HAW HAMBURG

**BACHELORTHESIS**
Ben Dzaebel

# Erweiterung des Wave Function Collapse Algorithmus zur Synthese von 3D-Voxel-Modellen anhand von nutzergenerierten Beispielen

**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**

Ben Dzaebel

# Erweiterung des Wave Function Collapse Algorithmus zur Synthese von 3D-Voxel-Modellen anhand von nutzergenerierten Beispielen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
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: 02. Dezember 2020

**Ben Dzaebel**

**Thema der Arbeit**

Erweiterung des Wave Function Collapse Algorithmus zur Synthese von 3D-Voxel-Modellen anhand von nutzergenerierten Beispielen

**Stichworte**

Computergrafik, Prozedurale Generierung, Wave Function Collapse, Textursynthese, Modellsynthese, Voxelgrafik

**Kurzzusammenfassung**

Der Wave Function Collapse Algorithmus (WFC) von Max Gumin erlaubt die Generierung von komplexen Bildern und Texturen im Stil eines gegebenen Beispiel-Bildes. Dieser Algorithmus wird hier auf drei Dimensionen, im Speziellen auf Voxel-Modelle erweitert, um Modelle im Stil von Input-Modellen zu generieren. Dabei werden Erkenntnisse aus den Themenbereichen der Prozeduralen Generierung, der Textursynthese und der Modellsynthese erklärt und genutzt, um eine Anwendung zu entwickeln, die den Import, Export und das Generieren von Voxel Modellen mithilfe der erweiterten Version des WFC Algorithmus ermöglicht. Diese Arbeit beschreibt außerdem die nötigen theoretischen Konzepte sowie die Implementierung der Anwendung und zeigt, dass sich mit diesem Ansatz gute Ergebnisse erzielen lassen, wenn die richtigen Input-Modelle gewählt werden.

**Ben Dzaebel**

**Title of Thesis**

Extending the Wave Function Collapse Algortihm for synthesis of 3D-Voxel-Models by user generated examples

**Keywords**

Computergraphics, Procedural Content Generation, Wave Function Collapse, Texturesynthesis, Modelsynthesis, Voxelgraphics

**Abstract**

The Wave Function Collapse Algortihm by Max Gumin allows the generation of complex pictures and textures that are stylistically similar to a given input image. This algorithm is extended to work in three dimensions, specifically with voxel models, to generate output models similar to a given input model. This work aims to explain and use insights from areas such as procedural generation, texture synthesis und model synthesis to develop an application capable of importing, exporting and generating voxel models, using the extended WFC algorithm. Furthermore any needed theoretical concepts and implementation details are explained. The results produced by the developed application show the viability of the used approach, if the input models used adhere to certain constraints.

# Inhaltsverzeichnis

# 1 Einleitung

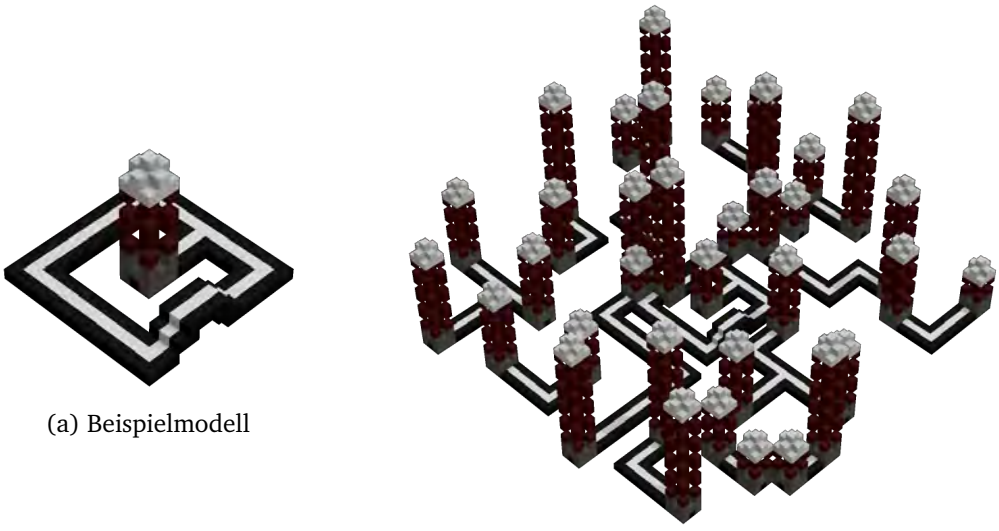
## 1.1 Motivation

Mit immer leistungsfähigeren Computern und raschen Fortschritten in Bereichen wie Computergrafik, Simulation und Spieleentwicklung sind über die letzten Jahre auch die Anforderungen an die benötigten Grafik-Assets stark gestiegen. Während ein durchschnittliches Charakter-Modell eines Videospiels noch vor einigen Jahren mit einigen Hundert Polygonen auskommen musste, ist es heute nicht unüblich Modelle mit bis zu 120.000 Polygonen [19] zu beobachten. Solch aufwendige Assets von Hand zu erstellen stellt natürlich eine große Zeit- und Geld-Investition dar.

Eine an Popularität gewinnende Lösung für dieses Problem ist die Prozedurale Content Generierung (PCG). Diese erlaubt es, Inhalte wie z. B. 3D-Modelle mithilfe von spezialisierten Systemen automatisch bzw. prozedural zu generieren, statt sie aufwendig manuell zu erstellen. Ein vergleichsweise neuer Ansatz aus diesem Fachbereich ist der von Maxim Gumin entwickelte Wave Function Collapse (WFC) Algorithmus [7]. WFC ist ein Algorithmus aus dem Bereich der Textursynthese, der es erlaubt, anhand von kleinen Beispielbitmaps beliebig große unterschiedliche Texturen im selben Stil zu synthetisieren. Auch wenn WFC zunächst für zwei Dimensionen entwickelt wurde, wie es in der Textursynthese üblich ist, kann er theoretisch auch im dreidimensionalen Raum angewendet werden. Aufgrund der guten Ergebnisse des WFC bei der Textursynthese lässt sich also die Hypothese stellen, dass dieser auch im dreidimensionalen Raum, also bei der Modellsynthese gute Ergebnisse erzielen könnte. Bei einer erfolgreichen Implementierung wäre es also möglich, komplexe 3D-Modelle anhand von kleineren nutzergenerierten Beispielmodellen zu synthetisieren und damit eventuell eine große Zeit- und Kostenersparnis für die Entwicklung von 3D-Contentbasierten Medien zu erreichen.

## 1.2 Problemstellung

Im Speziellen ist das Ziel dieser Arbeit, eine Erweiterung des WFC Algorithmus für die Generierung von 3D-Voxel-Modellen anhand von nutzergenerierten Beispielmodellen zu entwickeln und im Rahmen dessen die theoretischen Hintergründe des WFC, sowie verwandte Themengebiete wie Prozedurale Content Generierung, Textursynthese, Modellsynthese etc. zusammenzufassen und zu beleuchten. Dabei wird ein funktionsfähiger Prototyp entwickelt, der es erlaubt, ein Voxel-Modell als Vorlage zu importieren und nach Angabe verschiedener Parameter in der Lage ist ein komplett neues Modell zu generieren, das nach subjektiven und objektiven Kriterien dem Stil des Beispielmodells entspricht. Es soll wie bei der 2D-Version des WFC auch möglich sein, bedeutend größere Modelle als das Beispielmodell zu generieren.



(a) Beispielmodell

(b) Generiertes Modell

Abbildung 1.1: Beispiel für ein Ergebnis aus dieser Arbeit. Das Beispielmodell (links) wird als Input verwendet um ein größeres Modell (rechts) im selben Stil zu generieren.

Um dieses Ziel zu erreichen, wird das importierte Voxel-Modell zunächst in dreidimensionale Muster der Größe $N \times N \times N$ aufgeteilt, wobei N vom Nutzer als Parameter gewählt werden kann und der Größe von erkennbaren Mustern im Beispielmodell entsprechen sollte. Diese

Muster werden katalogisiert und ihre Häufigkeit gezählt. Daraufhin werden anhand der Position der Muster im Beispielmodell Regeln abgeleitet, welche Muster in welcher Richtung benachbart sein dürfen, diese Regeln werden auch *Adjacency Constraints* genannt. Danach wird der in Kapitel 2.4.1 beschriebene Teil des WFC angewendet, um ein Output-Modell in der gewünschten Größe zu generieren, welches diese Adjacency Constraints erfüllt. Grob erklärt wird dazu mit jeder Iteration des WFC eine Position im Output-Modell ausgewählt und für diese *Zelle* ein konkretes Muster bestimmt. Anschließend werden die Auswirkungen dieser Auswahl auf benachbarte Zellen propagiert. Dadurch verringert sich die Anzahl der möglichen Muster in den betroffenen Zellen. Dieser Prozess wird wiederholt, bis alle Zellen genau ein konkretes Muster beinhalten oder ein Widerspruch, also ein Zustand, in dem für eine Zelle kein Muster mehr möglich ist, eintritt.

Abbildung 1.2: Das Input-Modell wird in $3 \times 3$ Muster aufgeteilt

### Neue Ansätze

Gerade im ersten Teil dieses Ablaufs, der Findung der Muster und Adjacency Constraints zeigt diese Arbeit einige neue Ansätze: Um rotierte Versionen der Muster mitzubeachten, wird jeweils das gesamte Beispielmodell gedreht und als zusätzliches Modell verarbeitet, statt jedes Muster einzeln zu rotieren. Dies bietet den Vorteil, dass die Findung der Adjacency Constraints der rotierten Muster vereinfacht wird und nicht als Spezialfall behandelt werden muss. Außerdem wird dem Input- und Output-Modell optional eine *Polsterung* mit leeren Mustern, auch *Padding* genannt, um die Ränder hinzugefügt, sowie ein spezielles *Floor-Pattern* eingeführt, das es ermöglicht, den Kontakt zum Boden als Adjacency Constraint

zu definieren. Das optionale Padding verringert die Wahrscheinlichkeit für ein Fehlschlagen des Algorithmus, da es Mustern am Rand des Inputs ermöglicht im Output mit leeren Mustern benachbart zu sein. Bevor der Algorithmus in seiner klassischen Form startet, werden alle Auswirkungen dieser Rand-Muster durch das gesamte Output-Grid propagiert. Diese Erweiterung ermöglicht wesentlich bessere Ergebnisse für Modelle, in denen das Verhältnis von Objekten zum Boden und zum leeren Raum relevant ist. Des Weiteren wird eine einfache Methode zum Backtracking im WFC Algorithmus vorgestellt, um die Geschwindigkeit und Erfolgsrate der Generierung zu verbessern.

## 1.3 Ziele

Zusammengefasst verfolgt diese Arbeit die folgenden Ziele:

- Entwicklung einer angepassten Version des WFC Algorithmus um 3D-Voxel-Modelle im Stil eines Beispielmodells zu generieren
- Entwicklung einer Anwendung mit den folgenden Features:
  - Import und Export von Voxel-Modellen im .vox Format
  - Darstellung von Voxel-Modellen in 3D
  - Grafische Oberfläche, die es erlaubt, Parameter des WFC Algorithmus zu verändern
  - Generierung und Darstellung von Modellen anhand eines importierten Modells mit dem angepassten WFC
- Zusammenfassung des aktuellen Forschungsstands zum Thema WFC und verwandten Themengebieten
- Vergleich des in dieser Arbeit entwickelten Ansatzes mit vergleichbaren Methoden zur beispielbasierten Generierung von 3D-(Voxel)-Modellen
- Identifikation möglicher Anwendungsgebiete

# 2 Stand der Technik

Im folgenden Kapitel werden für diese Arbeit relevante theoretische Grundlagen zusammengefasst, sowie ein Blick auf den aktuellen Stand der Technik in verschiedenen, mit dem Thema dieser Arbeit verwandten Bereichen der Informatik geworfen. Es werden außerdem einige Arbeiten genauer beleuchtet, die entweder als Voraussetzung oder als paralleler Lösungsansatz für die Problemstellung dieser Arbeit gesehen werden können.

## 2.1 Prozedurale Content Generierung

Als Prozedurale Content Generierung (PCG) bezeichnet man die automatische Generierung von Inhalten in digitalen Medien mithilfe von Algorithmen bzw. Prozeduren [29]. Diese Inhalte können grafischer Natur sein, wie z. B. die automatische Generierung von 3D-Modellen in Videospielen oder Filmen, aber auch akustisch, konzeptuell o. Ä. Entscheidend ist nur, dass Inhalte mit begrenztem oder gerichteten menschlichem Input [12] mithilfe von dafür spezialisierten Algorithmen generiert werden und diese vorher festgelegten Anforderungen entsprechen.

Ein bekanntes Beispiel aus der Videospiel-Industrie ist das Sandbox-Survival-Spiel Minecraft[1]. Hier wird PCG zur Generierung von prozeduralen Voxel-Landschaften genutzt. Im Speziellen wird eine Kombination aus 3D-Perlin-Noise [20] und Whittaker Diagrammen verwendet [32] um verschiedenste Variationen von Terrain und Biomen zu generieren.

[1] https://www.minecraft.net/ (abgerufen: 03.09.2020)

Abbildung 2.1: Beispiel einer prozedural generierten Voxel Landschaft in Minecraft [2]

An diesem Beispiel zeigt sich wie die Kombination von zwei relativ einfachen Konzepten, dem Perlin Noise und der Whittaker Diagramme, sehr komplexe und ansprechende Ergebnisse erzielt werden können.

Natürlich umfasst der Begriff der Prozeduralen Generierung eine große Menge an verschiedenen Techniken die sich aufgrund der Popularität des Themas ständig weiter entwickeln. Auch das Thema dieser Arbeit lässt sich nach der obigen Definition als eine Art der Prozedurale Content Generierung verstehen.

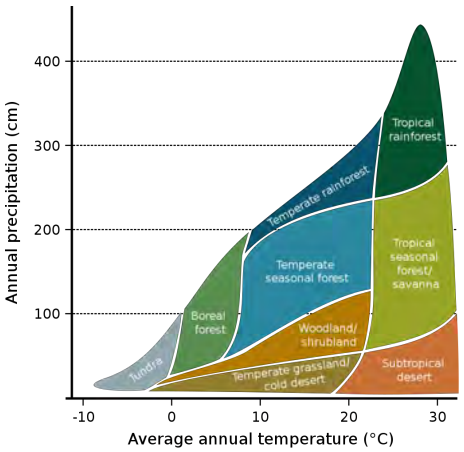



Abbildung 2.2: Ein Whittaker Diagramm [3]

Im Folgenden werden beispielhaft einige dieser Techniken vorgestellt:

[2] Bildquelle: https://pixabay.com/illustrations/minecraft-world-forest-773807/ (abgerufen: 12.09.20)
[3] Bildquelle: https://en.wikipedia.org/wiki/Biome (abgerufen: 12.09.20)

### 2.1.1 L-Systeme

L-Systeme oder Lindenmayer Systeme wurden erstmals 1967 vom Biologen Aristid Lindenmayer eingeführt, um die Wachstumsmuster von Pflanzen in einem mathematischen Modell zu formalisieren [15]. Zusammen mit Przemyslaw Prusinkiewicz stellt Lindenmayer in seiner Arbeit *Graphical modeling using L-systems* [21] Möglichkeiten vor, diese Systeme zur Modellierung von grafischen Objekten zu nutzen. Grundsätzlich können L-Systeme als Satz von Regeln zur Ersetzung oder Transformation von bestimmten Symbolen oder Formen gesehen werden. Jede dieser Regeln beschreibt dabei die Transformation eines bestimmten Symbols aus einem endlichen Alphabet $V$ in ein anderes (oder mehrere). Die Menge dieser Transformation wird als die Folge P notiert. Außerdem wird ein Startwort $w$ definiert, dass aus Symbolen aus $V$ besteht. Ein Beispiel für eine Solches L-System (G) wäre z. B. :

$$G =< V, w, P > \tag{2.1}$$

mit

$$\begin{aligned} V &: (a, b) \\ w &: a \\ p_1 &: b \to a \\ p_2 &: a \to ab \end{aligned} \tag{2.2}$$

Mit diesem L-System ließen sich beispielsweise Wörter wie *aab* (Anwendung von $p_2 \to p_1 \to p_2$) generieren.

Interessant für die prozedurale Generierung werden die L-Systeme nun, wenn die Symbole des Alphabets jeweils für bestimmte grafische Formen oder Operationen stehen. Ein Beispiel dafür sind die sogenannten *turtle graphics* [3], die aus dem gleichnamigen Grafikprogramm der Programmiersprache Logo kommen. In diesem System gibt es verschiedene Symbole, die jeweils ein bestimmtes grafisches Kommando repräsentieren:

- F : Bewege dich um d Einheiten nach vorne und zeichne eine Linie
- f : Bewege dich um d Einheiten nach vorne, ohne eine Linie zu ziehen
- $+$ : Rotierte um den Winkel $\delta$ nach links
- $-$ : Rotierte um den Winkel $\delta$ nach rechts

d und δ sind dabei Konstanten, die dem Algorithmus als Parameter mitgegeben werden.

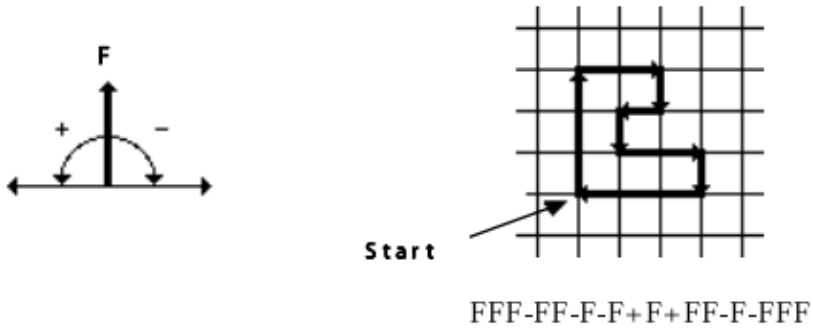



Abbildung 2.3: Die turtle graphics Symbole werden in eine Zeichnung umgesetzt [21]

Mit diesem einfachen Ansatz ist es nun möglich, diverse L-Systeme zu definieren, indem verschiedene Regeln zur Ersetzung P und Startwörter *w* aus dem Alphabet $\{F, f, +.-\}$ gewählt werden, die interessante zweidimensionale Muster generieren können.

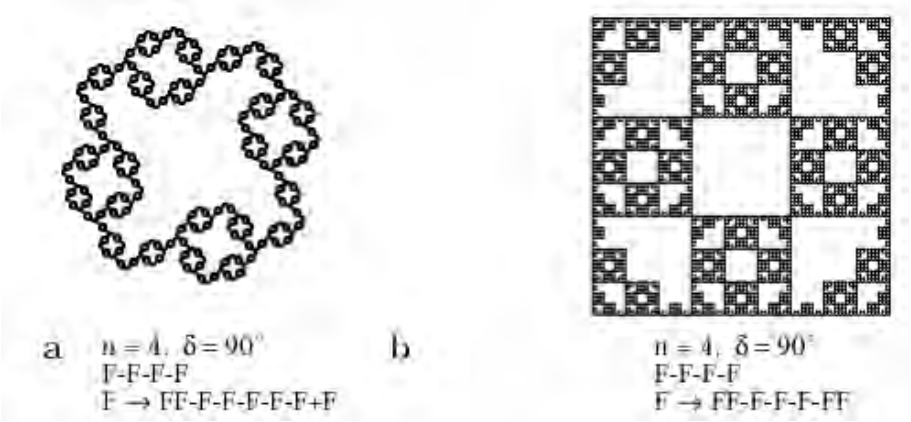



Abbildung 2.4: Besipiele für einfache L-Systeme mit dem turtle graphics Alphabet [21]

Je mehr Symbole und Parameter dem L-System hinzugefügt werden, desto komplexere Ergebnisse lassen sich erzielen. So lassen sich z. B. durch das Hinzufügen von nur zwei neuen Symbolen wesentlich komplexere Strukturen generieren:

- [ : Speichere den aktuellen Zustand der Turtle (Position, Ausrichtung usw.) in einem Last-In-First-Out (LIFO)-Speicher
- ] : Stelle den obersten Zustand der Turtle aus dem LIFO-Speicher wieder her

Diese Variante der L-Systeme werden auch *Bracketed L-Systeme genannt*.

