HAW HAMBURG

MASTER THESIS
Christian Dorn

Beispiel-basierte Synthese polygonaler Netze mit dem Wave-Function-Collapse Algorithmus durch Distanzfunktionen mit Marching Cubes

FAKULTÄT TECHNIK UND INFORMATIK
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Christian Dorn

Beispiel-basierte Synthese polygonaler Netze mit dem Wave-Function-Collapse Algorithmus durch Distanzfunktionen mit Marching Cubes

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Automatisierung*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Christian Lins

Eingereicht am: 19. Juni 2023

Christian Dorn

Thema der Arbeit

Beispiel-basierte Synthese polygonaler Netze mit dem Wave-Function-Collapse Algorithmus durch Distanzfunktionen mit Marching Cubes

Stichworte

Marching Cubes, Wave-Function-Collapse, Distanzfunktion, Prozedural, Synthese, Beispiel-basiert, Dreiecksnetze

Kurzzusammenfassung

Der Wave-Function-Collapse (WFC) Algorithmus ermöglicht es, durch beispiel-basierte Eingaben prozedural ähnlich aussehende Ausgaben zu erzeugen. Je nach Implementierung können verschiedenste Datentypen und Dimensionen benutzt werden, weswegen im Rahmen von vorherigen Projekten ein generisches Framework erstellt wurde, welches sich modular für neue Typen erweitern lässt. Bisherige Versuche eine Variante mit Dreiecksnetzen zu implementieren sind wegen zu hoher Komplexität des dreidimensionalen Raums gescheitert, weswegen in dieser Arbeit eine Lösung entwickelt wurde. Möglich war dies durch eine Detailreduktion mithilfe von Distanzfunktionen, Nutzung von WFC und anschließender Visualisierung mithilfe von Marching Cubes und zeigt trotz langer Laufzeit und vereinfachten Ausgaben die Möglichkeit auf, den WFC-Algorithmus auf beliebige Dreiecksnetze anwenden zu können.

Christian Dorn

Title of Thesis

Example-driven synthesis of triangle meshes using Wave-Function-Collapse with Marching Cubes and signed distance functions

Keywords

Marching, Cubes, Signed Distance Function, Synthesis, Example-driven, Triangle Mesh, Wave-Function-Collapse, Procedural

Abstract

The Wave-Function-Collapse (WFC) algorithm is able to procedurally generate outputs using features from an example input. Different implementations enable the usage of unique data types and dimensions leading to the development of a generic WFC-Framework, which makes it possible to create new modules if needed. Due to the inherent complexity and detail of three dimensional space an implementation using triangle meshes was yet not achieved. In this work a solution using signed distance functions and marching cubes for visualization is created to be able to use the Wave-Function-Collapse algorithm together with triangle meshes. Although the generation process can take multiple minutes, it shows the possibility to use the algorithm with any triangle meshes.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

CSG Constructive Solid Geometry.

CT computed tomography.

HAW Hochschule für Angewandte Wissenschaften.

LUT Look-Up-Table.

MC Marching Cubes.

MRI magnetic resonance imaging.

MS Marching Squares.

WFC Wave-Function-Collapse.

1 Einleitung

Prozedurale Generierung von Inhalten bietet die Möglichkeit mit nur wenig Aufwand viele abwechslungsreiche Inhalte zu generieren. Durch verschiedenste Algorithmen können mithilfe von Parametrisierung und Zufall unter anderem Inhalte wie 3D-Modelle, Musik oder Texturen erzeugt werden [31]. Dies hat den Vorteil, dass schnell neue Vorlagen für das weitere Vorgehen erzeugt oder sogar direkt als Inhalte übernommen werden können.

Eine Unterkategorie von solchen Algorithmen bildet die Textursynthese, bei welcher mithilfe einer Inputtextur eine neue größere Ausgabe unter Nutzung des Inhalts der Eingabe erzeugt wird. Hier werden verschiedenste Verfahren genutzt, begonnen bei einfachem aneinanderreihen der Eingabe und kaschieren von entstehenden Kanten [13], übergehend zur Synthese nach Efros und Leung [7], bis hin zu modernen Machine-Learning Verfahren mit Faltungsnetzen [9].

Ein besonderes Beispiel für einen Texturesynthesealgorithmus bildet der Wave-Function-Collapse (WFC) Algorithmus. Durch die visuell ansprechende Generierung hat der WFC-Algorithmus auf GitHub und in der Indie-Spiele-Entwickler-Szene viel Aufmerksamkeit gewonnen [21]. Anders als die bisherigen Texturesynthesealgorithmen lässt sich dessen allgemeines Konzept nicht nur für Texturen anwenden, sondern auch für weitere Datenstrukturen. Dementsprechend wurden viele Implementierungen erstellt (siehe Abschnitt 2.1.5), die unter anderem das Kernkonzept auf weitere Datenstrukturen erweitert haben.

1.1 Motivation

Die vielen verschiedenen, spezialisierten Implementierungen des WFC-Algorithmus haben zur Folge, dass mit jeder neuen Implementierung der Algorithmus speziell für den Anwendungszweck neu implementiert werden musste, während der Kernalgorithmus gleich bleibt. Dies hat sich auch im Rahmen der Computergrafik Forschungsgruppe an der

Hochschule für Angewandte Wissenschaften (HAW) zu einem Problem entwickelt, da der WFC-Algorithmus wiederholt von Grund auf erarbeitet wurde. Daher wurde eine Implementierung des WFC-Algorithmus erstellt, welche eine generische Plattform bietet, bei der zusätzliche Module für beliebige Datentypen in verschiedenen Dimensionen hinzugefügt werden können [4, 5]. Unter anderem lassen sich bisher Texte, Bitmaps, Voxel und auch 3D-Tilesets für die prozedurale Generierung mithilfe des WFC-Algorithmus nutzen.

Weiterhin wurde in eigener Arbeit versucht ein Modul für Dreiecksnetze zu implementieren, welches es ermöglicht, beliebige Modelle als Eingabe zu benutzen und anhand dieser ein neues Modell prozedural zu generieren. Diese Versuche sind bisher durch die Komplexität des dreidimensionalen Raums und der verwendeten Datenstrukturen gescheitert, weswegen nun im Rahmen dieser Arbeit eine Lösung erarbeitet werden soll.

1.2 Zielsetzung

Das bisher implementierte WFC-Framework soll um eine weitere Komponente ergänzt werden, welche die Beispiel-basierte Generierung von Dreiecksnetzen ermöglicht. Dabei soll die Struktur den bisherigen weiteren Modulen gleichen und dieselbe Implementierung des WFC-Algorithmus nutzen. Um die zuvor begegneten Probleme zu umgehen, müssen die eingegebenen Polygonnetze abstrahiert werden, sodass die damit verbundene Komplexität umgangen werden kann.

Als zusätzliche Komponente soll eine Benutzeroberfläche hinzugefügt werden. Durch die bisherigen Probleme mit der Komplexität des Problems soll diese als Hilfestellung bei der Entwicklung, aber auch bei der Nutzung zur Generierung dienen.

1.3 Aufbau

Die Arbeit ist in folgende sieben Kapitel aufgeteilt:

Einleitung

Grundlagen und Stand der Technik Es wird der WFC-Algorithmus und seine Mechanismen, Variationen und Stand der Technik vorgestellt. Zusätzlich werden Distanzfunktionen und der Marching Cubes (MC) Algorithmus erläutert.

Analyse Die in den Grundlagen erwähnten Aspekte des WFC-Algorithmus werden aufgegriffen und daraus resultierende Probleme festgestellt. Weiterhin werden anhand der Zielsetzung Anforderungen an die Software bestimmt und benötigte Zusatzkomponenten ausgewählt.

Konzept Basierend auf der Analyse wird ein Softwarekonzept entwickelt und dessen Details vorgestellt. Die Vorgehensweise und zu entwickelnde Algorithmen werden erläutert.

Implementierung Das Endprodukt wird präsentiert. Es wird auf Details der Architektur und die Kommunikation der einzelnen Komponenten eingegangen. Zusätzlich werden während der Entwicklung entstandene Probleme beschrieben.

Evaluation Es wird die entwickelte Software auf verschiedene Aspekte bewertet. Unter anderem wird die Qualität der prozeduralen Modelle und Laufzeit der Algorithmen analysiert.

Fazit Zusammenfassung aller Ergebnisse und ein Ausblick auf mögliche Erweiterungen.

1.4 Umfang

Dieses Projekt wurde iterativ im Rahmen des Informatik Masters an der HAW ausgearbeitet. Dieses besteht jeweils aus dem Grundprojekt, Hauptprojekt und abschließendem Masterprojekt. In dem Grundprojekt wurden die Grundlagen des WFC-Algorithmus erarbeitet und das generische Framework implementiert, welches eine dimensions- und datentypunabhängige Nutzung ermöglicht [4].

Im Rahmen des Hauptprojekts wurden erste Schritte im dreidimensionalen Raum durchgeführt und Erweiterungen für das Framework erstellt. Konkret wurden Implementierungen für Voxel und Tilesets aus Dreiecksnetzen betrachtet [5].

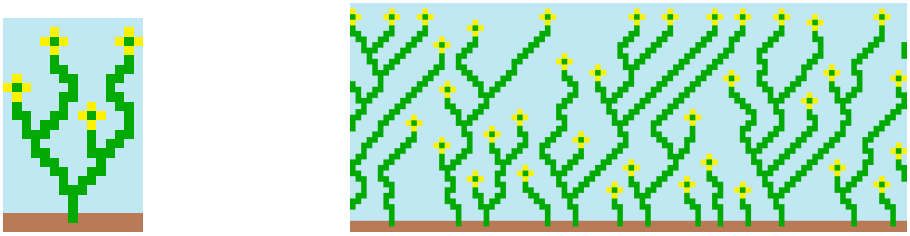
Zuletzt soll nun im Folgenden erarbeitet werden, wie sich beliebige Dreiecksnetze als Beispieleingabe für den WFC-Algorithmus nutzen lassen. Hierfür werden Distanzfunktionen als Abstraktion eingeführt und eine vorhandene Implementierung des Marching Cubes Algorithmus adaptiert.

2 Grundlagen und Stand der Technik

In diesem Kapitel werden die für das Projekt benötigten drei Kerngrundlagen aufbereitet dargelegt. Es wird jeweils der WFC-Algorithmus, MC-Algorithmus und das Konzept von Distanzfunktionen erläutert.

2.1 Wave-Function-Collapse

Der WFC-Algorithmus [12] ist ein Textursynthesalgorithmus, welcher anhand von einer Bitmap weitere Bitmaps erzeugen kann, welche lokale Ähnlichkeiten aufweisen. Tauchen Muster mehrfach im gleichen Kontext auf, so werden diese erfasst und nahtlos in der Ausgabe nebeneinander platziert und bietet damit die Möglichkeit, die Ausgabe beliebig groß zu gestalten. Abbildung 2.1a und 2.1b zeigen jeweils ein typisches Beispiel, wo eine einfache Blume im Pixel-Art-Stil eingegeben wird und anhand dessen ein deutlich größeres Bild erzeugt wird.



(a) Eingabe. (b) Ausgabe.

Abbildung 2.1: Beispielein- und Ausgabe aus Gumins Repository [12].

Entwickelt wurde der Algorithmus 2016 von Maxim Gumin und lässt sich auf GitHub finden[1]. Gumin hat dabei Inspiration aus den Arbeiten von Merrell gezogen [26], [28], [29]. Merrell zeigt in diesen einen Ansatz zur Modellsynthese, bei dem ein 3D-Modell eingegeben und basierend darauf ein Modell mit lokalen Ähnlichkeiten generiert wird.

[1] https://github.com/mxgmn/WaveFunctionCollapse

Dieses Konzept wurde von Gumin übernommen und für die Textursynthese eingesetzt, mit dem Resultat des WFC-Algorithmus.

Gumin präsentiert hierbei zwei verschiedene Varianten des Algorithmus, das SimpleTiledModel und das OverlappingModel. Beide nutzen das gleiche Konzept und unterscheiden sich nur im Eingabetyp und Verhalten beim Einlesen. Es werden zuerst die beiden Modelle mit ihren Eigenschaften erläutert und anschließlich der Kernalgorithmus.

2.1.1 SimpleTiledModel

Das SimpleTiledModel nimmt als Eingabe *Tilesets*. Diese bestehen aus mehreren, gleichförmigen Bildern, welche zusammen angeordnet verschiedene Strukturen, Levelgeometrien oder größere Bilder ergeben. Zusätzlich dazu muss ein *Constraint-File* angegeben werden, welches definiert, inwiefern die einzelnen Tiles nebeneinander liegen dürfen. Für jedes Tile muss bestimmt werden, welche Tiles jeweils in den vier verfügbaren Richtung (Oben, Unten, Links, Rechts) liegen dürfen.

Sollen Tiles rotiert werden können, so muss zusätzlich das Symmetrieverhalten näher bestimmt werden. Je nachdem ob ein Tile punktsymmetrisch, achsensymmetrisch oder gar keine Symmetrie hat, treten verschiedene Sonderfälle bei den Constraints und der späteren Generierung auf.

Am Ende der Generierung liegt ein Gitternetz von angeordneten Tiles vor, welche die definierten Nachbarschaftsregeln respektieren und abschließend in ein geeignetes Format gespeichert werden können.

2.1.2 OverlappingModel

Anders als das SimpleTiledModel benötigt das OverlappingModel keine zusätzlichen Regeln in Form eines Constraint-Files. Hier reicht eine Eingabe in Form eines Bildes und einer Fenstergröße als Parameter. Beim OverlappingModel werden die Tiles mithilfe eines gleitenden Fensters erfasst, welches durch die Fenstergröße definiert wird. Das Bild wird dann pixelweise mit dem Fenster abgetastet und jedes neue Muster in einem Tile speichert. Nachdem alle Muster identifiziert worden sind, werden die Regeln durch die mögliche Überlappung von verschiedenen Mustern inferiert.

Auch hier liegt nach der Generierung ein Gitternetz aus angeordneten Tiles unter Beachtung der Nachbarn vor. Anders als beim SimpleTiledModel werden die Muster allerdings beim Übertragen in ein geeignetes Format einander überlappend platziert, da durch das Abtasten mit dem gleitenden Fenster die Muster überlappend abgetastet wurden und nun wieder überlappend platziert werden müssen. Dies führt zu sehr organischen Übergängen zwischen den einzelnen Tiles.

2.1.3 Algorithmus

In der originalen Implementierung von Gumin arbeitet der WFC-Algorithmus auf einem Gitternetz. Ziel und Terminierungskriterium des Algorithmus ist es, dieses Gitternetz von Zellen mit jeweils einem gültigen Tile zu versehen. Jede Zelle beinhaltet einen Zustandsraum (Liste von beinhalteten Tiles), welche initial mit allen verfügbaren Tiles gefüllt ist.

Ab hier werden zwei Schritte abwechselnd durchgeführt: Kollaps und Propagation. Es wird eine Zelle nach einem Schema (Zufall, wenigste Zustände oder Entropie) ausgewählt und aus dessen übrigen Zustandsraum ein Zustand ausgesucht und für diese Zelle festgelegt. Im nächsten Schritt muss diese Änderung an die Nachbarzellen weiter propagiert werden, da durch die Nachbarschaftsrelationen der festgelegten Zelle nun auch die Zustandsräume der Nachbarn angepasst werden müssen. Wurden die Nachbarn angepasst, so müssen diese ebenfalls ihre Änderungen an deren Nachbarzellen weiter propagieren, da auch dessen Nachbarn nun durch die Regeln ihren Zustandsraum verringern müssen. Wurden alle nötigen Zellen aktualisiert, kann eine weitere Zelle zum Kollabieren gewählt werden. Listing 2.1 zeigt diesen Prozess in Pseudocode.

```
# Collapsing
collcell = getNextCell() #
collapsedState = collcell.getRandomState() # Use weights
collcell.states.clear()
collcell.states.add(collapsedState)

Stack toUpdate = Stack()
# Propagation
while toUpdate is not empty:
    cell = toUpdate.pop()
    for each cell neighbor:
        if neighbor is collapsed:
            break
        allowedStates = []
        # Collect all allowed states in direction of neighbor
        for each state:
            allowedStates.addAll(state.allowedStatesToNeighbor)
        neighbor.states.intersect(allowedStates) # set operation
        if neighbor.states has changed:
            neighbor.push(toUpdate)
```

Listing 2.1: Pseudocode für Propagation und Kollabierung im WFC-Algorithmus

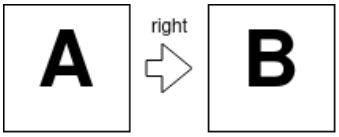
Dieser Prozess bildet den namensgebenden „Collapse“ in Anlehnung an Mechanismen der Quantenmechanik, wo nach Betrachten von Elementarteilchen dessen Zustand festgelegt wird.

Rotation und Symmetrie

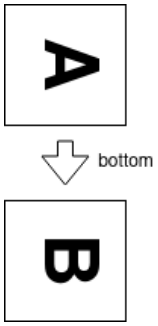
Gefundene Muster können zusätzlich rotiert und gespiegelt werden, um noch mehr Varietät in die Ausgabe zu bringen. Diese lassen sich als neue Muster betrachten, allerdings muss bei diesem Prozess zusätzlich auf die abgeleiteten Regeln geachtet werden. Wird ein Muster rotiert, so müssen auch die Regeln, sowie das betroffene Muster innerhalb der Regeln angepasst werden, wie in Abbildung 2.2a und 2.2b dargestellt.

Konflikt

Ist die Eingabe besonders komplex, so kann es zu einem Konflikt in der Generierung kommen, sodass ein Tile plötzlich gar keine Zustände mehr übrig hat, da benachbarte Tiles



(a) Festgelegte Regel: B darf nur rechts von A sein.



(b) Abgeleitete Regel bei Rotationen. Um 90° gedrehtes B darf nur unter 90° gedrehtes A sein.

Abbildung 2.2: Regelverhalten bei Rotationen.

zu viele Regeln einfordern. Anstatt durch das Propagieren auf einen Zustand reduziert zu werden, hat diese Zelle nun gar keinen möglichen Zustand und ist damit ungültig. An dieser Stelle können zwei Schritte durchgeführt werden. Einerseits kann die Generierung neu gestartet werden, in der Hoffnung, dass ein solcher Konflikt nicht erneut eintritt, andererseits kann auch ein Backtracking-Mechanismus genutzt werden, welcher die getätigten Schritte bis zur letzten wichtigen Entscheidung zurückverfolgt und den Konflikt damit umgeht.

Zellengewichtung

Tauchen in der Eingabe bestimmte Muster vermehrt auf, so muss diese Häufigkeitsverteilung auch beim WFC-Algorithmus berücksichtigt werden. Hier kann das vermehrte Auftreten von bestimmten Mustern gezählt werden, sodass eine Wahrscheinlichkeitsverteilung während des Auswählens eines Tiles beim Kollabieren benutzt werden kann.

