oder durch ein definiertes Iterationslimit vorher terminiert wird. Dabei besteht eine Iteration aus der Identifizierung der Position mit der niedrigsten Entropie, dem Kollabieren dieser Position und dem Propagieren der Änderungen. Für den Fall eines Konflikts, welcher durch eine Entropie von 0 einer Position definiert ist, wird zu Beginn jeder Iteration eine temporäre Kopie der aktuellen *wave* erzeugt. Bei einem Konfliktszenario können somit alle propagierten Änderungen durch ein vereinfachtes *Backtracking* rückgängig gemacht werden. Die im Algorithmus 2 genutzte Funktion *minEntropy* ist für die Identifizierung der Position zuständig, welche die niedrigste Entropie besitzt und wählt diese für die aktuelle Position aus. Die Funktion *collapse* kollabiert die gewählte Position zu einem der Prototypenzu-

Algorithm 2 Wave Function Collapse

```
function WFC(wave, max)
    i ← 0
    while wave not collapsed and i < max do
        i ← i + 1
        temp ← wave
        position ← minEntropy(wave)
        collapse(position)
        conflict ← propagate(position)
        if conflict then
            wave ← temp
        end if
    end while
    return wave
end function
```

stände. Hierfür wird von den möglichen Zuständen einer Position der Zustand mit der höchsten Wahrscheinlichkeit gewählt (Formel 2.2). Dabei ist die Wahrscheinlichkeit durch die relative Häufigkeit des Prototypen innerhalb des Eingabemodells ausgedrückt. Falls mehrere Zustände dieselbe Wahrscheinlichkeit haben, wird ein pseudozufälliger Zustand aus diesen gewählt.

Der Kern einer Iteration liegt in der *propagate* Funktion (Algorithmus 3). Die Funktion versucht die Änderungen der zuvor kollabierten Position in der *wave* zu verarbeiten. Da die Zustände aller Position in einer Nachbarschaftbeziehung zueinanderstehen, wird Schritt für Schritt evaluiert, welche Nachbarschaften nach dem Kollabieren noch gültig sind. Um dies umzusetzen, wird ein Stapel für die noch zu propagierenden Positionen mit der kollabierten Position initialisiert. Solange der Stapel eine Position beinhaltet, gibt es noch Änderungen die propagiert werden müssen. Die oberste Position wird vom Stapel entfernt und geprüft, ob die Entropie 0 ist. Falls ja, gibt es einen Konflikt und der Prozess wird terminiert. Falls es keinen Konflikt gibt, wird eine Liste aller noch möglichen Zustände für diese Position und jeweils für die aller Nachbarpositionen initialisiert. Hierfür werden die gültigen Nachbarn der entsprechenden Richtung aller möglichen Zustände der Position zu der Liste hinzugefügt. Anschließend wird nur noch verglichen, ob die möglichen Zustände der Nachbarpositionen in der Liste der möglichen Nachbarzustände vorhanden sind. Wenn nicht, wird der entsprechende Zustand der Nachbarposition entfernt. Falls mindestens ein Zustand entfernt ist, wird die Position dem Stapel hinzugefügt. Die Funktion terminiert, wenn der Stapel abgearbeitet ist.

Algorithm 3 Propagieren

```
function PROPAGATE(position)
    stack ← init empty stack
    stack push position
    while stack not empty do
        currentPosition ← pop stack
        if entropy of currentPosition = 0 then
            return true
        end if
        currentPossibleStates ← possible states of currentPosition
        for all dir in directions do
            adjacentPosition ← position in dir relative to currentPosition
            currentPossibleNeighbours ← init empty set
            for all possibleState in currentPossibleStates do
                currentPossibleNeighbours add all neighbours of possibleState in dir
            end for
            adjacentPossibleStates ← possible states of adjacentPosition
            modifiedPosition ← false
            for all adjacentState in adjacentPossibleStates do
                if currentPossibleNeighbours not contains adjacentState then
                    remove adjacentState from adjacentPosition
                    modifiedPosition ← true
                end if
            end for
            if modifiedPosition and stack not contains adjacentPosition then
                stack push adjacentPosition
            end if
        end for
    end while
    return false
end function
```

4.5.3 Einbindung in bestehende Strukturen

Um eine Kombination mit dem zellularen Automaten zu ermöglichen, muss ein Möglichkeit gefunden werden, den Wave Function Collapse auf eine bestehende Struktur anzuwenden [31]. Hierfür wird die Strukur wie ein Modell behandelt und durch den *PrototypeManager* weitere Prototypen generiert. Die Menge an Prototypen ist somit eine Zusammenführung der beiden Modelle. Dabei werden bei der Erzeugung der Prototypen aus dem Modell des zellularen Automaten nur die nötigsten Muster extrahiert. Das heißt, dass keine rotierten Kopien erzeugt werden und in Schritten der Größe vom Kernel durch das Modell iteriert wird. Es sollen nur soviel Prototypen erzeugt werden, dass gerade noch alle Teile des Modells vom zellularen Automaten abgebildet werden können. Die grundsätzliche Höhlenstruktur soll nicht modifiziert werden, sondern nur durch neue

Algorithm 4 WFC Vorverarbeitung einer bestehenden Struktur

```
function PRECOLLAPSE(wave, context)
    for all position ∈ wave do
        possibleStates ← all possible states of position
        for all possibleState ∈ possibleStates do
            positionContext ← contextState(position, context)
            if invalidInContext(possibleState, positionContext) then
                possibleStates ← possibleStates remove possibleState
            end if
        end for
    end for
    for all position ∈ wave do
        temp ← wave
        conflict ← propagate(position)
        if conflict then
            wave ← temp
            position ← contextState(position, context)
        end if
    end for
end function
```

Objekte und Details ergänzt werden. In Sektion 3.3 wird das Konzept des nun notwendigen Vorverarbeitungsschritts beschrieben, welche durch eine Funktion *precollapse* implementiert ist. Sie wird genutzt, wenn eine bestehende Zellenstruktur als *context* vorhanden ist und wird vor dem eigentlichen WFC Algorithmus ausgeführt. Die Vorbedingung hierfür ist, dass der *context* bereits durch den *PrototypeManager* verarbeitet und in der *wave* an den entsprechenden Positionen hinterlegt ist.