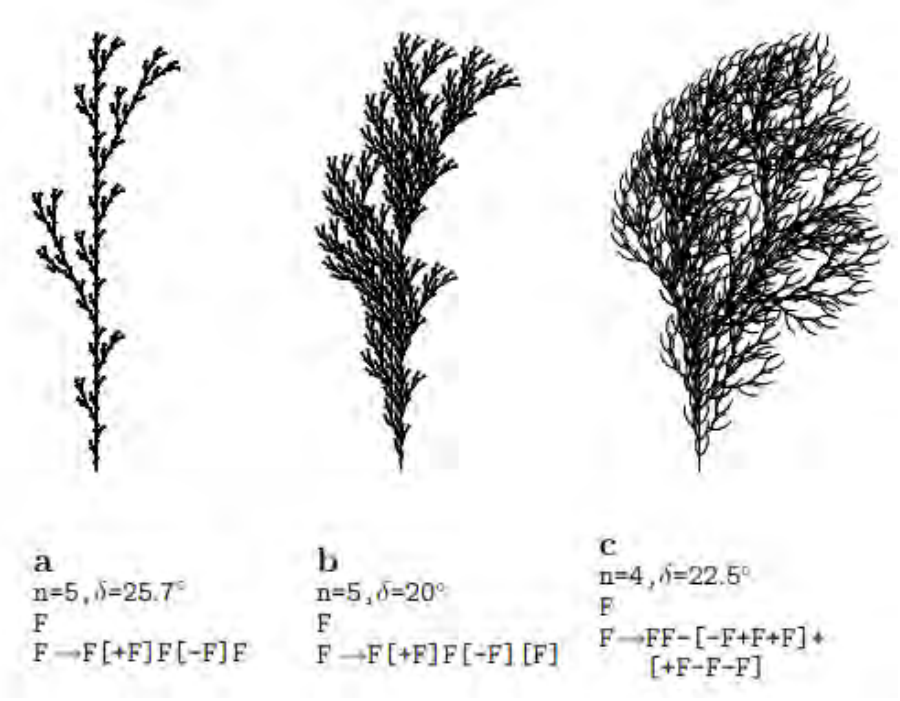



Abbildung 2.5: Beispiele für Bracketed L-Systeme [21] mit je verschiedenen Parametern für (a), (b) und (c)

Die Komplexität und Varianz der erzeugten Bilder bzw. Modelle kann durch die Erweiterung mit zusätzlichen Symbolen und anderen Techniken, sowie der Einführung von weiteren Parametern, beliebig weit in die Höhe getrieben werden. Heutzutage werden L-Systeme z. B. über Technologien wie *Speed Tree* [25] in vielen modernen Spielen genutzt um Pflanzen und Vegetation zu generieren.

### 2.1.2 Shape Grammars

Der Begriff der *Shape Grammars* wurde das erste Mal 1971 von George Stiny und James Gips in ihrer Arbeit *Shape Grammars and Generative Specification* [28], als regelbasiertes System zur Beschreibung und Erstellung zweidimensionalen Grafiken eingeführt. Ähnlich wie L-Systeme sind die Shape Grammars ein grammatikbasiertes System mit der Besonderheit, dass statt Symbolen zwei oder auch dreidimensionale Formen verwendet werden. Um eine Shape Grammar zu definieren, müssen zunächst die sogenannten Terminalsymbole

festgelegt werden. Dies sind Symbole bzw. Formen, die nicht weiter in eine andere Form transformiert werden können, also die kleinsten *Bausteine* der Grammatik. Dann wird ein Satz von Regeln bestimmt, die die Transformation einer Form in eine andere bestimmen. Schließlich muss noch eine initiale Form festgelegt werden, mit der der Generationsprozess startet. Formal kann eine Shape Grammar SG als das folgende 4-Tupel definiert werden:

$$SG = \{V_T, V_M, R, I\} \tag{2.3}$$

- $V_T$ : Eine begrenzte Menge an Formen, die nicht weiter transformiert werden können (Terminalsymbole)
- $V_T{}^*$ : Die Menge der möglichen Formen die durch die Kombination von Elementen aus $V_T$ in verschiedenen Ausrichtungen und Kombinationen entstehen können
- $V_M$ : Die Menge der nicht-terminalen Formen, für die gilt $V_T{}^* \cap V_M = \emptyset$
- R : Transformations-Regeln definiert als Menge von Tupeln $(u, v)$ wobei $u$ die Ausgangsform und $v$ die neue Form repräsentiert
- I : Die initiale Form

Sind alle nötigen Definitionen erfolgt, beginnt der Generationsprozess mit der initialen Form I und wendet rekursiv mögliche Regeln aus R an, bis nur noch Terminalsymbole vorhanden sind, also keine Regel aus R mehr angewendet werden kann. Bei der Anwendung der Regeln werden auch die Rotation und Skalierung der Ausgangsform beachtet und entsprechend auf die Ergebnisform angewendet.

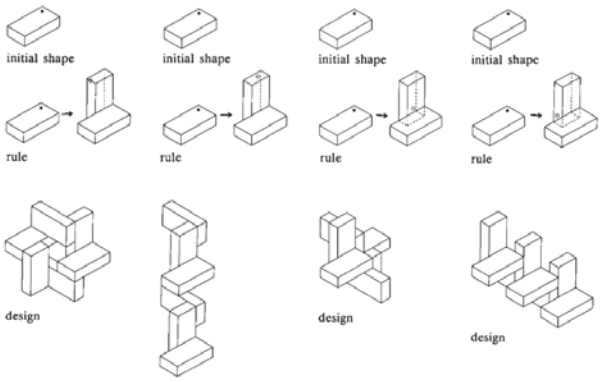



Abbildung 2.6: Beispiel für möglichen Designs die sich aus verschiedenen einfachen Shape Grammars ableiten lassen [4]

## 2.2 Textursynthese

Das der Computergrafik angehörige Feld der Textursynthese beschäftigt sich mit dem Problem der automatischen Erzeugung von Texturen, anhand eines lokal begrenzten Beispiels bzw. Samples. Dabei wird angenommen, dass eine beliebig große Textur existiert (größer als das Sample), deren Teile strukturelle Ähnlichkeiten zueinander aufweisen und dann durch verschiedene Methoden bestimmte Regeln aus dem Sample inferiert, die genutzt werden können, um neue Samples zu generieren [4].

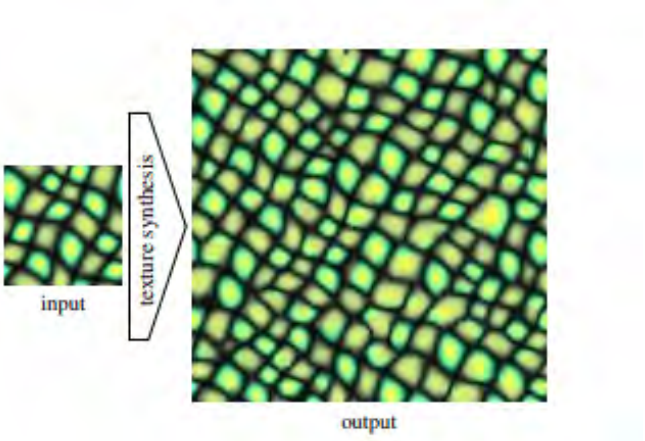



Abbildung 2.7: Ergebnis eines Textursynthese-Algorithmus [30]

Es gibt verschiedene Ansätze, das Problem der Textursynthese zu bearbeiten, jedoch bauen die meisten davon auf die im Folgenden beleuchtete Arbeit auf:

### 2.2.1 Textursynthese durch nicht-parameterisierte Stichproben

In ihrer Arbeit *Texture Synthesis by Non-parametric Sampling* [4] von 1999 stellen die Autoren ein vergleichsweise einfaches, aber effektives Verfahren zur Textursynthese vor. Ihr Ansatz basiert auf der Idee von C.E. Shannon [23] englische klingende Wörter mithilfe von Markov-Ketten zu generieren, indem durch die Analyse von großen Mengen an Beispieltext eine Wahrscheinlichkeitsverteilung für den nächsten Buchstaben für alle möglichen $n$-Gramme (Kombinationen von $n$ Buchstaben) errechnet wird. Diese Idee übertragen die Autoren auf die zweidimensionale Struktur einer Textur:

[4] Bildquelle: https://www.ocw.mit.edu/courses/architecture/4-520-computational-design-i-theory-and-applications-fall-2005/lecture-notes/lect8.pdf (abgerufen: 14.09.20)

Eine Textur wird als *Markov Random Field* (MRF) modelliert, es wird also angenommen, dass der Farbwert eines Pixels unabhängig von den Farbwerten von Pixeln außerhalb seiner lokalen Nachbarschaft ist. Die lokale Nachbarschaft ist ein quadratisches Fenster um das betrachtete Pixel, dessen Größe als Parameter $k$ mitgegeben werden kann. Bei der Synthese startet der Algorithmus mit einem zufälligen Wert in der Mitte des Output-Bildes und *wächst* nach außen. Dabei werden für jedes neue Pixel alle bereits synthetisierten Pixel in seiner lokalen Nachbarschaft betrachtet, um im Input-Bild die lokale Nachbarschaft (Quadrat aus $k \times k$ Pixeln) zu finden, die der des zu synthetisierenden Pixels am ähnlichsten ist. Dann wird lediglich der Farbwert des Pixels in der Mitte dieser Nachbarschaft (aus dem Input-Bild) als Wert für das Ziel-Pixel verwendet. Dieser Prozess wiederholt sich bis jedes Pixel einen Wert hat.

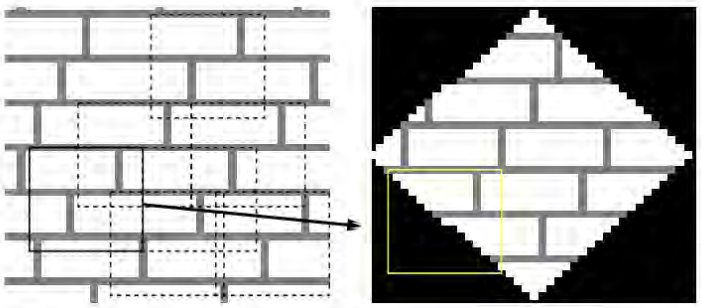


Abbildung 2.8: Soll ein neues Pixel im Output-Bild(rechts) synthetisiert werden, wird dafür die lokale Nachbarschaft des Pixels(gelbes Quadrat) mit den Nachbarschaften aus dem Input-Bild(links) verglichen und schließlich das zentrale Pixel der ähnlichsten Nachbarschaft als Wert gewählt [4].

Es gibt bereits verschiedene Verbesserungen dieses Algorithmus; Li-Yi Wei und Marc Levoy zeigen in *Fast Texture Synthesis using Tree-structured Vector Quantization* zum Beispiel einen Ansatz zur Verbesserung der Performance des Algorithmus [31].

## 2.3 Modellsynthese

Analog zur Textursynthese beschäftigt sich die Modellsynthese mit der Generierung von 3D-Modellen anhand von Beispielmodellen. Der Begriff wurde 2009 von Paul Merrel in seiner gleichnamigen Dissertation [18] eingeführt:

### 2.3.1 Model Synthesis (Paul Merrel)

Paul Merrel zeigt in seiner Dissertation *Model Synthesis* [18] von 2009 einen neuartigen Ansatz, um 3D-Modelle anhand von Beispielen zu generieren. Inspiriert von Prinzipien aus der Textursynthese und der Prozeduralen Modellierung werden zwei verschiedene Ansätze vorgestellt: *Discrete Model Synthesis* und *Continuous Model Synthesis*:

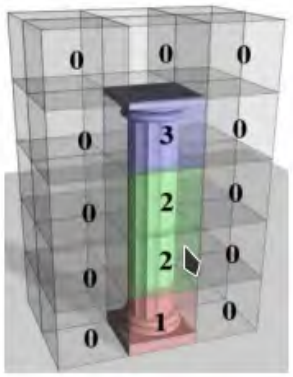



Abbildung 2.9: Bausteine und Label

Im **diskreten** Modell wird das Besipiel-Modell vom Benutzer zunächst in diskrete Bausteine von gleicher Dimension aufgeteilt. Jeder dieser Bausteine bekommt ein einzigartiges *Label*, sodass jedem Punkt im dreidimensionalen Raum des Beispielmodells ein solches Label zugewiesen werden kann. Schließlich werden daraus Regeln abgeleitet, welche Label neben welchen anderen Labeln erlaubt sind. Diese Regeln werden *Adjacency Constraints* genannt. Das Ziel der Modellsynthese ist es, aus diesen mit Labeln versehenen *Bausteinen* ein neues Modell zu generieren, dass diese Adjacency Constraints erfüllt. Ähnlich wie beim WFC, unterscheidet sich die Modellsynthese damit von der Textursythese, in der Adjacency Constraints nicht immer erfüllt sein müssen. Dieser diskrete Ansatz funktioniert gut, wenn sich die Bausteine des Beispielmodells auf ein Gitter von fester Größe anordnen lassen, dies ist allerdings nicht immer der Fall.

Um dieses Problem zu lösen, werden dazu bei der **Continuous Model Synthesis** auch Positionen beachtet, die sich nicht auf einem einfachen Gitter anordnen lassen. Da es unendlich viele solcher Punkte im dreidimensionalen Raum gibt, wird zunächst folgende Annahme getroffen: Alle Flächen des Modells liegen auf einer begrenzten Menge von Ebenen[5]. Innerhalb dieser Beschränkung gibt es nun eine begrenze Anzahl von möglichen Positionen für Kanten (an Schnittpunkten von zwei Ebenen) und Vertices (an Schnittpunkten von drei Ebenen). So können schließlich die einzelnen Kanten und Vertices des Beispielmodells mit Labeln versehen werden und ihre möglichen benachbarten Labels bestimmt werden.

[5] parallel plane assumption genannt

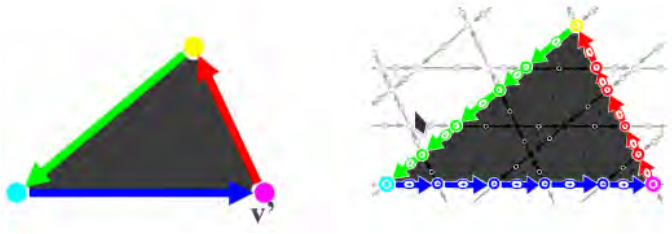
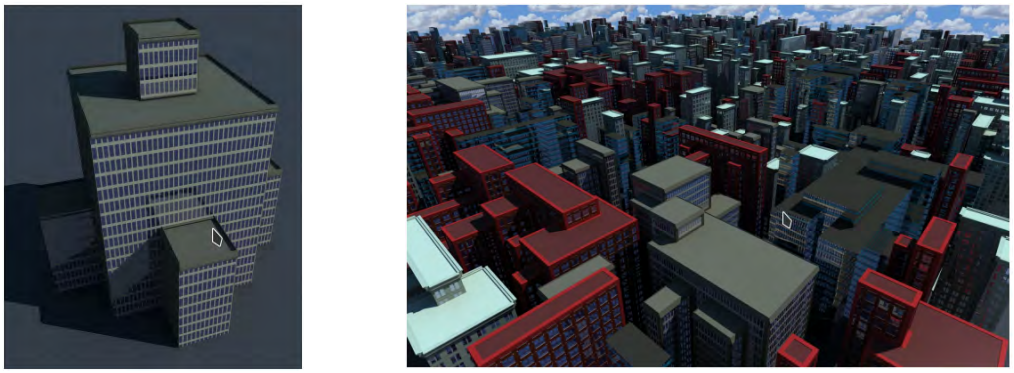


Abbildung 2.10: Beispiel-Form wird von parallelen Linien aufgeteilt

Der Ablauf des eigentlichen Algorithmus zu Generierung des Modells ist bei der diskreten und der *kontinuierlichen* Variante identisch:

1. Zunächst wird ein Katalog aller möglichen Labels für jede Position im Output initialisiert
2. Ein zufälliges Label wird für eine zufällige Position ausgesucht
3. Alle nun nicht mehr möglichen Labels werden aus angrenzenden Positionen gelöscht (wird rekursiv durch alle relevanten Position propagiert)
4. Schritt 2 und 3 werden wiederholt, bis jede Position ein konkretes Label besitzt und schließlich das generierte Modell ausgegeben



(a) Beispielmodell

(b) Generiertes Modell

Abbildung 2.11: Beispiel für die Ergebnisse der Modellsynthese

Wie in Abbildung 2.11 zu sehen, liefert die Modellsynthese beeindruckende **Ergebnisse** und erlaubt es, große und abwechslungsreiche 3D-Modelle aus einem einfachen Beispiel zu generieren, ohne dabei die Ähnlichkeit zu diesem zu verlieren. Die diskrete Variante funktioniert gut, wenn das Beispielmodell sich auf ein Gitter aufteilen lässt, sonst muss die *continuous*

Variante gewählt werden. Diese ist nur durch die *parallel planes assumption* eingeschränkt. Es muss also bei der Wahl von Beispielmodellen darauf geachtet werden, dass sich die Flächen dieses Modells auf einer begrenzten Anzahl von parallelen Ebenen befinden um gute Ergebnisse erwarten zu können. Paul Merrel entwickelte mit dieser Arbeit den ersten Algorithmus zur beispielbasierten prozeduralen Generierung von 3D-Modellen und legte damit den Grundstein für viele andere Arbeiten, wie z. B. den WFC.

## 2.4 Wave Function Collapse

Der 2016 von Maxim Gumin veröffentlichte *Wave Function Collapse* Algorithmus wurde entwickelt, um Bilder anhand von kleineren Beispielen zu generieren, die lokale Ähnlichkeit zu diesen besitzen. Lokale Ähnlichkeit bedeutet, dass jedes NxN Muster von Pixeln im Output-Bild mindestens einmal im Input-Bild zu finden ist. Daraus folgend haben die generierten Bilder denselben *Stil* wie das Input-Bild. Als Inspiration für die Arbeitsweise und den Namen des Algorithmus diente das gleichnamige Konzept aus der Quantenmechanik: Die Wellenfunktion beschreibt dort den Zustand eines Quantensystems, genauer kann sie als Wahrscheinlichkeitsverteilung für mögliche Zustände gesehen werden. Wird diese Wellenfunktion nun durch Beobachtung *kollabiert*, reduzieren sich die möglichen Zustände auf einen konkreten Zustand [34]. Max Gumin nutzt diese Idee, um die möglichen Farbwerte der Pixel im Output-Bild als Wellenfunktion darzustellen. Im Verlauf der Generierung werden dann nach und nach einzelne Pixel *kollabiert* und die daraus folgenden Konsequenzen für die möglichen Zustände benachbarter Pixel propagiert, bis jeder Pixel nur noch einen konkreten Wert hat [7].

WFC kombiniert diese Idee außerdem mit Erkenntnissen aus den Bereichen der Textursynthese, des Constraint Solving und der Prozeduralen Generierung. WFC weist jedoch entscheidende Unterschiede zu anderen Textursynthese Algorithmen auf [9].