Entropie

Wenn die nächste Zelle zum Kollaps ausgewählt wird, kann zufällig vorgegangen werden. Dies hat allerdings den Nachteil, dass eher Konflikte auftreten können, da dadurch zwei nahe bei einander liegende Zellen eventuell nicht vereinbare Zustände erhalten, wodurch die Generation nicht fortgesetzt werden kann.

Als Nächstes bietet sich es sich am ehesten an, Zellen zu wählen, welche zum aktuellen Zeitpunkt die wenigsten Zustände haben und noch nicht auf einen Zustand reduziert wurden. Dies sorgt dafür, dass sich die Generierung von der Startzelle aus ausbreitet, da die benachbarten Zellen einer kollabierten Zelle durch die Nachbarschaftsrelationen immer weniger mögliche Zustände haben, als Zellen in einem Bereich, wo noch keine Zelle kollabiert wurde.

Problem an diesem Ansatz ist, dass hier die Gewichtung der Zellen nicht beachtet wird. Hat beispielsweise eine Zelle nur drei übrige Zustände mit jeweils sehr hohen Gewichtungen und eine weitere Zelle zwei Zustände mit sehr niedrigem Gewicht, so wäre es sinnvoller die Zelle mit hohen gewichteten Zellen zu kollabieren. Daraus ergibt sich der typisch anschauliche Generierungsprozess, bei dem zuerst grobe Konturen und dann die Details generiert werden.

Um das Maß von Zustandsanzahl und Gewichtung zu kontrollieren, wird die Shannon-Entropie eingesetzt. Für jede Zelle wird die Entropie berechnet, welche den Informationsgehalt einer Zelle bestimmt, woraus eine Kombination aus der Zustandsanzahl und Gewichtung entsteht und mit der Formel

$$\text{Entropie} = \log\left(\sum_{i=1}^{n} w_i\right) - \left(\frac{\log(\sum_{i=1}^{n} w_i * \log(w_i)}{\sum_{i=1}^{n} w_i}\right)$$

beschrieben wird (mit w als Gewichtung eines Zustands). Nach diesem Entropiewert kann nun sortiert werden, um die Zelle mit dem niedrigsten Entropiewert als Nächstes zu kollabieren.

2.1.4 Unterschiede zur Modelsynthesis

Im Kern arbeiten der Algorithmus von Merrell und Gumins WFC-Algorithmus ähnlich und nutzen das gleiche Konzept, aber unterscheiden sich dennoch in manchen Details. Primäre Hauptunterschiede formuliert Merrell als verschiedenes Vorgehen während der Auswahl zum Kollabieren der nächsten Zelle und Gumins Neuerung vom Überlappen von Tiles [27].

Bei der Modelsynthesis geht der Algorithmus zeilen- und spaltenweise vor, während beim WFC-Algorithmus das Maß der Entropie genutzt wird. Als Folge dessen schlägt der WFC-

Algorithmus öfter bei sehr großen Ausgaben fehl, während die Modelsynthesis große Ausgaben ohne Probleme handhaben kann.

Um große Ausgaben zu ermöglichen wird in der Modelsynthesis zusätzlich das Problem in Blöcke unterteilt, welche nacheinander generiert werden. WFC dagegen besitzt keine solchen Mechanismen und schlägt deswegen insbesondere im Dreidimensionalen öfter fehl.

2.1.5 Stand der Technik

Der WFC-Algorithmus hat den Vorteil, dass sobald die Relationen der einzelnen Muster festgelegt worden sind, der eigentliche Inhalt keine Rolle für den restlichen Ablauf des Algorithmus spielt. Dadurch ist es möglich, nicht nur Texturen zu nutzen, sondern auch weitere Datentypen und auch weitere Dimensionen. Dies hat viele weitere Implementierungen ermöglicht, welche exemplarisch im Folgendem gezeigt werden.

Implementierung mit speziellen Datentypen

Durch die Popularität des WFC-Algorithmus gibt es viele Implementierungen mit neuen Datentypen. Gumin verweist innerhalb seines GitHub-Repositories auf Implementierungen, welche seinen Ansatz nutzen. Unter anderem lassen sich dort Implementierungen für generierte Gedichte[2], Voxelvarianten in 3D[3], Städtegenerierung[4] und Musikerzeugung[5] finden.

Erweiterungen

Auch größere Erweiterung zum Kernalgorithmus von WFC wurden erstellt. Die Autoren Kim et al. haben sich mit der Gitterabhängigkeit vom WFC-Algorithmus beschäftigt [22]. Hier wurde eine Variante implementiert, welche auf Graphen basiert und damit beliebige Dimensionen und Formen annehmen kann.

[2] https://github.com/mewo2/oisin
[3] https://github.com/MatveyK/Kazimir
[4] https://marian42.itch.io/wfc
[5] https://github.com/bbaltaxe/wfc-piano-roll

In [30] wurde eine Variante entwickelt, welche sich mithilfe von Growing Grids auf prozedurale Städte spezialisiert.

Weiterhin zeigen die Autoren von [8] eine Implementierung innerhalb einer Augmented Reality Anwendung, welche Räume einscannen und mithilfe von WFC dynamisch neue Räume generieren kann.

2.2 Marching Cubes

Der MC-Algorithmus ermöglicht es, aus einem Skalarfeld Isoflächen zu identifizieren und diese mit einem angenäherten Dreiecksnetz darzustellen. Dadurch ist es möglich, aus einem gitterförmigen Datensatz vollständige Dreiecksnetze zu generieren und damit diesen Datensatz zu visualisieren. Abbildung 2.3 zeigt dies im zweidimensionalen anhand eines Polygons (grün), einem Gitternetz aus Skalarwerten, welche angeben ob diese sich inner- ober außerhalb des Polygons befinden (rot und blau) und der anschließenden Rekonstruierung (rosa) mit Marching Cubes (Squares, siehe Abschnitt 2.2.2)

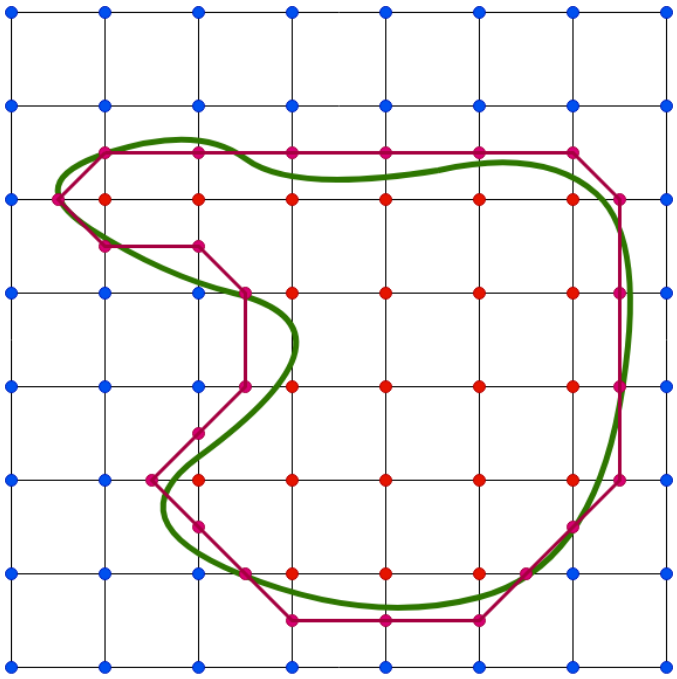


Abbildung 2.3: Darstellung des Marching Cubes (Squares) Algorithmus. Polygon (grün) wird durch Skalarfeld (rot und blau) rekonstruiert (rosa).

Entwickelt wurde der Algorithmus von Lorensen und Cline in 1987. Das primäre Einsatzgebiet ist die Visualisierung für medizinische 3D-Scans. In Verfahren wie *computed tomography (CT)* und *magnetic resonance imaging (MRI)* wird scheibenweise menschliches Gewebe gescannt und durch die verschiedenen Dichtewerte des menschlichen Gewebes ein zweidimensionales Bild erzeugt. Durch wiederholtes Schalten durch die einzelnen Schichten kann ein Mediziner aus diesen Scans eine ungefähre Vorstellung für die dritte Dimension des Scans erhalten. Da dies jedoch viel Erfahrung und eine spezielle Ausbildung benötigt, bietet es sich hier an, dieses Verfahren zu verbessern. An dieser Stelle kommt MC in Einsatz und ermöglicht es, diese Scans in Oberflächenmodelle zu konvertieren und ohne größeren Hardwareaufwand darstellbar zu machen.

2.2.1 Algorithmus

Der Algorithmus benötigt als Eingabe nur das zu konvertierende Skalarfeld und einen Dichtewert ρ. Dieser gibt eine Grenze an, ab wann ein Skalar im Feld durch das zu modellierenden Dreiecksnetz eingeschlossen ist.

Das Skalarfeld wird als Gitternetz betrachtet in dem jede Zelle einen Würfel bildet und jeder Eckpunkt des Würfels ein Skalar beinhaltet. Nun wird durch jede dieser Zellen iteriert und die jeweiligen Eckpunke betrachtet. Ziel ist es, für jede Position des Würfels eine Konfiguration von Dreiecksnetzen zu finden, welches bestmöglich die Eckpunkte über und unterhalb des Dichtewerts voneinander trennt. Demnach wird für jeden Eckpunkt bestimmt, ob dieser über oder unter dem Dichtewert ρ liegt. Die einzelnen Eckpunkte bilden dadurch jeweils acht boolesche Werte, welche zusammen einen Index bilden. Dieser Index kann für eine Look-Up-Table (LUT) genutzt werden. Diese LUT beinhaltet 256 verschiedene Dreiecksnetze, die den Würfel von Innen- und Außenseite trennen (siehe Abbildung 2.4). Diese lassen sich auf 14 einzigartige Variationen reduzieren, da Rotationen und Spiegelungen ebenfalls mit enthalten sind.

Wird per Index das passende Dreiecksnetz gefunden, so wird dieses an der Stelle des Würfels eingefügt. Somit wird Schritt für Schritt die Oberfläche des Modells anhand des Skalarfeldes konstruiert und bildet damit den namensgebenden Prozess eines marschierenden Würfels der Stück für Stück das Oberflächenmodell generiert.

Standardmäßig befinden sich die Eckpunkte des Dreiecksnetzes in der LUT mittig zwischen den Eckpunkten des Würfels, was das resultierende Modell sehr gleichmäßig wirken lässt. An dieser Stelle kann lineare Interpolation genutzt werden, da die Skalare an den

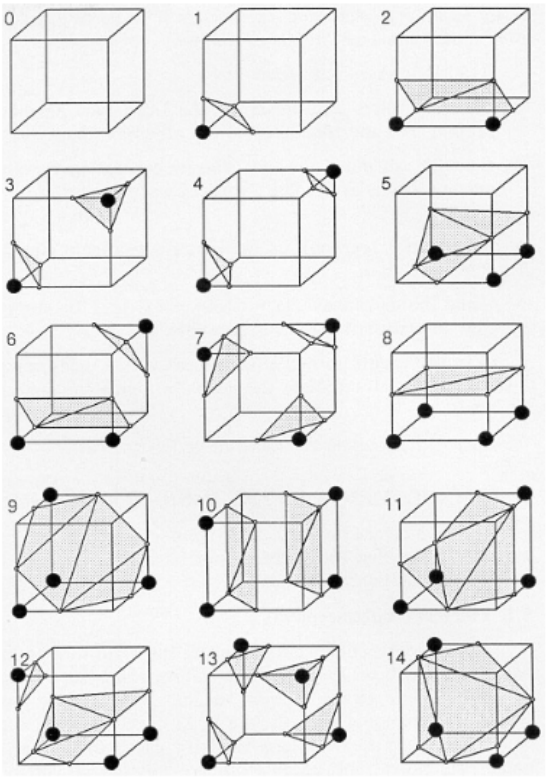


Abbildung 2.4: 14 einzigartige Permutationen der 256 möglichen Fälle und dessen resultierendes Dreiecksnetz in Marching Cubes. Quelle: [24].

Eckpunkten bekannt sind und zusammen mit dem Dichtewert ein Gewicht entlang der Kanten des Würfels bestimmt werden kann, womit die Dreiecke richtig positioniert werden.

2.2.2 Anwendungszwecke

Neben dem ursprünglichen Anwendungszweck in der Medizin findet MC auch weitere Anwendungen in weiteren Bereichen.

Marching Squares

Auch im Zweidimensionalen findet MC Anwendung, allerdings unter dem Namen Marching Squares (MS). Hier wird anstelle eines Würfels ein Quadrat genutzt, welches ein

zweidimensionales Feld abfährt und anstatt von Dreiecksnetzen einzelne Kanten für ein Polygon zusammenstellt.

Anwendung findet MS unter anderem in der Erstellung von topografischen Karten. Die Konturen von Bergen werden schichtweise mit den jeweiligen Höhenwerten eingezeichnet, wie in Abbildung 2.5 dargestellt. MS verwendet hier die Höhendaten als Skalarfeld und kann mit verschiedenen Dichtewerten die jeweilige Kontur eines Höhenniveaus ermitteln.

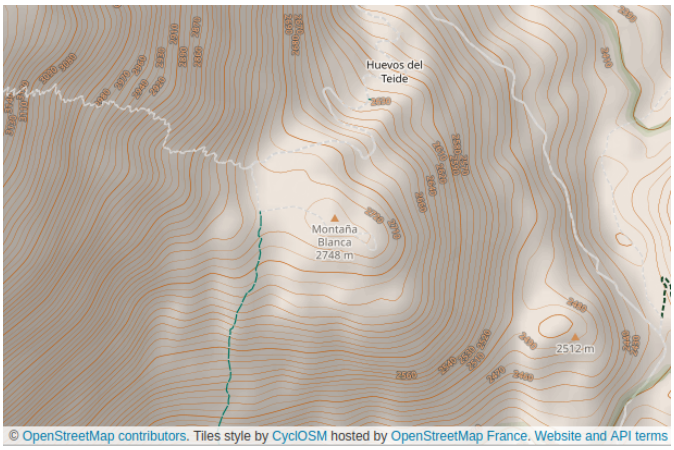


Abbildung 2.5: Landkarte mit eingezeichneten Höhenlinien, welche die Höhe des Berges in einzelnen Schichten darstellt. Quelle: OpenStreetMap.

Konturen lassen sich nicht nur für Berge erstellen, sondern auch für Bilddaten. In der Bildbearbeitung müssen häufig bestimmte Objekte in einem Bild verändert und dadurch umständlich selektiert werden. Auch hier kann MS genutzt werden, sodass eine Maske für ein gewünschtes Objekt erstellt wird, welche genau dieses einschließt. Hier dienen die Pixeldaten als Skalarfeld, wodurch MS die Konturen des Objekts identifizieren kann und damit eine Maske erstellt. Eine Einschränkung ist dabei, dass das Objekt sich genug vom Hintergrund abhebt.

3D-Datenvisualisierung

In vielen industriellen Applikationen wird ebenfalls MC eingesetzt. Beispielsweise wird MC der Finite-Elemente-Methode eingesetzt, wo transparente Kräfteeinwirkungen wie Magnetfelder, visualisiert werden können, wie die Arbeit [25] zeigt.

Ähnlich wie in [25] wird auch im Paper [17] MC genutzt, um nicht sichtbare Kräfte zu visualisieren. Konkret werden hier Moleküle und dessen Elektronendichte betrachtet, wie es Abbildung 2.6 zeigt. Hier wird zusätzlich zum Molekül als Stäbchenmodell die Elektronendichte mithilfe des MC-Algorithmus visualisiert und die Darstellung ergänzt, um ein besseres Bild der Kräfte darstellen zu können.

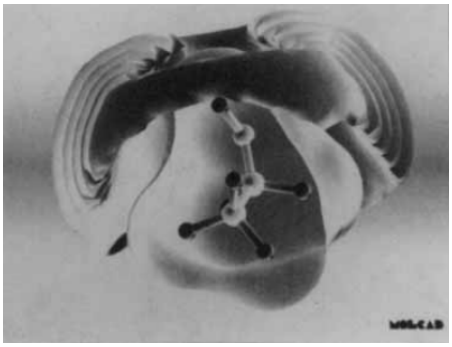


Abbildung 2.6: Darstellung eines Ethanolmoleküls als Stäbchenmodell mit Elektronendichte visualisiert durch Marching Cubes. Quelle: [17]

Auch in der Analyse von der Umwelt lässt sich MC einsetzen. In [33] wird MC für die Visualisierung von Wasserqualität innerhalb einer Bucht genutzt.

2.2.3 Stand der Technik

Durch seine simple Herangehensweise ist MC bereits in seiner einfachen Form sehr populär und lässt sich schnell anwenden. Dennoch hat der Algorithmus einige Probleme und Optimierungspotential, die von einigen Arbeiten untersucht wurden und hier exemplarisch vorgestellt werden.

Mehrdeutige Konfigurationen

Kurz nach der Veröffentlichung von Lorensen und Cline hat Dürst in [6] erste Probleme von MC aufgezeigt. Betrachtet man einige Konfigurationen der Eckpunkte einer Zelle, so können diese auf verschiedene Weisen interpretiert und die Flächen unterschiedlich platziert werden. Dies tritt vor allem bei zu niedrig aufgelösten Datensätzen auf und kann zu nicht wasserdichten Modellen führen. Mithilfe einer Berechnung mit der bilinearen Variation der Flächen wird in [1] und [10] gezeigt, wie sich die Mehrdeutigkeit vermeiden lässt und topologisch korrekte Modelle erzeugt werden können.

Viele Iteration über leere Zellen

Je nach Eingabedaten und Dichtewert kann es vorkommen, dass viele der Zellen kein Dreiecksnetz erhalten, da alle Eckpunkte über oder unter dem Dichtewert liegen. Da diese Zellen keine Dreiecke beinhalten würden, bietet es sich, diese Zellen zu überspringen. Hierfür können verschiedene Ansätze genutzt werden wie hierarchische [34], intervallbasierte [2] oder propagationsbasierte Ansätze [32].

2.3 Distanzfunktionen

Distanzfunktionen nehmen als Eingabe einen Punkt im Raum und liefern den kürzesten Abstand zu einer Oberfläche zurück. Je nach Vorzeichen des Abstands befindet sich der Punkt inner- oder außerhalb des Volumens beschrieben durch die Oberfläche. Somit ist es möglich durch eine Distanzfunktion Formen, Körper und Objekte implizit zu beschreiben [15].

Das einfachste Beispiel bildet hier die Distanzfunktion einer Kugel. Da die Oberfläche einer Kugel überall den gleichen Abstand zur Mitte m hat, beschrieben durch den Radius r, lässt sich die Distanzfunktion f zu einem beliebigen Punkt p mit

$$f_{circle}(p) = \|m - p\| - r$$

definieren. Mit komplexeren Distanzfunktionen lassen sich auch weitere Körper wie Würfel, Tori und auch einzelne Dreiecke im Rahmen von Dreiecksnetzen abbilden.