Wie im Algorithmus 4 zu sehen ist, besteht der Vorverarbeitungsschritt aus 2 Schritten. Zuerst werden alle möglichen Zustände der Positionen in der *wave* durch die Funktion *invalidInContext* gefiltert. Hierbei wird der *context* durch die Funktion *contextState* herbeigezogen, welcher genau die Zellenmenge darstellt, welche an der gegebenen Position in der Zellenstruktur des zellularen Automaten vorhanden sind. Es wird anschließend geprüft, ob jede Zelle der Prototypzustände mit den Zellen des *context* mit den Bedingungen in Sektion 3.3 übereinstimmt. Also ob Zellen auch Höhlenzellen sind, welche auch im *context* Höhlenzellen sind. Aber auch ob Tunnelzellen des *context* an denselben Stellen im Prototyp keine Höhlenzellen sind. Falls einer der Zellen die Bedingungen nicht erfüllt, wird der Zustandsprototyp für die entsprechende Position entfernt. Der zweite Teil des Vorverarbeitungsschritts versucht anschließend die aus der Filterung entstandenen Änderungen auf die *wave* zu propagieren.
Im Falle eines Konflikts wird die Änderungen der aktuellen Iteration wie im eigentlichen WFC zurückgesetzt. Zusätzlich wird die Ausgangsposition, welche den Konflikt ausgelöst hat, in den Zustand des *context* kollabiert. Das Resultat des Vorverarbeitungsschritts ist, das alle Zustände der Positionen an den *context* angepasst sind. Es bleiben somit nur noch die Zustände in den Positionen der *wave* übrig, welche alle Bedingungen zum *context* erfüllen. Im schlechtesten Fall wird zurück in die Ausgangsstruktur des zellularen Automaten kollabiert. Dies passiert genau dann, wenn alle anderen Prototypen außerhalb vom *context* innerhalb der Struktur ungültig sind oder zu viele Konflikte aufgetreten sind.

5 Evaluierung

Das folgende Kapitel setzt sich mit den Ergebnissen aus der Umsetzung auseinander. Es soll evaluiert werden, ob die Annahmen und die ausgearbeiteten Konzepte zu den gewünschten Ergebnissen kommen. Des Weiteren sollen aufgetretene Probleme erläutert und bewertet werden.

5.1 Auswertung der Umsetzung

Die Bewertung der Umsetzung nimmt Bezug auf die definierten Anforderungen und soll einen Überblick über die erreichten Ergebnisse geben.

5.1.1 Zellularer Automat

Die Umsetzung des zellularen Automaten stellt eine generische Implementierung dar, welche unter anderem durch die flexible Definierung der Nachbarschaftsregeln vielfältige Ergebnisse erlaubt.

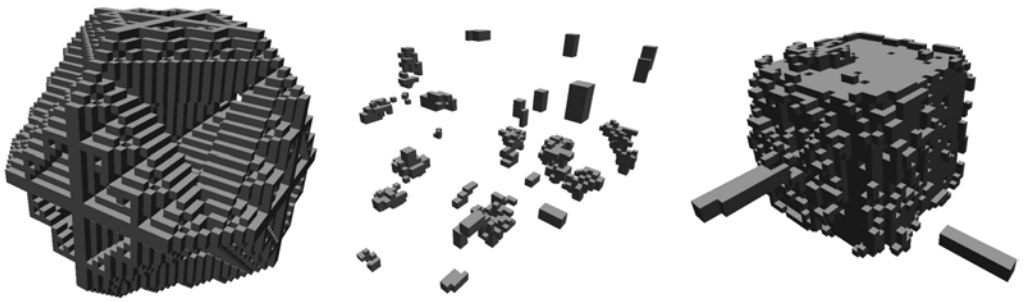


Abbildung 5.1: Durch CA generierte 3D-Strukturen (Darstellung in Anlehnung [46])

Die Arbeit hat sich im Speziellen auf die Generierung von Höhlenstrukturen fokussiert. Genau hier können die zellularen Automaten ihre Stärke zeigen. Die Resultate zeigen organische und komplexe Höhlen sowohl im zweidimensionalen als auch im dreidimensionalen Raum. Durch Nachverarbeitungsprozesse wie dem *Flood Fill* (Anhang A) werden Rauschen und Ungenauigkeiten bereinigt. Die spätere Darstellung kann die Zellengitter auf zwei verschiedene Arten darstellen. Zum einen als Kubusgitternetz und zum anderen durch den *Marching Cube* Algorithmus (Anhang B).

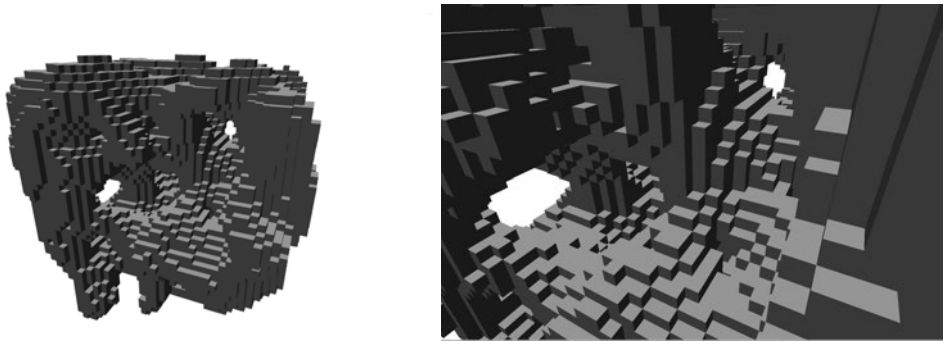


Abbildung 5.2: Durch zellularen Automaten generierte 3D-Höhlenstruktur (Darstellung in Anlehnung [46])

Eines der Stärken von zellularen Automaten liegt in dem einfachen Aufbau und der daraus resultierenden vorhersehbaren Komplexität. Für denselben Automaten und denselben Gitterdimensionen bleibt die Laufzeit stets konstant und in einem Rahmen, welche sich für Echtzeitberechnung nutzen lassen kann (Tabelle 5.1). Dieser Umstand macht den CA zu einem nützlichen Werkzeug für die prozedurale Generierung.

Tabelle 5.1: Messungen einiger Berechnungswerte des zellularen Automaten

cells per axis	generations	total cells	neighbour compares	time
10	10	1000	209520	0.022s
20	10	8000	1871120	0.03s
30	10	27000	6544720	0.072s
40	10	64000	15790320	0.168s
50	10	125000	31167920	0.323s

Weniger vorhersehbar sind jedoch die generierten Modelle. Vor allem in Anbetracht der lokalen Details kann wenig Einfluss genommen werden. Genau hierfür versucht die beschriebene Umsetzung einen Lösungsansatz vorzuschlagen. Der Ansatz zeigt eine Möglichkeit, Teilbereiche des Modells unabhängig vom Rest des Gitternetzes zu manipulieren. Der Teilbereich bleibt jedoch stets konsistent zum umliegenden Kontext und den definierten Nachbarschaftsregeln.