Der WFC Algorithmus löst in seiner klassischen Form vier grundlegende Aufgaben: [9]

1. Extraktion von lokalen Mustern aus dem Input-Bild
2. Erkennung von Adjacency Constraints zwischen Mustern durch Analyse des Input-Bildes
3. Inkrementelle Generierung des Output-Bildes unter Beachtung der erkannten Constraints
4. Ausgabe des Output-Bildes im selben Format wie das Beispiel

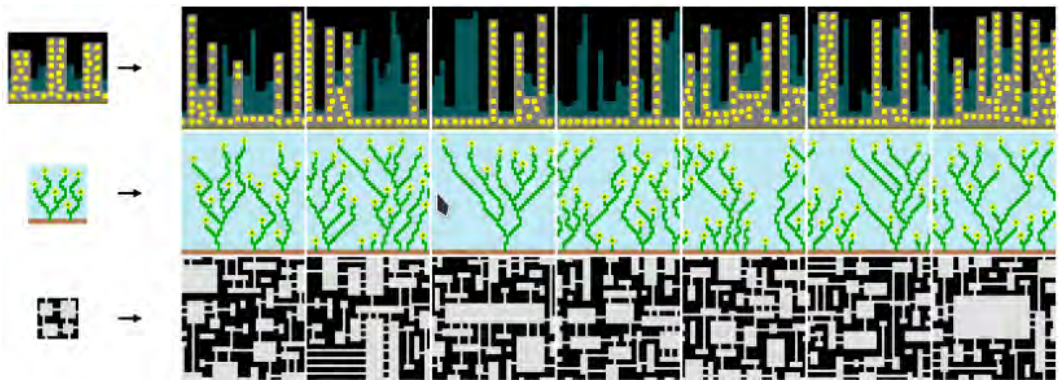


Abbildung 2.12: Beispiele für generierte Bilder (rechts) aus kleineren Input-Bildern (links) von Max Gumins offizieler Implementierung [7]

Die Extraktion der Muster und Erkennung der Adjacency Constraints ist in Tilemap-basierten Variante des Algorithmus nicht nötig (siehe Simple Tiled Model). In der klassischen Version, dem Overlapping Model, wird in (1) und (2) das Input-Bild systematisch durchsucht, um alle möglichen N $\times$ N Muster zu extrahieren. Dabei wird die Frequenz, mit der die einzelnen Muster im Input-Bild vorkommen, gespeichert und später genutzt, um die Wahrscheinlichkeit zu bestimmen, mit der das jeweilige Muster beim Kollabieren ausgewählt wird. Danach werden die Pixel-Werte der Muster miteinander verglichen, um festzustellen, ob zwei Muster beim Überlappen dieselben Farbwerte für die überlappenden Pixel aufweisen. Ist dies der Fall, wird das jeweilige Muster als möglicher Nachbar gespeichert. Dieser Vorgang findet für alle möglichen Verschiebungen der Muster statt, bei denen eine Überlappung existiert. Diese Informationen werden in (3) dann zur Generierung des Output-Bildes genutzt [9].

### 2.4.1 Der Algorithmus

Ablauf des Algorithmus wie ursprünglich von Max Gumin beschrieben: [7]

**Wave Function Collapse Algorithmus**

1. Lesen des Input-Bildes und Zählen der $N \times N$ Muster
   a) (optional) Rotierte und reflektierte Versionen der Muster hinzufügen
2. Erstellung eines Arrays mit den gewünschten Dimensionen des Output-Bildes (*Wave* genannt). Jedes Element repräsentiert den Zustand einer $N \times N$ Region des Output-Bildes. Der Zustand beinhaltet alle noch möglichen Muster an dieser Position der Wave und wird durch Boolean-Koeffizienten für alle möglichen Muster modelliert. `false` bedeutet das jeweilige Muster ist an dieser Position verboten, `true` es ist noch erlaubt.
3. Initialisierung der Wave im komplett unbeobachteten Zustand, d. h. alle Koeffizienten sind `true`
4. Wiederholung der folgenden Schritte
   a) Beobachtung:
      i. Finden des Elements mit dem kleinsten Entropiewert der nicht Null ist. Wenn keines gefunden werden kann, beende die Schleife (4) und gehe zu (5)
      ii. Kollabiere dieses Element in einen konkreten Zustand unter Beachtung seiner Koeffizienten und der Häufigkeitsverteilung der Muster
   b) Propagierung: Propagierung der Auswirkung dieser Änderung an benachbarte Elemente
5. Nun sind entweder alle Elemente der Wave in einem konkreten Zustand (genau ein Koeffizient ist `true`) oder in einem Zustand des Widerspruchs (alle Koeffizienten sind `false`). Im ersten Fall wird das Ergebnis ausgegeben. Im zweiten Fall beende den Algorithmus ohne Ergebnis.

Diese relativ allgemein gehaltene Beschreibung des WFC, illustriert die Arbeitsweise des Algorithmus bei der Generierung, lässt jedoch offen, welche Methode zur Feststellung von Adjacency Constraints der Muster verwendet wird. Das Kernstück des Algorithmus ist dabei die Schleife (4). Mit jeder Iteration werden hier zwei grundlegende Schritte durchgeführt: Beobachtung (4a) und Propagierung (4b).

Im Beobachtungsschritt wird zunächst das Element der Welle mit dem kleinsten Entropiewert ungleich null gesucht. Entropie ist in diesem Fall ein Maß dafür, wie stark ein Element

bereits eingeschränkt ist, oder vereinfacht, wie viele Muster an dieser Position der Wave noch möglich sind, gewichtet durch die Frequenz der jeweiligen Muster. Je weniger Möglichkeiten existieren, desto niedriger die Entropie. Diese Heuristik (Minimum-Entropy-Heuristik [14]) zur Auswahl des nächsten Elements wurde von Max Gumin bewusst gewählt, nachdem er beobachtete, dass Menschen bei der Lösung ähnlicher Probleme zu einem Vorgehen neigen, das dieser Heuristik ähnelt [7]. Ist nun ein Element ausgewählt, wird dieses in einen konkreten Zustand kollabiert, d. h. eines der noch möglichen Muster wird ausgewählt. Wurde kein Element gefunden, wird die Schleife verlassen.

Im Propagierungsschritt werden die neu gewonnenen Informationen ausgehend vom gewählten Objekt durch die Wave propagiert. Dabei wird für jedes benachbarte Element anhand der vorher festgestellten Adjacency Constraints der Muster berechnet, welche Muster in benachbarten Elementen aufgrund der Kollabierung des gewählten Elements nun nicht mehr möglich sind. Verursacht dies eine Änderung im Zustand des jeweiligen Elements, wird auch diese Änderung an dessen Nachbarn propagiert.

### 2.4.2 Simple Tiled Model

Diese Variante des WFC verzichtet auf die Analyse eines einzelnen Input-Bildes und überlässt die Aufgabe der Definition von Mustern stattdessen dem Anwender des WFC. Wie der Name vermuten lässt, kann als Eingabe hier eine Tilemap, bestehend aus mehreren gleichgroßen Bildern oder Tiles, die in bestimmten Ausrichtungen zusammenpassen, inklusive der vom Anwender definierten Adjacency Constraints genutzt werden [7].

### 2.4.3 Overlapping Model

Für das Overlapping Model werden die möglichen Muster und ihre Adjacency Constraints aus dem Input-Bild inferiert und optional um rotierte und reflektierte Versionen ergänzt. Eine hilfreiche Vorstellung für den Ablauf dieses Prozesses lässt sich folgendermaßen skizzieren: Ein Quadrat der Größe $N \times N$ mit $N$ als Größe der Muster läuft Pixel für Pixel über alle möglichen Positionen dieses Quadrates auf dem Input-Bild. An jeder neuen Position wird dann das Muster innerhalb des Quadrates als neues Muster gespeichert.

Der nächste Schritt besteht darin, alle möglichen Konfigurationen zu überprüfen, in denen diese Muster überlappen können und daraus zu schließen, welche Muster in welcher Rich-

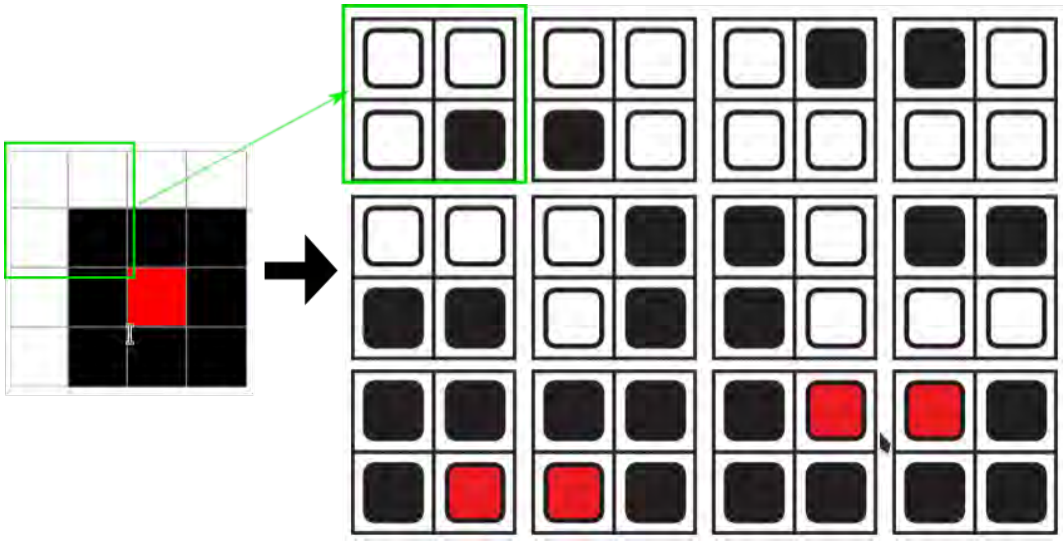


Abbildung 2.13: Zeigt alle möglichen 2 × 2 Muster (rechts) inklusive Rotation und Reflexion die aus dem 4 × 4 Pixel großen Input-Bild generiert werden können. (Zusammengesetzt aus Grafiken aus: [9])

tung benachbart sein dürfen. Zwei Muster in einer bestimmten Konfiguration passen zusammen, wenn alle überlappenden Pixel dieselben Farbwerte haben [9].

### 2.4.4 Unterschiede zu klassischen Textursynthese-Algorithmen

Der entscheidende Unterschied zu anderen Textursynthese Algorithmen besteht in der strikten Erhaltung von Adjacency Constraints aus dem Input-Muster, da zum einen keine Interpolation von Farbwerten erlaubt wird und zum anderen bei einem Widerspruch in Bezug auf Adjacency Constraints der Algorithmus abgebrochen wird. Das hat zwar den Nachteil, dass WFC nicht immer zu einem Ergebnis kommt, bringt aber auch den signifikaten Vorteil mit sich, dass WFC in Bereichen außerhalb der Textursynthese angewendet werden kann, in denen eine strikte Einhaltung von bestimmten Adjacency Constraints gewünscht ist. Ein Beispiel dafür wäre die Levelgenerierung in Spielen oder Generierung von urbanen Landschaften aus vorher entworfenen Tiles.

## 2.5 Voxel Grafik

Der Volume-Pixel oder Voxel wird in der Computergrafik als Repräsentation für dreidimensionale *Volumen* auf einem festen 3D-Raster genutzt. Ein Voxel beschreibt dabei eine kubische Einheit von Volumen an einem diskreten Punkt dieses Rasters und kann damit als Gegenstück zum Pixel in der 2D-Grafik gesehen werden [10]. Dieses Raster wird in der Regel als 3D-Array modelliert, in dem jede Position Daten über den jeweiligen Voxel enthält. Da die Position des Voxels durch seine Position im Array impliziert wird, muss dieser seine Position nicht selbst speichern. Welche Daten für den jeweiligen Voxel gespeichert werden variiert je nach Anwendung. Im einfachsten Fall beschreibt eine Null oder Eins ob ein Voxel an der Position vorhanden ist. Häufig beinhalten diese Daten auch einen Farbwert oder Informationen über die Dichte und Oberflächenbeschaffenheit für den jeweiligen Voxel [11].

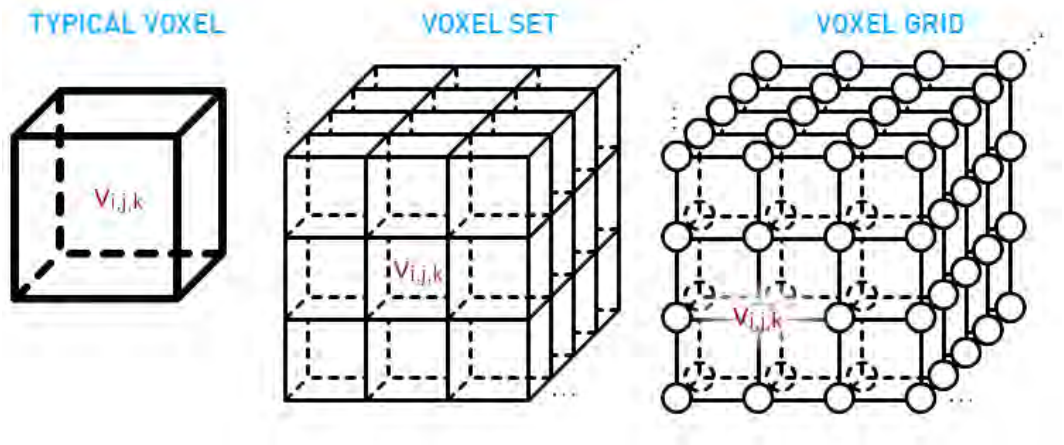



Abbildung 2.14: Ein Voxel beschreibt eine kubische Einheit von Volumen im Voxel Raster (Voxel Grid) an der Position $(i, j, k)$ [6]

[6] Bildquelle: http://www.aoktar.com/octane/OCTANE%20HELP%20MANUAL.html?WhatisVolumetrics.html (abgerufen: 20.09.20)

## 2.6 WFC zur 3D-Modellsynthese

### 2.6.1 Voxel Synthesis for Generative Design

Matvey Khokhlov, Immanuel Koh und Jeffrey Huang stellen in ihrer Arbeit *Voxel Synthesis for Generative Design* [13] von 2018 einen *Proof of Concept* für ein automatisiertes System vor, um Modelle aus dreidimensionalen Besipielmodellen (speziell Voxel-Modelle) zu synthetisieren. Dabei mischen die Autoren Erkenntnissen aus Paul Merrel's *Model Synthesis* [18] mit dem Wave Function Collapse Algorithmus [7].

Es werden zwei verschiedene Ansätze vorgestellt, die beide ihre Vor- und Nachteile aufweisen: das *Simple Model* und das *Convolutional Model*.

#### 2.6.1.1 Simple Model

Dieser Ansatz besitzt große Ähnlichkeit zu Paul Merrel's diskreter Modellsythese und dem Simple Tiled Model des WFC. So gibt es auch hier zwei grundlegende Phasen: die Initialisierung und den Beobachtungs- und Propagierungszyklus.

In der **Initialisierungsphase** wird das Input-Modell zunächst in $N \times N \times N$ große Voxel-Muster aufgeteilt, anders als beim Convolution Model werden hier tatsächlich nur die Muster beachtet, die durch das *Auseinanderschneiden* des Modells entstehen können. Bei einem $4 \times 4 \times 4$ Modell wären das z. B. $2 \times 2 \times 2$ Muster. Jedes Muster wird durch ein Label markiert, dabei erhalten mehrmals vorkommende Muster dasselbe Label. Schließlich werden Adjacency Constraints aus dem Input-Modell inferiert, indem für jedes Muster in den sechs verschiedenen Richtungen im dreidimensionalen Raum ($\pm X, \pm Y, \pm Z$) festgestellt wird, welches Muster in dieser Richtung benachbart ist. Dieses wird dann als möglicher Nachbar gespeichert. Optional kann zusätzlich noch eine Analyse der *Randvoxel* der Muster vorgenommen werden, um Muster, dessen Voxel in der jeweiligen Richtung übereinstimmen, als mögliche Nachbarn zu speichern. Abschließend wird ein dreidimensionales Array in der gewünschten Ausgabegröße initialisiert, das die möglichen Muster für die jeweilige Position im Output-Modell enthält. Jede Position kann zu Beginn alle Muster enthalten.

Der **Beobachtungs- und Propagierungszyklus** ist dem gleichnamigen Schritt im WFC sehr ähnlich. Zunächst wird eine Position im Ausgabearray gewählt und diese kollabiert (ein kon-

kretes Muster aus den Möglichen ausgewählt). Die Auswahl erfolgt anhand der Häufigkeitsverteilung der Muster sowie der Anzahl der noch verfügbaren Muster. Dieser Prozess nennt sich Beobachtung oder *Observation*. Darauf folgend wird die Information über nicht mehr mögliche Muster in alle sechs Richtungen durch das Ausgabearray propagiert und dort werden anhand von Adjacency Constraints die möglichen Muster eingeschränkt. Dieser Zyklus wiederholt sich schließlich für alle Positionen, an denen eine Änderung stattgefunden hat, bis das gesamte Modell generiert ist oder ein Widerspruch entdeckt wurde.

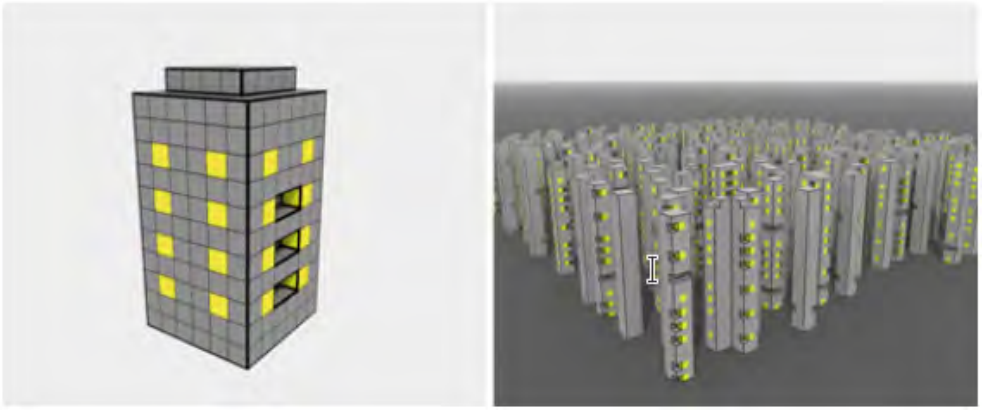


Abbildung 2.15: Anwendung des Simple Models, um eine Stadt aus einem einzelnen Haus zu generieren.

Das Simple Model scheint gute **Ergebnisse** für einfache Input-Modelle zu liefern. Wichtig ist, dass der richtige Wert N für die Größe der Muster gewählt wird, um ein gutes Ergebnis zu erzielen. Außerdem hilft der optionale Schritt des Vergleichs der Randvoxel, Widersprüche im Algorithmus zu vermeiden.

#### 2.6.1.2 Convolutional Model

Das Convolutional Model ist vom Overlapping Model des WFC inspiriert und versucht, dieses auf drei Dimensionen zu erweitern. Im Gegensatz zum Simple Model wird das Input-Modell hier nicht einfach *auseinandergeschnitten*, sondern auch die möglichen überlappenden Muster beachtet. Diese Überlappungen oder *Convolutions* werden dann genutzt, um eine Nachbarschaftsmatrix für jedes einzigartige Muster zu definieren. Durch diese Vorgehensweise ist das Ergebnis nicht nur eine Zusammensetzung von bekannten Mustern aus dem Input-Modell, sondern kann auch neuartige Muster enthalten, die trotzdem dem Stil des Beispiels entsprechen.