Die implizite Definition von Körpern durch Distanzfunktionen hat den Vorteil, dass runde Flächen perfekt beschrieben werden können. Explizite Beschreibungen wie Dreiecksnetze dagegen können dies nur durch äußert hohe Auflösungen näherungsweise erreichen.

2.3.1 Stand der Technik

Distanzfunktionen lassen sich an vielen Stellen einsetzen und bilden insbesondere beim Rendering eine wichtige Rolle.

Ray-Marching

Ein großer Anwendungszweck für Distanzfunktionen ist bei Ray-Marching. Hierbei handelt es sich um eine Unterart von Ray-Tracing, wo ebenfalls eine Szene mithilfe von Strahlen abgetastet wird. Anders als beim herkömmlichen Ray-Tracing wird nicht für jeden Strahl der Schnittpunkt mit Objekten in der Szene berechnet, sondern die Distanzfunktionen bestimmt [15].

Vom Ursprung des Strahls wird der kürzeste Distanzwert von allen Objekten in der Szene berechnet und damit die Distanz festgelegt, an dem am Strahl "marschiert" werden kann, ohne ein weiteres Objekt zu treffen. Liegt der Distanzwert nahe bei null, so wurde ein Objekt getroffen. Wurde stattdessen eine große Distanz über mehrere Iterationen zurückgelegt und ist der Distanzwert kontinuierlich gestiegen, so trifft der Strahl kein Objekt. Dieses Konzept lässt sich im zweidimensionalen Raum in Abbildung 2.7 betrachten und zeigt die berechneten Distanzwerte mithilfe von Kreisen visualisiert.

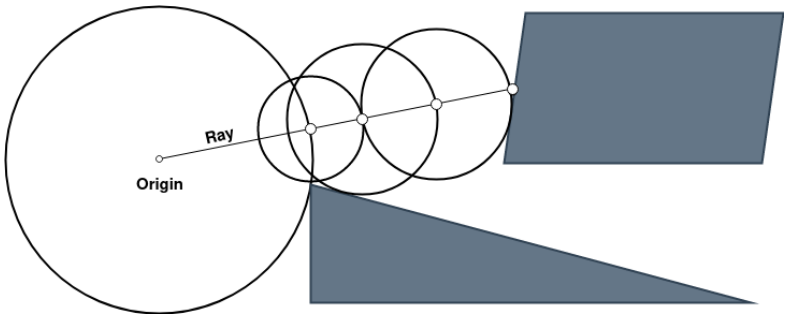


Abbildung 2.7: Visualisierung vom Ray-Marching Algorithmus in 2D. Für jeden Schritt wird mithilfe des kürzesten Distanzwerts der nächste Punkt auf dem Strahl bestimmt.

Mithilfe von Ray-Marching lassen sich komplexe Szenen wie Fraktale umsetzen, welche mit herkömmlichen Ray-Tracing-Methoden nicht darstellbar wären [14].

Constructive Solid Geometry

Interaktives Bearbeiten von Dreiecksnetzen kann sich als schwierig herausstellen, insbesondere wenn ungleichmäßige Flächen verändert oder mehrere Dreiecksnetze kombiniert werden sollen. An dieser Stelle kann Constructive Solid Geometry (CSG) eingesetzt werden. Mithilfe von Distanzfunktionen lässt sich CSG einfach umsetzen, indem Minimum

und Maximum Funktionen auf die Distanzwerte von zwei Modellen angewandt werden [16].

Konvertieren von Modellen zu Distanzfunktionen

Zwar ist es möglich mithilfe von CSG und Distanzfunktionen komplexe Modelle zu erstellen, allerdings ist dieser Prozess sehr umständlich. Um dies zu vereinfachen, gibt es mehrere Ansätze mit neuronalen Netzen, um vorhandene Modelle in Distanzfunktionen zu konvertieren.

In [3] präsentieren die Autoren Dai et al. ein Netz, welches bereits fehlerhafte Voxelfelder, definiert durch Distanzfunktionen, von Modellen ergänzen und reparieren kann, sodass ein vollständiges Modell entsteht. Weiterhin wurden in [11] und [23] Netze entwickelt, welche eingegebene Punktwolken in ein Modell, angenähert mit Distanzfunktionen, konvertieren können.

3 Analyse

In diesem Kapitel soll die Ausgangslage genauer betrachtet und erläutert werden. Zusätzlich soll bestimmt werden, welche externen Softwarekomponenten und Formate benötigt werden, um das Projekt umzusetzen.

3.1 Problemstellung

Durch das zuvor in [4] und [5] erarbeitete generische WFC-Framework konnten verschiedenste Datentypen in diversen Dimensionen implementiert und getestet werden. Dabei wurden die Eigenschaften und Nachteile des WFC-Algorithmus festgehalten. Um nun eine Variante mit Dreiecksnetzen zu konzipieren müssen diese Eigenschaften und insbesondere Nachteile genauer betrachtet werden, damit mögliche Lösungen entwickelt werden können.

3.1.1 Hoher Detailgrad in Modellen

Wie Merrell bereits festgehalten hat, funktioniert der WFC-Algorithmus am besten, wenn niedrig aufgelöste Texturen als Eingabe genutzt werden [27]. Kernpunkt des WFC-Algorithmus ist das wiederholte Auftauchen von Mustern in der Eingabe, da somit die erlaubten Nachbarn eines Musters festgestellt werden können. Taucht dieses Muster mehrfach auf, so können mehr mögliche Nachbarn hinzugefügt werden. Hat die Eingabe eine sehr hohe Auflösung, etwa wie bei einer Textur erfasst durch ein fotogrammetrisches Verfahren, so werden zwangsläufig sehr viele Muster gefunden. Allerdings sind diese in den meisten Fällen einzigartig und es werden keine Duplikate gefunden. Dies führt dazu, dass jedes Tile nur seine eigenen Nachbarn als Relation hat und der WFC-Algorithmus somit die Eingabe nachbaut.

Verallgemeinert man das Problem auf weitere Datentypen als Texturen, so lässt sich feststellen, dass Eingaben mit vielen einzigartigen Details schnell dafür sorgen, dass zu wenig gleiche Muster gefunden werden und somit der WFC-Algorithmus keine guten prozeduralen Ergebnisse erzielt.

3.1.2 Kontinuierlicher Raum

Ein weiteres Problem bildet die Komplexität des dreidimensionalen Raumes. Wird mit Pixeln oder Voxeln gearbeitet, so gibt es eine limitierte Anzahl von Permutationen bei den Mustern, da innerhalb eines Musters nur begrenzt viele Anordnungen möglich sind. Wird allerdings ein kontinuierlicher Raum mit z. B. Vertices und Vektoren betrachtet, so treten weitere Probleme auf.

Durch die kontinuierliche Natur gibt es nun unendlich viele Permutationen, die ein Muster haben kann (abhängig von der Auflösung des Datentyps für Positionen), da nun beliebig viele Vertices im Muster existieren können. Dies führt dazu, dass Muster zwar identisch aussehen können, aber durch minimalste Unterschiede innerhalb des Dreiecksnetzes dennoch verschieden gehandhabt werden können.

Dazu kommen noch weitere Probleme wie Float Ungenauigkeiten, welche ebenfalls dafür sorgen können, dass zwei identisch aussehende Muster als verschieden angesehen werden.

Insgesamt führt dies dazu, dass vermehrt einzigartige Muster gefunden werden, welche sich zwar visuell stark ähneln, allerdings vom WFC-Algorithmus als verschiedene Tiles gehandhabt werden.

3.1.3 Aufteilung in Muster

Das Absuchen der Eingabe mithilfe eines gleitenden Würfels gestaltet sich in der dritten Dimension ähnlich wie im zweidimensionalen Raum, solange Voxel eingesetzt werden. Hier wird anstatt eines gleitenden Fenster ein Würfel benutzt, welcher zusätzlich die neue Dimension abfährt und anstatt der Pixeldaten nun Voxel speichert. Dies wurde im Rahmen einer Erweiterung für das zuvor entwickelte generische WFC-Framework umgesetzt [5]. Im Zuge dessen wurde testweise ein Konzept aufgestellt, welches dieses Konzept mit Dreiecksnetzen kombinieren sollte.

Dieses Konzept wurde allerdings vorerst verworfen, da ohne weitere Änderungen das Einlesen sich als schwierig gestaltet. Anstatt von Voxeln werden bei Dreiecksnetzen mithilfe des gleitenden Würfels die Positionen und Relationen der beinhalteten Vertices gespeichert. Dies funktioniert allerdings nur solange sich die Dreiecke vollständig innerhalb des Würfels befinden. Sobald ein Dreieck den Würfel überschneidet, müssen zusätzliche Schritte eingeleitet werden. Um das Muster erfolgreich zu extrahieren, muss die beinhaltende Fläche des Dreiecks richtig abgeschnitten und neu trianguliert werden. Abbildung 3.1 zeigt exemplarisch einige dieser Randfälle, die im zweidimensionalen Raum auftreten können.

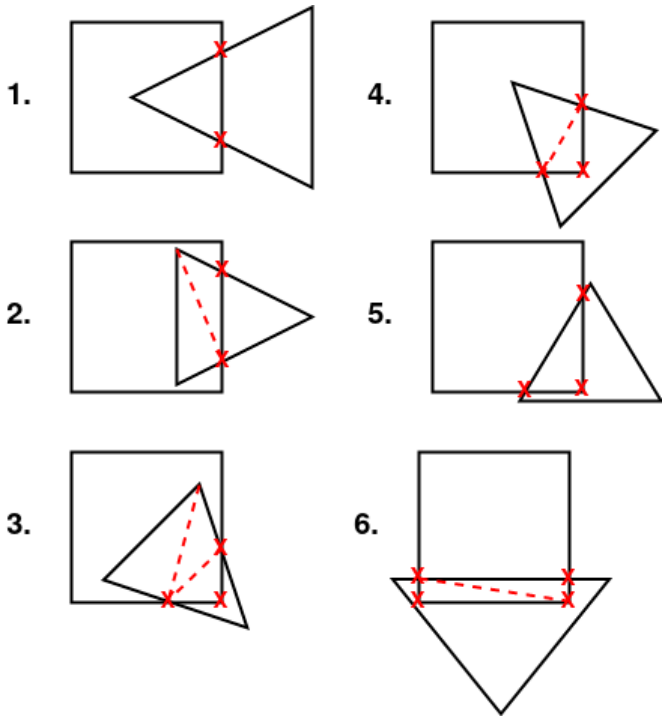


Abbildung 3.1: Triangulation von abgeschnittenen Dreiecken durch gleitendes Fenster in 2D. Rote Kreuze zeigen neue Vertices, gestrichelte Linien neue Kanten.

Durch den hohen Implementierungsaufwand wurde dieser Ansatz nicht weiter verfolgt. Weiterhin kristallisieren sich durch das Anpassen der Muster beim Abtasten weitere Probleme mit dem OverlappingModel.

3.1.4 OverlappingModel im dreidimensionalen Raum

Eine der Neuerungen gegenüber von Merrells Model synthesis war das Einführen des Überlappens von Mustern innerhalb des OverlappingModels. Durch das Abtasten mithilfe des gleitenden Fensters kann garantiert werden, dass benachbarte Muster einander überlappen können und weiterhin Teile der Eingabe darstellen. Dies ist gerade bei der Textursynthese sehr hilfreich, da dadurch gleichmäßige Übergänge zwischen einzelnen Mustern sichergestellt werden können.

An dieser Stelle bildet sich die Frage, inwiefern das OverlappingModel bei anderen Datentypen sinnvoll ist. Gerade beim dreidimensionalen Raum entstehen durch die neue Dimension zusätzliche Komplexität innerhalb der Muster, sodass diese seltener mehrfach auftreten.

Weiterhin bildet auch hier das Verhalten des kontinuierlichen Raumes aus Abschnitt 3.1.2 ein Problem, da durch Anpassung der Tiles, wie in Abschnitt 3.1.3, oder durch Ungenauigkeiten von Datenpunkten, das saubere Überlappen von Mustern nicht mehr möglich ist und diese leicht untereinander variieren. Wird ein Muster neu trianguliert, da dieses abgeschnitten werden musste, dann unterscheidet sich dieses möglicherweise von einem anderen Muster, welches zwar die identische Fläche darstellt, aber anders trianguliert wurde.

3.2 Anforderungen

Mit der formulierten Zielsetzung können konkrete Anforderungen an die Software gestellt werden. Für die zu benutzenden Methoden werden Bibliotheken und Architekturentscheidungen benötigt, welche im Folgendem genauer erläutert werden.

3.2.1 2D-Variante Prototyp

Da durch das generische WFC-Framework beliebige Dimensionen genutzt werden können, soll ein Prototyp im zweidimensionalen Raum erstellt werden. Das Vorgehen ist analog zur dreidimensionalen Variante. Anstatt von Dreiecksnetzen sollen Polygone betrachtet werden, welche von einem Quadrat abgetastet und Muster extrahiert werden

und anschließlich durch den WFC-Algorithmus neu angeordnet werden, sodass ein neues Polygon entsteht.

Anhand dessen lässt sich die Funktionalität verifizieren und durch die geringere Komplexität einfacher testen, sowie die Ergebnisse leichter visualisieren.

3.2.2 Benutzeroberfläche

Bisher ließ sich im Rahmen des WFC-Frameworks der Code nur als Bibliothek nutzen. Diese hat dabei möglichst wenig Informationen nach außen präsentiert, um die Nutzung möglichst einfach zu halten und die Schnittstelle verständlich zu gestalten. Da das zu implementierende Verfahren wesentlich mehr Schritte beinhaltet, soll eine Benutzeroberfläche erstellt werden, welche den Nutzer den Generierungsprozess interaktiv mitverfolgen und gestalten lässt. Dort sollen Parameter eingestellt und dessen Auswirkungen direkt in einer Vorschau betrachtet werden können.

Weiterhin wäre eine solche Benutzeroberfläche auch für die Entwicklung äußerst hilfreich, da die visualisierten Daten einfacher verifiziert werden können.

3.2.3 Dateiformat für Dreiecksnetze

Da im Projekt Dreiecksnetze eingelesen und nach erfolgreicher Generierung wieder gespeichert werden können müssen, braucht es ein geeignetes Dateiformat. Da innerhalb des Projektes im Computergrafik-Framework gearbeitet wird, soll das .obj Dateiformat eingesetzt werden. Hierfür wurden bereits Reader und Writer Klassen implementiert, sodass diese direkt benutzt werden können. Weiterhin wurde bereits in vorherigen Projekten mit .obj Dateien gearbeitet, sodass dieses Format sich am besten für das Projekt eignet.

3.2.4 Darstellung von Dreiecksnetzen

Innerhalb der Benutzeroberfläche soll das eingelesene Dreiecksnetz interaktiv betrachtet werden können. Hierfür wird ein Framework benötigt, welches diese verwalten und darstellen kann. Zusätzlich muss Interaktion wie eine bewegliche Kamera und das dynamische Hinzufügen und Entfernen von weiteren Modellen möglich sein. Gesucht wird daher eine geeignete 3D-Engine, welche diese Eigenschaften erfüllt.

Innerhalb des Computergrafik Frameworks wird standardmäßig für solche Aufgaben die jMonkey-Engine eingesetzt. Als Game-Engine bietet jMonkey alle benötigten Funktionen und erlaubt mit wenig Aufwand Szenen darzustellen.

3.2.5 Bibliothek für Benutzeroberfläche

Neben jMonkey als Darstellung werden auch weitere Kontrollelemente benötigt, mit welchen sich die Applikation steuern lässt und mögliche Anpassungen an Parametern durchgeführt werden können, um die resultierende Generation zu beeinflussen. An dieser Stelle ließe sich ebenfalls jMonkey einsetzen, da es als Game-Engine auch viele Möglichkeiten bietet grafische Bedienelemente anzuzeigen.

Beachtet man jedoch, dass eine 2D-Variante implementiert werden soll, so können diese Bedienelemente nicht für die zweidimensionale Variante genutzt werden, ohne die jMonkey-Engine zu starten.

Stattdessen soll Java Swing eingesetzt werden. Die Darstellung von Bedienelementen und Polygonen im 2D ist ohne Probleme durch Komponenten möglich. Zusätzlich kann das jMonkey-Fenster in die Swing GUI eingebettet und ausschließlich zum Anzeigen genutzt werden.

4 Konzept

In diesem Kapitel wird der Lösungsansatz für die Probleme der vorigen Analyse präsentiert und daraus ein Ablauf abgeleitet, welcher die Probleme bestmöglich mitigiert. Es werden die einzelnen Schritte des Ablaufs durchgegangen und erläutert.

4.1 Lösungsansatz

Zusammenfassen lassen sich die Probleme als nicht geeignete Nutzung des WFC-Algorithmus bei detailreichen Eingaben, kombiniert mit der Komplexität des dreidimensionalen Raums. Demnach ist es sinnvoll die Eingabe vor der Generierung möglichst weit zu vereinfachen, sodass erfolgreich Muster erfasst werden können. Dabei gilt es zwei Dinge zu beachten. Zuerst müssen die Details von einzelnen Mustern weitesgehend reduziert werden, sodass diese vergleichbar sind. Als Zweites muss ein Konzept entwickelt werden, welches ermöglicht die Muster miteinander zu vergleichen und Ähnlichkeiten sowie Unterschiede festzustellen.

Um dies zu erreichen sollen Distanzfunktionen und der MC-Algorithmus eingesetzt werden.

4.1.1 Distanzfunktionen

Anstatt innerhalb von Tiles Dreiecksnetze zu nutzen, sollen Distanzwerte eingesetzt werden, welche den Abstand zur Oberfläche darstellen. Somit ist es möglich, die Oberfläche innerhalb eines Musters sehr effizient anzunähern, ohne die Dreiecke des Models weiter verändern zu müssen.

Dies hat den großen Vorteil, dass mögliche komplexe Details auf der abgetasteten Oberfläche nicht mehr beachtet werden und sich dadurch leichter identische Muster identifizieren lassen.

4.1.2 Marching Cubes

Wurde mithilfe des WFC-Algorithmus eine geeignete Konfiguration von Mustern mit Distanzwerten generiert, so muss diese anschließend wieder in ein Dreiecksnetz konvertiert werden. Dies soll mithilfe des MC-Algorithmus umgesetzt werden, welcher die Muster als Skalarfeld einliest und daraus ein Oberflächenmodell erzeugt.

4.1.3 Änderung OverlappingModel

Da das OverlappingModel sich nicht für eine Implementierung mit Dreiecksnetzen eignet, muss eine andere Variante genutzt werden. Jedoch bildet das SimpleTiledModel die einzige Alternative, welches nur Eingaben in Form von fertigen Tilesets mit bereits definierten Regeln erlaubt.