5.1.2 Wave Function Collapse

Die Arbeit setzt eine auf Zellen basierende Implementierung des Wave Function Collapse um. Diese Abwandlung des WFC extrahiert Prototypen in einer definierten Größe aus einem Eingabemodell. Als Nachbarschaftregel wird eine Zellengleichheit der überlappenden Ränder zweier Prototypen verwendet. Ein Netz an Superpositionen mit diesen Prototypen wird kollabiert und das Resultat sind neue Modelle, die an allen Stellen eine lokale Ähnlichkeit und Verhältnismäßigkeit zum Eingabemodell aufweisen.

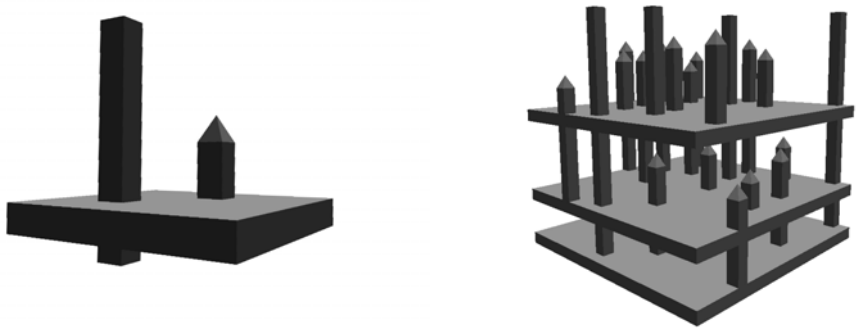


Abbildung 5.3: Vom WFC generierte Säulenstruktur (eigene Darstellung)

Hierbei ist ein großer Wert auf die Korrektheit und Konsistenz der extrahierten Nachbarschaftsverhältnisse gelegt. Um dies sicher zustellen, wird über alle kollabierten Positionen iteriert und geprüft, ob die Nachbarpositionen valide sind. Die Ergebnisse zeigen, dass die generierten Modelle über mehrere Stichproben hinweg mit den definierten Adjacency Contraintss konsistent bleiben.
Im Gegensatz zu den zellularen Automaten ist der Prozess um ein Vielfaches komplexer

und garantiert keine konstante Laufzeit. Es ist auch nicht garantiert, dass überhaupt ein Ergebnis in einer akzeptablen Zeit erzeugt wird. Hier entscheiden einige Faktoren über die Leistung und dem Erfolg. Zum einen das Eingabemodell, welches durch hohe Varianz und Komplexität in den verwendeten Zellen zu einer erhöhten Anzahl an extrahierten Prototypen führt. Zum anderen die Größe der Ausgabe. Der schlimmste anzunehmende Fall für den Algorithmus ist es, wenn das Eingabemodell sehr klein von der Dimensionalität ist, aber trotzdem möglichst viele und unterschiedliche Kombinationen von Zellen aufweist. Dieser Umstand führt dazu, dass ein Prototyp pro Seite nur einen oder gar keine validen Nachbarn besitzt. Viele Prototypen mit möglichst wenig validen Kombinationen führt zu einer hohen Konfliktanfälligkeit, welches dadurch in das Iterationslimit läuft und dementsprechend nicht vollständig kollabieren kann. Den Optimalfall stellen Eingabemodelle dar, welche eine begrenzte Varianz an unterschiedlichen Zellen aufweist und eine klare und einheitliche Struktur aufweisen. Hier kann der Algorithmus oft wiederholdende Muster extrahieren. Daraus resultieren weniger Prototypen mit mehr Kombinationen der Nachbarschaften. In diesem Fall hat der Wave Function Collapse mehr Spielraum beim Kollabieren, da ein ausgeschlossener Zustand einer Position nicht direkt das gesamte Netz beeinflusst.

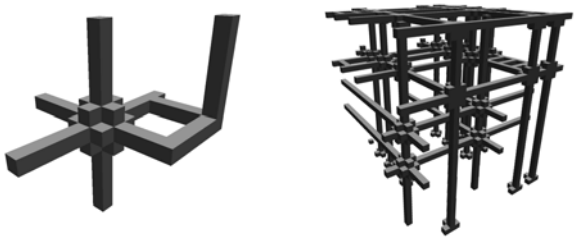


Abbildung 5.4: Für WFC optimales Modell durch klare Struktur (eigene Darstellung)

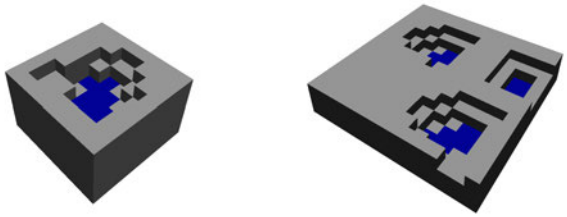


Abbildung 5.5: Für den WFC schwierig zu verarbeitenes Modell (eigene Darstellung)

Um dies zu verdeutlichen, wurden zwei Beispiele für Eingabemodelle durch den WFC auf eine Ausgabegröße von 15^3 Zellen mit unterschiedlichen Kernelgrößen angewendet. Das Eingabemodell aus Abbildung 5.4 stellt ein optimales Modell dar, da es symmetrische und klare Strukturen aufweist. Dagegen soll das Modell aus Abbildung 5.5 ein schwierieger zu verarbeitendes Modell zeigen. Da hier viele asymmetrische Formen vorhanden sind, müssen dementsprechend viele individuelle Prototypen mit vergleichsweise wenig Nachbarschaften generiert werden.

Tabelle 5.2: WFC Messwerte für Modell aus Abbildung 5.4

kernel size	prototypes	positions	total states	avg. neighbours	iterations	time
5	118	125	14750	46	8	0.2s
4	198	343	67914	72	28	3.3s
3	352	3375	1188000	49	378	189.5s

Tabelle 5.3: WFC Messwerte für Modell aus Abbildung 5.5

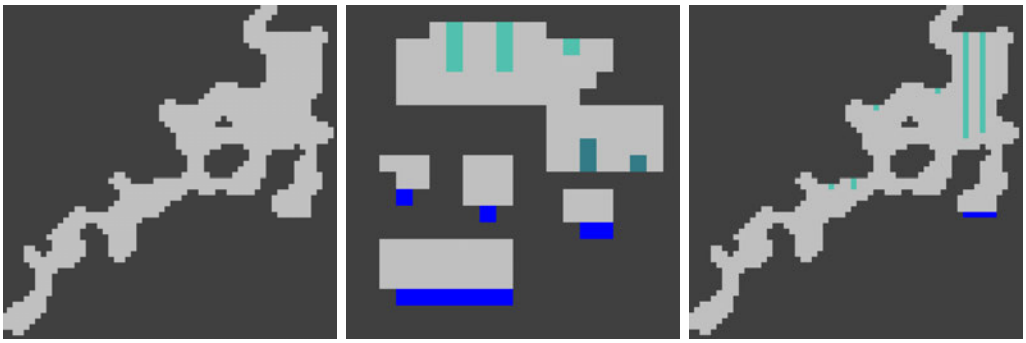
kernel size	prototypes	positions	total states	avg. neighbours	iterations	time
5	34	125	4250	6	11	0.3s
4	154	343	52822	14	13	0.98s
3	466	3375	1572750	16	33	229.93s