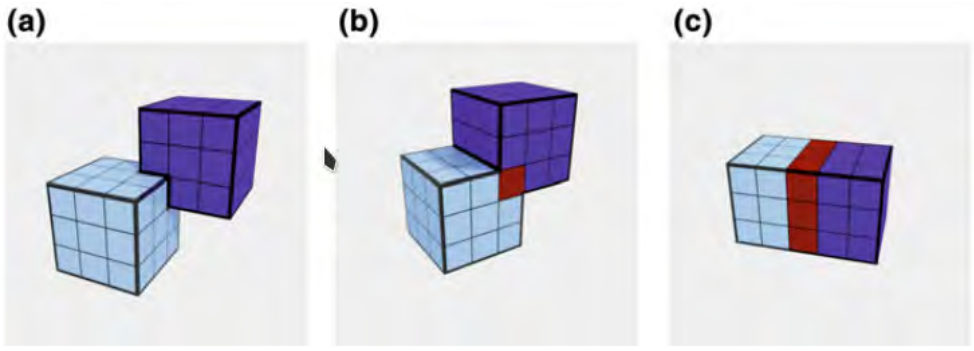



Abbildung 2.16: Verschiedene Überlappungen von zwei $3 \times 3 \times 3$ Mustern

Die **Initialisierungsphase** unterscheidet sich hier nur durch die unterschiedliche Vorgehensweise zur Muster- und Nachbarschaftserkennung. Auch der **Oberservations -und Propagierungszyklus** läuft wie beim Simple Model ab; mit dem einzigen Unterschied, dass nicht nur in sechs Richtungen geprüft und propagiert wird, sondern für alle möglichen Convolutions (abhängig von der Mustergröße).

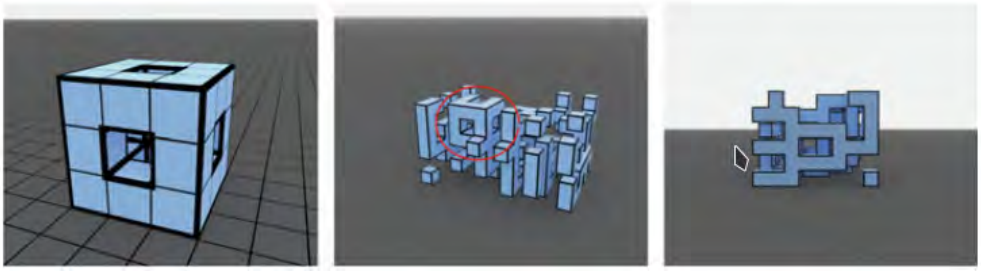


Abbildung 2.17: Ergebnis des Convolutional Models. Der rote Kreis zeigt wie sich Teile des Beispielmodells in der Ausgabe wieder finden

Die **Ergebnisse** des Convolutional Models zeigen, dass es mit diesem Ansatz möglich ist neue Variation von Mustern zu erzeugen, wenn ein simples Beispielmodell gewählt wird. Auch hier ist die Wahl des richtigen Wertes für die Mustergröße N extrem wichtig: Ist es zu klein, hat das Ergebnis meist wenig Ähnlichkeit zum Beispiel. Ist es zu groß, werden zwar mehr Eigenschaften des Beispiels festgehalten, jedoch wird die Ausführung des Algorithmus gerade für größere Modelle schwerer, da es zu viele Einschränkungen gibt.

### 2.6.2 Andere Anwendungen von WFC in 3D

Gerade außerhalb des akademischen Kontexts (insbesondere in der Spieleentwicklung) trafen die Ideen des WFC auf großen Anklang. Im Folgenden einige interessante Anwendungsbeispiele des WFC.

#### 2.6.2.1 Prozedurale generierte Inseln von Oscar Stalberg (Townscaper)

Oskar Stalberg kombiniert in seiner experimentellen Software [26] zur Generierung von prozeduralen Inseln irreguläre Grids mit dem Tiled Model des WFC, um es dem Anwender zu erlauben, interaktiv beeindruckende Inseln zu generieren. Dabei kann der Anwender durch Mausklicks entscheiden, an welcher Position ein neues Tile hinzugefügt werden soll. Um zu entscheiden, welches Tile an der entsprechenden Position eingesetzt wird und wie sich die angrenzenden Tiles daraufhin verändern, nutzt Stalberg eine modifizierte Version des WFC. Dieser Ansatz erlaubt es, dass das entstandene Modell immer konsistent mit den vorher definierten Adjacency Constraints der Tiles ist und damit ein visuell sinnvolles und ansprechendes Ergebnis garantiert ist [27].

Abbildung 2.18: Screenshot einer in Townscaper erstellten Insel [26]

#### 2.6.2.2 Prozedurale Levelgenerierung in Caves of Qud

Caves of Qud [6] ist ein erfolgreiches Roguelike-Spiel, das sich verschiedenen Techniken aus der Prozeduralen Content Generierung bedient. Unter anderem wird der WFC Algorithmus in Caves of Qud genutzt, um zweidimensionale Gebäudekomplexe zu generieren. Dafür wird der WFC durch zwei zusätzliche Schritte erweitert: Zunächst werden mit klassischen PCG Methoden zufällige Bereiche innerhalb des Levels festgelegt, in dem Gebäude generiert werden sollen. Dieser Schritt vermeidet ein zu homogenes Ergebnis, in dem das gesamte Level mit Gebäuden gefüllt ist. Schließlich werden die Gebäude anhand von Beispiel-Inputs mithilfe des Overlapping Models des WFC generiert und anschließend ein weiterer Schritt durchgeführt, der alle Räume auf Verbindungen zur Außenwelt oder anderen Räumen prüft und diese gegebenenfalls einfügt. Entwickler Brian Bucklew schätzt den WFC für die große Variabilität in seinen Ergebnissen im Vergleich zu traditionellen Methoden [2].

Abbildung 2.19: mithilfe von WFC generiertes Level in Caves of Qud [7]

[7] Bildquelle: https://twitter.com/unormal/status/819725864623603712 (abgerufen: 23.09.20)

# 3 Konzept

Im folgenden Kapitel werden die Anforderungen an den in dieser Arbeit entwickelten Prototypen analysiert sowie eine konzeptuelle Beschreibung der verwendeten bzw. erweiterten Algorithmen und Ideen durchgeführt.

Das Ziel dieser Arbeit ist die Entwicklung einer Software, die in der Lage ist, aus einfachen Voxel-Modellen beliebige größere Voxel-Modelle prozedural zu generieren, die erkennbare Ähnlichkeit zum Input-Modell aufweisen. Dazu wird der WFC Algorithmus als Grundlage genommen und durch verschiedene Änderungen und Erweiterungen für diesen speziellen Use-Case angepasst.

## 3.1 Funktionale Anforderungsanalyse

Die im Rahmen dieser Arbeit entwickelte Software zielt darauf ab, die folgenden funktionalen Anforderungen zu erfüllen:

1. **Voxel-WFC**: Der WFC Algorithmus soll um die folgenden Anforderungen erweitert werden:

    a) **Funktionalität in 3D**: Um mit Voxel-Modellen arbeiten zu können, muss eine Erweiterung für drei Dimensionen stattfinden. Folglich muss es möglich sein, dem Algorithmus ein Voxel-Modell in Form eines 3D-Arrays als Input bereitzustellen und ein 3D-Array des Output-Modells als Ausgabe zu erhalten.

    b) **Parameterisierbarkeit**: Der Voxel-WFC soll durch die in Abschnitt 3.2.2 beschriebenen Parameter anpassbar sein, um dem Nutzer möglichst viel Kontrolle über das Ergebnis zu erlauben.

    c) **Input-Modell**: Der Voxel-WFC akzeptiert eine Repräsentation eines Voxel-Modells als Input.

d) **Output-Modell**: Als Ergebnis liefert der Voxel-WFC eine Repräsentation des Output-Modells, die sowohl die Farben als auch die Positionen der einzelenen Voxel enthält. Dabei soll dieses Modell eindeutige Ähnlichkeiten zum Input aufweisen, auch wenn es sich stark in der Größe unterscheidet.

e) **Performance**: Die Performance des Generierungsprozesses soll, wo möglich, optimiert werden.

2. **User-Interface**: Es gibt eine grafische Oberfläche zur Konfiguration und Visualisierung des Algorithmus

   a) **3D-Viewer**: Die Applikation beinhaltet ein Fenster, das es ermöglicht, Voxel-Modelle in 3D zu betrachten und per Mausbewegung zu drehen sowie mithilfe des Mausrads zu vergrößern. Dieses Fenster soll zunächst das Input-Modell anzeigen und nach der erfolgreichen Generierung das Output-Modell. Wird ein neues Input-Modell geladen, wird wieder dieses angezeigt.

   b) **Parameter**: Die Applikation bietet UI-Elemente, um alle in Abschnitt 3.2.2 definierten Parameter konfigurieren zu können. Je nach Art des Wertes sollen dafür angemessene Elemente genutzt werden (z. B. Checkbox für boolsche Werte).

   c) **Generierung**: Es existiert ein *Generate*-Button, der eine neue Instanz des Voxel-WFC mit den aktuell eingestellten Parametern aufruft und das Ergebnis im 3D-Viewer darstellt.

   d) **Import und Export**: UI-Elemente, die den Import und Export von Input bzw. Output Voxel-Modellen ermöglichen, sind vorhanden.

3. **(De-)Serialisierung von Voxel-Modellen**: Um den Import und Export von Voxel-Modellen zu ermöglichen, muss es möglich sein, Modelle im .vox Format zu Serialisieren und Deserialisieren.

## 3.2 Erweiterung des WFC für 3D-Voxel-Modelle

In diesem Abschnitt wird die erweiterte bzw. angepasste Version des WFC Algorithmus, die im Rahmen dieser Arbeit erarbeitet wurde, im Detail vorgestellt. Der größte Unterschied zum ursprünglichen WFC ist dabei die Erweiterung von zwei auf drei Dimensionen und die

Spezialisierung auf Voxel-Modelle. Dabei wurden unter anderem Konzepte aus *Voxel Synthesis for generative Design* [13] (2.6.1), insbesondere aus dem *Simple Model* genutzt, aber auch einige neue Ansätze eingeführt, die in Abschnitt 3.3 erläutert werden.

Auf oberster Ebene lässt sich der Algorithmus analog zum Original wie folgt strukturieren:

**Algorithm 1** Voxel-WFC auf oberster Ebene

| | |
|---|---|
| 1: Vorbereitung und Initialisierung | ▷ siehe 3.2.5 |
| 2: Finden und Verarbeiten von Mustern | ▷ siehe 3.2.6 |
| 3: Finden von Adjacency Constraints | ▷ siehe 3.2.7 |
| 4: Lösungsroutine | ▷ siehe 3.2.8 |
| 5: Generierung des Output-Modells | ▷ siehe 3.2.9 |

In den folgenden Abschnitten der Arbeit werden diese Schritte jeweils detailliert erklärt.

### 3.2.1 Koordinatensystem

Um in drei Dimensionen Arbeiten zu können muss ein einheitliches Koordinatensystem gewählt werden. Für diese Arbeit werden deshalb die folgenden Achsenbelegungen definiert. $+x$: nach rechts, $+y$: nach unten, $+z$: in die Tiefe. Diese Belegung wird auch *Rechtshändiges Kartesisches Koordinatensystem* genannt und wurde aufgrund der verwendeten Grafik-Library gewählt und ist austauschbar.

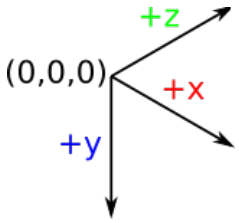



Abbildung 3.1: Rechtshändiges Kartesisches Koordinatensystem

### 3.2.2 Parameter

Die folgenden Parameter beeinflussen den Ablauf und das Ergebnis der Generierung und erlauben es, den Algorithmus für den jeweiligen Use-Case anzupassen.

- **InputModel**: Das Input-Voxel-Modell
- **OutputSize**: Die gewünschte Größe des Output-Modells als 3D-Vektor der Form $(x, y, z)$
- **PatternSize**: Die Größe $N$ der $N \times N \times N$ Muster, die aus dem Input extrahiert werden
- **Rotation**: Wahr $\rightarrow$ Rotierte Versionen des Inputs werden beachtet; Falsch $\rightarrow$ Rotierte Versionen werden nicht beachtet (siehe 3.3.1)
- **AvoidEmptyPattern**: Wert zwischen 0 und 1, der bestimmt wie stark leere Voxel im Output vermieden werden sollen. 0 $\rightarrow$ Keine zusätzliche Vermeidung; 1 $\rightarrow$ Maximal mögliche Vermeidung (siehe 3.3.6)
- **Seed**: Optionaler Seed[1] um generierte Output-Modelle reproduzieren zu können
- **EmptyPadding**: Boolscher Wert, der bestimmt ob Padding mit leeren Voxeln verwendet werden soll (siehe 3.3.3)
- **UseFloor**: Boolscher Wert, der bestimmt ob ein sogenanntes *Floor-Muster* genutzt werden soll (siehe 3.3.2)

### 3.2.3 Variablen

Um für den Algorithmus nötige Daten festzuhalten, müssen einige abstrakte Variablen definiert werden:

- $L_p$ := Liste aller eindeutigen Muster, wobei der Index $I$ der Liste dem Index des Musters entspricht
- $L_f$ := Liste der Frequenzen, mit der ein Muster im Input-Modell vorkommt
- $W$ := Die Wave ist eine Liste von Listen von Muster-Indizes, bei der jedes Element für die noch möglichen Muster an der korrespondierenden Position des Output-Modells steht
- $L_e$ := Liste der Entropie Werte für das Muster des jeweiligen Index
- $S_i$ := 3D-Vektor der Größe des Input-Modells

[1] Eine Zahl die als Basis für einen Zufallszahlengenerator genutzt wird, um reproduzierbare Zufallszahlen zu generieren.

### 3.2.4 Repräsentation der Voxel-Modelle

Um mit geladenen und generierten Voxel-Modellen arbeiten zu können, muss eine geeignete Repräsentation gewählt werden, die alle wichtigen Informationen über das Modell encodiert. Im Fall dieser Arbeit sind diese Informationen zum einen die **Position** der einzelnen Voxel sowie der **Inhalt** der Voxel. Der Inhalt wird als einzelner Integer $v$ repräsentiert und hat die folgende Bedeutung:

- $v = -1$ : Ein leerer Voxel
- $v \geqslant 0$ : Der Index einer Farbe in einer vorher festgelegten Farbpalette

Diese Werte werden in einem dreidimensionalen Array gespeichert, wobei die Position des Wertes im Array die Position des entsprechenden Voxels im Modell encodiert. Die Indizes des Arrays stehen wie folgt im Zusammenhang mit der Position im Koordinatensystem: $A[z][y][x]$

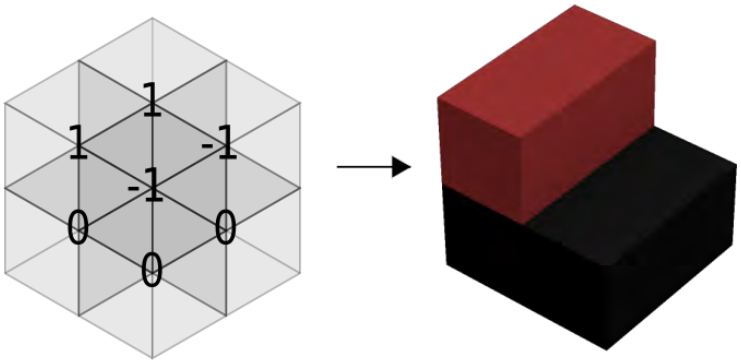



Abbildung 3.2: Die 3D-Array Repräsentation (links) für das Voxel-Modell (rechts)

In Abbildung 3.2 lässt sich erkennen, wie die Werte des Arrays in ein Voxel-Modell umgesetzt werden. In diesem Beispiel wird dafür die folgende Farbpalette genutzt:
$\{0 \rightarrow$ schwarz, $1 \rightarrow$ rot$\}$
Die Belegung des 3D-Arrays $A$ sähe für dieses Beispiel wie folgt aus:

$$A[0][0][0] = 1 \qquad A[0][0][1] = -1 \qquad A[0][1][0] = 0 \qquad A[0][1][1] = 0$$
$$A[1][0][0] = 1 \qquad A[1][0][1] = -1 \qquad A[1][1][0] = 0 \qquad A[1][1][1] = 0$$

### 3.2.5 Vorbereitung und Initialisierung

In diesem ersten Schritt des Voxel-WFC werden zunächst einige für spätere Schritte relevante Werte initialisiert. Zum einen wird die Größe des Input-Modells eingelesen und in der Variable $S_i$ gespeichert. Außerdem wird, sofern der Parameter *Padding* wahr ist, die Input sowie Output-Größe um jeweils eine Mustergröße pro Dimension erweitert (mehr dazu in 3.3.3).

### 3.2.6 Musterfindung

Ist die Initialisierung abgeschlossen, werden aus dem Input-Modell alle eindeutigen Muster extrahiert und verschiedene Informationen über diese gespeichert.

#### 3.2.6.1 Repräsentation von Mustern

Ein Muster beschreibt ein Voxel-Modell der Größe $N \times N \times N$, wobei $N$ durch den Parameter *PatternSize* festgelegt wird. Ein Muster wird genau wie das Input -und Output-Modell durch ein 3D-Array $A$ repräsentiert, beinhaltet aber zusätzlich noch andere Informationen. Zum einen wird jedem Muster ein eindeutiger Index $i$ zugewiesen, der dieses Muster identifiziert. Außerdem gehört zu jedem Muster eine Liste $L_n$ von Nachbarn in jeder Richtung der drei Dimensionen. Ein Muster $P$ kann also wie folgt definiert werden:

$$P := \{A, i, L_n\}$$
$$\text{mit}$$
$$L_n := \{N_{left}, N_{right}, N_{top}, N_{bottom}, N_{forward}, N_{backward}\}$$

Die Nachbarschaftslisten wie $N_{left}$ beinhalten die Indizes aller erlaubten Muster in der entsprechenden Richtung. Es gibt außerdem zwei spezielle Muster für die Indizes -1 und 0 reserviert sind:

- $i = -1$: Das Floor-Muster
- $i = 0$: Das leere Muster

#### 3.2.6.2 Extraktion der Muster aus dem Input-Modell

Um die Muster aus dem Input-Modell zu extrahieren wird das Array des Input-Modells in Schritten der Mustergröße N in allen drei Dimensionen durchlaufen, um es in Muster der Größe $N \times N \times N$ *auseinanderzuschneiden*. Dabei wird jedes neue Muster in $L_p$ eingefügt, sofern diese das Muster noch nicht beinhaltet, ansonsten wird die Frequenz des Musters in $L_f$ erhöht. Ist der Parameter *Rotation* wahr, wird dieser Prozess für jede Rotation des Input-Modells noch einmal durchgeführt. Dieser Ansatz ist stark vom *Simple Model* in 2.6.1.1 inspiriert.

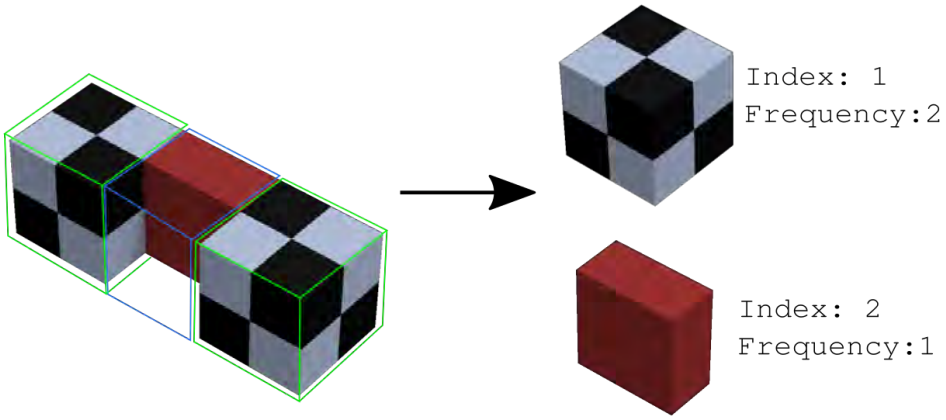



Abbildung 3.3: Einfaches Beispiel für $2 \times 2 \times 2$ Muster (rechts) und deren Indizes sowie Frequenzen aus einem Input-Modell (links)

Im Detail lässt sich der Ablauf der Musterfindung durch den folgenden Algorithmus beschreiben:

**Algorithm 2** Muster Extraktion

```
1:  procedure FINDPATTERNS
2:      L_p ← ∅                                          ▷ Liste der Muster
3:      L_f ← ∅                                          ▷ List der Muster Frequenzen
4:      I ← InputModel
5:      if Rotation then
6:          L_r ← List of rotated Versions of I
7:          for I_r in L_r do
8:              I ← I ∪ I_r
9:          end for
10:     end if
11:     for x ← 0 to S_i.x step PatternSize do           ▷ S_i : Größe des Inputs
12:         for y ← 0 to S_i.y step PatternSize do
13:             for z ← 0 to S_i.z step PatternSize do
14:                 P ← Pattern at Position (x,y,z)
15:                 if P is not in L_p then
16:                     L_p ← L_p ∪ P
17:                     L_f ← Index of P
18:                 else
19:                     I_p ← Index of P
20:                     L_f[I_p] ← L_f[I_p] + 1
21:                 end if
22:             end for
23:         end for
24:     end for
25: end procedure
```

Ist die Musterfindung erfolgreich abgeschlossen, sollten die Listen $L_p$ und $L_f$ mit den entsprechenden Werten gefüllt sein, um zum nächsten Schritt überzugehen. Zu bemerken ist noch, dass am Ende der Musterfindung die Frequenzen der Muster in $L_f$ jeweils durch die Gesamtanzahl von möglichen Mustern im Input-Modell geteilt werden, um den Frequenzwert von einem absoluten zu einem relativen Wert zu transformieren. Dieser kann als relative Häufigkeit eines Musters im Input interpretiert werden.

