

Masterarbeit

Prozedurale Spielweltgenerierung: Dungeons

Maximilian Hönig

4. Juli 2012

betreut durch:
Dr. rer. nat. habil. Peter Schenzel

Arbeitsgruppe Computergrafik
Naturwissenschaftliche Fakultät III
Institut für Informatik

Martin-Luther-Universität Halle-Wittenberg
Halle (Saale)
Sachsen-Anhalt, Deutschland



Dies ist eine ergänzte und korrigierte Version der ursprünglichen Arbeit. Diese Version stammt vom 27. August 2012.

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst, noch nicht anderweitig für andere Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die den benutzten Werken wörtlich oder sinngemäß entnommenen Stellen als solche gekennzeichnet habe.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

1. Einleitung	1
1.1. Zielsetzung	1
1.2. Aufbau der Arbeit	1
1.3. Voraussetzungen	2
1.4. Ergebnisse in Kurzform	3
2. Vorbetrachtung	4
2.1. Höhlen und Dungeons	4
2.2. Prozedurale Generierung	4
2.2.1. Prozedurale Inhalte in Computerspielen	7
2.2.2. Roguelikes und prozedurale Dungeons	7
3. Aufbau der Generatorpipeline	10
3.1. Erste Ansätze	10
3.1.1. Hierarchische bausteinartige Generierung	10
3.1.2. Übereinanderlegung und Vereinigung polygonaler Netze	10
3.2. Iterative Erstellung mittels Subgeneratoren für einzelne Dungeonbestandteile	11
3.3. Allgemeine Anmerkungen	12
3.3.1. Die Irrlicht-Engine	12
3.3.2. Abstandsmaße	14
3.3.3. Zufallsgenerator	14
4. Erstellung der Höhle im Voxelraum	16
4.1. Generierung von Lindenmayer-System-Fraktalen	16
4.1.1. Lindenmayer-Systeme	17
4.1.2. Fraktalgenerator	19
4.2. Turtle-Grafik im Voxelraum	19
4.2.1. Drehungen und Radiusänderungen	20
4.2.2. Zeichnen von Strichen mit gegebenem Radius	21
4.2.3. Abarbeitung der Zeichenanweisungen	26
4.3. Nachbearbeitung	27
4.3.1. Erosion	27
4.3.2. Filterung schwiegender Fragmente	28
4.3.3. Anmerkung zur Optimierung	29
5. Umrechnung der Voxelhöhle in Dreiecksnetz	31
5.1. Umwandlungsalgorithmus	31
5.2. Berechnung der Vertexkoordinaten	34
5.2.1. Glättung des Dreiecksnetzes	36
5.3. Berechnung der Normalen	39
5.3.1. Normalen an Meshgrenzen	40
5.4. Texturierung und Beleuchtung der Höhle	42

Inhaltsverzeichnis

6. Konzept der Gänge und Räume	44
6.1. Gänge als polygonale Schläuche	44
6.1.1. Verlauf des Gangs per kubischem Hermite-Spline	45
6.1.2. Adapter	48
6.1.3. Objekte in Gängen	55
6.2. Räume als Subszenen	56
6.2.1. Ermittlung der Andockstellen	57
7. Hinzufügen von Räumen und Gängen zum Dungeon	59
7.1. Platzierung von Räumen und Verbindung durch Gänge	61
7.2. Test für Andocken an Höhle	62
7.2.1. Filterung der Scankarten	62
7.2.2. Test auf potentielle Kollision von Gang und Höhle	64
7.3. Andocken an Höhle	68
7.4. Grenzen des Verfahrens	70
8. Methoden zur Reduzierung des Renderaufwandes	72
8.1. Sichtbarkeitsgraph	72
8.1.1. Sichtbarkeitsinformationen der Höhlensubnetze	73
8.1.2. Sichtbarkeitsinformationen der Gänge	73
8.2. Reduktion von Dreiecksnetzen	76
8.2.1. Reduktion der Höhlensubnetze	77
9. Programmstruktur	86
9.1. Anmerkung zum Rendering	86
10. Auswertung	88
10.1. Untersuchungen zu Parametern und Laufzeiten	88
10.2. Weitere Probleme	90
10.3. Zusammenfassung und Bewertung	91
10.4. Ausblick	92
A. Export	93
B. Weitere Beispiele für Dungeons und Höhlen	95
C. CD-Inhalt	99
Abbildungsverzeichnis	101
Literaturverzeichnis	103