Die Messwerte aus den Tabellen 5.2 und 5.3 zeigen, dass bei einer vergleichsweise kleinen Kernelgröße die Berechnungszeit stark ansteigt, da deutlich mehr Prototypen generiert werden und es somit insgesamt mehr Zustände zum Kollabieren gibt. Die Laufzeiten überschreiten bei einem zu kleinen Kernel die nicht funktionalen Anforderungen von 10 Sekunden deutlich.
Je nach Modell gibt es somit zwei Möglichkeiten für das Ergebnis. Entweder alle Positionen kollabieren komplett und konsistent. Oder alle Positionen laufen in ein Iterationslimit rein, da sie immer wieder in Konflikte laufen, da zu viele Prototypen mit zu wenig gültigen Nachbarschaften vorhanden sind. Die Schlussfolgerung aus diesem Umstand ist, dass die vorgeschlagene Implementierung des WFC zu vielfältigen und konsistenten Ergebnissen führt, aber die Zuverlässigkeit stark an das Eingabemodell und die verwendeten

Parameter gekoppelt ist. Diese Inkonsistenz wird allgemein als großen Nachteil gesehen und hindert den WFC häufig an größeren kommerziellen Anwendungen [28].

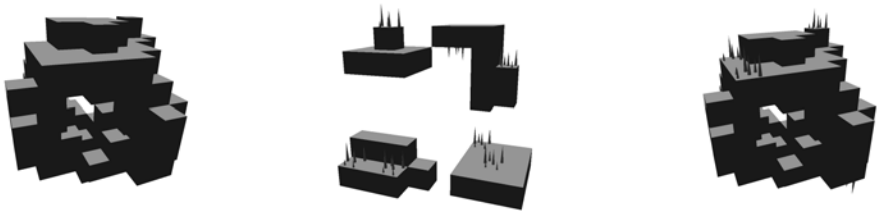
5.1.3 Kombination der Verfahren

Das Ziel der Kombination aus CA und WFC ist es, die Nachteile gegenseitig auszugleichen und die Stärken zu betonen. Hierfür sollte der zellulare Automat für die Generierung der grundlegenden Höhlenstruktur zuständig sein und der Wave Function Collapse für die Platzierung von Details und komplexeren Objekten. Hierbei ist eine der Anforderungen, dass die Höhlenstruktur vom WFC beibehalten werden soll.



(a) CA Höhlenstruktur (b) WFC Eingabemodell (c) Ausgabemodell

Abbildung 5.6: Kombination der Verfahren anhand einer 2D-Höhlenstruktur (eigene Darstellung)



(a) CA Höhlenstruktur (b) WFC Eingabemodell (c) Ausgabemodell

Abbildung 5.7: Kombination der Verfahren anhand einer 3D-Höhlenstruktur (eigene Darstellung)

Die vorgestellte Umsetzung ist hierbei in der Lage, die Anforderungen zum Teil zu erfüllen und zeigt bei der Laufzeit ähnliche Limitierungen wie der isolierte WFC. Durch die

Umwandlung der CA Struktur in weitere Prototypen wird eine natürliche Kombination mit dem Wave Function Collapse ermöglicht. Das anschließende Vorfiltern der Prototypen garantiert die Erhaltung der Höhlenstruktur, lässt jedoch noch genug Spielraum für die additive Ergänzung und dem Kollabieren der restlichen Ausgabe.
Die Implementierung wird im schlechtesten Fall die ausgehende CA Höhlenstruktur ausgeben, da garantiert ist, dass die Prototypen der Höhlenstruktur untereinander valide sind. Selbst wenn keines der zusätzlichen Prototypen eine konfliktfreie Position findet, kann immer auf den Prototypen der ausgehenden Struktur kollabiert werden. Im besten Fall ist das Eingabemodell für den WFC in einer Form, welche gut mit der Höhlenstruktur kombinierbar ist. Hierfür müssen lokale Muster in der Höhlenstruktur auch im WFC Eingabemodell vorkommen und an diesen Stellen mit zusätzlichen Zellen ergänzen. Der Algorithmus arbeitet dementsprechend genau dann am besten, wenn das WFC Eingabemodell stark an die abzubildende Höhlenstruktur angepasst ist.

Tabelle 5.4: WFC Messwerte für Modell aus Abbildung 5.7

kernel size	prototypes	positions	total states	avg. neighbours	iterations	time
5	1124	125	557	208	24	21.51s
4	1158	343	3986	244	3000	35.8s
3	1583	3375	44848	567	33	201.96s

Die Laufzeit hier unterscheidet sich unwesentlich von der Laufzeit des Wave Function Collapse ohne den zellularen Automaten, wie die gemessenen Werte in Tabelle 5.4 zeigen. Die zu kollabierenden Zustände sind jedoch deutlich niedriger, da bereits die meisten durch den Kontextfilter (siehe Algorithmus 4) aus den Positionen entfernt sind. Die höhere durchschnittliche Nachbarschaftsmenge pro Position, welche aus der Kombination zweier Modelle folgt, relativiert die Reduktion der Zustandskomplexität jedoch wieder.

5.2 Probleme

Die Umsetzung der Arbeit zeigt, dass eine Kombination beider Algorithmen zu den gewüsnchten Ergebnissen kommen kann. Es verdeutlicht jedoch auch einige Probleme. Diese limitieren die Software vor allem in der Leistung und der Anwendungsgebiete.

- **WFC Modellierung**
  Eines der schwerwiegendsten Limitierungen stellt die vom WFC präferierten Modelle dar. Wie in Sektion 5.1 beschrieben, bevorzugt der Algorithmus Eingabemodelle, welche eine klare Struktur und eine niedrige Varianz an unterschiedlichen Zellen aufweisen. Komplexere Eingabemodelle sind mit der Implementierung zu konfliktanfällig, was den Umgang mit dem Wave Funtion Collapse erschwert. Eine Möglichkeit hier auszubessern wäre die Teilmengen des Eingabemodells noch weiter künstlich zu erweitern. Neben dem Rotieren können auch gespiegelte Prototypen zu einer größeren Menge an validen Nachbarschaften führen. Dies würde jedoch die Berechnungszeit der Propagation weiter erhöhen.