### 3.2.7 Adjacency Constraints

Um die Adjacency Constraints, also die erlaubten benachbarten Muster für jedes eindeutige Muster zu finden, wird das Input-Modell wieder in Schritten der Mustergröße durchlaufen. In jedem Iterationsschritt wird zunächst festgestellt, welches Muster sich an der aktuellen Position befindet und dieses in $L_p$ gesucht. Dann werden alle sechs benachbarten Positionen geprüft und jedes valide Muster dem aktuellen Muster als möglicher Nachbar in der entsprechenden Richtung hinzugefügt.

**Algorithm 3** Adjacency Constraints

```
1:  procedure FINDNEIGHBOURS
2:      I ← InputModel
3:      D_irs ←[left, right, up, down, forward, backward]
4:      L_r ← List of rotated Versions of I
5:      for I_r in L_r do                                  ▷ Für jede Rotation
6:          for x ← 0 to S_i.x step PatternSize do         ▷ S_i : Größe des Inputs
7:              for y ← 0 to S_i.y step PatternSize do
8:                  for z ← 0 to S_i.z step PatternSize do
9:                      P ← Pattern at Position (x, y, z) in I_r
10:                     L_n := List of list of Neighbours for P
11:                     for d in D_irs do
12:                         P* ← Pattern at Position (x, y, z) + d
13:                         L_n[d] ← P*
14:                     end for
15:                 end for
16:             end for
17:         end for
18:     end for
19: end procedure
```

Die Richtungen in $D_{irs}$ (Zeile 3) können als 3D-Vektoren repräsentiert werden, die mit der aktuellen Position addiert, die Position des entsprechenden Nachbar-Musters zu ergeben. Der rechte Nachbar der Position $(2, 3, 4)$ wäre mit $\mathtt{right} := (1, 0, 0)$ z. B. $(2, 3, 4) + (1, 0, 0) = (3, 3, 4)$.

### 3.2.8 Generierung

Mithilfe der gefundenen Muster und deren Adjacency Constraints kann nun der eigentliche Generierungsprozess gestartet werden. Der grundlegende Ablauf unterscheidet sich dabei nicht vom Original-WFC Algorithmus (siehe 2.4.1), es wurden lediglich einige Anpassungen getroffen, um in drei Dimensionen zu funktionieren. Zunächst wird die Wave $W$ entsprechend der gewünschten Größe des Output-Modell *OutputSize* mit allen gefundenen Muster-Indizes als möglichem Wert initialisiert. Dann beginnt der *Main Loop* des Algorithmus, in dem jeweils die Zelle mit dem geringsten Entropiewert ausgewählt wird, um aus dessen möglichen Muster-Indizes ein Konkretes auszuwählen (die Zelle wird *kollabiert*). Die Auswirkungen dieser Änderung werden dann unter Beachtung der Adjacency Constraints rekursiv durch die gesamte Wave propagiert, um im nächsten Iterationsschritt eine weitere Zelle zu kollabieren. Hat jede Zelle genau einen konkreten Musterindex, ist die Generierung abgeschlossen. Kommt es zu einem Widerspruch, wird der Algorithmus abgebrochen bzw. neu gestartet.

#### 3.2.8.1 Die Wave

Die Wave beschreibt den aktuellen Zustand des Output-Modells und kann als dreidimensionales Array von Listen von Musterindizes repräsentiert werden. Jede Position der Wave beinhaltet eine *Zelle*, in der eine Liste aller noch möglichen Musterindizes gespeichert wird.

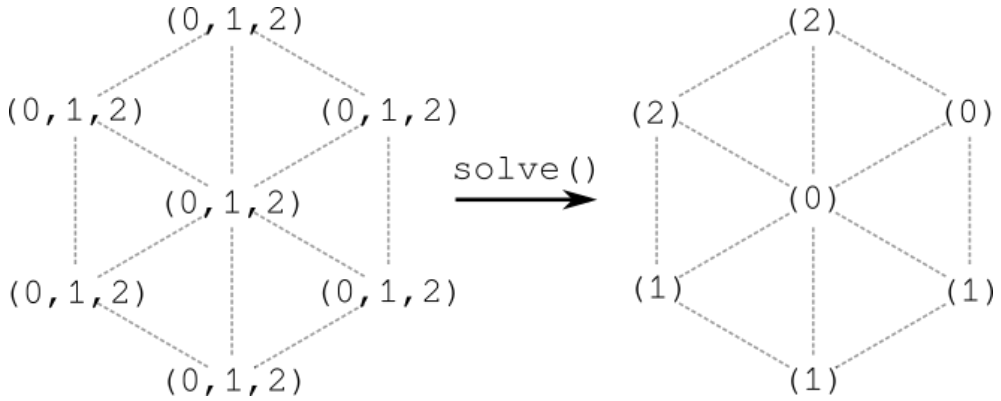



Abbildung 3.4: Die Wave im initialen Zustand (links) und nach der erfolgreichen Generierung (rechts)

#### 3.2.8.2 Initialisierung der Wave

Zunächst wird die Wave wie folgt initialisiert:

**Algorithm 4** Initialisierung der WAve

```
1: procedure INITIALIZEWAVE()
2:     L_pi ← List of all pattern indices
3:     for cell in W do
4:         cell ← L_pi          ▷ Initialisiere jede Zelle der Wave mit allen Musterindizes
5:     end for
6: end procedure
```

#### 3.2.8.3 Lösungsroutine

Diese Schleife hat als Herzstück des Algorithmus die Aufgabe, eine konkrete Belegung der Wave mit Mustern zu errechnen, die keine Adjacency Constraints verletzt. Dabei wird bei einer Verletzung dieser Constraints, also einem Widerspruch (contradiction), wieder von vorn angefangen, bis eine gewisse Anzahl von maximalen Versuchen überschritten ist.

**Algorithm 5** Solve Routine

```
1: procedure SOLVE()
2:     initializeWave()
3:     T_max ← maximumTries
4:     T_n ← 0                                   ▷ Aktuelle Anzahl der Versuche
5:     while Number of uncollapsed cells in W > 0 and T_n < T_max do
6:         collapseLowestEntropyCell()           ▷ Erklärung folgend
7:         propagate()
8:         if Contradiction then
9:             initializeWave()
10:            T_n ← T_n + 1
11:        end if
12:    end while
13:    if All cells in W have exactly one value then
14:        generateOutput()                      ▷ Generierung war erfolgreich
15:    end if
16: end procedure
```

Die **collapseLowestEntopyCell** Funktion stellt den Beobachtungsschritt des WFC dar und hat die Aufgabe, die Zelle der Wave mit dem niedrigsten Entropiewert zu kollabieren. Zur Entropie (E)-Berechnung wird beim WFC eine modifizierte Version der *Shannon Entropy* [35] verwendet:

$$E = \log \sum_{i=0} p_i - \frac{\sum_{i=0} p_i \log p_i}{\sum_{i=0} p_i}$$

p ist dabei die Liste der aus dem Input-Modell berechneten Frequenzen der Musterindizes (aus $P_f$) die in der betrachteten Zelle vorkommen. $\sum_{i=0} p_i$ beschreibt also die Summe dieser Frequenzen. In der Regel ist dieser Entropiewert umso niedriger, je weniger Musterindizes in der Zelle vorhanden sind, allerdings gewichtet durch die Frequenz der Muster. Ist die Zelle mit der niedrigsten Entropie gefunden, wird diese **kollabiert**: Das heißt, einer der Musterindizes wird zufällig ausgewählt (gewichtet durch die Frequenzen) und alle anderen werden entfernt.

#### 3.2.8.4 Propagierung

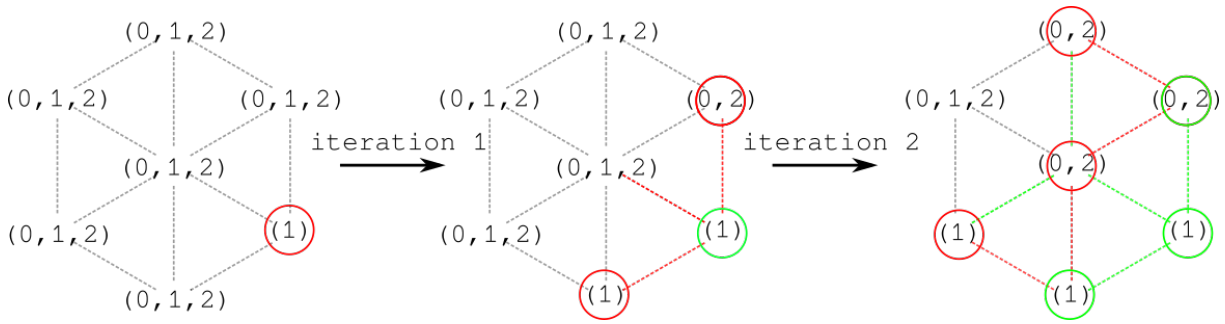



Abbildung 3.5: Drei Iterationen der Propagierung, zeigt jeweils die veränderten Zellen der aktuellen (rot) und der vorherigen (grün) Iterationen

Nachdem eine Zelle kollabiert wurde, folgt der Propagierungs-Schritt (*propagate()* im Algorithmus). Dabei werden, ausgehend von der Position der kollabierten Zelle, zunächst die Zustände von benachbarten Zellen in allen sechs Richtungen entsprechend der Adjacency Constraints aktualisiert. Wurde z. B. das Muster $P_1$ für die aktuelle Zelle ausgewählt, werden aus der Zelle links neben der aktuellen nun alle Musterindizes entfernt, die laut der Adjacency Constraints nicht links neben dem Muster $P_1$ liegen dürfen. Alle Zellen, die eine

Zustandsänderung erfahren, werden einer Liste Q hinzugefügt, um sie ebenfalls für die Propagierung zu markieren. Ein Propagierungszyklus ist abgeschlossen, wenn Q leer ist. Tritt eine Contradiction ein, wird der Prozess abgebrochen.

**Algorithm 6** Propagate

```
1:  procedure PROPAGATE()
2:      C ← collapsed cell
3:      Q ← Q ∪ C
4:      while Q ≠ ∅ do
5:          C_i ← next cell in Q
6:          for d in D_irs do                    ▷ Für jede der sechs Richtungen
7:              C_n ← Neighbouring cell to C_i in direction d
8:              N_d ← List of allowed neighbours of C_i in direction d
9:              for Pattern Index P_i in C_n do
10:                 if P_i ∉ C_i then
11:                     Remove P_i from C_n
12:                 end if
13:             end for
14:             if C_n has changed then Q ← C_n
15:             end if
16:             if |C_n| == 0 then
17:                 Abort propagation            ▷ Es gab einen Widerspruch: Abbruch
18:             end if
19:         end for
20:         Remove C_i from Q
21:     end while
22: end procedure
```

### 3.2.9 Generierung des Output-Modells

Wurde die Lösungsroutine erfolgreich abgeschlossen, müssen nun nur noch die konkreten Musterindizes aus der Wave wieder in ein Voxel-Modell umgesetzt werden. Dafür wird für jede Zelle der Wave, das während der Musterfindung gespeicherte 3D-Array, für den jeweiligen Musterindex abgerufen und an der korrekten Position O im Output-Modell positioniert. Diese Position ist abhängig von der Mustergröße N und lässt sich wie folgt errechnen:

$$(O_x, O_y, O_z) = (W_x * N, W_y * N, W_z * N)$$

Um die absolute Position eines einzelnen Voxels zu bestimmen, kann O nun als Offset auf die Position des Voxels in der 3D-Array-Repräsentation addiert werden. Dies impliziert natürlich, dass die Größe des Output-Modells ebenfalls um den Faktor N größer ist als die Wave.

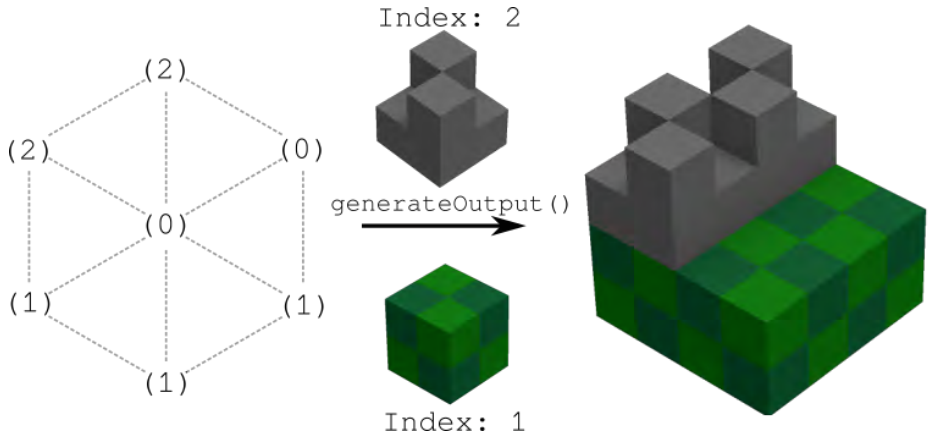



Abbildung 3.6: Aus der Wave (links) wird mithilfe der Musterindizes ein Output-Modell (rechts) generiert

## 3.3 Spezielle Erweiterungen des WFC

### 3.3.1 Rotation

Eine Strategie, die im klassischen WFC angewendet wird, um eine größere Variabilität im Output zu erreichen, ohne einen komplexeren Input zu benötigen, ist das Hinzufügen von rotierten bzw. reflektierten Versionen der gefundenen Muster. Dafür werden die gefundenen

Muster einzeln rotiert und der Liste der Muster hinzugefügt. Dieses Vorgehen macht das Finden der korrekten Adjacency Constraints für die rotierten Muster verhältnismäßig schwer, da sich für jedes Muster gemerkt werden muss, welche Rotation stattgefunden hat, um dann die Adjacency Constraints des ursprünglichen Musters entsprechend anzupassen und dem neuen zuzuweisen [7].

Hier wird ein anderer Ansatz zur Rotation verfolgt: Für jede gewünschte Rotation wird das komplette Input-Modell kopiert und als Ganzes rotiert. Diese rotierten Versionen werden genau wie der ursprüngliche Input behandelt und erfordern daher auch keine Spezialbehandlung beim Finden der Adjacency Constraints. Im Fall dieser Arbeit wird sich auf Rotationen um die y-Achse beschränkt es spricht jedoch nichts dagegen, dieses in zukünftigen Arbeiten um weitere Raumachsen zu erweitern.

#### 3.3.1.1 Rotation von 3D-Koordinaten

Um einen Punkt im dreidimensionalen Raum zu rotieren, können Rotations-Matrizen [8] genutzt werden:

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

Wird diese Rotations-Matrize mit einem Positions-Vektor multipliziert, stellt das Ergebnis die Position des um $\theta$ um die y-Achse rotierten Punktes dar. Da in unserem Fall eine Rotation um $90^\circ$ gewünscht ist, sieht die entspreche Rotations-Matrize wie folgt aus:

$$R_y(90^\circ) = \begin{bmatrix} \cos 90^\circ & 0 & \sin 90^\circ \\ 0 & 1 & 0 \\ -\sin 90^\circ & 0 & \cos 90^\circ \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

Ein einzelner Punkt kann dann durch die folgende Berechnung um $90^\circ$ um die y-Achse rotiert werden:

$$R_y(90°) \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} Z \\ Y \\ -X \end{bmatrix}$$

Diese Transformation muss nun lediglich dreimal auf das Input-Modell angewendet werden, um alle vier 90° Rotation zu erhalten.

### 3.3.2 Floor-Muster

Da an der Realität orientierte Voxel-Modelle häufig in Bezug auf ihr Verhältnis zum Boden eingeschränkt sind, ein Gebäude würde z. B. in einem realistischen Modell immer Kontakt zum Boden haben und nicht in der Luft schweben, ergibt es Sinn auch diese Information aus dem Input-Modell zu extrahieren. Dafür wird ein sogenanntes *Floor-Muster* eingeführt, welches der Wave unter jeder Zelle mit niedrigsten Y-Positionen hinzugefügt wird. Wie in Abschnitt 3.2.6.1 definiert wird dieses Muster durch den Musterindex -1 repräsentiert. Während der Analyse von Adjacency Constraints muss nun lediglich der Musterindex -1 für alle Muster auf der niedrigsten Y-Ebene des Input-Modells als Nachbar in Richtung *Down* hinzugefügt werden. Damit ist sichergestellt, dass alle Muster, die im Input-Modell Kontakt zum Boden haben, auch im Output-Modell nur dort erlaubt sind.

### 3.3.3 Input und Output Padding

Ein weiteres Problem, das bei einigen Input-Modellen auftreten kann, ist der Fakt, dass ein Muster, das genau am Rand des Modells positioniert ist, keinerlei Adjacency-Constraints in Richtung des Randes erhält. Auch bei der Generierung des Output-Modells stellen die Rand-Zellen ein Problem dar, da diese nicht auf Constraints in Richtung des Randes überprüft werden. Dies kann zu *abgeschnittenen* Output-Modellen führen.

Als Lösungsansatz wird dafür ein *Padding* des In -und Outputs mit dem leeren Muster (Index 0) eingeführt. Das heißt, das jeweilige Modell wird in jeder Richtung (außer in Richtung des Bodens) von leeren Mustern umschlossen.

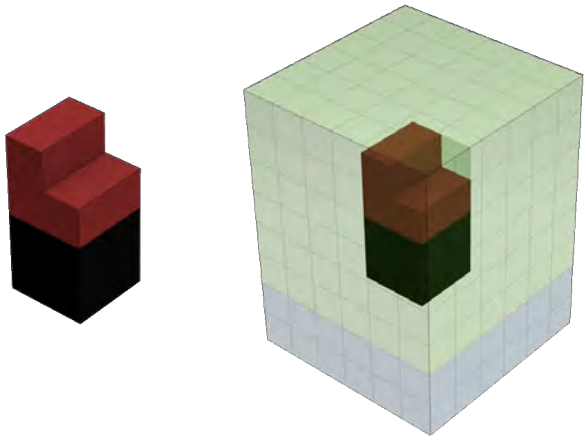


Abbildung 3.7: Ein Input-Modell mit (rechts) und ohne (links) Padding. Leere Muster sind grün gekennzeichnet, Floor-Muster blau.