Demnach soll ein Hybrid aus beiden Varianten entwickelt werden. Es wird die Musteridentifikation aus dem OverlappingModel übernommen, allerdings wird auf den überlappenden Teil verzichtet, sodass Muster, wie beim SimpleTiledModel, nebeneinander angeordnet werden.

4.1.4 Ähnlichkeitsmaß

Mithilfe von Distanzfunktionen lassen sich Muster zwar vereinfachen, allerdings bestehen weiterhin Probleme wie Ungenauigkeiten. Um diese zu kompensieren, kann ein Ähnlichkeitsmaß hinzugefügt werden. Da es sich bei Distanzwerten nur um skalare Größen handelt, können diese miteinander verglichen werden. Liegen diese dicht genug beieinander, so liegen die dadurch beschriebenen Oberflächen ebenfalls nah genug aneinander, sodass diese als identisch betrachtet werden können.

4.2 Ablauf

Zerlegen lässt sich der Ablauf des Konzepts in drei Hauptteile und wird in Abbildung 4.1 dargestellt.

Zuerst muss für die Nutzung vom WFC-Algorithmus die Eingabe eingelesen und aufbereitet werden. Dieser Prozess besteht hier aus dem Berechnen der Distanzwerte aller

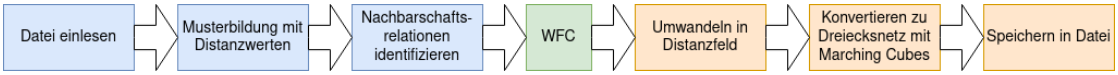


Abbildung 4.1: Ablauf des Generierungsprozesses.

Zellen und der Erstellung von Tiles. Diese werden auf Ähnlichkeiten geprüft und anhand dessen Nachbarschaftsrelationen abgeleitet. Zusammen mit den Mustern können diese Relationen schließlich in den WFC-Algorithmus eingegeben werden. Dessen Ausgabe muss wieder zu dem ursprünglichen Datentyp verarbeitet werden. Dies geschieht durch eine Konvertierung des Feldes von Tiles zu einem Skalarfeld, welches vom MC-Algorithmus wieder in ein Dreiecksnetz konvertiert werden kann, welches abschließend gespeichert wird.

Optional wird eine vierte Komponente als interaktive Benutzeroberfläche hinzugefügt, wo sich die zuvor durchgeführten Schritte visualisieren lassen.

4.3 Konvertieren von Dreiecksnetzen zu Distanzfunktionen

Anders als bei herkömmlichen Distanzfunktionen werden hier nicht einfache Primitive betrachtet. Stattdessen sollen beliebige Dreiecksnetze geladen werden können, weswegen eine vollständig mathematische Beschreibung der Oberfläche nicht ohne weiteres bestimmt werden kann.

Um dies zu Umgehen, soll eine Punkt-Mesh Abstandsberechnung implementiert werden, welche für jeden Punkt den Abstand zum Dreiecksnetz bestimmen kann.

Es wird ein Gitternetz über das Modell gelegt. Für jede Zelle innerhalb des Gitternetzes muss für jeden Eckpunkt ein Abstandswert zu dem Dreiecksnetz bestimmt werden. Dies bedeutet, das für jedes Dreieck iterativ der Abstand bestimmt werden muss und der kürzeste davon ausgewählt wird. Dieses Konzept wird in Abbildung 4.2 im zweidimensionalen Raum für eine Zelle dargestellt. Die vier kürzesten Distanzen der Eckpunkte zum Polygon bilden damit die Abstraktion für die Zelle und können mit dem WFC-Algorithmus eingesetzt werden.

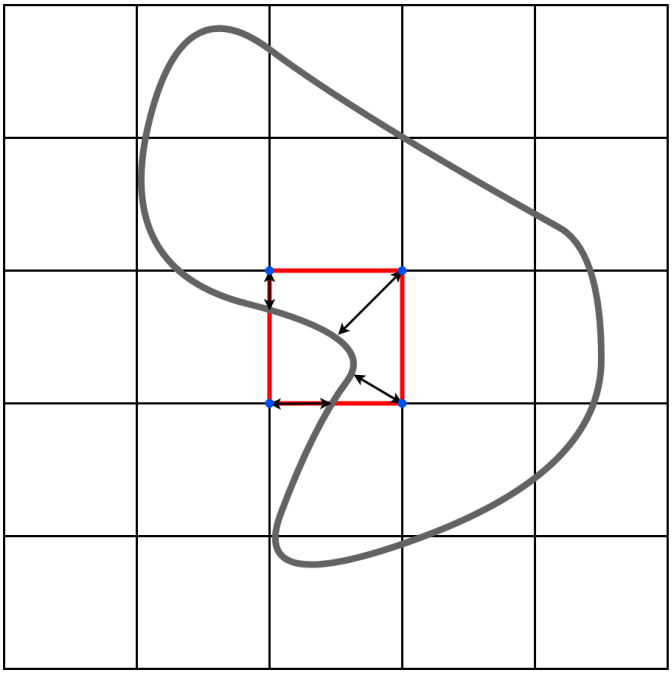


Abbildung 4.2: Polygon mit übergelegtem Gitternetz. In Rot markierte Zelle zeigt die kürzesten Abstände von jedem Eckpunkt zum Polygon.

4.3.1 Abstandsberechnungen

Um den Abstand eines Punktes zu einem Dreieck im Raum zu berechnen, werden mehrere Schritte benötigt. Zwar existieren fertige Lösungen für Distanzfunktionen mit Dreiecken bereits[1], allerdings werden hier die meisten Schritte stark zusammengefasst, weswegen nun im Folgendem die einzelnen Schritte vereinfacht und separat dargestellt werden.

Insgesamt gibt es drei Anordnungen von der Position des Punkts und des Dreiecks, die jeweils in Abbildung 4.3 visualisiert sind. In der ersten befindet sich der Punkt $P3$ am nächsten zu einer der Eckpunkte des Dreiecks. In der zweiten mit Punkt $P2$ am nächsten zu einer der Kanten und zuletzt mit $P1$ am nächsten zu der Fläche des Dreiecks. Je nachdem welche dieser Fälle eintritt müssen diese unterschiedlich berechnet werden. Um den kürzesten Abstand zu bestimmen, muss zuerst geprüft werden, ob der dritte Fall eingetreten ist. Hier zu wird eine Ebene gespannt, welche das Dreieck beinhaltet. Mithilfe der Normalen n der Ebene wird der Punkt $P3$ auf die Ebene projiziert. Durch die baryzentrischen Koordinaten lassen sich nun die Flächeninhaltsanteile Aa, Ab und

[1] https://iquilezles.org/articles/distfunctions/

Ac berechnen, welche summiert zusammen 1 ergeben, falls sich der Punkt innerhalb des Dreiecks befindet. Dadurch lässt sich bestimmen, ob sich der Punkt oberhalb oder unterhalb der Dreiecksfläche befindet.

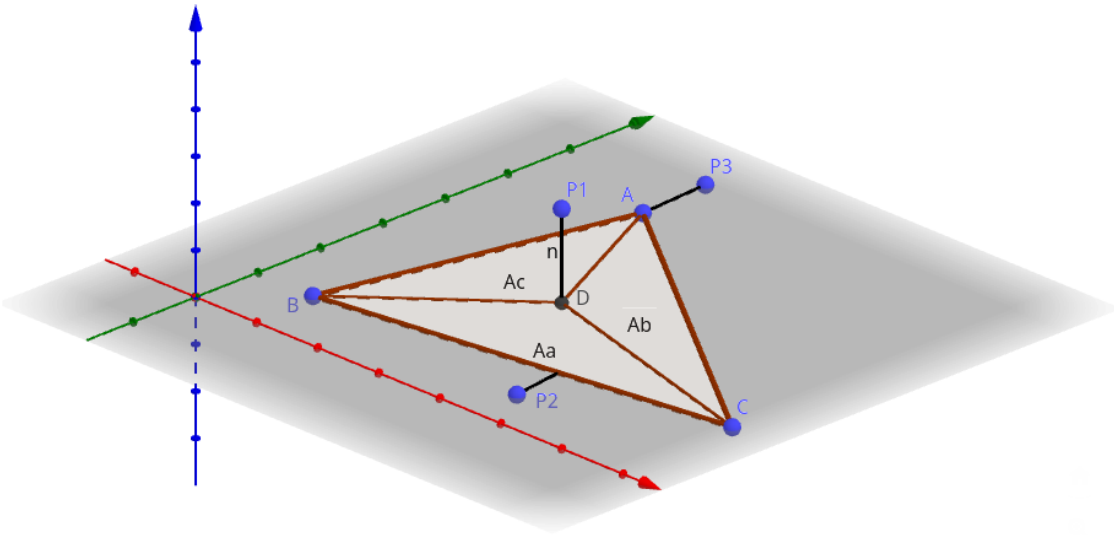


Abbildung 4.3: Dreieck bestehend aus Punkten A, B und C zusammen mit jeweiligen Abständen zu Punkten $P1$, $P2$ und $P3$, welche die jeweiligen möglichen Konfigurationen darstellen.

Ergeben die baryzentrischen Koordinaten zusammen nicht 1, so kann der Punkt zwar nahe an der gespannten Ebene liegen, befindet sich jedoch nicht in der Nähe der Fläche, wodurch der erste oder zweite Fall eintreten muss. Daher wird als Nächstes jeweils der zweite und dritte Fall berechnet und die kürzeste Distanz von diesen ausgewählt.

Fall 1 und 2 lassen sich mithilfe einer Punkt-Segment-Abstandsberechnung bestimmen.

4.3.2 Optimierung

Wird die Abstandsberechnung so umgesetzt, bedeutet dies, dass für jeden Eckpunkt einer Zelle der Abstand zu allen Dreiecken des Dreiecknetzes berechnet werden muss. Gerade bei hochaufgelösten Dreiecksnetzen ist dies ein sehr großer Aufwand und sollte daher optimiert werden.

Möglich ist dies, indem jeweils Dreiecke nur betrachtet werden, wenn in der Nähe der aktuell betrachteten Zelle sind. Daher wird für jede Zelle zuerst bestimmt, welche Dreiecke

die Zelle überschneiden oder vollständig eingeschlossen werden. Für diesen Satz Dreiecke kann dann die Abstandsberechnung aller Eckpunkte der Zelle durchgeführt werden.

Zusätzlich die Berechnung der einzelnen Distanzwerte je Zelle parallelisierbar. Jede Berechnung einer Zelle ist unabhängig voneinander, weswegen dieser Prozess simultan durchgeführt werden kann.

4.3.3 Innen- und Außen Unterscheidung

Für den MC-Algorithmus ist die Unterscheidung zwischen Innen- und Außenseite wichtig, da sonst die Orientierung der Isoflächen nicht identifiziert werden kann. Bei der bisherigen Abstandsberechnung gibt es keine Unterscheidung, da nur einzelne Dreiecke betrachtet wurden und der Distanzwert immer positiv war.

Um die Unterscheidung nun umsetzen zu können, muss das vollständige Dreiecksnetz betrachtet werden. Die Unterscheidung lässt sich mithilfe der *Even-odd-rule* umsetzen. Diese besagt, dass wenn ein Strahl durch das Dreiecksnetz geschossen wird, dieser immer eine gerade Anzahl von Schnittpunkten hat, wenn der Strahl außerhalb startet und eine ungerade Anzahl, wenn er von innen beginnt.

Daher muss für jeden Eckpunkt pro Zelle nun ein Strahl in eine beliebige Richtung geschossen werden und die Schnittpunkte mit dem Dreiecksnetz gezählt werden. Wird eine ungerade Anzahl festgestellt, so muss das Vorzeichen des Distanzwerts negativ werden, sodass der Distanzwert sich von anderen Werten auf der Außenseite unterscheidet.

4.3.4 Leere Zellen

Das bisherige Vorgehen erzielt jedoch das gewünschte Ergebnis nur bei Zellen, die Dreiecke tatsächlich beinhalten. Leere Zellen dagegen erhalten allerdings gar keine Abstandswerte, da durch die bisherigen Optimierungen leere Zellen nun keine Dreiecke beinhalten, zu denen der Abstand berechnet werden darf. Daher wird der theoretisch größtmögliche Abstand, die Diagonale der Zelle, genommen.

Dies führt dazu, dass alle leeren Zellen den maximal möglichen Distanzwert erhalten und damit in weiteren Berechnungen als identisch angesehen werden können.

4.3.5 Erweiterung mit leerem Rand

Weiterhin ist auch relevant, wie das Gitternetz aus Zellen über das Dreiecksnetz gespannt wird. Umschließt dieses das Netz perfekt, sodass die Ecken des Gitters auch gleichzeitig die minimalen und maximalen Punkte des Netzes sind, so können sich bei der Regelfindung Probleme ergeben.

Der ursprüngliche WFC-Algorithmus hat beim Abtasten mit dem gleitenden Fenster Überläufe durchgeführt, sodass beim Überschreiten der Bildgrenze mit dem Fenster dies einen Wrap-around durchführt. Dies ermöglicht eine beliebige Vergrößerung des Eingabebilds und war bei Texturen sehr nützlich, da diese in den meisten Fällen im gesamten Bild eine ähnliche Struktur aufweisen. Dieses Konzept ergibt für 3D allerdings keinen Sinn, da hier nicht Texturen, sondern konkrete Modelle betrachtet werden.

Da das Gitternetz nun zusammen mit dem Rand des Modells abschließt und kein Wrap-around durchgeführt wird, führt dies dazu, dass die Tiles am Rand keine Nachbarn haben. Zwangsläufig resultiert daraus, dass diese Rand-Tiles auch in der Ausgabe am Rand auftauchen müssen, da sonst die Generierung fehlschlägt, weil die Zustandsräume der Nachbarn direkt leer sind. Um dies zu verhindern, kann das Gitternetz erweitert werden, sodass insgesamt zwei Schichten leerer Tiles hinzugefügt werden. Dadurch kann erzwungen werden, dass mindestens zwei leere Tiles nebeneinander liegen und diese die Relation bilden, dass beliebig viele leere Tiles nebeneinander angeordnet werden dürfen.

4.3.6 Tile Substitution

Nachdem alle Tiles erfolgreich erstellt wurden, können diese substituiert werden. Sind zwei Tiles sich ähnlich genug, so können diese als identisch angesehen und durch das gleiche Tile repräsentiert werden. Hierzu wird ein Verfahren benötigt, welches diese vergleichen kann.

Es werden von zwei Tiles jeweils die Distanzwerte jeder Ecke miteinander verglichen. Es wird die Differenz aus zwei Distanzwerten gebildet und mit einem maximalen Grenzwert verglichen. Dieser Grenzwert bildet sich aus einem frei wählbaren Prozentwert multipliziert mit der diagonalen Länge eines Tiles. Solange nun der Ausdruck

$$\text{TileDiagonal} * \text{Threshhold} > |(\text{DistA} - \text{DistB})|$$

für jeden Eckpunkt der Zellen gilt, können diese als identisch angesehen werden.

4.3.7 Regelableitung

Im finalen Schritt können nun die Regeln anhand der Anordnung von Tiles abgeleitet werden. Hier zu wird über jedes Tile iteriert und die aktuellen Nachbarn und dessen Richtungen festgehalten. Durch die Tile Substitution erhalten Tiles, die durch die Substitution mehrfach auftauchen, mehrere mögliche Nachbarn in verschiedenen Richtungen, wodurch der WFC-Algorithmus die Ausgabe abwechslungsreich gestalten kann.

4.4 Wave-Function-Collapse

Da der WFC-Algorithmus keine Informationen aus dem Inhalt einzelner Tiles ableitet, werden diese zu Beginn zufällig angeordnet. Dies kann dazu führen, dass die Ausgabe Modelle zwar generiert, diese aber durch die Begrenzung der Ausgabe abgeschnitten sind.

Bereits zuvor wurde zur Schnittstelle eine Funktion hinzugefügt, welche es ermöglicht, vor dem Generierungsprozess Tiles festzulegen und den Algorithmus umliegende Tiles bei der Generation selbstständig ergänzen zu lassen. Dies war bisher allerdings nur innerhalb des SimpleTiledModels nutzbar, da es zur Laufzeit keinen Weg gab, die abgetasteten Muster zu referenzieren. Diese Funktion lässt sich jedoch hier einsetzen, um zu verhindern, dass Modelle abgeschnitten werden, da leere Tiles am Rand platziert werden können. Die Distanzwerte von leeren Tiles sind zur Laufzeit bekannt und können daher genutzt werden, um vor der Generierung einen Rand von leeren Tiles zu erzeugen, wodurch immer vollständige Modelle erzeugt werden.

4.5 Benutzeroberfläche

Die Benutzeroberfläche soll eine optionale Visualisierung bieten. Die bisherigen Komponenten des WFC-Frameworks ließen sich komplett ohne GUI benutzen, weswegen nun auch die neuen Bestandteile unabhängig nutzbar sein sollen.

Alle bisherigen Module ließen sich zwar ohne Probleme benutzen, allerdings war oftmals nicht klar, inwiefern sich die Parameter und Eingaben auf die Ausgabe auswirken. Um dies zu kompensieren, soll eine interaktive Benutzeroberfläche alle Schritte visualisieren.

4.5.1 Mockup

Abbildung 4.4 zeigt ein mögliches Mockup einer solchen Benutzeroberfläche.

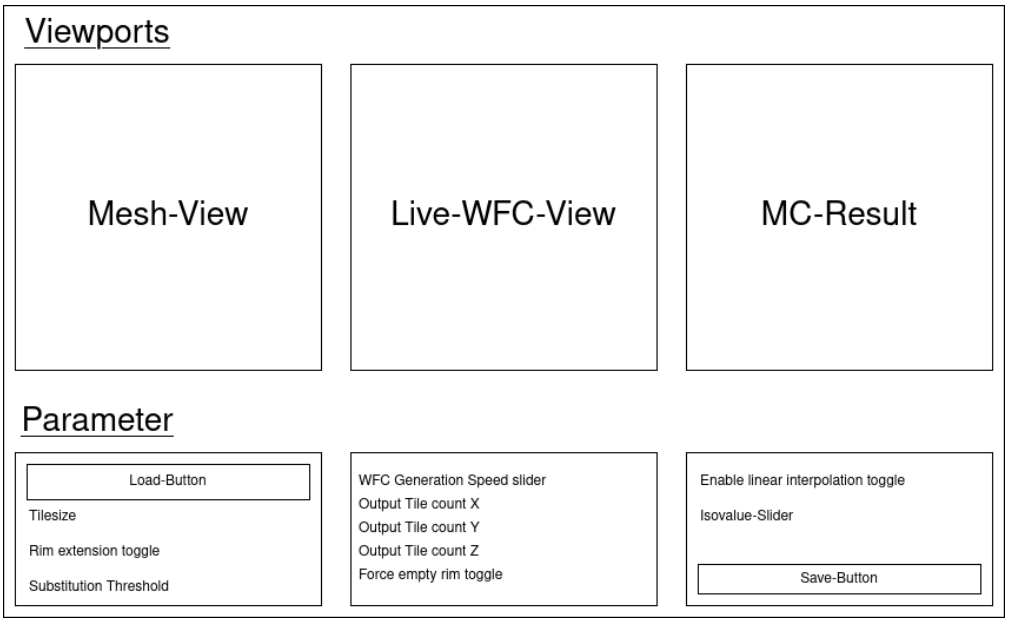


Abbildung 4.4: Ein Mockup für die GUI, die die benötigten Komponenten angeordnet zeigt.