- **Tradeoff der Nachbarschaftsdefinitionen**
  Die Umsetzung unterstützt zwei Arten der Nachbarschaftbedingungen. Beim Ersten werden von zwei Prototypen nur die entsprechend äußeren Sockel verglichen, beim Zweiteren die Kernsockel mit dem Außensockel des anderen Prototyps. Zweiteres führt zu einer Überlappung der Prototypen und zu einer erhöhten Genauigkeit. Dies schränkt jedoch die möglichen Nachbarschaften in einem Maße ein, dass es die Laufzeit vom WFC stark erhöht. Es gibt also einen Tradeoff zwischen Genauigkeit und Leistung.

- **Positionsumfang schränkt WFC ein**
  Wenn der Wave Function Collapse eine zu große Anzahl an Positionen kollabieren muss und das Eingabemodell eine gewisse Größe aufweist, kann der Algorithmus alle Positionen theoretisch kollabieren. Jedoch kann es praktisch dazu führen, dass zu viele Iterationen benötigt werden. Durch enorme Zustandsgrößen und durch potentielle Konflikte kann der Algorithmus das Iterationslimit schnell erreichen. In solchen Fällen muss die Anzahl der Prototypen möglichst gering gehalten werden.

- **Prototypen sind im Kontext stark eingeschränkt**
  Wenn der Wave Function Collapse auf eine bestehende Höhlenstruktur angewendet werden soll, werden im ersten Schritt die Prototypen in den Positionen gefiltert, welche im Kontext der bestehenden Struktur nicht in der entsprechenden Position kollabiert werden können. Da die bestehende Struktur beibehalten werden soll, sind die in Sektion 3.3 beschriebenen Bedingungen sehr strikt und schränken die möglichen Prototypen stark ein. Dies kann dazu führen, dass keine zusätzlichen Zellen durch den WFC eingefügt werden können.

6 Fazit

Dieses Kapitel soll abschließend einen Überblick über die Arbeit geben und zusammenfassen, welche Schlüsse aus den erreichten Zielen gezogen werden können. Zudem wird ein Ausblick auf Ansätze und Möglichkeiten gegeben, wie an die Ergebnisse der Arbeit angeschlossen werden kann, um diese zu verbessern und zu erweitern.

6.1 Zusammenfassung der Arbeit

Die Arbeit beschreibt eine prototypische Umsetzung, welche dreidimensionale Höhlenstrukturen durch die Kombination von zellularen Automaten und dem Wave Function Collapse erzeugt. Die vorgestellte Implementation zeigt, wie sich prozedurale Generierungsverfahren in Kombination verhalten und ergänzen können.
Die Arbeit verdeutlicht unter anderem, dass eine Kombination von sehr unterschiedlichen Algorithmen gute Ergebnisse liefern kann, diese jedoch genau aufeinander abgestimmt werden müssen. Die Stärken der jeweiligen Verfahren müssen identifiziert werden und das Anwendungsfeld im Hinblick hierauf ausgerichtet und aufgeteilt werden. Im Anwendungsbeispiel dieser Arbeit werden die zellularen Automaten nur für die Generierung der Höhlenstrukturen genutzt, da eine detaillierte Generierung von komplexen Objekten und Zusammenhänge mit einem CA schwer umsetzbar ist. Der Wave Function Collapse wird wiederum genau hierfür genutzt, sodass eine strukturelle Arbeitsteilung geschaffen wird. Die Umsetzung bildet dieses Verhalten deutlich ab und zeigt das Potential des Konzepts. Jedoch werden auch Limitierungen deutlich. Die Modellierung des Eingabemodells für den WFC muss stark an die grundsätzliche Höhlenstruktur angepasst sein, damit valide Kombinationen gefunden werden. Zudem hat der auf Zellen basierte Wave Function Collapse einen zu schmalen optimalen Bereich zwischen der Modellgröße und der Komplexität, in welchem dieser konsistent arbeitet kann. Genau hier liegt der größte Optimierungsbedarf.

Insgesamt zeigt das Ergebnis der Arbeit eine Möglichkeit, organische Höhlenstrukturen durch minimalen Modellierungsaufwand zu generieren. Der Anwendungsbereich für eine solche Umsetzung liegt in der automatischen Generierung von virtuellen Welten. Durch weitere Anpassung kann dies auf andere Domänen erweitert werden. Das kann die Generierung von Dungeons oder auch die allgemeine Platzierung von Objekten und Details in einer Spielwelt sein. Um dies im größeren Rahmen umzusetzen, muss der Algorithmus weiter optimiert und verfeinert werden. Die Arbeit hat jedoch gezeigt, welche Ergebnisse prototypisch durch Kombination der behandelten prozeduraler Generierungsverfahren möglich sind.

6.2 Ansätze für Verbesserungen

Abschließend werden Ansätze zur Verbesserung und Optimierung aufgezeigt. Alle im Folgenden aufgelisteten Punkte sind entweder zur Steigerung der Leistung gedacht oder sollen einen Ausblick geben, in welcher Form das Projekt erweitert werden kann.

- **Speichern der Prototypen**
  Ein gewisser Teil der Berechnungszeit wird beim Wave Function Collapse für die Extrahierung der Prototypen verbraucht. Diese werden in der beschriebenen Implementierung anschließend in einen Cache geschrieben, der jedoch bei einem neuen Modell geleert wird. Dieser Schritt kann beschleunigt werden, wenn die Prototypen persistent gespeichert werden. Damit entfällt beim erneuten Ausführen der Extraktionsschritt.

- **Parallisieren der Propagation**
  Der wohl größte Berechnungsaufwand des Wave Function Collapse stellt das Propagieren dar. Die Implementierung setzt dies sequenziell um, was bei einer großen Anzahl von Positionen schnell uneffizient wird. Eine Parallelisierung der Propagation hat großes Potential und würde die Berechnungszeit stark kürzen. In dem speziellen Anwendungsfall ist eine parallele Verarbeitung schwierig, da Änderungen an einer Position direkte Auswirkung auf die benachbarten Positionen hat. Ansätze für eine solche Umsetzung [25] zeigen deutlich den Mehrwert für den Wave Function Collapse.

- **Vermeidung homogener Strukturen**
  Die Umsetzung des Wave Function Collapse versucht stets die Frequenz der Prototypen an das Eingabemodell anzugleichen. Dies führt dazu, dass die Verhältnisse in der lokalen Betrachtung korrekt sind. Global betrachtet entsteht jedoch eine homogene Verteilung über den gesamten Zustandsraum. Dem kann entgegengesteuert werden, indem bestimmte Zonen definiert werden, in welchem bestimmte Prototypen häufiger auftreten können[1]. Dies kann beispielweise nützlich für die Definierung von Biomen innerhalb einer Landschaftsgenerierung sein.