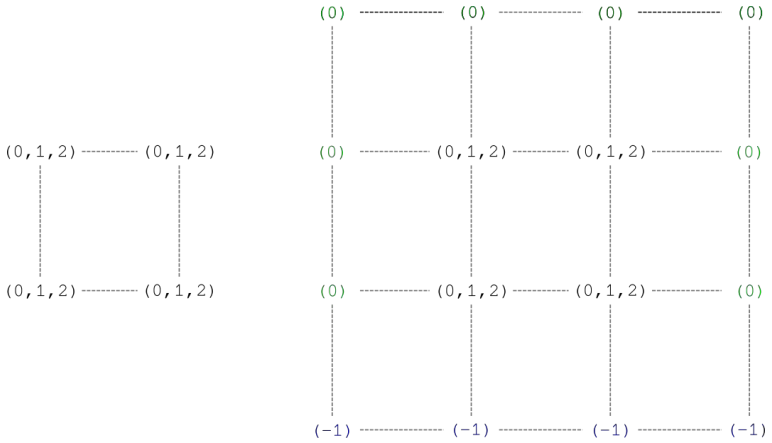



Abbildung 3.8: Zeigt eine zweidimensionale Darstellung der Wave nach der Initialisierung mit (rechts) und ohne (links) Output Padding

### 3.3.4 Prepropagation

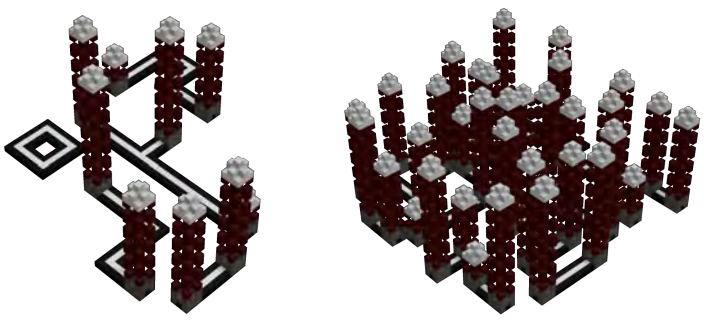
Bedingt durch das Output Padding, sind am Anfang der Löungsroutine bereits alle Randzellen der Wave kollabiert. Das heißt, jeder Randzelle ist schon ein eindeutiger Musterindex zugewiesen (-1 am Boden und 0 an allen anderen Rändern). Deshalb muss, bevor die eigentliche Lösungsroutine beginnt, eine sogenannte *Prepropagation* durchgeführt werden. Dafür wird die Liste aller Randzellen durchgegangen und für jede Zelle die Propagierung wie in 3.2.8.4 beschrieben durchgeführt.

### 3.3.5 Backtracking

Im klassischen WFC wird auf ein Widerspruch während der Lösungsroutine mit einem kompletten Neustart des Algorithmus reagiert. Diese Lösung ist zwar funktional, führt aber zu langen Generierungszeiten, wenn viele Widersprüche auftreten. In dieser Arbeit wird ein einfaches System zum Backtracking genutzt, um dieses Problem zu vermeiden. Dafür wird vor jeder Iteration der Lösungsroutine, also bevor eine Zelle ausgesucht und auf ein zufälliges Muster kollabiert wird, eine Kopie (Snapshot) der Wave in ihrem aktuellen Zustand gespeichert. Erzeugt die anschließende Propagierung nun einen Widerspruch, wird dieser Snapshot wiederhergestellt und ein erneuter Versuch gestartet. Erst wenn auch diese Backtracking Routine eine gewisse Anzahl (konfigurierbar) von Fehlversuchen durchlaufen hat, wird komplett von vorne angefangen.

### 3.3.6 Vermeidung von leeren Zellen

In größeren Input-Modellen sind häufig viele leere Voxel vorhanden. Das heißt also, dass bei der Musterfindung das leere Muster meist eine hohe Frequenz in $L_f$ aufweist, da dieses zunächst als ganz normales Muster behandelt wird. Korrekterweise führt das dazu, dass auch im Output-Modell große Teile des verfügbaren Raumes leer bleiben. Um dem Benutzer mehr Kontrolle darüber zu geben, wie stark der Output *gefüllt* werden soll, wurde der Parameter *AvoidEmptyTiles* $A_e$ als Wert zwischen 0 und 1 eingeführt. Ist dieser Wert 0, verläuft der Voxel-WFC ohne Modifikation und die tatsächliche Frequenz des leeren Musters wird genutzt. Je näher der Parameter der 1 ist, desto mehr wird die Frequenz des leeren Musters verringert, um einen *gefüllteren* Output zu erreichen. Die angepasste Frequenz des leeren Musters $F'_e$ berechnet sich wie folgt ($F_e :=$ tatsächliche Frequenz): $F'_e = F_e(1 - A_e)$.



(a) $A_e = 0$ (b) $A_e = 0.95$

Abbildung 3.9: Vergleich von Output-Modellen mit unterschiedlichen Werten für $A_e$

# 4 Umsetzung

Im folgenden Kapitel werden die Spezifika der Umsetzung bzw. Implementierung der in Kapitel 3 vorgestellten Konzepte erläutert. Dabei werden auch Entscheidungen bezüglich der genutzten Technologien sowie der Architektur der Anwendung gerechtfertigt. Der Quellcode der Anwendung ist in einem Gitlab-Repository der HAW-Hamburg[1] und einem öffentlichen Github-Repository[2] zu finden.

## 4.1 Nicht funktionale Anforderungen

Zusätzlich zu den in Abschnitt 3.1 vorgestellten funktionalen Anforderungen wurden einige Anforderungen in Bezug auf die Qualität der Software und des Entwicklungsprozesses erarbeitet, dabei wurde sich an bekannten Prinzipien des Software-Engineerings orientiert [16, 24]:

1. **Verständlichkeit und Wartbarkeit**: Einem Entwickler, dem das Projekt unbekannt ist, soll es möglich sein, schnell die grundlegende Struktur der Applikation und die Zuständigkeiten der einzelnen Komponenten zu identifizieren. Dies wird erreicht durch:
   a) Sprechende Klassen, -Methoden- und Variablen-Namen
   b) Java-Doc-Kommentare
   c) Kommentare an wichtigen oder schwer verständlichen Teilen des Codes
   d) Teilung von großen Methoden in mehrere kleinere
2. **Verfügbarkeit:** Das Projekt soll möglichst einfach aufzusetzen zu sein und minimale Abhängigkeiten besitzen.

[1] https://git.haw-hamburg.de/acg303/wfc
[2] https://github.com/Bendzae/WFCVoxelModelGenerator

3. **Erweiterbarkeit:** Die Architektur der Anwendung soll die nachträgliche Erweiterung von Funktionalität einfach machen.
4. **Modularität:** Unterschiedliche Aufgaben (hier z. B. Generierung und Darstellung) sollen auch von unterschiedlichen Modulen oder Klassen übernommen werden.
5. **Minimale Code Duplizierung**: Wird derselbe Code an mehreren Stellen verwendet, sollte dieser in eine eigene Methode oder Klasse abstrahiert werden.
6. Einhaltung der folgenden **SOLID** [16] Prinzipien (nicht alle sind für dieses Projekt sinnvoll):
    a) **Single Responsibility Principle:** Jede Klasse/Komponente der Anwendung sollte, wenn möglich, nur für eine Aufgabe verantwortlich sein.
    b) **Open/Closed Principle:** Klassen der Anwendung sollten erweiterbar sein, ohne den existierenden Code stark verändern zu müssen.

## 4.2 Vorgehensmodell

Als Vorgehensmodell zur Entwicklung des Prototyps wurde ein iterativer und inkrementeller Ansatz gewählt, der sich stark an agilen Methoden wie Scrum [1] orientiert. Diese Entscheidung wurde hauptsächlich getroffen, da die Anforderungen an die Anwendung nicht von Anfang an definiert werden konnten und sich erst im Laufe der Entwicklung und Nachforschung entwickelt haben, was gegen ein klassisches Vorgehensmodell wie z. B. das Wasserfall-Modell [33] spricht. Der Entwicklungsprozess wurde also in mehrere Iterationen aufgeteilt, an deren Ende immer eine funktionierende Anwendung stehen sollte. Jeder dieser Entwicklungszyklen beinhaltet vier Phasen: Planung, Design und Implementierung, Testing und Evaluation (siehe Abbildung 4.1). So konnten z.

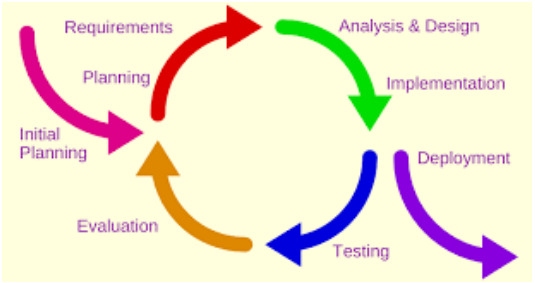



Abbildung 4.1: Ein Zyklus bei der Iterativen/Inkrementellen Entwicklung [3]

[3] Bildquelle: https://de.wikipedia.org/wiki/Inkrementelles_Vorgehensmodell (abgerufen: 16.10.20)

B. bei dieser Anwendung nach jeder Iteration bessere Anforderungen für die nächste definiert werden. Die folgenden Meilensteine wurden in je einer Iteration bearbeitet:

1. 2D-WFC-Implementierung in Java (orientiert an der original C# Verison [7])
2. Prototyp einer UI zur Parameterisierung des Algorithmus
3. Entwicklung eines 3D-Viewers für Voxel-Modelle
4. Grundlegende 3D-Implementierung des WFC
5. Importer und Exporter für Voxel-Modelle (im .vox Format)
6. Erweiterung und Spezialisierung der der 3D-Implementierung für Voxel-Modelle

Dabei wurde großen Wert auf eine saubere Versionskontrolle mithilfe von Git gelegt, um es jederzeit möglich zu machen zu einem früheren Stand der Applikation zurückzukehren und den Entwicklungsprozess anhand von detaillierten Commit-Messages nachzuvollziehen. Zum selben Zweck wurde auch das Branching-System von Git genutzt.

## 4.3 Verwendete Technologien

### Java

Die Programmiersprache Java wurde aufgrund der großen Beliebtheit, der Präferenz des Autors und der Verfügbarkeit von relevanten Bibliotheken als Implementierungswerkzeug gewählt. Die Software wurde mit der JDK Version 11[4] entwickelt.

### Git Version Control

Git[5] ist ein Tool zur Versionskontrolle von Software-Projekten und wurde hier für eben diesen Zweck genutzt.

[4] https://www.oracle.com/java/technologies/javase-jdk11-downloads.html (abgerufen: 16.10.20)
[5] https://git-scm.com/ (abgerufen: 16.10.20)

### Maven

Maven[6] wurde in diesem Projekt verwendet, um das Auflösen von Abhängigkeiten und den Build-Prozess zu vereinfachen. Das Projekt ist so aufgesetzt, dass ein Nutzer lediglich das Git-Repository klonen muss und ohne weitere Abhängigkeiten aufzulösen eine funktionsfähige Anwendung starten kann.

### IntelliJ Idea

IntelliJ Idea[7] ist eine Entwicklungsumgebung für Java-Anwendungen und wurde hier für die Entwicklung des Prototyps genutzt.

### Magicka Voxel

Magica Voxel[8] ist ein kostenloser Editor für Voxel-Modelle. In dieser Arbeit wurde die Software genutzt, um Input-Modelle zu erstellen. Es wurde sich für diesen Voxel Editor aufgrund seiner freien Verfügbarkeit und großen Beliebtheit [17] entschieden. Außerdem bietet der Entwickler eine genaue Spezifikation des hauseigenen .vox Formats, was den Import und Export von Modellen in diesem Format vereinfacht. Alternativ wurde auch noch der Open-Source Voxel-Editor Goxel[9] verwendet. Dieser hat ähnliche Features, bietet allerdings auch Support für das Linux Betriebssystem an.

## 4.4 Verwendete Bibliotheken

### JavaFX

Zur interaktiven Darstellung von Voxel-Modellen in drei Dimensionen wurde die Grafik-Bibliothek JavaFX[10] ausgewählt. Die Bibliothek bietet die nötige Funktionalität, um die Anforderungen an den 3D-Viewer sowie das User-Interface umzusetzen und lässt sich als einfache Maven-Dependency in das Projekt einbinden. Daher wird keine zusätzliche Software

[6] https://maven.apache.org/ (abgerufen: 16.10.20)
[7] https://www.jetbrains.com/idea/ (abgerufen: 16.10.20)
[8] https://ephtracy.github.io/ (abgerufen: 16.10.20)
[9] https://goxel.xyz/ (abgerufen: 16.10.20)
[10] https://openjfx.io/ (abgerufen: 16.10.20)

zur Entwicklung benötigt wie es z. B. bei Game-Engines wie Unity oder Unreal Engine der Fall wäre.

### JVox Voxel Parser

JVox [11] ist eine unter der MIT-Lizenz verfügbare Bibliothek um Modelle im .vox Format in eine Java Anwendung zu laden und in angemessenen Datenstrukturen zu speichern.

### Gson Json Parser

Gson[12] (Apache-2.0 Lizenz) ist eine von Google entwickelte Bibliothek zum Serialisieren und Deserialisieren von JSON-Dokumenten und wird hier genutzt, um Parameter für verschiedene Input-Modelle im JSON-Format zu persistieren.

## 4.5 Architektur der Anwendung

Für die grundlegende Architektur der Anwendung wurde eine vereinfachte Version des *Model-view-controller* [5] Design-Patterns verwendet, in der die View und Controller Komponenten vereint sind (Model/View Architektur) [22]. Grund für diese Architektur ist die so erreichte Entkoppelung von Anwendungs -und Darstellungslogik. Das erhöht die Wiederverwendbarkeit der einzelnen Komponenten. Wenn z. B. eine andere Grafik-Bibliothek verwendet werden soll, muss lediglich die View Komponente ausgetauscht werden.

Auf oberster Ebene lässt sich die entwickelte Anwendung in drei grundlegende Komponenten einteilen: die **Model** Komponente, die **View** Komponente und die **Serializer** Komponente. Die Model Komponente enthält alle nötigen Datenstrukturen und Logik, für Durchführung des Voxel-WFC-Algorithmus und ist agnostisch gegenüber der verwendeten Grafik Bibliothek. Die View Komponente beinhaltet die gesamte Darstellungslogik, d. h. das 3D-Rendering der Voxel-Modelle, das User-Interface und die Verarbeitung von Eingaben des Benutzers. Die Serializer Komponente hat lediglich die Aufgabe der (De-)Serialisierung von Modellen im .vox Format und wird von der View Komponente genutzt, um Voxel-Modelle zu importieren und exportieren.

[11] https://github.com/Lignum/JVox (abgerufen: 16.10.20)
[12] https://github.com/google/gson (abgerufen: 16.10.20)

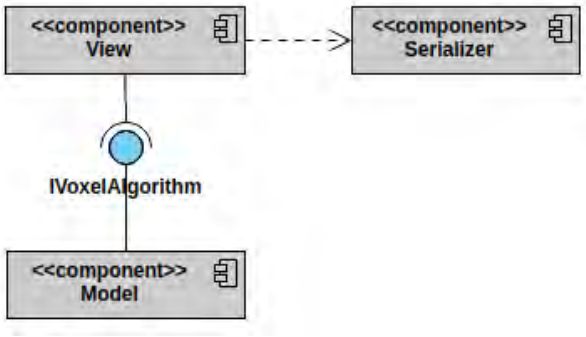



Abbildung 4.2: Top-Level Architektur der Applikation

## 4.6 Model

Die Model Komponente beinhaltet alle Klassen, die für die Ausführung des Voxel-WFC benötigt werden. Das Herzstück der Komponente ist dabei die Klasse **VoxelWfcModel**. Diese Klasse beinhaltet den Großteil der Anwendungslogik und bietet die Funktionalität der Komponente über das Interface **IVoxelAlgortihm** nach außen an.

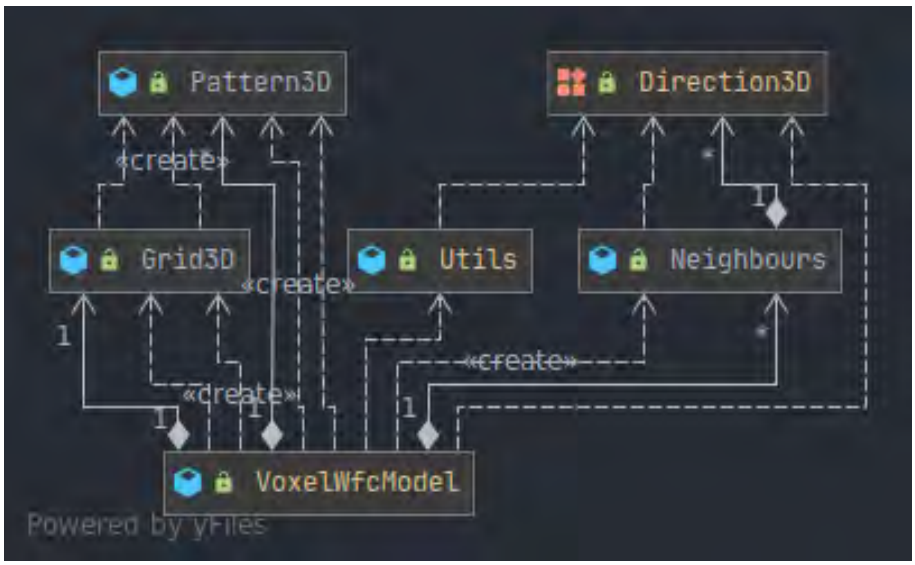



Abbildung 4.3: UML Klassendiagramm der **Model** Komponente

### 4.6.1 Schnittstellendefinition

Um zu definieren, welche Funktionalität die Model Komponente nach außen anbietet, wird das Interface **IVoxelAlgortihm** definiert. Besonders wichtig ist hier die *solve()* Methode, die den Algorithmus startet und das generierte Modell als 3D-Integer-Array wiedergibt.

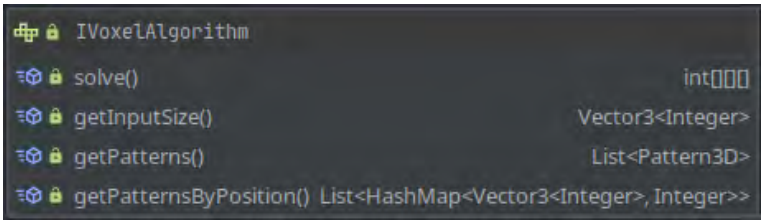



Abbildung 4.4: UML-Diagramm des IVoxelAlgorithm Interface

### 4.6.2 Wichtige Klassen

#### Pattern3D

Die Pattern3D Klasse wird als Datenstruktur zur Repräsentation der beim WFC gefundenen Muster genutzt. Interessant ist hier das Überschreiben der equals() Methode, um einen einfachen Vergleich von Mustern im Algorithmus zu ermöglichen.

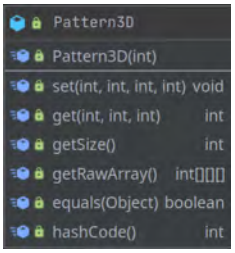



Abbildung 4.5: UML Diagramm der Pattern3D Klasse

#### Grid3D

Diese Klasse wird als Abstraktion für die 3D-Array Repräsentation der Voxel-Modelle genutzt und beinhaltet unter anderem Methoden, um den Wert von Voxeln an bestimmten Positionen auszulesen oder eine rotierte Version des Models zu erhalten.

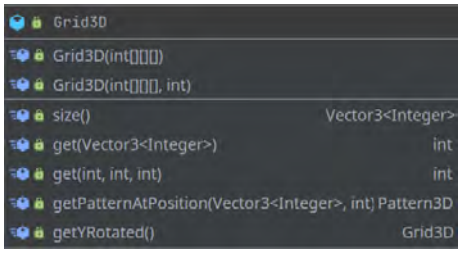



Abbildung 4.6: UML Diagramm der Grid3D Klasse

### Neighbours

Die Klasse *Neighbours* implementiert eine Datenstruktur die genutzt wird, um eine Liste von Nachbarn für ein Muster in alle sechs Richtungen zu verwalten.