Wie der Generierungsprozess soll auch die GUI aus drei Komponenten bestehen. Diese Komponenten werden mithilfe von drei unabhängigen Viewports dargestellt, in welchen eine jeweils interaktive 3D-Szene zu betrachten ist und dessen Kameraparameter beliebig verändert werden können.

Mesh-View In der Mesh-View wird das aktuell geladene Dreiecksnetz dargestellt. Zusätzlich wird ein Gitternetz mit farbigen Würfeln angezeigt, welches die verschiedenen erkannten Mustern darstellt, wobei gleiche Farben das gleiche Muster sind.

Live-WFC-View Wird die WFC-Generierung gestartet, so lässt sich in der Live-WFC-View beobachten, wie die verschiedenen Muster gitternetzförmig nacheinander angeordnet werden. Dabei werden keine Teile des originalen Dreiecksnetzes genutzt, sondern nur die farbcodierten Muster platziert.

MC-Result Nach der Generierung wird das Ergebnis des WFC-Algorithmus durch den MC-Algorithmus wieder zu einem Dreiecksnetz konvertiert und im MC-Result Viewport dargestellt.

4.5.2 Parameter

Zusätzlich sollte jede Phase des Generierungsprozesses durch Parameter konfigurierbar sein. Das initiale Konvertieren zu Distanzwerten lässt sich unter anderem durch Werte wie der Mustergröße und eines Substitutionsgrenzwerts steuern. Weiterhin werden Knöpfe benötigt, welche ein Ein- und Ausschalten der farbcodierten Musterwürfel ermöglichen, sowie das Hinzufügen eines breiteren Randes mit leeren Mustern.

Die WFC-Generierungsphase soll mithilfe eines Geschwindigkeitsreglers steuerbar sein, sodass der Nutzer mit beliebiger Geschwindigkeit die einzelnen Schritte des Algorithmus nachvollziehen kann. Auch Parameter wie Ausgabegröße in Form von Tileanzahl soll hier festgelegt werden zusammen mit einer Option einen leeren Rand zu erzwingen.

Zuletzt lässt sich das Konvertieren zu Dreiecksnetz mithilfe von Marching Cubes konfigurieren. Konkret lässt sich beim MC-Algorithmus lineare Interpolation nutzen, um akkuratere Isoflächen zu erhalten. Zusammen mit einer Schaltfläche zum Einschalten und einem Schieberegler zum Verstellen des Grenzwerts soll das Ganze nach der Generierung nachträglich veränderbar sein und die Veränderungen in Echtzeit innerhalb des Viewports dargestellt werden.

4.6 2D-Variante

Das Kernkonzept ist für eine Variante im zweidimensionalen Raum identisch und die Algorithmen auf dreidimensionaler Basis lassen sich ohne Probleme für die zweite Dimension anpassen.

Bei der Konvertierung zu Distanzwerten wird nur ein 2D-Gitternetz mit Quadraten betrachtet, welche aus vier statt acht Eckpunkten mit Distanzwerten bestehen. Außerdem werden Polygone betrachtet, weswegen nun der Abstand zu einzelnen Segmenten gemessen wird. Die Unterscheidung zwischen Innen- und Außenraum lässt sich ebenfalls mit der *Even-odd-rule* umsetzen.

Durch die Dimensionsunabhängigkeit und frei wählbaren Datentyps kann das WFC-Framework ohne Änderungen mit der 2D-Variante genutzt werden.

Das Konvertieren des Generierungssergebnis zu einem Polygon wird mithilfe des Marching Squares Algorithmus durchgeführt.

Die Benutzeroberfläche bleibt ebenfalls gleich aufgebaut, da alle Parameter für die 2D-Variante benutzt werden können. Die Viewports bleiben ebenfalls bestehen, nur, dass sie nun ein Canvas benutzen, auf dem das Polygon, der Generierungsprozess und das visualisierte Ergebnis gezeichnet werden kann.

5 Implementierung

In diesem Kapitel wird das zuvor entwickelte WFC-Framework und die im Rahmen des Projekts entwickelte Software genauer betrachtet. Es wird die Architektur und Struktur der Implementierung vorgestellt, sowie die unerwarteten Probleme, die bei der Entwicklung aufgetreten sind.

5.1 Generisches, dimensionsunabhängiges Wave-Function-Collapse-Framework

Das im Grundprojekt entwickelte Framework, bietet die Möglichkeit, neue Datentypen für den WFC-Algorithmus einzubauen [4, 5]. Abbildung 5.1 zeigt den Aufbau des WFC-Frameworks mit den wichtigsten Klassen. Da der WFC-Algorithmus im Kern nur Tiles anordnet, ist der eigentliche Inhalt (Muster) für den Algorithmus nicht relevant. Durch eine geschickte Einteilung in PreProzessor, WFC-Generierung und PostProzessor ist es möglich, die WFC-Generierung dimensions- und datentypunabhängig zu gestalten. Die Prozessoren übernehmen die Rolle des Einlesens und Zerlegens in Muster, sowie das wieder Zusammensetzen zum ursprünglichen Datentyp mit dem Ergebnis der Generierung. Dabei muss jeweils pro Datentyp ein eigener Pre- und PostProzessor gegen eine gegebene Schnittstelle implementiert werden. Es wurden bereits Implementierungen für Texte, das SimpleTiled- und OverlappingModel für Bitmaps und zuletzt erste Schritte im 3D mit jeweils Tilesets mit Dreiecksnetzen und einer Overlapping-Variante mit Voxeln umgesetzt.

5.1.1 PreProzessor

Die Schnittstelle des PreProzessor fordert nur zwei Methoden ein. Die erste Methode benötigt eine Liste von Tiles zusammen mit ihren Relationen. Innerhalb der Relationen

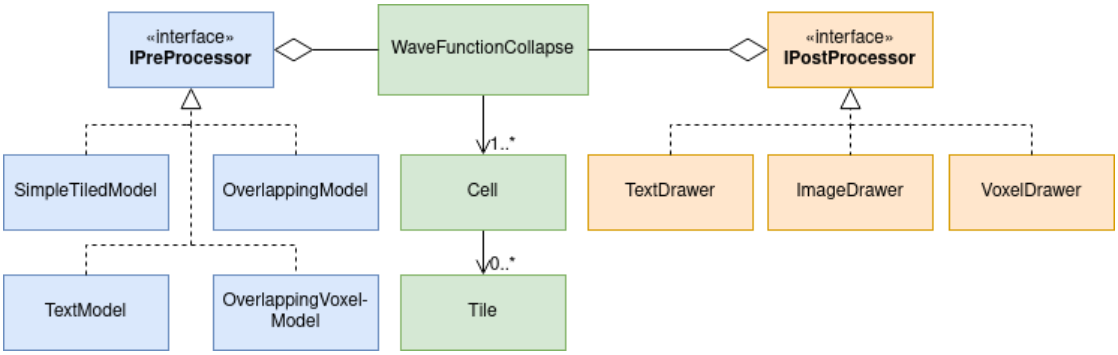


Abbildung 5.1: Wichtigste Klassen des WFC-Frameworks und dessen Relationen.

referenzieren sich die Tiles gegenseitig und geben jeweils eine Richtung an, in der die Nachbarn auftauchen dürfen. Diese Richtungen werden mit `String`-Objekten benannt. Der Kontext dieser Strings wird mit der zweiten Methode definiert, welche die Richtungsnamen zu Richtungsvektoren zuordnet. Anhand dieser beiden Methoden kann der WFC-Algorithmus die Generierung durchführen.

5.1.2 Wave-Function-Collapse Generierung

Im Hintergrund arbeitet das Framework auf einem Graphen aus Zellen, welche sich jeweils über Koordinaten oder Nachbarn referenzieren lassen. Diese Zellen sind immer gitterförmig positioniert und werden je nach Initialisierung ein-, zwei- oder dreidimensional angeordnet. Soll ein Nachbar in einer Richtung referenziert werden, können die Offsets aus dem PreProzessor als Richtungsvektoren genutzt werden, um anhand des Gitternetzes den Nachbarn zu referenzieren. So kann beispielsweise mit (1 0 0) für einen String festgelegt werden, dass sich ein Nachbar rechts von der aktuell betrachteten Zelle befindet. Dieser Mechanismus ist nötig, da je nach Dimension sich die Bedeutungen von Richtungen unterscheiden können (z. B. „oben“ in 2D vs. 3D). Somit ist es möglich nicht nur Gitternetze zu unterstützen, sondern auch andere Netze, wie Hex-Grids, solange sich diese auf normale Gitternetze abbilden lassen. Dies ermöglicht eine dimensionsunabhängige Nutzung des WFC-Frameworks.

Der Generierungsprozess bleibt wie in Abschnitt 2.1.3 bestehen. Es werden Zellen mit der niedrigsten Entropie kollabiert und die Änderungen der Zustandsräume der Zellen an die Nachbarn propagiert, bis keine Änderungen mehr erfolgen.

Die Datentypunabhängigkeit wurde mithilfe von Generics innerhalb der `Tile`-Klasse umgesetzt. Dies ermöglicht den Pre- und PostProzessoren beliebige Datentypen zu behandeln und gleichzeitig den WFC-Algorithmus isoliert zu nutzen.

5.1.3 PostProzessor

Wurde die WFC-Generierung abgeschlossen, muss mithilfe des PostProzessors nur über die Zellen des Graphen iteriert werden, welche am Ende nur noch einen Zustand jeweils besitzen. Diese Zustände können nun genutzt werden, um die Ausgabe wieder zusammenzubauen. Die PostProzessor-Schnittstelle fordert daher nur eine Methode ein, welche die Graphendatenstruktur benötigt und den ursprünglichen Datentyp zurückliefert.

5.2 Architektursicht

Als Nächstes werden die neuen Erweiterungen betrachtet. Durch das bereits gegebene WFC-Framework musste dieses nur in Form von Pre- und Postprozessor Modulen erweitert werden. Abbildung 5.2 zeigt exemplarisch die neuen Klassen des WFC-Frameworks rot umrandet.

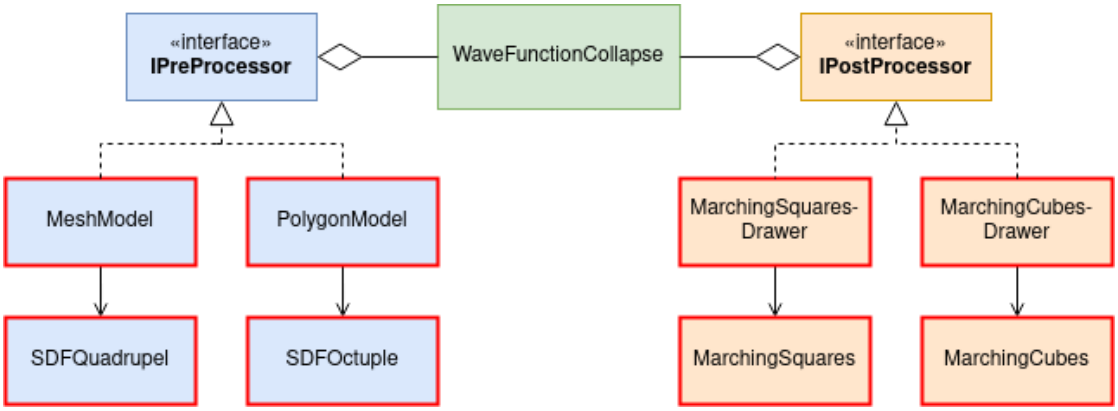


Abbildung 5.2: Exemplarisches Klassendiagramm der neuen Klassen des WFC-Frameworks in Rot.

PreProcessor Im PreProzessor wurden jeweils die `MeshModel`- und `PolygonModel-Klasse` ergänzt. Diese verwalten jeweils das Einlesen von Dreiecksnetzen und Polygonen, die Berechnung von Distanzwerten und dem Erstellen der Tiles. Da für

die Erstellung von Tiles ein Typ benötigt wird, wurden die `SDFQuadrupel`- und `SDFOctuple`-Klassen eingeführt, welche die Distanzwerte speichern.

PostProcessor Ähnlich wie bei der PreProzessor Schnittstelle werden hier auch zwei neue Klassen hinzugefügt. `MarchingSquaresDrawer` und `MarchingCubesDrawer` verwalten jeweils das Wiederzusammenbauen der vollständig kollabierten Welle zum ursprünglichen Datentyp. Anders als bei den ursprünglichen PostProzessor-Klassen werden mehrere Schritte benötigt, wie das Rekonstruieren mithilfe von Marching Squares / Cubes.

Weiterhin lässt sich in Abbildung 5.3 ein Klassendiagramm zu der Benutzeroberfläche finden und wie diese mit den vorhandenen Klassen des WFC-Frameworks zusammenhängt.

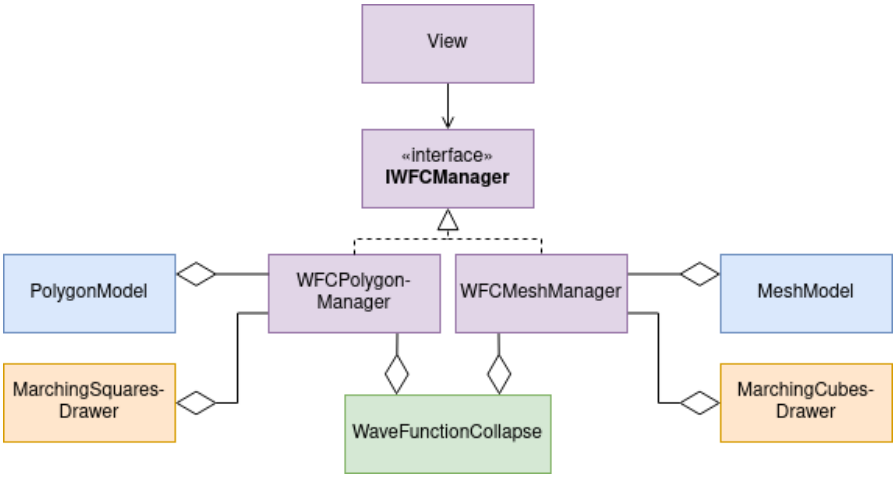


Abbildung 5.3: Klassendiagramm der GUI-Komponenten und dessen Relation zum WFC-Framework.

Die Benutzeroberfläche ist für die 2D- und 3D-Variante dieselbe, und benutzt die IWFC-Manager Schnittstelle. Diese Schnittstelle bietet eine einfache Swing-Panel-Komponente, welche in die Oberfläche eingebunden werden kann. Je nach Implementierung durch WFCPolygonManager oder WFCMeshManager befinden sich innerhalb dieses Panels weitere Swing-Komponenten zur Polygondarstellung oder ein Canvas für die jMonkey-Engine.

Zusätzlich werden alle Events durch Parameterveränderungen in der Oberfläche an die Schnittstelle weitergeleitet, die, je nach Implementierung, die Darstellung korrekt verändern.

5.3 Laufzeitsicht

In den Abbildungen 5.4 und 5.5 finden sich zwei Sequenzdiagramme, welche die Kommunikation zwischen den Klassen darstellen. Hier wird allerdings nur der WFC-Prozess und nicht die Benutzeroberflächekomponenten dargestellt, da diese nur aus Weiterleiten und Parameterveränderungen besteht.

Abbildung 5.4 beschreibt das Initialisieren der WFC-Klasse und wie diese direkt die Muster zusammen mit den Relationen von dem `MeshModel` abrufen.

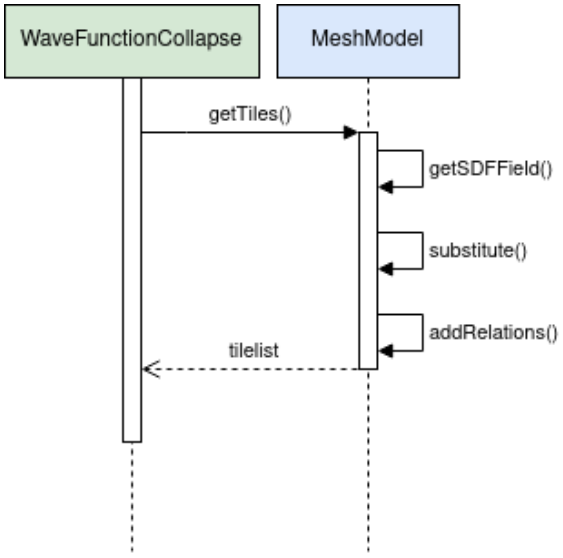


Abbildung 5.4: Sequenzdiagramm des Initialisierens des WFC-Frameworks mit Schritten vom PreProzessor.

In Abbildung 5.5 lässt sich dagegen der Generierungsprozess beobachten. Nach dem Aufruf der `generate()`-Funktion wird die Generierung gestartet und der WFC-Algorithmus legt durch abwechselndes Kollabieren und Propagieren die Zustände der einzelnen Zellen fest. Nach der Beendigung wird das Gitternetz an den `MarchingCubesDrawer` weitergeleitet, welcher das Netz aus Mustern in ein Feld aus Isowerten konvertiert, was wiederum durch MC in ein Dreiecksnetz konvertiert werden kann und zuletzt in einen definierten Pfad gespeichert wird.