- **Genereller Implementierung des WFC**
  Für die Kombination von zellularen Automaten und dem Wave Function Collapse nutzt in der Implementierung der WFC ein Zellengitter als Datenmodell, womit Berechnungen deutlich vereinfacht sind. Die Kombination mit anderen Datenstrukturen ist so jedoch stark limitiert. In der Computergrafik werden vor allem Dreieckspolygonnetze aus Eckpunkten und Kanten verwendet, um Modelle abzubilden. Dreidimensionale Implementierungen des Wave Function Collapse nutzen klassischerweise solche Modelle [20] und würde die Implementierung mit vielen weiteren prozeduralen Generierungsverfahren kombinierbar machen.

- **Weitere Kombinationen prozeduraler Generierungen**
  Die Arbeit stellt eine Möglichkeit vor, zwei prozeduralen Generierungsverfahren zu kombinieren. Dass dies oft zu erstaunlichen Ergebnissen führt, zeigen bereits andere Ansätze [31]. Vor allem der Wave Function Collapse bietet durch seine Vielfältigkeit hohen Kombinationspotential. Eine sehr ähnliche Kombination zu den zellularen Automaten wäre *Drunkard's Walk*, welcher für die Generierung von Dungeons genutzt wird. Der Wave Function Collapse könnte hier für die Platzierung von Gegnern und Truhen verwendet werden.

[1] Ansätze zur Vermeidung homogener Strukturen zeigt das roguelike *Caves of Qud*, welches Entwickler Brian Bucklew 2019 in seiner Präsentation bei *Roguelike Celebration* verdeutlicht.

Literaturverzeichnis

[1] Aljasser, Khalid: Implementing design patterns as parametric aspects using ParaAJ: The case of the singleton, observer, and decorator design patterns. 45 (2016), S. 1–15

[2] Amato, Alba: *Procedural Content Generation in the Game Industry*. S. 15–25, 03 2017

[3] Bach, Esben ; Madsen, Andreas: *Procedural Character Generation: Implementing Reference Fitting and Principal Components Analysis*, Aalborg University, Diplomarbeit, 2007

[4] Bays, Carter: Candidates for the Game of Life in Three Dimensions. In: *Complex Systems* 1 (1987), 01, S. 373–400

[5] Bays, Carter: *Patterns for Simple Cellular Automata in a Universe of Dense-Packed Spheres*. Bd. 1. S. 853–875, 1987

[6] Boris: *Wave Function Collapse Explained*. 2020. – URL https://www.boristhebrave.com/2020/04/13/wave-function-collapse-explained/. – Zugriffsdatum: 2022-01-09

[7] Bourke, Paul: *Polygonising a scalar field*. 1994. – URL http://paulbourke.net/geometry/polygonise/

[8] Carli, Daniel ; Bevilacqua, Fernando ; Pozzer, Cesar ; d'Ornellas, Marcos: A Survey of Procedural Content Generation Techniques Suitable to Game Development, 11 2011, S. 1–2

[9] Cheng, Darui ; Han, Honglei ; Fei, Guangzheng: *Automatic Generation of Game Levels Based on Controllable Wave Function Collapse Algorithm*. S. 37–50, 01 2020

[10] François, Alexandre: Software architecture for computer vision: Beyond pipes and filters. (2003), 08, S. 3–7

[11] Freiknecht, Jonas: *Procedural content generation for games*, Universität Mannheim, Dissertation, 2021

[12] Gasch, Cristina ; Chover, Miguel ; Remolar, Inmaculada ; Rebollo, Cristina: Procedural modelling of terrains with constraints. In: *Multimedia Tools and Applications* 79 (2020), Nr. 41-42, S. 1–10

[13] Gobron, Stéphane ; Coltekin, Arzu ; Bonafos, Hervé ; Thalmann, Daniel: GPGPU Computation and Visualization of Three-dimensional Cellular Automata. (2010), 08, S. 3–6

[14] Gumin, Maxim: *WaveFunctionCollapse*. 2017. – URL https://github.com/mxgmn/WaveFunctionCollapse/blob/master/README.md. – Zugriffsdatum: 2022-03-22

[15] Harrison, Neil ; Heesch, Uwe ; Sobernig, Stefan ; Sommerlad, Peter ; Filipczyk, Martin ; Fülleborn, Alexander ; Musil, Angelika ; Musil, Juergen: Software Architecture Patterns: Reflection and Advances: [Summary of the MiniPLoP Writers' Workshop at ECSA'14]. In: *ACM SIGSOFT Software Engineering Notes* 40 (2015), 02, S. 30–34

[16] Harzallah, Hassen: *L-systems : draw nice fractals and plants*. 2020. – URL https://medium.com/@hhtun21/l-systems-draw-your-first-fractals-139ed0bfcac2. – Zugriffsdatum: 2022-01-21

[17] He, Yixuan ; Hu, Tianyi ; Zeng, Delu: Scan-flood Fill(SCAFF): an Efficient Automatic Precise Region Filling Algorithm for Complicated Regions, 2019, S. 1–3

[18] Henrich, Dominik: *Space-efficient Region Filling in Raster Graphics, The Visual Computer*. Bd. 10. S. 205–215, 1994

[19] Johnson, Lawrence ; Yannakakis, Georgios ; Togelius, Julian: Cellular automata for real-time generation of infinite cave levels. (2010), S. 1–3

[20] Karth, Isaac ; Smith, Adam M.: WaveFunctionCollapse is constraint solving in the wild. In: *Proceedings of the 12th International Conference on the Foundations of Digital Games*, ACM, 2017, S. 1–10. – URL https://doi.org/10.1145/3102071.3110566

[21] Khaled, Rilla ; Nelson, Mark J. ; Barr, Pippin: Design Metaphors for Procedural Content Generation in Games. In: *Proceedings of the SIGCHI Conference on Human*

Factors in Computing Systems. New York, NY, USA : Association for Computing Machinery, 2013 (CHI '13), S. 1509–1518

[22] Kim, Hwanhee ; Lee, Seongtaek ; Lee, Hyundong ; Hahn, Teasung ; Kang, Shinjin: Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm, 08 2019, S. 1–4

[23] Krasner, Glenn ; Pope, Stephen: A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk80 System. In: *Journal of Object-oriented Programming - JOOP* 1 (1988), 01, S. 3–10