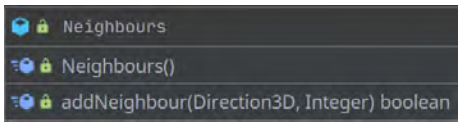



Abbildung 4.7: UML Diagramm der Neighbours Klasse

### VoxelWfcModel

Diese Klasse implementiert die Funktionalität des Voxel-WFC wie sie in Kapitel 3 beschrieben wurde. So setzt z. B. die Methode solve() die Lösungsroutine (3.2.8.3) um, findPatterns() die Mustersuche (3.2.6) usw. Wie in Abbildung 4.8 zu sehen ist, wurde darauf geachtet viele Unter-Methoden mit sprechenden Namen zu nutzen, um die Lesbarkeit zu erhöhen. Die Parameter für den Algorithmus (siehe 3.2.2) werden über den Konstruktor der Klasse initialisiert.

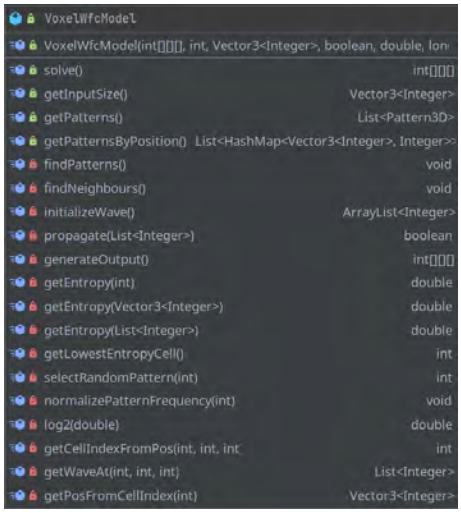



Abbildung 4.8: UML Diagramm der VoxelWFCModel Klasse

## 4.7 View

Die View-Komponente enthält die gesamte Darstellungslogik der Anwendung. Das beinhaltet zum einen das User-Interface (UI) und den 3D-Viewer, zum anderen aber auch das Einlesen von Parametern über das UI sowie die Ausführung des Voxel-WFC mit diesen Parametern. Die Klasse **App** dient außerdem als Einstiegspunkt in die Applikation und startet in ihrer main() Methode die JavaFX-Anwendung.

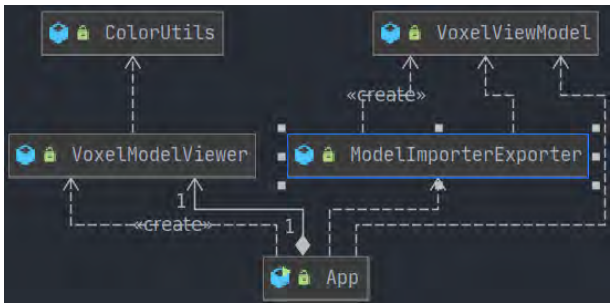



Abbildung 4.9: UML Klassen Diagramm der View Komponente

### 4.7.1 Voxel-Model-Viewer

Die dreidimensionale Darstellung der Input und Output Voxelmodelle wird durch die Klasse **VoxelModelViewer** implementiert. Außerdem ermöglicht sie die Rotation der Ansicht mithilfe von Mausbewegungen sowie ein Vergrößern und Verkleinern des Bildes. Die Klasse nimmt dafür über die Methode *setModel()* ein Voxel-Modell als Integer 3D-Array entgegen und erstellt für jede Array-Position eine 3D-Box in der durch den Wert des Arrays und der Farbpalette spezifizierten Farbe. Zur Darstellung der Boxen wir die *Box*-Klasse des JavaFX Frameworks genutzt, die es auf einfache Weise ermöglicht, verschiedenfarbige Boxen im dreidimensionalen Raum zu positionieren. Für die 3D-Darstellung und Rotation wurde zum Teil Code eines JavaFX Tutorials[13] zum Vorbild genommen.

### 4.7.2 Die App Klasse

Die Klasse **App** kann als zentraler Punkt der Anwendung gesehen werden, in dem die gesamte Funktionalität zusammengeführt wird. Unter anderem speichert diese Klasse die aktuell gewählten Parameter für den Voxel-WFC und verbindet die Funktionalität der anderen Komponenten mit dem User Interface.

#### User Interface

Das User-Interface wurde als Teil der **App** Klasse umgesetzt und nutzt verschiedene JavaFX Komponenten wie z. B. Textfelder, Buttons oder Checkboxen, um die in Abschnitt 3.1 beschriebenen Anforderungen zu erfüllen. Dabei wurden auf ein minimalistisches und funktionales Design geachtet, um es dem Nutzer möglichst einfach zu machen, sich in der Applikation zurechtzufinden. Die Hauptaufgabe der Oberfläche ist es, dem Nutzer Eingaben der Parameter für den Voxel-WFC zu ermöglichen und diese für die nächste Generierung zwischenzuspeichern. Aber auch das Starten der Generierung sowie das Bereitstellen von Dialogen zum Import und Export von Modellen.

[13] https://github.com/afsalashyana/JavaFX-3D (abgerufen: 16.10.20)

#### Generierung

Durch einen Klick auf den *Generate-Button* wird eine neue Instanz der Klasse VoxelWFC-Model mit den aktuellen Parametern erstellt und dessen *solve()* Methode aufgerufen, um ein neues Modell zu generieren. Dieses wird dann ebenfalls zwischengespeichert und mithilfe des Voxel-Models-Viewers dargestellt. Der Generierungsprozess wird in einem eigenen Thread aufgerufen, damit die Applikation trotzdem noch auf Eingaben reagieren kann.

## 4.8 Serializer

Die Serializer-Komponente realisiert das Serialisieren und Deserialisieren von Modellen im .vox Format und besteht zum Großteil aus dem JVox-Voxel-Parser [14]. Im Rahmen dieser Arbeit wurde die Bibliothek lediglich um die Fähigkeit erweitert, Voxel-Modelle im .vox Format zu speichern.

### 4.8.1 Magicka Voxel Datei Format

Das Magicka Voxel Datei Format (.vox) ist ein einfaches und häufig genutztes Binary-Format um Voxel-Modelle zu speichern. Der Entwickler stellt online eine Spezifikation[15] des Formats zur Verfügung, die hier genutzt wurde, um den Exporter für Voxel-Modelle zu realisieren. Grob erklärt ist eine solche Datei in sogenannte *Chunks* eingeteilt, die verschiedene Informationen über das Modell beinhalten. Ein *XYZI-Chunk* codiert beispielsweise die Positionen und Farbindizes der Voxel in einem Modell. Ein Eintrag in diesem Chunk besteht aus genau 4 Byte (x, y, z, colorIndex).

### 4.8.2 Deserialisierung

Die Deserialisierung oder das Lesen von Voxel-Modellen wird von der JVox-Bibliothek über die Klasse **VoxReader** angeboten und in der Klasse **ModelExporterImporter** der View Komponente genutzt, um Modelle zu laden.

[14] https://github.com/Lignum/JVox (abgerufen: 16.10.20)
[15] https://github.com/ephtracy/voxel-model/blob/master/MagicaVoxel-file-format-vox.txt (abgerufen: 16.10.20)

### 4.8.3 Serialisierung

Die Serialisierung oder das Speichern wurde als Erweiterung der JVox-Bibiliothek in der Klasse **VoxSerializer** selbst implementiert. Dafür erstellt die Klasse zunächst eine neue Datei und schreibt dann mithilfe eines FileOutputStreams und unter Beachtung der Spezifikation die entsprechenden Bytes in die Datei. Dafür muss der Methode *writetoVox()* das Voxel-Modell, die genutzte Farbpalette und die Ausgabedatei mitgegeben werden. Auch diese Klasse wird schließlich in der **ModelExporterImporter** der View Komponente genutzt.

# 5 Evaluation

Im folgenden Kapitel werden die Ergebnisse dieser Arbeit präsentiert und diskutiert. Dabei wird auch versucht, mögliche Probleme in Konzept oder Umsetzung zu identifizieren und Vorschläge zur Verbesserung zu nennen.

## 5.1 Präsentation von Ergebnissen

### 5.1.1 Generierte Modelle

Nachfolgend werden einige ausgewählte Ergebnisse vorgestellt. Dabei ist zu beachten, dass für die meisten dieser Ergebnisse mehrere Versuche mit verschiedenen Parametern durchgeführt wurden, um ein möglichst ansprechendes Ergebnis präsentieren zu können. Außerdem sind die Input-Modelle in diesen ersten Beispielen speziell für die Anwendung mit dieser Applikation kreiert worden, d. h. sie lassen sich in der Regel in sinnvolle $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ Muster einteilen, die wiederholt im Objekt vorkommen.

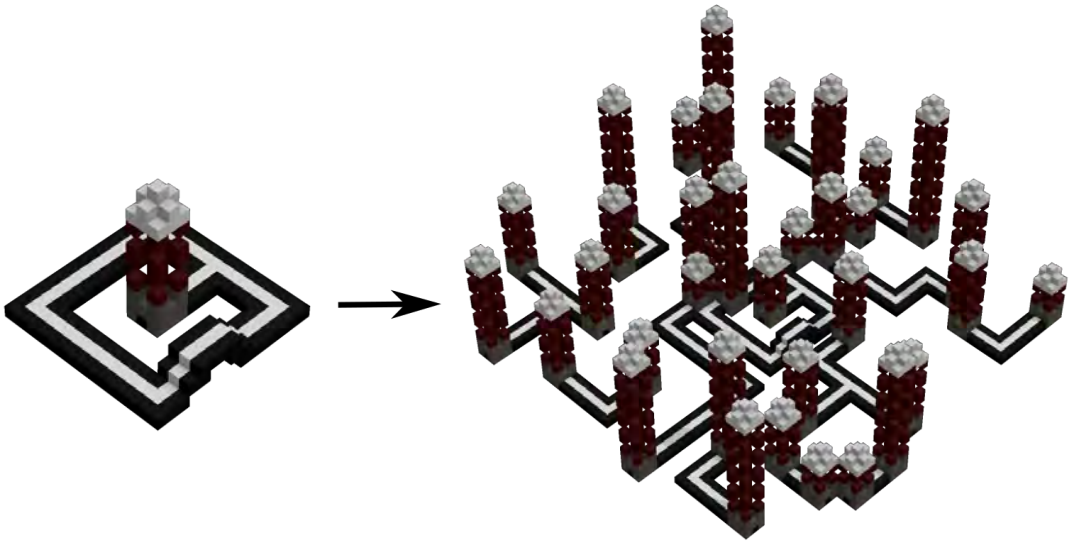


Abbildung 5.1: Ergebnis des Voxel-WFC mit einem Gebäude-ähnlichen Input-Modell

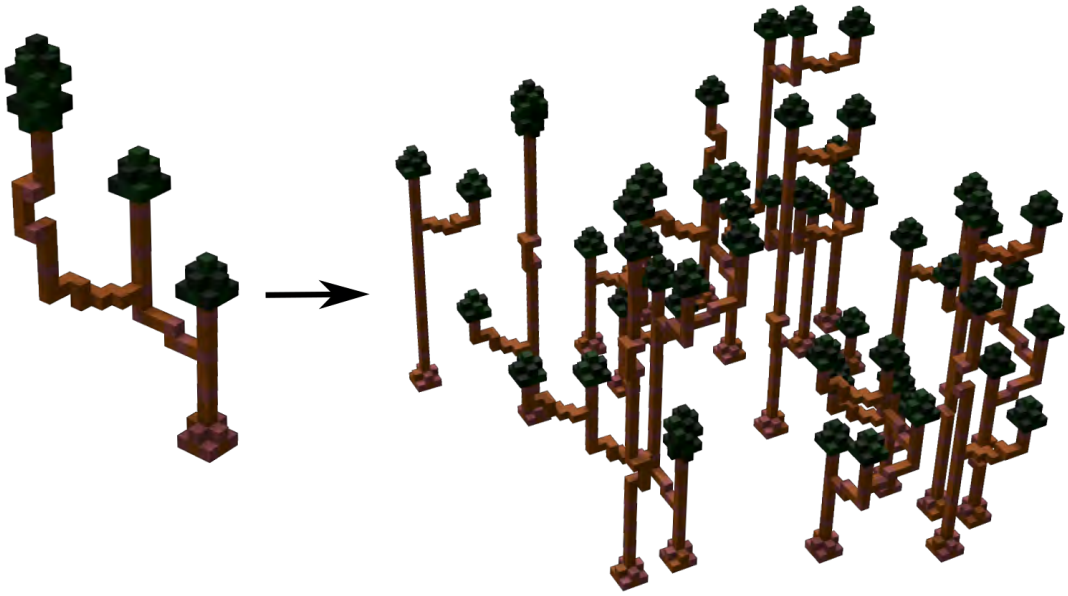


Abbildung 5.2: Ergebnis des Voxel-WFC mit einem Baum als Input-Modell

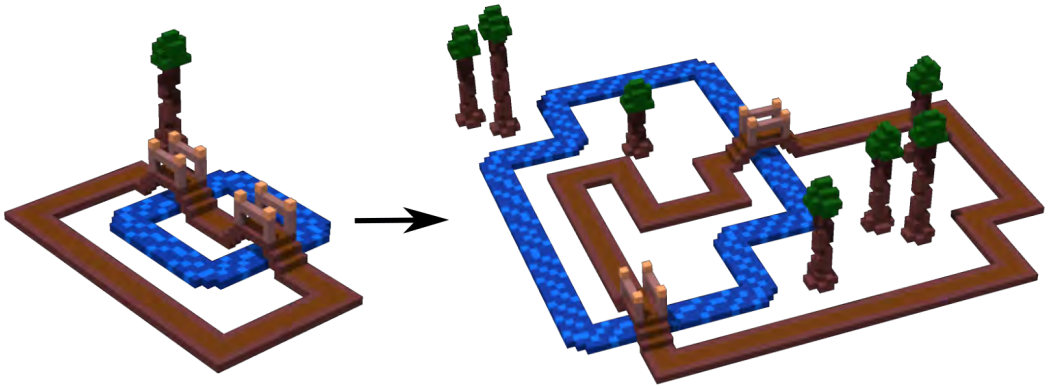


Abbildung 5.3: Ergebnis des Voxel-WFC mit einer Natur-Szene als Input-Modell

Am Ergebnis in Abbildung 5.1 lässt sich gut erkennen, wie Muster aus dem Input-Modell in verschiedenen Variationen im Output-Modell wiederzufinden sind. Zu beachten ist außerdem, dass durch die Einführung des Floor-Musters (siehe 3.3.2) im Output-Modell nur Muster Bodenkontakt haben, die diesen auch im Input-Modell haben (z. B. die Straßen Elemente).

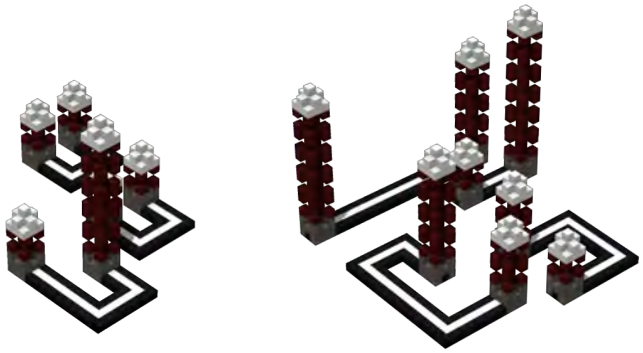
Das Baum-Modell aus Abbildung 5.2 demonstriert besonders gut das Verhalten des Algorithmus bei Modellen, die in die Höhe gehen. Auch hier lässt sich klar erkennen, welche Adjacency Constraints aus dem Input-Modell inferiert wurden. Eine Baumkrone kann z. B. nur am oberen Ende eines Stammes positioniert sein, eine Verzweigung muss horizontal an einen Stamm angrenzen.

In der Natur-Szene aus Abbildung 5.3 zeigt sich die Anwendbarkeit der Applikation als Generator für Szenen mit verschiedenen Modellen.

### 5.1.2 Parameter

Im Folgenden wird präsentiert, inwiefern sich einige der in 3.2.2 definierten Parameter auf die Ergebnisse des Algorithmus auswirken.

### Rotation

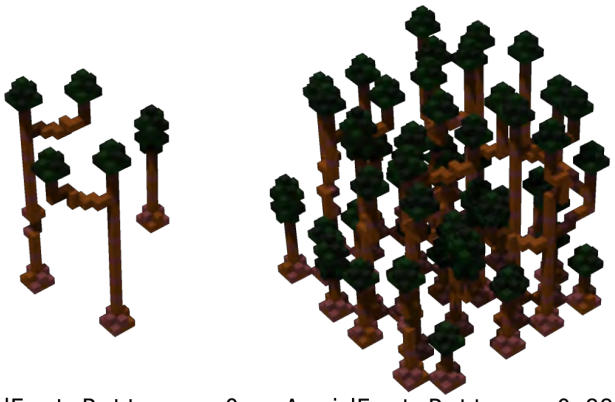


Rotation = false Rotation = true

Abbildung 5.4: Vergleich von generierten Modellen mit und ohne aktiviertem Rotations-Parameter

In Abbildung 5.4 lässt sich erkennen, dass z. B. die Richtung, in der ein Turm mit einer Straße verbunden ist, bei der Version mit nicht-aktivierter Rotation immer dieselbe ist. Ist der Parameter aktiviert, gibt es vier mögliche Richtungen. In den meisten Fällen führt eine Aktivierung des Parameters zu interessanteren Ergebnissen mit mehr Variation.

### Avoid Empty Pattern



AvoidEmptyPattern = 0 AvoidEmptyPattern= 0.99

Abbildung 5.5: Vergleich von generierten Modellen mit unterschiedlichen Werten für den AvoidEmptyPattern Parameter

Abbildung 5.5 zeigt deutlich, dass ein höherer Wert des AvoidEmptyPattern Parameters zu einer stärkeren *Füllung* des verfügbaren Raumes führt. Bei der Generierung ist es je nach gewünschtem Ergebnis häufig sinnvoll, mit diesem Parameter zu experimentieren, gerade wenn das Input-Modell viel leeren Raum enthält.

### 5.1.3 User-Interface


Abbildung 5.6: Screenshot des User-Interface der Applikation mit Markierungen für unterschiedliche Funktionsbereiche

Wie in Abbildung 5.6 zu sehen, setzt das User-Interface die in Abschnitt 3.1 definierten Anforderungen um und lässt sich dabei in verschiedene Bereiche aufteilen. In Bereich (1) kann der Nutzer ein Voxel-Modell importierten oder ein bereits geladenes Modell auswählen. In Bereich (2) können die verschiedenen Parameter des Algorithmus definiert werden. Bereich (3) enthält verschiedene Buttons, um z. B. die Generierung zu starten oder das Modell zu exportieren. Die Darstellung des Input bzw. Output-Modells findet schließlich in Bereich (4) statt.

Die Oberfläche erfüllt die gewünschten Anforderungen und bietet damit eine einfache Schnittstelle, um die Parameterisierung und Ausführung des Voxel-WFC zu kontrollieren. Der 3D-Viewer hilft außerdem, schnell ein Modell mit den gewünschten Eigenschaften zu generie-

ren, da das Ergebnis nicht erst exportiert und in einer anderen Software evaluiert werden muss.

## 5.2 Evaluation des Prototyps

### 5.2.1 Ziele und Anforderungen

Die in Rahmen dieser Arbeit entwickelte Anwendung hat die meisten der vorher definierten Ziele und Anforderungen erfüllt und liefert bei sorgfältiger Auswahl von Input-Modellen (subjektiv) gute Ergebnisse.

Die Erweiterung des WFC für Voxel-Modelle, war als zentrale Idee der Arbeit in vollem Umfang erfolgreich, ist allerdings stärker als zunächst gedacht von der Qualität der Input-Modelle abhängig. Die Anwendung ermöglicht außerdem das einfache Laden und Speichern von Modellen im gängigen *.vox* Format, was die parallele Arbeit mit anderer Voxel-Software einfach macht. Die Anforderungen an das User-Interface wurden wie schon in 5.1.3 beschrieben, ebenfalls umfassend erfüllt.