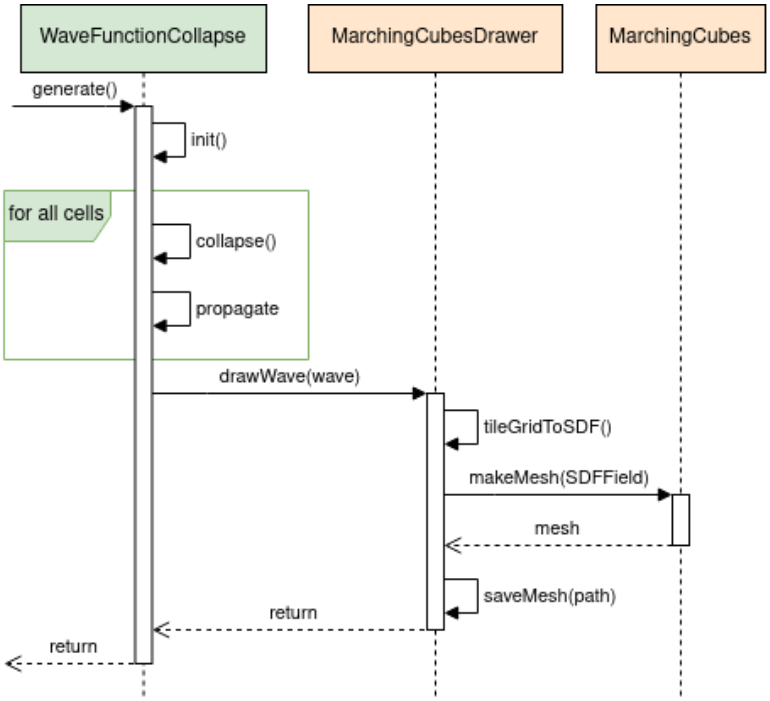


Abbildung 5.5: Sequenzdiagramm vom WFC-Prozess und abschließendes Generieren eines Dreiecksnetzes mit MC im PostProzessor.

5.4 Ergebnisse

Als Nächstes wird die Umsetzung aus Nutzersicht betrachtet und wie mit der Software interagiert werden kann.

5.4.1 2D-Variante

Als Erstes wurde die 2D-Variante implementiert, um zu prüfen, ob dieses Konzept als Grundlage für die Anwendung auf die dritte Dimension dienen kann. Hier wurden nur einfache Polygone als Eingabe genutzt, welche die korrekte Funktionsweise vom Erstellen der Muster und des WFC-Algorithmus verifizieren sollte. Abbildung 5.6 zeigt eines der primär genutzten Beispielpolygone, dessen WFC-Ausgabe und die Rekonstruktion mit Marching Squares.

Das geladene Polygon lässt sich in dem ersten Fenster links betrachten und ist ein einfaches Quadrat, dessen obere Kante um wenige Pixel eingedellt ist, sodass sich die Distanz-

werte ab der Hälfte minimal unterscheiden. Die farbigen Quadrate stellen die identifizierten Tiles dar und sind identisch, wenn sie die gleiche Farbe besitzen. Das mittlere Fenster zeigt die fertige WFC-Generierung, wo die farbcodierten Tiles aus dem ersten Fenster nach den Nachbarschaftsrelationen platziert wurden. Das Ergebnis der Generierung lässt sich mit MS visualisiert im dritten Fenster betrachten

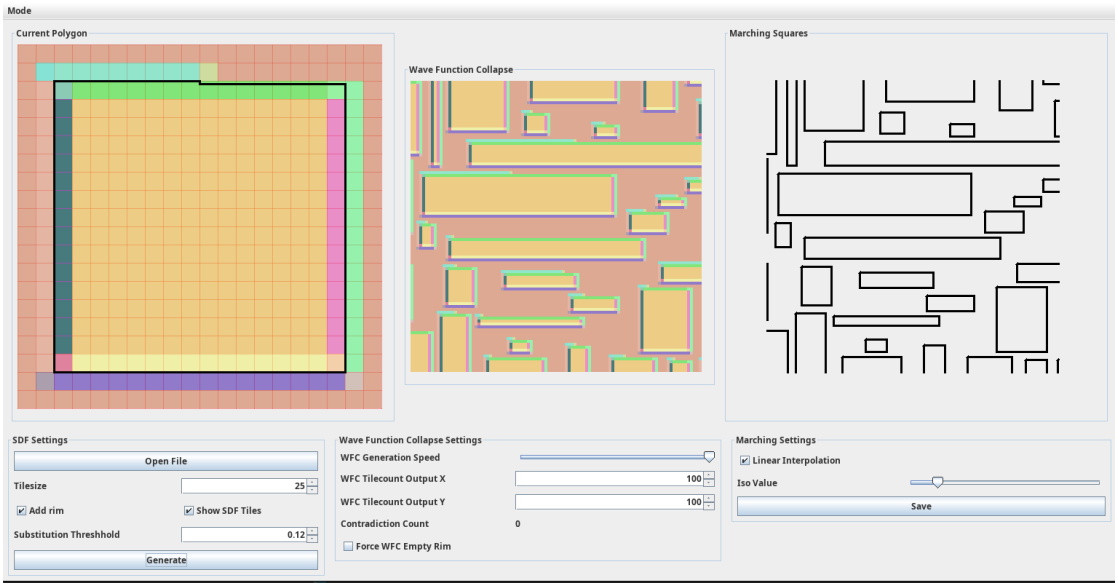


Abbildung 5.6: Fertige Benutzeroberfläche im 2D Modus mit geladenem Polygon und daraus generierter Ausgabe.

Anhand des Würfels in Abbildung 5.6 lässt sich erkennen, dass die Muster erfolgreich erkannt und dessen Relationen bestimmt werden können. So hat die Innenfläche des Würfels immer gelbe Tiles, die Tiles in der Außenfläche einen hautfarbenen Farbton, und die Tiles auf den jeweiligen Kanten eine einzigartige Farbe.

Auch Mechanismen wie das Ähnlichkeitsmaß, getestet durch die Delle in der oberen Kante, konnten erfolgreich eingesetzt werden. Es ist zu sehen, wie trotz der Kante, die gleichen farbcodierten Tiles (grün) innerhalb der linken Anzeige platziert wurden.

Zuletzt konnte die Funktion des WFC-Algorithmus verifiziert werden, indem die Anordnung der farbcodierten Muster mit dem Original verglichen werden. Auch hier zeigt sich, dass alle Muster nur neben denen liegen, die auch in der Eingabe nebeneinander waren, weswegen ohne Bedenken die dreidimensionale Variante implementiert werden konnte.

5.4.2 3D-Variante

In der dreidimensionalen Variante wurden zum Testen der Funktionalität ebenfalls einfache Formen, wie Würfel, benutzt. Abbildung 5.7 zeigt eins der Ergebnisse mit der Benutzeroberfläche im dreidimensionalen Modus mit einem Testwürfel als geladenem Modell und der daraus generierten Ausgabe. Vom Konzept her sind die 2D- und 3D-Variante identisch und können mit denselben Bedienelementen gesteuert werden.

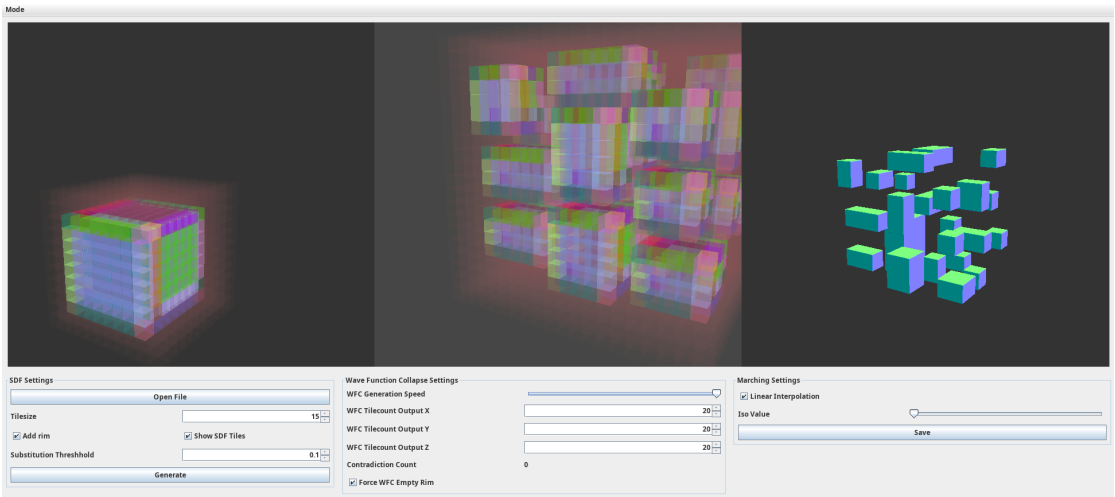


Abbildung 5.7: Fertige Benutzeroberfläche im 3D-Modus mit geladenem Dreiecksnetz und daraus generierter Ausgabe.

Wie in der 2D-Variante lässt sich auch hier anhand der farbcodierten Muster die korrekte Funktionsweise des Algorithmus verifizieren.

Die Darstellung der farbcodierten Muster hat sich in der 3D-Variante schwieriger gestaltet, da durch das Überlappen von transparenten Würfeln zunehmend die Sichtbarkeit einschränkt wird und das originale Modell nicht mehr erkennbar ist.

5.5 Aufgetretene Probleme

Im Rahmen der Entwicklung sind einige Probleme aufgetreten, welche bei der Analyse und Konzeption nicht ersichtlich waren und kleinere Änderungen benötigten. Diese sollen im Folgenden diskutiert werden.

5.5.1 Legacy Probleme von Swing

Um jMonkey innerhalb einer Benutzeroberfläche einzubetten, kann Swing eingesetzt werden. Standardmäßig benutzt jMonkey die *Lightweight Java Game Library* (LWJGL) der Version 3. Diese Unterstützung zur Integration in Swing Oberflächen wurde jedoch eingestellt, weswegen auf die ältere Version 2 ausgewichen werden muss.

Damit bilden sich allerdings neue Probleme, da LWJGL2 keine mehrfache Instanziierung erlaubt, weswegen nicht drei verschiedene Instanzen für die drei Ansichten innerhalb der Benutzeroberfläche genutzt werden können. Stattdessen musste eine gemeinsame Instanz genutzt werden, dessen Ansicht mit drei Kameras verwaltet wurde, welche jeweils nur einen Teil des Szenengraphen betrachten können. Der einzige Nachteil an diesem Ansatz ist, dass die Benutzeroberfläche nicht richtig skaliert, da die Breite der einzelnen Ansichten unabhängig von Swings Komponenten arbeiten und somit die Konfigurationen der drei Hauptkomponenten nicht genau unter den Viewports sind.

5.5.2 jMonkey Asset-Loader

Um zur Laufzeit Dreiecksnetze im .obj Format von einem Speichermedium einlesen zu können, muss eine Komponente diese Einlesen und in entsprechende Objekte konvertieren. jMonkey bietet hierfür einen Asset-Loader an, welcher Ressourcen von vordefinierten Pfaden laden kann.

Um allerdings von beliebigen Pfaden Objekte einlesen zu können muss jedes Mal beim Asset-Loader der neue Pfad als Asset-Path registriert werden, die Ressource geladen und anschließend der Asset-Path wieder entfernt werden.

Zusätzlich bildet sich das Problem, dass der Asset-Loader zusammen mit der jMonkey Engine initialisiert wird. Diese wird jedoch nur mit der Oberfläche zusammen genutzt, sodass im Falle einer Nutzung ohne Benutzeroberfläche keine Modelle mehr geladen werden können.

Daher musste die ObjReader-Klasse aus dem Computergrafik Framework benutzt werden.

5.5.3 Postprozessor-Schnittstelle

Die Postprozessor-Schnittstelle fordert nur eine Methode ein, welche als Eingabe die vollständig kollabierte Welle des WFC-Algorithmus annimmt und die zusammengebaute Ausgabe zurückliefert. Die Ausgabe entspricht dabei dem Datentyp eines einzelnen Musters. Dies war bisher der sinnvollste Weg, da immer Datentypen genutzt wurden, dessen Anteile aus dem gleichen Datentyp bestanden (z. B. Bild-Objekt als Gesamtbild und mehrere Bild-Objekte als Tiles). In der Implementierung verhält es sich jetzt anders, da das Gesamtbild nun ein Dreiecksnetz ist, während die Generierung über ein Informationsobjekt läuft (SDFQuadrupel und SDFOctuple), weswegen kein gültiges Objekt zurückgeliefert werden kann.

5.5.4 Gewichtung von Mustern

Im WFC-Algorithmus lassen sich Muster mit Gewichten versehen, sodass sich die Häufigkeitsverteilung in der Ausgabe von diesem Muster verändert. Je nach Dreiecksnetz kann es allerdings vorkommen, dass durch das Gitternetz unverhältnismäßig viele leere Zellen entstehen und diese damit ein sehr hohes Gewicht erhalten. Dies tritt insbesondere auf, wenn ein leerer Rand um das Model erzwungen wird. Als Folge dessen haben die Ausgaben häufig ebenfalls sehr viel leeren Raum und oftmals wird gar kein Modell generiert. Je nach Eingabe kann es daher sinnvoll sein die Gewichtung von Mustern zu deaktivieren.

5.5.5 Punkt innerhalb eines Dreiecksnetzes Überprüfung

jMonkey bietet unter anderem ein `Collidable`-Interface an, was von den meisten Klassen, die im Szenengraphen eingehängt werden können, unterstützt wird. Mit diesem lassen sich komplexe Kollisionsabfragen mit beliebiger Geometrie durchführen. Die `Ray`- und `Mesh`-Klasse bieten ebenfalls dieses Interface an und lassen sich dazu benutzen, um zu bestimmen, ob ein Strahl ein Dreiecksnetz schneidet. Hierzu wird die Even-odd-rule aus Abschnitt 4.3.3 angewendet.

Initial wurde diese Schnittstelle benutzt, allerdings hat sich herausgestellt, dass bei der Kollisionsbestimmung mit Dreiecksnetzen Probleme auftreten. Schneidet ein Strahl die Kante eines Dreiecks werden zwei Schnittpunkte gezählt, da zwei Dreiecke sich immer

eine Kante teilen. Dies führt dazu, dass zwei Schnittpunkte mit der Oberfläche gezählt werden und die Even-odd-rule nicht mehr aufgeht.

Also wurde eine eigene Schnittpunktimplementierung erstellt, welche diesen Fehler kompensiert. Je nachdem in welche Richtung die Normale des getroffenen Dreiecks zeigt, muss dieser Fall anders behandelt werden. Um dies zu erzielen wird die Art des Schnittpunkts und dessen Position gespeichert, sodass dieser beim erneuten Auftreten ignoriert werden kann.

5.5.6 Unsaubere Dreiecksnetze

Ein weiteres Problem bilden manche Arten von Dreiecksnetzen. Damit ein Dreiecksnetz als wohlgeformt gilt, muss dieses mehrere Eigenschaften erfüllen. Jedes Dreieck muss aus jeweils drei Vertices und drei Kanten bestehen. Jede dieser Kanten muss zwei anliegende Dreiecke besitzen und selber aus zwei Vertices bestehen.

Zwar gilt ein Netz mit diesen Eigenschaften als wohlgeformt, allerdings kann die Topologie des Netzes fehlerhaft sein, sodass diese sich selbst überschneidet oder Löcher besitzt und nicht wasserdicht ist.

Die implementierte Überprüfung, ob ein Punkt sich innerhalb des Netzes befindet, kann bei solchen fehlerhaften Netzen gegebenenfalls ein falsches Ergebnis zurückliefern. Ist dieses zum Beispiel nicht wasserdicht, so trifft der Strahl der Überprüfung eine Fläche zu wenig und erkennt einen Punkt fälschlicherweise auf der falschen Seite des Netzes. Gleichermaßen gilt dies für Flächen innerhalb des Netzes, welche sonst nicht sichtbar sind.

Insgesamt führt dies dazu, dass sich außen befindende Tiles teilweise oder vollständig als innen klassifiziert werden und dadurch sich von den umliegenden Tiles unterscheiden, was wiederum zu falsch abgeleiteten Nachbarschaftsregeln führt.

Lösen lässt sich das Problem bisher nur durch gezieltes Prüfen von importierten Modellen und bei Bedarf einer Anpassung des Modells.

5.5.7 Widerspruchpotenzial

Durch die Umsetzung eines erzwungenen Rands innerhalb der WFC-Ausgabe kann sichergestellt werden, dass Modelle immer wasserdicht und vollständig generiert werden. Die können zwar immer noch beliebig innerhalb des Raums positioniert sein, allerdings wird ein Abschneiden durch Begrenzung der Ausgabegröße verhindert.

Aus diesem Konzept hat sich allerdings ein weiteres Problem kristallisiert. Der WFC-Algorithmus wählt für die nächste zu kollabierende Zelle die der wenigsten Entropie aus. Da zu Beginn die Zellen am Rand kollabiert wurden, beginnt der Algorithmus folglich an einer am Rand anliegenden Zelle. Bleiben während des Generationsprozesses temporär nur noch leere Tiles übrig, kann dies dazu führen, dass der Algorithmus parallel von mehreren Seiten beginnt zu generieren. Dadurch entsteht ein hohes Widerspruchpotenzial, da sich bei zwei annähernden Seiten schnell verschiedene Strukturen bilden und dadurch Konflikte in den Nachbarschaftsrelationen entstehen, sobald die Seiten aufeinander treffen.

6 Evaluation

Nach der erfolgreichen Umsetzung des Prototyps muss dieser als Nächstes evaluiert werden. Primärer Fokus wird auf die dreidimensionale Variante gelegt und in den drei Aspekten von Performance, Modellqualität und zuletzt Nutzbarkeit genauer analysiert. Insgesamt wurden mehrere Modelle getestet und exemplarisch ausgewählt, um mögliche Probleme und Ergebnisse aufzuzeigen.

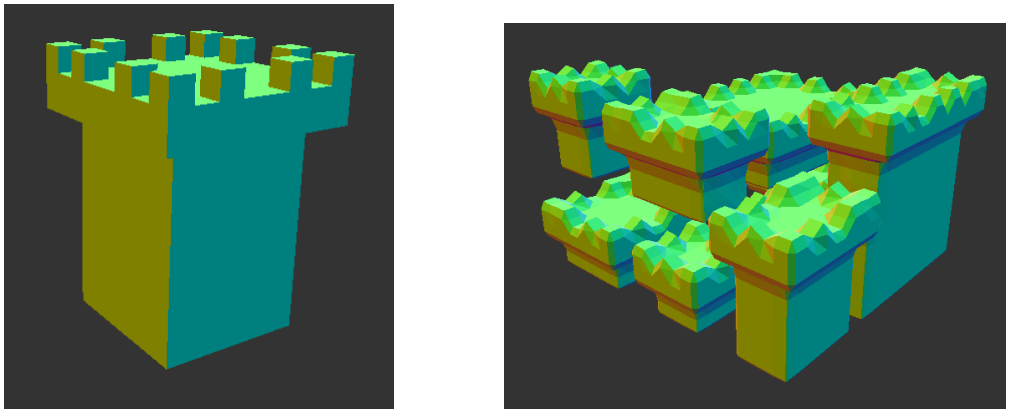
6.1 Qualität der generierten Modelle

Zuerst soll die Qualität der Ausgaben der 3D-Implementierung betrachtet werden. Dabei kann die Qualität jeweils auf zwei Weisen evaluiert werden. Einerseits wird die Rekonstruierung mit MC bewertet, da mit diesem Verfahren Details verloren gehen und hier geprüft werden muss, wie genau Teile des originalen Modells wiederhergestellt werden können. Andererseits muss geprüft werden, wie gut der WFC-Algorithmus die Komponenten neu anordnet und damit geeignet das Modell prozedural generieren kann.