[24] Lagae, Ares ; Lefebvre, Sylvain ; Cook, Rob ; Derose, T. ; Drettakis, George ; Ebert, David ; Lewis, J.P. ; Perlin, Ken ; Zwicker, Matthias: A Survey of Procedural Noise Functions. In: *Computer Graphics Forum* 29 (2010), 12, S. 1–10

[25] Lee, Amy ; Orlowski, Jan: *Parallel Wave Function Collapse*. 2019. – URL https://amylh.github.io/WaveCollapseGen/. – Zugriffsdatum: 2022-02-24

[26] Lindenmayer, Aristid: Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. In: *Journal of Theoretical Biology* 18 (1968), Nr. 3, S. 300–315. – ISSN 0022-5193

[27] Lorensen, William ; Cline, Harvey: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In: *ACM SIGGRAPH Computer Graphics* 21 (1987), 08, S. 163–169

[28] Marian: *Infinite procedurally generated city with the Wave Function Collapse algorithm*. 2019. – URL https://marian42.de/article/wfc/. – Zugriffsdatum: 2021-12-19

[29] Mark, Benjamin ; Berechet, Tudor ; Mahlmann, Tobias ; Togelius, Julian: *Procedural Generation of 3D Caves for Games on the GPU*. 2015

[30] Marvie, Jean-Eudes ; Perret, Julien ; Bouatouch, Kadi: The FL-System: A Functional L-System for Procedural Geometric Modeling. 21 (2005), jun, Nr. 5, S. 329–339

[31] Minini, Pedro ; Assunção, Joaquim: Combining Constructive Procedural Dungeon Generation Methods with WaveFunctionCollapse in Top-Down 2D Games, 11 2020, S. 395–398

[32] Mount, Dave: *Procedural Generation: Perlin Noise, CMSC 425.* Kap. 14, 2018

[33] Newman, Timothy S. ; Yi, Hong: A survey of the marching cubes algorithm. In: *Computers & Graphics* 30 (2006), oct, Nr. 5, S. 854–879

[34] Packard, Norman H. ; Wolfram, Stephen: Two-dimensional cellular automata. In: *Journal of Statistical Physics* 38 (1985), Nr. 5-6, S. 901–946

[35] Pavlidis, Theo: Contour Filling in Raster Graphics. New York, NY, USA : Association for Computing Machinery, 1981, S. 29–36. – URL https://doi.org/10.1145/800224.806786

[36] Perlin, Ken: An Image Synthesizer. New York, NY, USA : Association for Computing Machinery, 1985, S. 287–296

[37] Perlin, Ken: Improving Noise. 21 (2002), Nr. 3

[38] Petrasch, Martin: *Prozedurale Städtegenerierung mit Hilfe von L-Systemen*, Technische Universitat Dresden, Diplomarbeit, 2008

[39] Prusinkiewicz, Przemyslaw ; Lindenmayer, Aristid: Graphical modeling using L-systems. In: *The Virtual Laboratory.* Springer New York, 1990, S. 1–50. – URL https://doi.org/10.1007/978-1-4613-8476-2_1

[40] Rodríguez Puente, Rafael ; Pérez Betancourt, Yadian ; Mufeti, Kauna: Cellular Automata And Its Applications In Modeling And Simulating The Evolution Of Diseases, 09 2015, S. 1–3

[41] Savidis, Anthony: There is more to PCG than Meets the Eye: NPC AI, Dynamic Camera, PVS and Lightmaps, 08 2018, S. 14–16

[42] Sherratt, Stephen: *Procedural Generation with Wave Function Collapse.* 2019. – URL https://www.gridbugs.org/wave-function-collapse/. – Zugriffsdatum: 2022-02-15

[43] Shiffman, Daniel: *The Nature of Code: Simulating Natural Systems with Processing.* Bd. 1. Kap. 7.1, 2012. – ISBN: 0985930802

[44] Smith, Alvy R.: Tint Fill. New York, NY, USA : Association for Computing Machinery, 1979, S. 276–283

[45] Smith, Gillian ; Gan, Elaine ; Othenin-Girard, Alexei ; Whitehead, Jim: PCG-based game design: Enabling new play experiences through procedural content generation. (2011), 01

[46] softologyblog: *3D Cellular Automata*. 2019. – URL https://softologyblog.wordpress.com/2019/12/28/3d-cellular-automata-3/

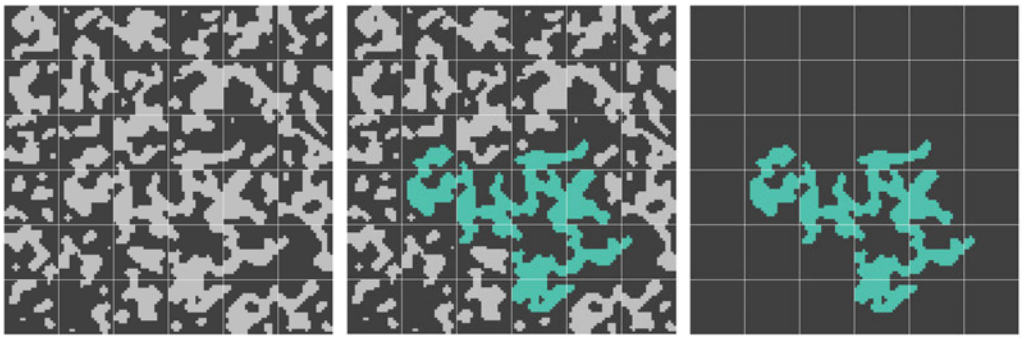
[47] Togelius, Julian ; Kastbjerg, Emil ; Schedl, David ; Yannakakis, Georgios: What is Procedural Content Generation? Mario on the borderline. (2011), 01, S. 1–2

[48] Wang, Xin ; Gao, Su ; Wang, Monan ; Duan, Zhenghua: An Marching Cube Algorithm Based on Edge Growth. (2021), S. 1–4. – URL https://arxiv.org/abs/2101.00631

[49] Wolfram, Stephen: *Cellular Automata and Complexity*. S. 3–48, 1994

A Flood Fill

Die Aufgabe des *Flood Fill* Algorithmus ist es, eine Teilmenge oder eine Sektion in einer Datenstruktur mit einem Wert zu füllen. Die Datenstrukturen können die sein, welche aus einer Menge an Elementen bestehen, die zueinander eine Beziehung aufweisen. Dazu zählen Pixelbilder, Polygonnetze oder Graphen. Hierfür liegen bereits einige Umsetzungen vor, welche auf verschiedenste Weise optimiert sind [18]. Grundsätzlich gibt es hier jedoch die Unterscheidung zwischen dem *Seed Filling* und dem *Edge Filling* [17][35].
In der Implementation der Arbeit wird der Flood Fill Algorithmus dafür genutzt, den größten zusammenhängenden Tunnel aus einer Höhlenstruktur zu extrahieren. Hierfür wird die Variante des *Seed Filling* [44] genutzt. Der Seed beschreibt hier einen Index innerhalb des Zellengitters der Höhle, von welchem aus der Algorithmus ausgeführt wird. Es wird der Seed gesucht, von welchem aus die meisten Zelen gefüllt werden.