Die Anforderung, wenn möglich die Performance der Anwendung zu optimieren, wurde nur bedingt durch die Einführung des *Backtrackings* bearbeitet, stellte sich aber darüber hinaus aufgrund der Funktionsweise des WFC, als schwierig dar.

### 5.2.2 Probleme

Die im Folgenden beschriebenen Beispiele zeigen, unter welchen Umständen die Anwendung keine guten Ergebnisse liefert und wo noch Verbesserungspotenzial besteht.

#### Spezifische Anforderungen an Input-Modelle

Um gute Ergebnisse zu erzielen, müssen die Input-Modelle sehr spezifische Anforderungen erfüllen. Es ist z. B. wichtig, dass sich das Input-Modell in Teil-Muster der gleichen Größe aufteilen lässt, die sich zumindest teilweise an verschiedenen Positionen im Modell wiederholen. Ist dies nicht der Fall (wie z. B. im Modell aus Abbildung 5.7) können keine nützlichen

Adjacency Constraints abgeleitet werden, da jedes Muster nur einen möglichen Nachbarn hat und es ist kein Ergebnis bzw. nur das leere Ergebnis möglich.

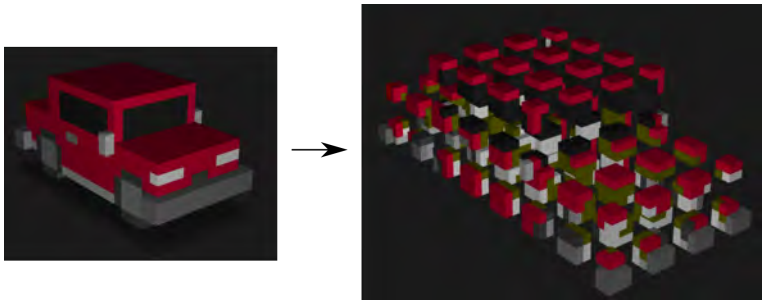


Abbildung 5.7: Beispiel für ein Input-Modell, dass nicht genug wiederholte Muster beinhaltet um es dem Voxel-WFC zu ermöglichen ein Ergebnis zu produzieren.[1]

### Schwebende Strukturen

Da es über die Adjacency Constraints hinaus keine weiteren Einschränkungen in Bezug auf die Positionierung der Muster im dreidimensionalen Raum gibt, kann es wie in Abbildung 5.8 dazu kommen, dass Teile des Outputmodells ohne Verbindung zum Boden oder dem Rest des Modells frei im Raum schweben. Dies muss natürlich je nach gewünschtem Ergebnis kein Problem sein. Aktuell gibt es aber auch keine Möglichkeit es zu verhindern.

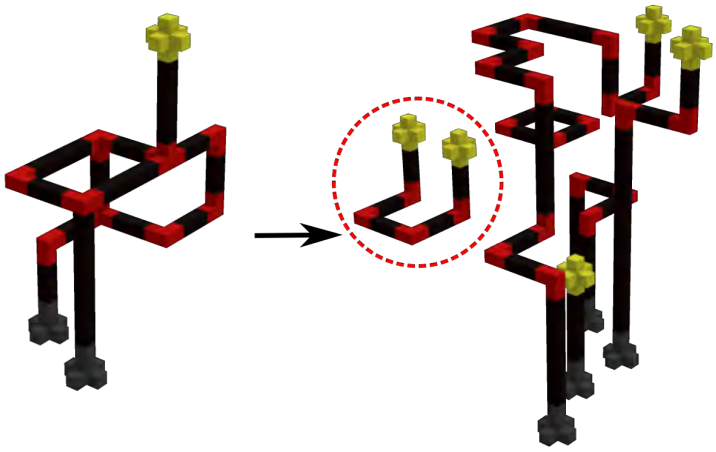


Abbildung 5.8: Ein Teil des Output-Modells (rot umkreist) schwebt losgelöst vom Rest der Struktur im Raum

[1] Voxel-Modell Quelle: https://github.com/mikelovesrobots/mmmm (abgerufen: 02.11.20)

#### Homogene Ergebnisse bei großen Input-Modellen

Ist das Input-Modell verhältnismäßig groß und beinhaltet sehr viele sich wiederholende Muster, kann es zu sehr homogenen Ergebnissen kommen, die bis auf die Farbe der Voxel wenig Ähnlichkeit mit dem Input-Modell haben.

#### Performance bei großen Modellen

Aufgrund der Arbeit in drei Dimensionen steigt der Aufwand des Algorithmus, insbesondere bei der Propagierung stark mit der gewünschten Größe des Output-Modells sowie der Größe des Input-Modells an. Zudem steigt auch die Wahrscheinlichkeit für einen Widerspruch mit der Größe des Output-Modells und damit auch die Laufzeit des Algorithmus. Außerdem macht es die Natur des WFC schwer, die Laufzeit vor der Generierung zu schätzen, da diese je nach Anzahl der Widersprüche stark variieren kann.

#### Toleranz bei der Musterfindung

Unterscheiden sich zwei Muster nur in der Farbe oder Position eines einzelnen Voxels, werden diese als zwei unterschiedliche Muster gespeichert. Diese Eigenschaft des Voxel-WFC schränkt die nutzbaren Input-Modelle weiter ein, da viele frei verfügbare Modelle in der Regel bewusste Variationen in sich wiederholenden Blöcken beinhalten.

## 5.3 Mögliche Anwendungsgebiete

Wie schon im ersten Kapitel angerissen, gibt es insbesondere in der Videospiel-Industrie einen großen Bedarf an 3D-Modellen und Methoden den Aufwand für deren Erstellung zu minimieren. Es liegt also nahe, dass die hier entwickelte Software sowie die gewonnenen Erkenntnisse in diesem Bereich Anwendung finden könnten. Ein spezieller Anwendungsfall wäre die Generierung von Modellen oder sogar ganzen Leveln für Spiele, die mit Voxel Grafik arbeiten. Ein 3D-Artist könnte in diesem Fall Beispielmodelle modellieren, um dann den Voxel-WFC zur Generierung von Modellen im selben Stil zu nutzen. Denkbar wäre auch die direkte Integration des Algorithmus in ein Spiel, um zur Laufzeit neue zufällige Inhalte zu generieren, wie es heute auch mit anderen Methoden zur prozeduralen Generierung üblich ist. Diese Arbeit könnte zudem als Vorlage dienen, um andere spezialisierte Versionen des

WFC in drei Dimensionen zu entwickeln und so z. B. die Arbeit mit normalen 3D-Modellen zu ermöglichen. Dies würde weitere Anwendungsmöglichkeiten in verschiedenen digitalen Medien erschließen.

## 5.4 Vergleich mit Voxel Synthesis for generative Design

Da die in Kapitel 2.6.1 vorgestellte Arbeit ebenfalls das Ziel verfolgt, größere Voxel-Modelle anhand von kleineren Beispielen zu generieren, wird hier ein kurzer Vergleich der beiden Ansätze vorgenommen. Das Simple-Model hat große Ähnlichkeit mit dem in dieser Arbeit entwickelten Ansatz, da es ebenfalls ein Input-Modell in $N \times N \times N$ große Teil-Muster einteilt und anhand deren Position im Input-Modell die entsprechenden Adjacency-Constraints erkennt.

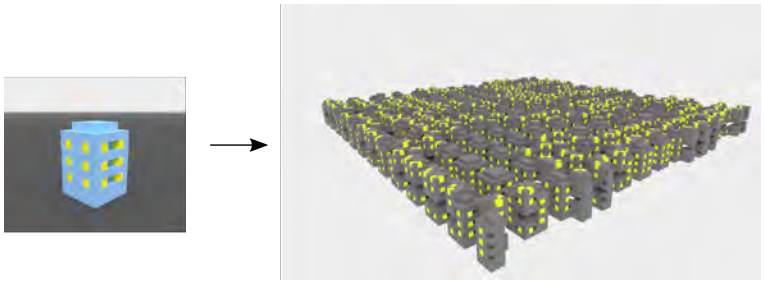


Abbildung 5.9: Ergebnis des Simple-Models mit einem einfachen Haus-Modell [2]

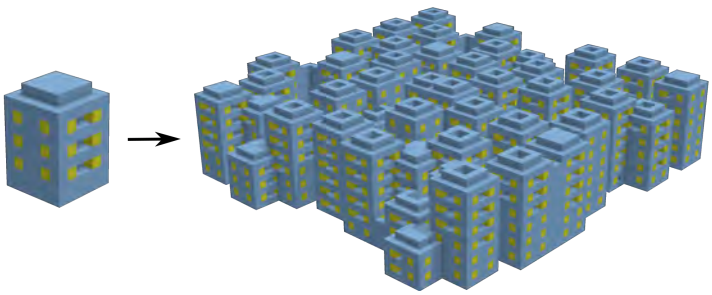


Abbildung 5.10: Ergebnis des hier entwickelten Voxel-WFC mit demselben Modell

[2]Bildquelle: https://github.com/MatveyK/Kazimir (abgerufen: 04.11.20)

# 6 Schluss

## 6.1 Zusammenfassung der Arbeit

Aufbauend auf der Arbeit von Max Gumin werden in dieser Arbeit ein Konzept und eine Anwendung vorgestellt, die den Wave Function Collapse Algorithmus für die Synthese von Voxel-Modellen erweitert. Die entwickelte Anwendung ist in der Lage, Voxel-Modelle als Beispiel zu laden, diese grafisch darzustellen und eine Konfiguration der Voxel-WFC Parameter über ein einfaches User-Interface zu ermöglichen. Auf Basis des geladenen Modells kann ein neues Modell im selben Stil generiert werden, wenn aus dem Input-Modell wiederholbare Muster inferiert werden können. Dabei hat der Benutzer Kontrolle über Parameter wie die Größe des Output-Modells, die Größe der zu suchenden Muster und weitere. Außerdem werden verschiedene verwandte Arbeiten und Grundlagen aus den Bereichen der Prozeduralen Generierung, der Computergrafik und der Textursynthese vorgestellt und mit der Problemstellung dieser Arbeit in Verbindung gesetzt.

Mit der vorgestellten Methode lassen sich durchaus beeindruckende Voxel-Modelle, die deutlich größer und komplexer als ihre korrespondierenden Input-Modelle sind, erzeugen. Es zeigt sich allerdings, dass die Qualität der Ergebnisse stark davon abhängt, ob sich die Input-Modelle in klar trennbare, sich wiederholende Muster einteilen lassen.

## 6.2 Ausblick

Im Laufe der Arbeit sind einige Möglichkeiten deutlich geworden, um die Funktionalität des Voxel-WFC zu erweitern oder zu verbessern, die den Rahmen der Arbeit überstiegen hätten:

### Overlapping-Model

Ähnlich wie die Autoren von *Voxel Synthesis for Generative Design* [13] hätte zusätzlich ein Overlapping Model implementiert werden können. Das heißt, anstatt das Input-Modell nur in diskrete Muster in bestimmten Abständen zu schneiden, würde Voxel für Voxel jedes mögliche Muster extrahiert werden und Adjacency Constraints aus dem Vergleich von überlappenden Voxeln am Rand der Muster erschlossen werden. Dies könnte die Anforderungen an Input-Modelle verringern und möglicherweise interessantere Ergebnisse liefern.

### Verschiedene Mustergrößen

Eine der größten Einschränkungen des Voxel-WFC ist, dass sich das Input-Modell aus gleichgroßen kubischen Mustern aufbauen muss, was die künstlerischen Möglichkeiten bei deren Design stark eingeschränkt. Fände eine Erweiterung statt, die es ermöglicht im selben Modell Muster in verschiedenen Größen zu erkennen, würde dies die mögliche Komplexität von Voxel-Modellen stark erhöhen.

### Connectivity Constraints

Ist es gewünscht, dass ein Output-Modell entweder mit dem Boden oder zu anderen Teilen des Modells verbunden ist (wie es z. B. bei einem Baum der Fall wäre) könnte eine optionale Einschränkung, hier Connectivity Constraint genannt, eingeführt werden. Diese wird im Algorithmus genau wie die Adjacency Constraints behandelt und schränkt ein welche Muster an welcher Position der Wave möglich sind.

### Heuristik zum Vergleich von Mustern

In dieser Arbeit werden Muster Voxel für Voxel verglichen, um festzustellen, ob zwei Muster äquivalent sind. Würde dieser Vergleich stattdessen durch eine Heuristik geschehen, die auch Muster mit großer Ähnlichkeit als äquivalent bewertet, könnte dies die Anforderungen an Input-Modelle senken und für größere Variation im Ergebnis sorgen.

# Literaturverzeichnis


[1] Agile Alliance: *Scrum*. – URL https://www.agilealliance.org/glossary/scrum/. – Zugriffsdatum: 27.10.2020

[2] Bucklew, Brian: *Dungeon Generation via Wave Function Collapse*. – URL https://www.youtube.com/watch?v=fnFj3dOKcIQ. – Zugriffsdatum: 01.10.2020

[3] c64 Wiki: *Turtle Graphics*. – URL https://www.c64-wiki.com/wiki/Turtle_Graphics_Interpreter. – Zugriffsdatum: 05.10.2020

[4] Efros, A. A. ; Leung, T. K.: Texture synthesis by non-parametric sampling. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision* Bd. 2, 1999, S. 1033–1038 vol.2

[5] Fowler, Martin: *Model View Controller Architecture*. – URL https://www.martinfowler.com/eaaDev/uiArchs.html#ModelViewController. – Zugriffsdatum: 25.10.2020

[6] Freehold Games: *Caves of Qud Steam Page*. – URL https://store.steampowered.com/app/333640/Caves_of_Qud/. – Zugriffsdatum: 03.10.2020

[7] Gumin, Maxim: *Official Git Page WFC*. – URL https://github.com/mxgmn/WaveFunctionCollapse. – Zugriffsdatum: 08.09.2020

[8] Harris, Frank E.: *Mathematics for Physical Science and Engineering*. 2014. – 163–167 S

[9] Karth, Isaac ; Smith, Adam: WaveFunctionCollapse is constraint solving in the wild, 2017

[10] Kaufman, Arie ; Cohen, Daniel ; Yagel, Roni: Volume Graphics. In: *IEEE Computer* 26 (1993), Nr. 7

[11] Kaufman, Arie E.: Voxels as a Computational Representation of Geometry, 1994

[12] KHALED, Rilla ; NELSON, Mark ; BARR, Pippin: Design metaphors for procedural content generation in games. In: *Conference on Human Factors in Computing Systems - Proceedings* (2013), 04, S. 1509–1518

[13] KHOKHLOV, Matvey: Voxel Synthesis for Generative Design. In: *Design Computing and Cognition '18* (2018), S. 227–244

[14] LIN YUAN ; KESAVAN, H. K.: Minimum entropy and information measure. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 28 (1998), Nr. 3, S. 488–491

[15] LINDENMAYER, Aristid: Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. In: *Journal of Theoretical Biology* 18 (1968), Nr. 3, S. 280 – 299

[16] MARTIN, Robert C.: *Design Principles and Design Patterns*. 2000. – URL https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf. – Zugriffsdatum: 25.10.2020

[17] MEGA VOXELS: *BEST VOXEL EDITORS FOR 2020*. – URL https://www.megavoxels.com/2019/08/best-voxel-editors.html. – Zugriffsdatum: 24.10.2020

[18] MERRELL, Paul C.: *Model Synthesis*. 2009

[19] N., Jordan: *Polycounts in next gen games*. – URL https://polycount.com/discussion/141061/polycounts-in-next-gen-games-thread. – Zugriffsdatum: 01.09.2020

[20] PERSSON, Markus A.: *Artikel zu Minecrafts Terrain Generation*. – URL https://notch.tumblr.com/post/3746989361/terrain-generation-part-1. – Zugriffsdatum: 02.09.2020

[21] PRUSINKIEWICZ, Przemyslaw ; LINDENMAYER, Aristid: Graphical modeling using L-systems. In: *The Algorithmic Beauty of Plants* (1990)

[22] QT DOCUMENTATION: *Model/View Programming*. – URL https://doc.qt.io/qt-5/model-view-programming.html. – Zugriffsdatum: 27.10.2020

[23] SHANNON, C. E.: A mathematical theory of communication. In: *Bell Sys. Tech. Journal* 27 (1948)

[24] SHUTE, Gary: *Principles of Software Engineering*. – URL https://www.d.umn.edu/~gshute/softeng/principles.html. – Zugriffsdatum: 29.10.2020

[25] SPEEDTREE INC.: *Speedtree Official Website*. – URL https://store.speedtree.com/speedtree/. – Zugriffsdatum: 06.10.2020

[26] STALBERG, Oscar: *Townscaper Steam Page*. – URL https://store.steampowered.com/app/1291340/Townscaper/. – Zugriffsdatum: 02.10.2020

[27] STEELAND, Merel: *Townscaper Oscar Stalberg*. – URL https://procedural3d.be/townscaper-oskar-stalberg/. – Zugriffsdatum: 02.10.2020

[28] STINY, George ; GIPS, James: Shape Grammars and Generative Specification. In: *IFIP Congress 71*, 1971

[29] TOGELIUS, Julian ; KASTBJERG, Emil ; SCHEDL, David ; YANNAKAKIS, Georgios N.: What is Procedural Content Generation? Mario on the borderline. (2011)

[30] WEI, L. ; LEFEBVRE, S. ; KWATRA, Vivek ; TURK, Greg: State of the Art in Example-based Texture Synthesis. In: *Eurographics*, 2009

[31] WEI, Li-Yi ; LEVOY, Marc: *Fast Texture Synthesis using Tree-structured Vector Quantization*. In proceedings of Siggraph 2000. 2000

[32] WIKI, PCG: *Artikel zu PCG in Minecraft*. – URL http://pcg.wikidot.com/pcg-games:minecraft. – Zugriffsdatum: 02.09.2020

[33] WIKIPEDIA: *Waterfall model*. – URL https://en.wikipedia.org/wiki/Waterfall_model. – Zugriffsdatum: 27.10.2020

[34] WIKIPEDIA: *Wave Function Collapse Definition Quantum Mechanics*. – URL https://en.wikipedia.org/wiki/Wave_function_collapse. – Zugriffsdatum: 20.09.2020

[35] WIKTIONARY: *Shannon Entropy*. – URL https://en.wiktionary.org/wiki/Shannon_entropy. – Zugriffsdatum: 18.10.2020

# Abbildungsverzeichnis

# Glossar

**Adjacency Constraints** Regeln die bestimmen, welche Nachbarn für ein bestimmtes Muster erlaubt sind.

**Backtracking** Wird z. B. bei der Durchführung eines Algorithmus ein früherer Zustand wiederhergestellt, also ein Schritt zurück getätigt, bezeichnet man dies als *Backtracking*.

**Grafik-Assets** Grafische Inhalte wie z.B. 3D-Modelle oder Bilder, die in digitalen Medien genutzt werden.

**Padding** Eine *Polsterung* eines Bildes oder Modells mit leeren Elementen an den Rändern des Objekts.

**Polygon** Element eines 3D-Modells, stellt eine Fläche des Objekts dar.

**Seed** Eine Zahl, die als Basis für einen Zufallszahlengenerator genutzt wird, um reproduzierbare Zufallszahlen zu generieren.

**Textur** Ein zweidimensionales digitales Bild.

**Tile** Ein einzelnes Element einer Tilemap.

**Tilemap** Eine Sammlung von Bildern, auch Tiles genannt, die in bestimmten Konfigurationen zusammenpassen, um ein sinnvolles neues Bild zu erzeugen.

# Abkürzungen

**LIFO** Last-In-First-Out.

**PCG** Prozedurale Content Generierung.

**UI** User-Interface.

**WFC** Wave Function Collapse.

## Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

*Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI*

## Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

**Erweiterung des Wave Function Collapse Algorithmus zur Synthese von 3D-Voxel-Modellen anhand von nutzergenerierten Beispielen**

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und 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 |