Es werden exemplarisch drei verschiedene Arten von Modellen präsentiert, welche für die Tests genutzt werden.

6.1.1 Turm-Modell

Begonnen wird mit einem simplen Modell. Abbildungen 6.1a und 6.1b zeigen einen einfachen mittelalterlichen Turm ohne Details, abgetastet mit $14 * 17 * 14$ Tiles, einem Substitutionsgrenzwert 5 % und einer Ausgabegröße von $20 * 20 * 20$.



(a) Eingabe. (b) Ausgabe.

Abbildung 6.1: Erstes Modell: Simpler mittelalterlicher Turm abgetastet mit $14 * 17 * 14$ Tiles, Substitutionsgrenzwert 5 %, Ausgabegröße $20 * 20 * 20$.

Rekonstruierungsqualität

Es fällt direkt auf, dass mithilfe der Rekonstruierung durch den MC-Algorithmus viele der harten Kanten verloren gegangen sind und stattdessen abgerundet wurden. Auch sind einige Zinnen des Turms miteinander verschmolzen worden und bilden rundliche Beulen. Dies lässt sich wahrscheinlich auf eine zu große Mustergröße zurückführen, da die Grundseiten des Turms weiterhin gerade Flächen mit 90° Ecken bilden. Die Mustergröße korrekt zu wählen bildet hier einen Balance-Akt, da durch eine zu große Mustergröße Details, wie die Zinnen, verloren gehen, allerdings durch eine zu kleine Mustergröße die Generierung beeinflusst wird und die Zinnen gar nicht oder in zufälligen Abständen platziert werden.

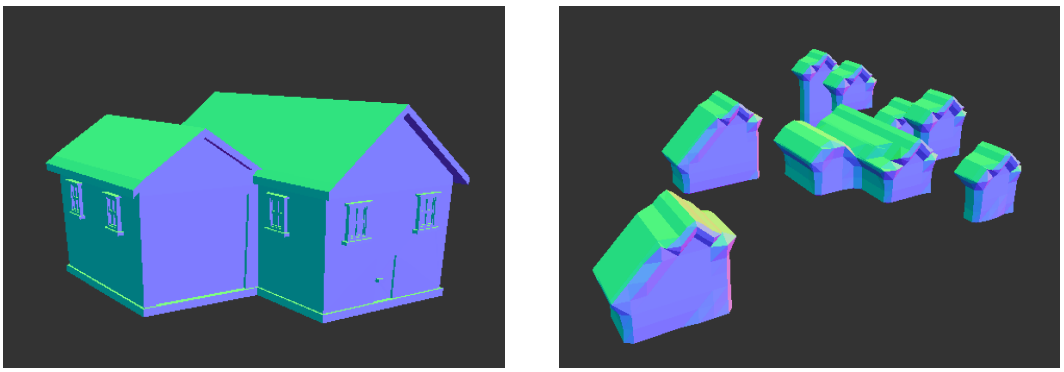
WFC-Qualität

Betrachtet man ausschließlich die WFC-Qualität, so fällt auf, dass der Turm erfolgreich in verschiedenen Größen repliziert werden kann. Dabei variiert nicht nur die Höhe des Turms, sondern auch die Grundflächengröße und dies sogar in verschiedenen Seitenverhältnissen, sodass auch Mauersegmente entstehen können, wie es der Turm ganz rechts in Abbildung 6.1b zeigt. Wie bereits zuvor beschrieben, gilt es auch hier die Mustergröße zu balancieren. In Abbildung 6.1a wurde die Mustergröße so gewählt, dass der Überhang des Turms oben aus nur einem Tile besteht und die Zinnen direkt aufliegen. Würde dieser

mehrere Tiles hoch sein, könnte auch dessen Überhang variabel werden und damit keine ikonische Turmform mehr besitzen.

6.1.2 Haus-Modell

Als etwas komplexeres Modell werden die Abbildungen 6.2a und 6.2b betrachtet. Hier handelt es sich um ein Haus mit Garagenanbau mit detaillierten Fenstern und Türen. Es wurde mit $14 * 11 * 14$ Tiles abgetastet und ein sehr hoher Substitutionsgrenzwert von 40 % gewählt und einer Ausgabegröße von $40 * 16 * 40$.



(a) Eingabe. (b) Ausgabe.

Abbildung 6.2: Zweites Modell: Haus mit komplexen Fenstern und zusätzlichem Garagenanbau, abgetastet mit $14 * 11 * 14$ Tiles, Substitutionsgrenzwert 40 %, Ausgabegröße $40 * 16 * 40$.

Rekonstruierungsqualität

Verglichen mit dem vorherigen Turm-Modell wurde hier ein sehr hoher Substitutionsgrenzwert von 40 % gewählt, welcher die, trotz des detaillierten Models, geringe Tileanzahl kompensieren sollte. Damit wurden die Details wie Fenster und Türen ignoriert und wie die umliegenden Wände betrachtet.

Dies zeigt sich auch in der Ausgabe, wo nur sehr grobe Formen von Grundfläche und dem Dach erkennbar sind, während Türen und Fenster vollständig verschwunden sind. Der Überhang des Daches ist wesentlich dicker als in der Eingabe und hat auch hier starke Rundungen durch den MC-Algorithmus. Anders als beim vorigen Turm-Modell sind die Ecken des Hauses abgerundet worden.

WFC-Qualität

Durch den WFC-Algorithmus zeigt hier ebenfalls eine variable Grundrissgröße, bei der das Haus in Richtung des Dachgiebels beliebig lang werden kann. Die Breite ist jedoch limitiert in der Größe des Daches. Dieses ist jedoch nicht immer symmetrisch, sondern hat ungleich große Dachflächen, wodurch eine Seite kürzer oder länger ist als die andere. Auch der Garagenanbau bildet sich variabel und bleibt manchmal aus oder wird ebenfalls in einer beliebigen Größe hinzugeneriert, wie es das hinterste Haus in Abbildung 6.2b zeigt.

6.1.3 Kuh-Modell

Als Letztes wird das Modell in Abbildung 6.3a genutzt. Verglichen mit den anderen Modellen handelt es sich hier um ein sehr hoch aufgelöstes Modell mit vielen runden Flächen. Da die letzten beiden Beispiele anorganische Modelle waren, soll hier mit organischen Figuren getestet werden. Wegen der Details wurde eine kleine Mustergröße von 6 gewählt, was beim skalierten Modell $21 * 15 * 9$ Tiles entspricht, zusammen mit 30 % Substituierung und einer Ausgabegröße von $20 * 20 * 30$.

(a) Eingabe. (b) Ausgabe.

Abbildung 6.3: Drittes Modell: Modell einer Kuh, abgetastet mit $21 * 15 * 9$ Tiles, Substitutionsgrenzwert 30 %, Ausgabegröße $20 * 20 * 30$.

Rekonstruierungsqualität

Anders als bei den bisherigen Modellen finden sich fast keine gleichmäßigen Flächen oder rechtwinklige aufeinander treffende Flächen. Dennoch lässt sich die Fläche trotz einer vergleichsweise hohen Tilegröße annähern, wodurch sich weiterhin grob ein vierbeiniges Säugetier erkennen lässt. Kleine Details, wie die Augen, Hörner oder Ohren dagegen, gehen verloren.

WFC-Qualität

Initial fällt direkt auf, dass das originale Modell mehrfach in der Ausgabe in Abbildung 6.3b auftaucht. Da durch die vielen, ungleichmäßigen Flächen fast nur einzigartige Tiles gefunden wurden, führt dies dazu, dass der WFC-Algorithmus das originale Modell in den leeren Raum rein generiert, da die Nachbarschaftsregeln keine anderen Kombinationen zulassen. Eine Ausnahme bilden in diesem Beispiel die Vorderbeine, welche sich hier durch unterschiedliche Längen variabel gestalten.

6.2 Performance

Als Nächstes wird die Performance des Generierungsprozesses genauer untersucht. Abhängig ist diese von der Wahl der verschiedenen Parameter und soll im Folgenden analysiert werden. Es werden jeweils die Laufzeiten der drei Hauptkomponenten (PreProzessor, WFC und PostProzessor) einzeln betrachten.

Alle Tests wurden auf der gleichen Computer durchgeführt, dessen Spezifikationen in Tabelle 6.1 aufgelistet sind.

Typ	Komponente
CPU	AMD Ryzen 7 5800x3D
RAM	32 GB CL16 3200 MHz
GPU	NVIDIA RTX 2070
JVM	AdoptOpenJDK 16
OS	Arch Linux Kernel 6.3.3

Tabelle 6.1: Hardware und Software des Computers, auf dem die Messungen durchgeführt wurden.

6.2.1 PreProzessor Distanzfunktionberechnung

Für das Testen der Laufzeit vom PreProzessor wurden drei Modelle geladen, welche jeweils aus einer aufsteigenden Anzahl an Dreiecken bestehen (2624, 5804 und 12946). Da die bisherigen Modelle aus vergleichsweise wenig Dreiecken bestanden, wird für diesen Test das Fandisk-Modell hinzugezogen [19]. Innerhalb des Tests wurden die Schritte des Berechnens von Distanzfunktionen, Tiles erstellen, Tiles substituieren und Nachbarschaftsregeln bilden durchgeführt. Diese Versuche wurden jeweils mit verschiedenen Tilegrößen getestet und insgesamt zehnmal durchgeführt und die durchschnittlichen Laufzeiten ermittelt. Die Ergebnisse sind in Abbildung 6.4 dargestellt.

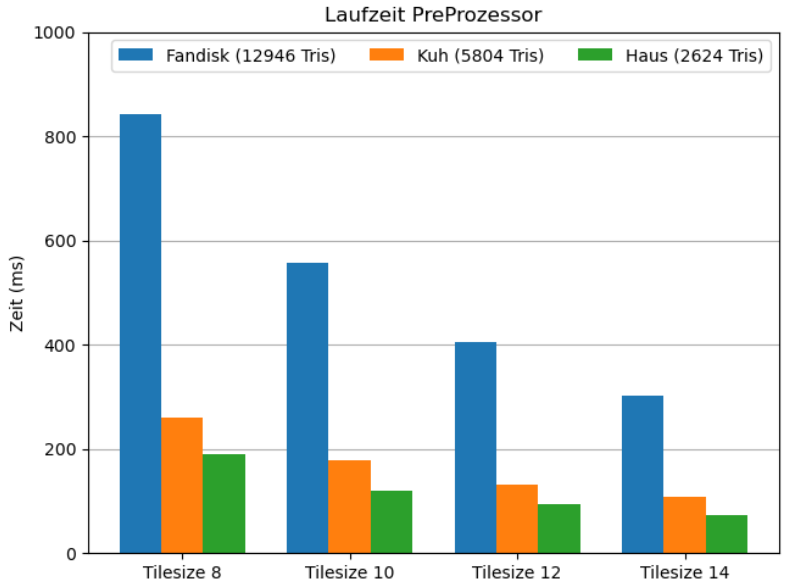


Abbildung 6.4: Laufzeitmessung des PreProzessors mit verschiedenen Dreiecksnetzen und verschiedenen Mustergrößen.

Es lässt sich erkennen, dass die Verarbeitungszeit mit einer höheren Anzahl von Dreiecken und kleineren Tilegrößen steigt. Gerade sehr hochaufgelöste Modelle, wie das Fandisk-Modell, müssen trotz Parallelisierung fast eine Sekunde verarbeitet werden. Weiterhin fällt auf, dass die Anzahl der Dreiecke deutlich maßgebender ist, als die Tilegröße, da die anderen beiden Modelle trotz verschiedener Tilegrößen sich in einem ähnlichen Bereich von 180 ms bewegen.

Zusätzlich wurde mithilfe von Profiling bestimmt, welche Schritte die längste Zeit beanspruchen. Hierbei hat sich gezeigt, dass alle Schritte bis auf die Erstellung der Tile-

Objekte von der beanspruchten Zeit vernachlässigbar sind. Die Berechnung der Tile-Objekte dagegen benötigt die meiste Zeit, mit jeweils 23,6 % für das Bestimmen der relevanten Dreiecke, 74,6 % für die Innen- und Außenunterscheidung und zuletzt nur 1,6 % für die eigentliche Abstandsberechnung. Dies erklärt auch die hohe Laufzeit von Modellen mit vielen Dreiecken, da für die Innen- und Außenraumunterscheidung mehrfach durch alle Dreiecke iterieren werden muss.

6.2.2 Wave-Function-Collapse

Als Nächstes wird der Kern der Applikation mit dem WFC-Algorithmus betrachtet und wie dieser sich in Bezug auf die Laufzeit verhält. Hier haben ebenfalls mehrere Parameter einen Einfluss auf die beanspruchte Zeit, wie etwa die Ausgabegröße, die Option einen leeren Rand zu erzwingen und die Anzahl der Tiles, welche durch den PreProzessor identifiziert wurden. Da durch die Nutzung von zufälligen Zahlen und das Auftreten von Widersprüchen innerhalb der Generierung starke Variationen entstehen, ist es hier schwierig Zeitmessungen durchzuführen. Je nachdem ob ein Widerspruch auftritt, kann es passieren, dass die Generierung von vorne begonnen wird, wodurch sich die Laufzeit massiv verlängert. Da sich zuvor gezeigt hat, dass sich durch den erzwungenen Rand eher Konflikte bilden können (siehe Abschnitt 5.5.7), werden die folgenden Tests ohne diese Option durchgeführt.

Es wird das Turm-Modell verwendet und mit verschiedenen Parametern getestet und die Ergebnisse in Abbildung 6.5 festgehalten. Die Qualität der Ausgaben wird in diesen Tests nicht weiter beachtet und Durchläufe mit Konflikten werden verworfen. Es wurden wieder insgesamt Zehn Durchläufe durchgeführt und der Durchschnitt gebildet. Trotz der Verwendung von zufälligen Werte während der Generierung streuen die Werte nur minimal und haben eine Abweichung von $\pm \approx 7\%$.

Es zeigt sich, dass eine höhere Mustergröße zu einer niedrigeren Generierungsdauer führt. Da durch eine Vergrößerung der Mustergröße sich auch die Anzahl an einzigartigen Tiles verringert, verkleinern sich auch die Zustandsräume beim WFC-Algorithmus und es werden insgesamt weniger Schritte benötigt, diese auf einen Zustand zu reduzieren.

Die Ausgabegröße wirkt sich, wie erwartet, auf die Laufzeit aus, da durch eine größere Ausgabe mehr Zellen mit einem Zustand versehen werden müssen. Durch die dritte Dimension steigt die Laufzeit umso stärker an, je größer die Ausgabe ist.

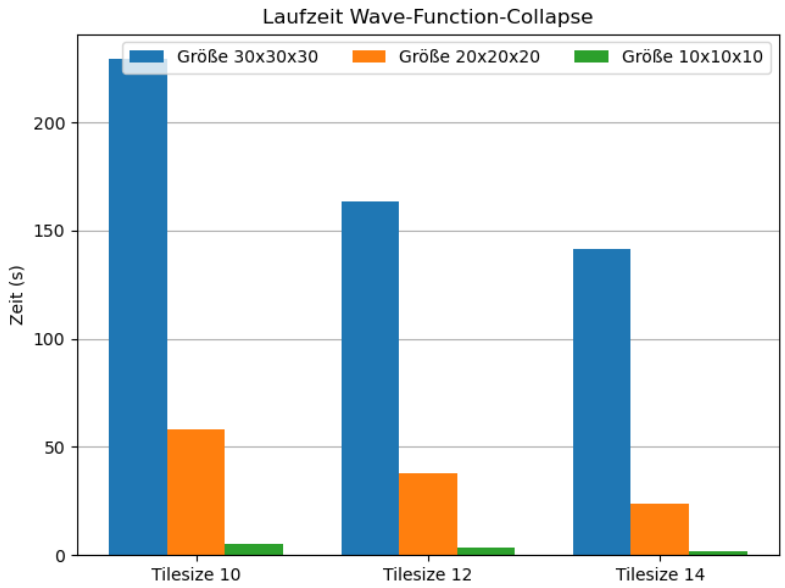


Abbildung 6.5: Laufzeitmessung des WFC-Algorithmus mit verschiedenen Ausgabegrößen und Mustergrößen. Die Ausgabegröße beeinflusst massiv die Laufzeit vom WFC-Algorithmus

Weiterhin fällt bei den Messungen das hohe Ausmaß der Generierungsdauer auf. Bereits im Zweidimensionalen konnte bei hohen Ausgabegrößen eine höhere Laufzeit festgestellt werden und es konnte mit Profiling bestätigt werden, dass der Algorithmus die meiste Zeit für Propagation benötigt. Durch die Erweiterung in die dritte Dimension wird dieses Problem nun verstärkt, da nun nicht nur mehr Zellen durch die extra Dimension existieren, sondern jede Zelle nun auch zwei neue potenzielle Nachbarn besitzt, an die propagiert werden muss. Demnach ist es ratsam, die Ausgabegröße möglichst kleinzuhalten, da diese sonst ein zu hohes Ausmaß annimmt und durch eine lange Generierungsdauer das Risiko eines Konflikts und damit eines Neustarts stetig steigt.

6.2.3 PostProzessor Marching Cubes

Zuletzt wird die Laufzeit des MC-Algorithmus betrachtet. Anders als die anderen beiden Komponenten hat MC als möglichen Parameter nur die Größe des eingegebenen Felds aus Isowerten. Abbildung 6.6 zeigt die Laufzeiten des MC-Algorithmus mit verschiedenen Mengen von Zellen.

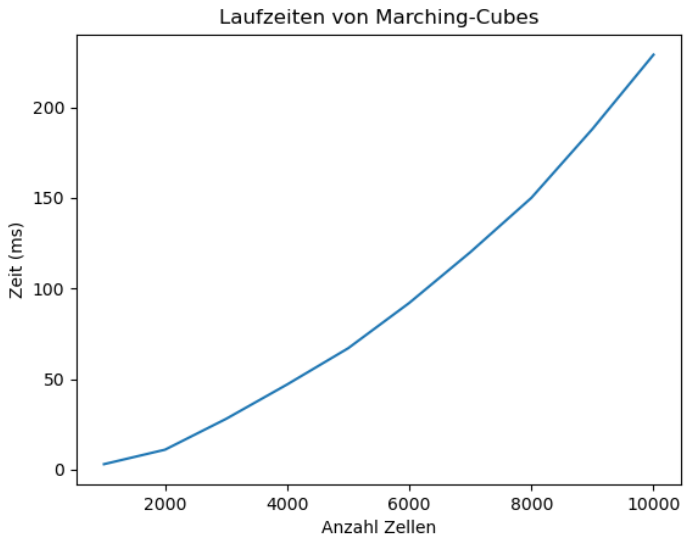


Abbildung 6.6: Laufzeitmessung des MC-Algorithmus mit verschiedenen Mengen von Zellen. Die Laufzeit steigt leicht quadratisch an.