(a) Eingabe (b) Größtes Tunnelsystem (c) Ausgabe

Abbildung A.1: Extraktion der größten zusammenhängenden Struktur (eigene Darstellung)

Sei ein Zellennetz R wie in Abbildung A.1a eine Höhlenstruktur. Der Algorithmus iteriert durch jede Zelle $c \in R$ und versucht von dieser Zelle als Seed aus das Netz zu fluten. Zu den gültigen Startpositionen gehören die Zellen, welche noch nicht geflutet sind und zu einem Tunnelsystem gehören.

Die Implementierung des Flutens verwendet einen Stapel an Zellen, wobei die an oberster Stelle liegende Zelle als Nächstes bearbeitet wird. Initial wird der Seed dem Stapel hinzugefügt. Für jede Zelle wird geprüft, ob sie Teil des Tunnelsystems ist und noch nicht markiert wurde. Ist dies der Fall, wird die Zelle markiert und alle Nachbarzellen dem Stapel hinzugefügt. In einem zweidimensionalen Zellennetz werden in der Implementierung nur alle 4 direkten Nachbarn mit einbezogen. Das Fluten terminierten genau dann, wenn alle Zellen des Tunnelsystems markiert sind und der Stapel somit leer ist.
Nachdem alle gültigen Zellen in R geflutet sind, kann zu jedem Tunnelsystem eine Menge an Zellen zugeordnet werden. Der Tunnel mit der höchsten Anzahl an gefluteten Zellen wird markiert (Abb. A.1b). Alle anderen Tunnelsysteme werden aus R entfernt. Somit bleibt nur noch das größte und isolierte Tunnelsystem übrig (Abb. A.1c).

B Marching Cube

Der Marching Cube (MC) Algorithmus wurde ursprünglich von Lorensen und Cline zur dreidimensionalen Darstellung von Anatomien in medizinischer Anwendung entworfen [27]. Hier werden die Datenpunkte von gescannten 2D-Schnittmengen einer Computertomographie in ein 3D-Model umgewandelt.

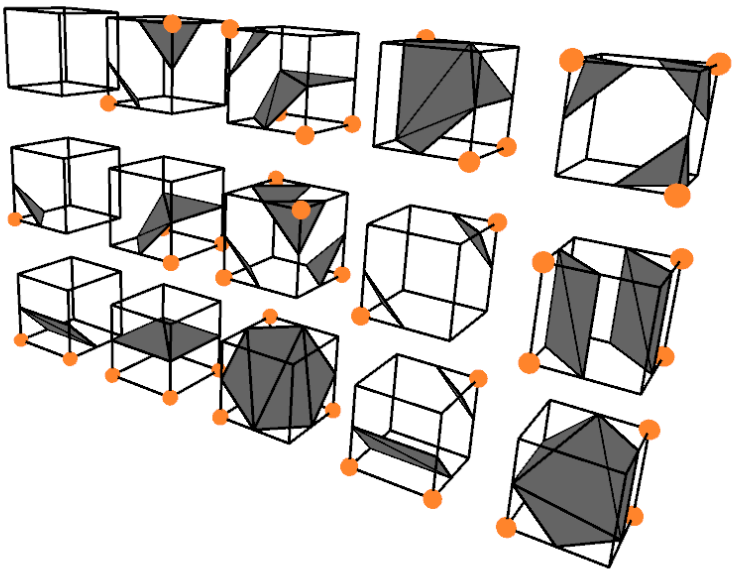


Abbildung B.1: Mögliche Oberflächenschnitte eines Volumens mit einem Kubus (Darstellung in Anlehnung [27])

Es soll ein Volumenmodell aus Datenpunkten generiert werden, welche in einem Datennetz angeordnet sind. Dabei wird für jeden Datenpunkt berechnet, ob dieser sich innerhalb oder außerhalb des Volumenmodells befindet. Durch das gesamte Punktnetz wird der namengebende Zeiger in Form eines Kubus iteriert, wobei alle 8 Ecken einen

Datenpunkt darstellen. Fallse es eine Schnittmenge mit der Volumenoberfläche und dem Kubus gibt, gilt dieser als aktiv. In diesem Fall gibt es mindestens einen Datenpunkt außerhalb und ein Datenpunkt innerhalb des Volumens [33]. Anhand der Anordnung und Werte der Datenpunkte kann in jedem Kubus der Oberflächenschnitt abgebildet werden. Alle 15 ursprünglichen Oberflächenschnitte mit dem Kubus sind in Abbildung B.1 dargestellt, welche von Rotationen und Invertierungen bereinigt sind [48].

(a) Zellendarstellung (b) Marching Cube

Abbildung B.2: Auswirkung von MC auf ein Zellenmodell (eigene Darstellung)

Die Umsetzung dieser Arbeit nutzt den *Marching Cube* Algorithmus für eine optionale Darstellung eines dreidimensionalen Zellengitters, welche jede Zelle als Platzhalter von einem beliebigen 3D-Modell wie einem Kubus darstellt. Der Algorithmus wird in einer vereinfachten Form auf die zuvor generierte Höhlenstruktur angewendet, bei welcher jede Zelle mit einem Wert von 0 als außerhalb des Volumens zählt und jede Zelle mit einer 1 gilt als Teil hiervon. Anschließend wird wie beschrieben durch das Zellennetz iteriert, wobei immer 8 Zellen als ein Marching Cube gezählt werden. Auf jeden Cube werden die möglichen Oberflächenschnitte angewandt. Um die Berechnung zu optimieren, wird eine statische Tabelle mit den entsprechenden Zellenwerten und den Ecken der resultierenden Oberfläche genutzt [7]. Das resultierende Modell (Abb. B.2b) zeigt weichere Übergänge zwischen den Zellen und ist für die praktische Anwendung der Höhlenstrukturen mehr geeignet als die sonstige Zellendarstellung (Abb. B.2a).

Glossar

Adjacency Contraints Bedingung, welche ein gültiges Verhältnis von zwei anliegenden Objekten definiert.

Kernel Faltungsmatrix oder Filter - abstrahiert einen Teilbreich aus einem Datenobjekt.

Seed Beliebiger Wert, mit welchem Ausgaben von pseudozufällige Algorithmen reproduziert werden können.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort Datum Unterschrift im Original