Initial wurde hier ein linearer Zusammenhang vermutet, jedoch zeigt die Kurve leicht quadratisches Wachstum. Jeder Bearbeitungsschritt einer Zelle sollte allerdings die gleiche Zeit benötigen, da die Implementierung mit einer LUT arbeitet und nur die passenden Dreiecksnetze basierend auf den Isowerten bestimmen muss. Dies liegt wahrscheinlich an der steigenden Zahl von Dreiecken, die in jedem Iterationsschritt mit dem Hauptdreiecksnetz vereint und dort wiederholt die Normalen neu berechnet werden.

Es kann noch optional lineare Interpolation aktiviert werden, allerdings handelt es sich hier nur um triviale Berechnungen, welche keinen Einfluss auf die Laufzeit haben.

Verglichen mit dem Laufzeitanteil vom WFC-Algorithmus nimmt der MC-Algorithmus nur einen sehr kleinen Teil ein. Während des Testens wurden maximal Ausgabegrößen im Bereich von $30 * 30 * 30$ (27.000 Zellen) verwendet, welche beim WFC-Algorithmus hohe Laufzeiten im Bereich von Minuten, aber beim MC nur bei wenigen Sekunden verursacht hat. Sollten dennoch Performance Probleme bei der Dreiecksnetzerzeugung mit MC auftreten, so kann Multithreading eingesetzt werden. MC lässt sich als *embarrassingly parallel* Algorithmus klassifizieren [18], da jede Zelle völlig unabhängig von seinen Nachbarn betrachtet wird und das finale Dreiecksnetz aus den Ergebnissen der Zelle zusammengebaut werden kann. Daher könnte hier noch zusätzlich Parallelität implementiert werden, um den Prozess weiter zu beschleunigen.

6.2.4 Viewport Bildrate

Zusätzlich ist aufgefallen, dass beim Darstellen von besonders vielen farbcodierten Tiles die Bildrate von jMonkey sich erheblich verlangsamt. Vor allem tritt dies bei sehr kleinen Tilegrößen auf oder wenn beim WFC-Algorithmus eine sehr hohe Anzahl von Tiles in der Ausgabe angegeben wird.

Dies ist auf die ineffiziente Darstellung der farbcodierten Tiles zurückzuführen, da jedes Mal eine weitere neue Instanz eines transparenten Würfels hinzugefügt wird. Somit werden mit einer großen Ausgabe oder einer kleinen Tilegröße mehr Tiles innerhalb des Szenengraphens dargestellt. Weiterhin ist jedes der Tiles transparent, wodurch kein Culling oder andere Sichtbarkeitsoptimierungen durchgeführt werden können.

Dies führt insgesamt schnell zum Einbruch der Bildrate innerhalb des jMonkey-Viewports, da die einzelnen Frames nicht schnell genug dargestellt werden können.

6.3 Nutzbarkeit

Zuletzt lässt sich die entwickelte Software auf die Nutzbarkeit analysieren. Hierbei lassen sich verschiedene Aspekte betrachten, unter anderem die Gestaltung der Software-Schnittstelle zur Einbindung in weitere Projekte und Bedienung der Benutzeroberfläche.

6.3.1 Software-Schnittstelle

Die Software-Schnittstelle wurde bestmöglich aus dem ursprünglichen WFC-Framework übernommen, sodass durch die Modularisierung eine klare Trennung zwischen den einzelnen Komponenten der Distanzwertbildung, WFC und MC möglich ist. Durch die Nutzung der Pre- und Postprozessor Schnittstellen ist auch eine Abkapselung zur gesamten Applikation gegeben, sodass die interne Logik nicht nach außen hin erreichbar ist.

Es ist innerhalb des PreProzessors möglich verschiedene Parameter wie Tilegröße, Substitutionsgrenzwert und optionalen Rand einzustellen. Insbesondere durch die Tilegröße lässt sich bereits die Ausgabe stark steuern, allerdings besteht ansonsten keine Möglichkeit weiter mit der Positionierung der einzelnen Tiles zu interagieren. Bisher wird nur von der untersten, hinteren linken Ecke aus beginnend Tiles in der festgelegten Größe

platziert, bis diese das Modell vollständig einschließen. An dieser Stelle wäre es wünschenswert, das Modell verschieben und beliebig skalieren zu können, sodass die Tiles nicht an der Grenze des Modells beginnen, sondern an eine beliebige Position geschoben werden können. Dadurch kann der Inhalt der Tiles besser kontrolliert werden, wie zum Beispiel bei sich wiederholenden Elementen innerhalb des Modells, sodass garantiert werden kann, dass der Algorithmus diese als identische Tiles identifizieren kann.

6.3.2 Benutzeroberfläche

Die grafische Oberfläche bildet die Visualisierung der genutzten Komponenten und bietet eine Konfiguration in Echtzeit mit der Möglichkeit den Generierungsprozess zu beobachten. Mithilfe des 3D-Viewports kann das Modell aus allen Perspektiven betrachtet werden und durch die Darstellung der farbcodierten Tiles der WFC-Prozess nachvollzogen werden.

Darüber hinaus bietet die Benutzeroberfläche keine Erweiterungen und greift nur auf dieselben Komponenten zu, wie sie von der Software-Schnittstelle vorgegeben werden. Mit dem Zusatz, dass hier auf die internen Prozesse geschaut werden kann, um diese besser nachzuvollziehen. Da allerdings nur die Softwareschnittstelle ohne Erweiterungen genutzt wird, bleiben die Einschränkungen in der bisherigen Konfigurierbarkeit bestehen.

7 Fazit

Final sollen nun im Folgendem die Ergebnisse zusammengefasst, ein abschließendes Fazit über das Projekt gezogen und potenzielle Erweiterungen für die Zukunft diskutiert werden.

7.1 Zusammenfassung

Zielsetzung für das Projekt war es, die Beispiel-basierte Version des OverlappingModels des Wave-Function-Collapse Algorithmus mit Dreiecksnetzen zu kombinieren. Um dies umzusetzen wurden die Erfahrungen und das WFC-Framework aus vorherigen Projekten genutzt und dadurch zuvor ausgearbeitete Probleme mithilfe von Distanzfunktionen und MS innerhalb eines 2D Prototypen gelöst. Nach der erfolgreichen Implementierung konnte dieses Konzept auf die dritte Dimension ausgebaut werden.

Durch die Detailreduktion mit Distanzfunktionen ist es möglich die hohe Vielfalt des dreidimensionalen Raums zu umgehen und damit den WFC-Algorithmus zu nutzen. Anschließend konnten die Ergebnisse wieder mit dem MC-Algorithmus visualisiert werden und zeigen damit die Möglichkeit auf, prozedurale Dreiecksnetze mithilfe von Beispieleingaben zu erzeugen.

Die Ergebnisse sind hierbei sehr unterschiedlich ausgefallen. Es hat sich gezeigt, dass sich organische Modelle mit vielen Details nur schwer für die Generierung eignen, da sich durch die vielen ungleichmäßigen Formen keine Nachbarschaftsrelationen ermitteln lassen. Anorganische Modelle dagegen, wie Gebäude oder Landschaften, die viele Flächen besitze, eignen sich deutlich besser und bieten die Möglichkeit, aus diesen neue Modelle mit viel Varietät zu generieren. Nachteil ist hier jedoch, dass die generierten Modelle ein deutlich kleineren Detailgrad besitzen und durch die geringe Auflösung der Zellen einen *low-poly-look* erhalten. Je nach Eingabe ist dieser Effekt unterschiedlich stark ausgeprägt.

Neben der qualitativen Eigenschaften ist auch die Laufzeit untersucht worden. Diese ist trotz sehr kleiner Ausgaben sehr hoch und kann bereits bei Ausgaben größer als 20∗20∗20 mehrere Minuten dauern. Dies ist ein großer Schwachpunkt, da dadurch die Interaktivität der Applikation verringert wird und nur langsam Ergebnisse erzeugt werden können.

7.2 Ausblick

Sollte die Applikation erweitert werden, so können mehrere Kernaspekte fokussiert werden. Hauptziel sollte es sein, die Performance zu verbessern, sodass eine interaktive Nutzung möglich ist.

Bisher wurde die Benutzeroberfläche nur zur Visualisierung genutzt, allerdings kann diese noch weiter auf Interaktivität ausgebaut werden. So können zwar leere Tiles am Rand der Ausgabe erzwungen werden, dies verhindert allerdings nicht, dass Modelle kontextlos im Raum nebeneinander schwebend erzeugt werden. An dieser Stelle wäre es denkbar, dass innerhalb des Viewports farbcodierte Tiles ausgewählt und innerhalb des WFC-Fensters platziert werden können, sodass diese vor Beginn der Generierung definiert werden und sich die Synthese besser steuern lässt.

Als letzten Aspekt sollte die Qualität der Abtastung und der Rekonstruierung verbessert werden. Bisher gehen viele Details nach der Rekonstruierung verloren, wodurch die einzelnen Segmente des originalen Modells nicht mehr wiedererkennbar sind. Hier wäre es sinnvoll anstatt Distanzwerte für Eckpunkte eines Tiles zu bestimmen, mehr Punkte zu bestimmen oder sogar mithilfe einer Annäherung als Funktion diese zu ergänzen, sodass der MC-Algorithmus mit einer höheren Auflösung arbeiten und damit höher aufgelöste Resultate erzeugt werden können. Weiterhin wäre es sinnvoll die Implementierung für Marching Cubes zu verbessern. Bisher bilden insbesondere scharfe Ecken ein Problem, da der MC-Algorithmus diese nicht replizieren kann. Daher sollte nach Alternativen Implementierung geschaut werden, wie etwa einem hierarchischen Ansatz mit verschiedenen Auflösungen oder einer Implementierung wie das Dual Contouring [20].

Literaturverzeichnis

[1] Chernyaev, E V.: Marching Cubes 33: Construction of Topologically Correct Isosurfaces. (1996)

[2] Chiang, Yi-Jen: Out-of-core isosurface extraction of time-varying fields over irregular grids. In: *IEEE Visualization, 2003. VIS 2003.*, 2003, S. 217–224

[3] Dai, Angela ; Qi, Charles R. ; Niessner, Matthias: *Shape Completion using 3D-Encoder-Predictor CNNs and Shape Synthesis.* – URL http://arxiv.org/abs/1612.00101. – Zugriffsdatum: 2023-04-18

[4] Dorn, Christian: Generische Implementierung des Wave Function Collapse Algorithmus. (2021), S. 18

[5] Dorn, Christian: Erweiterung des generischen WFC-Frameworks in die dritte Dimension. (2022)

[6] Dürst, Martin J.: Re: additional reference to "marching cubes". 22 (1988), Nr. 5, S. 243. – URL https://dl.acm.org/doi/10.1145/378267.378271. – Zugriffsdatum: 2022-11-29. – ISSN 0097-8930

[7] Efros, A.A. ; Leung, T.K.: Texture synthesis by non-parametric sampling. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*, IEEE, 1999, S. 1033–1038 vol.2. – URL http://ieeexplore.ieee.org/document/790383/. – Zugriffsdatum: 2021-07-13. – ISBN 978-0-7695-0164-2

[8] Elor, Aviv ; Conde, Samantha: *Exploring the Creative Possibilities of Infinite Photogrammetry through Spatial Computing and Extended Reality with Wave Function Collapse.* – URL https://papers.ssrn.com/abstract=3689390. – Zugriffsdatum: 2022-08-23

[9] Gatys, Leon A. ; Ecker, Alexander S. ; Bethge, Matthias: *Texture Synthesis Using Convolutional Neural Networks.* – URL http://arxiv.org/abs/1505.07376. – Zugriffsdatum: 2023-05-22

[10] Gelder, Allen van ; Wilhelms, Jane: Topological considerations in isosurface generation. 13 (1994), Nr. 4, S. 337–375. – URL https://dl.acm.org/doi/10.1145/195826.195828. – Zugriffsdatum: 2023-04-20. – ISSN 0730-0301

[11] Groueix, Thibault ; Fisher, Matthew ; Kim, Vladimir G. ; Russell, Bryan C. ; Aubry, Mathieu: *AtlasNet: A Papier-M\^ach\'e Approach to Learning 3D Surface Generation*. – URL http://arxiv.org/abs/1802.05384. – Zugriffsdatum: 2023-04-18

[12] Gumin, Maxim: *mxgmn/WaveFunctionCollapse*. – URL https://github.com/mxgmn/WaveFunctionCollapse. – Zugriffsdatum: 2021-07-01. – original-date: 2016-09-30T11:53:17Z

[13] Guo, Baining ; Xu, Ying-Qing: Chaos Mosaic: Fast and Memory Efficient Texture Synthesis.

[14] Hart, J. C. ; Sandin, D. J. ; Kauffman, L. H.: Ray tracing deterministic 3-D fractals. 23 (1989), Nr. 3, S. 289–296. – URL https://dl.acm.org/doi/10.1145/74334.74363. – Zugriffsdatum: 2023-04-14. – ISSN 0097-8930

[15] Hart, John C.: Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. 12 (1996), Nr. 10, S. 527–545. – URL http://link.springer.com/10.1007/s003710050084. – Zugriffsdatum: 2023-04-14. – ISSN 01782789

[16] Haugo, Simen ; Stahl, Annette ; Brekke, Edmund: Continuous Signed Distance Functions for 3D Vision. In: *2017 International Conference on 3D Vision (3DV)*, 2017, S. 116–125. – ISSN: 2475-7888

[17] Heiden, W. ; Goetze, T. ; Brickmann, J.: Fast generation of molecular surfaces from 3D data fields with an enhanced "marching cube" algorithm. 14 (1993), Nr. 2, S. 246–250. – URL https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.540140212. – Zugriffsdatum: 2022-11-29. – ISSN 1096-987X

[18] Herlihy, Maurice ; Shavit, Nir: *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012. – Google-Books-ID: vfvPrSz7R7QC. – ISBN 978-0-12-397795-3

[19] Hoppe, Hugues ; DeRose, Tony ; Duchamp, Tom ; Halstead, Mark A. ; Jin, Hubert ; McDonald, John A. ; Schweitzer, Jean ; Stuetzle, Werner: Piecewise smooth surface reconstruction. In: Schweitzer, Dino (Hrsg.) ; Glassner, Andrew S. (Hrsg.) ; Keeler, Mike (Hrsg.): *Proceedings of the 21th An-*

nual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994, Orlando, FL, USA, July 24-29, 1994, ACM, 1994, S. 295–302. – URL https://doi.org/10.1145/192161.192233

[20] Ju, Tao ; Losasso, Frank ; Schaefer, Scott ; Warren, Joe: Dual contouring of hermite data. In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, Association for Computing Machinery, 2002 (SIGGRAPH '02), S. 339–346. – URL https://doi.org/10.1145/566570.566586. – Zugriffsdatum: 2023-06-16. – ISBN 978-1-58113-521-3

[21] Karth, Isaac ; Smith, Adam M.: WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning. 14 (2022), Nr. 3, S. 364–376. – Conference Name: IEEE Transactions on Games. – ISSN 2475-1510

[22] Kim, Hwanhee ; Lee, Seongtaek ; Lee, Hyundong ; Hahn, Teasung ; Kang, Shinjin: Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm. In: *2019 IEEE Conference on Games (CoG)*, 2019, S. 1–4. – ISSN: 2325-4289

[23] Liu, Shaohui ; Zhang, Yinda ; Peng, Songyou ; Shi, Boxin ; Pollefeys, Marc ; Cui, Zhaopeng: DIST: Rendering Deep Implicit Signed Distance Function With Differentiable Sphere Tracing, URL https://openaccess.thecvf.com/content_CVPR_2020/html/Liu_DIST_Rendering_Deep_Implicit_Signed_Distance_Function_With_Differentiable_Sphere_CVPR_2020_paper.html. – Zugriffsdatum: 2023-04-18, 2020, S. 2019–2028

[24] Lorensen, William E. ; Cline, Harvey E.: Marching cubes: A high resolution 3D surface construction algorithm. 21 (1987), Nr. 4, S. 163–169. – URL https://dl.acm.org/doi/10.1145/37402.37422. – Zugriffsdatum: 2022-11-18. – ISSN 0097-8930

[25] Matsuda, H. ; Cingoski, V. ; Kaneda, K. ; Yamashita, H. ; Takehara, J. ; Tatewaki, I.: Extraction and visualization of semitransparent isosurfaces for 3-D finite element analysis. 35 (1999), Nr. 3, S. 1365–1368. – ISSN 1941-0069

[26] Merrell, Paul: Example-based model synthesis. In: *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, Association for Computing Machinery, 2007 (I3D '07), S. 105–112. – URL https://doi.org/10.1145/1230100.1230119. – Zugriffsdatum: 2022-08-22. – ISBN 978-1-59593-628-8

[27] Merrell, Paul: Comparing Model Synthesis and Wave Function Collapse. (2021), S. 13

[28] Merrell, Paul ; Manocha, Dinesh: Continuous model synthesis. In: *ACM SIGGRAPH Asia 2008 papers*, Association for Computing Machinery, 2008 (SIGGRAPH Asia '08), S. 1–7. – URL https://doi.org/10.1145/1457515.1409111. – Zugriffsdatum: 2022-08-22. – ISBN 978-1-4503-1831-0

[29] Merrell, Paul ; Manocha, Dinesh: Model Synthesis: A General Procedural Modeling Algorithm. 17 (2011), Nr. 6, S. 715–728. – Conference Name: IEEE Transactions on Visualization and Computer Graphics. – ISSN 1941-0506

[30] Møller, Tobias ; Billeskov, Jonas ; Palamas, George: Expanding Wave Function Collapse with Growing Grids for Procedural Map Generation. (2020), S. 4

[31] Shaker, Noor ; Togelius, Julian ; Nelson, Mark J.: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016

[32] Shekhar, R. ; Fayyad, E. ; Yagel, R. ; Cornhill, J.F.: Octree-based decimation of marching cubes surfaces. In: *Proceedings of Seventh Annual IEEE Visualization '96*, 1996, S. 335–342

[33] Stein, R. ; Shih, A.M. ; Baker, M.P. ; Cerco, C.F. ; Noel, M.R.: Scientific visualization of water quality in the Chesapeake Bay. In: *Proceedings Visualization 2000. VIS 2000 (Cat. No.00CH37145)*, 2000, S. 509–512

[34] Sutton, P. ; Hansen, C.D.: Isosurface extraction in time-varying fields using a Temporal Branch-on-Need Tree (T-BON). In: *Proceedings Visualization '99 (Cat. No.99CB37067)*, 1999, S. 147–520. – ISSN: 1070-2385

Erklärung zur selbstständigen Bearbeitung

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

Ort	Datum	Unterschrift im Original