1. Einleitung

Diese Arbeit beschäftigt sich mit der prozeduralen Generierung von Dungeons, die als Spielwelten in Computerspielen verwendet werden können. Computerspiele stellen ein interaktives Unterhaltungsmedium dar, welches in den letzten Jahren massiv an Bedeutung gewonnen hat. Die entwickelten Spiele sind mitunter Großproduktionen, die in ihrem Budget und Aufwand mit Hollywood-Blockbustern vergleichbar sind. Herausforderungen bei der Entwicklung von Computerspielen sind unter anderem: die Einhaltung des Budgets, die Erstellung von vielen Spielinhalten und das Einhalten von Speicherplatzbeschränkungen. Diese Punkte stehen allerdings im Widerspruch zueinander.

Dieser Widerspruch kann durch den Einsatz prozeduralen Generierungsmethoden entspannt werden. Mittels entsprechender Algorithmen lassen sich hiermit aus wenigen Eingabeparametern komplexe Inhalte, wie beispielsweise Landschaften, erzeugen. Somit können Spielinhalte vergleichsweise schnell und einfach erstellt werden. Die Algorithmen werden mitunter direkt in Spiele integriert. Durch die Verwendung zufälliger Werte für ausgewählte Generierungsparameter kann bei jedem Start des Spiels eine neue Spielwelt erzeugt werden. Hierdurch wird für eine hohe Wiederholbarkeit gesorgt. Methoden der prozeduralen Generierung erlauben es ebenfalls, große Mengen an Inhalten mit wenig Speicherverbrauch abzulegen, da nur die Generierungsparameter gespeichert werden müssen und nicht das Gesamtresultat.

Dungeons, wörtlich übersetzt Verliese oder Kerker, stellen ein wichtiges Element in Computerspielen dar. Sie bestehen aus untereinander verbundenen Höhlen, Gängen und Räumen. Insbesondere in der sehr weit verbreiteten Gruppe der Rollenspiele bilden Dungeons oft den Hauptteil der Spielwelt. Bis vor einigen Jahren waren diese primär zweidimensional aufgebaut, jedoch gewinnen zunehmend dreidimensionale Strukturen an Bedeutung. Im Gegensatz zu zweidimensionalen Dungeons, für die es bereits viele Generierungswerkzeuge gibt, ist die Erstellung dreidimensionaler Dungeons deutlich weniger gut erforscht.

1.1. Zielsetzung

Ziel der Arbeit ist es, ein Generierungsverfahren für realistische 3D-Dungeons zu entwickeln. Zu diesen Dungeons gehören zerklüftete Höhlensysteme, gewundene Gänge und passend ausgestattete Räume. Schwerpunkt der Arbeit liegt auf der Generierung der Geometrien dieser Objekte sowie der passenden Platzierung und nahtlosen Verbindung aller Bestandteile. Die entwickelten Algorithmen sollen effizient bezüglich ihrer Laufzeit und des Speicherverbrauchs sein und konkret in einem Programm implementiert werden.

1.2. Aufbau der Arbeit

Der Aufbau der Arbeit folgt, nach Betrachtung der Grundbegriffe *Höhle*, *Dungeon* und *prozedurale Generierung* und Beispielen für diese, dem eigentlichen Generierungsprozesses des Dungeons. Dabei werden die jeweiligen Grundlagen in den Abschnitten erläutert, in denen diese Verwendung finden. Dies hat den Vorteil, dass gleich darauf Bezug genommen werden kann.

1. Einleitung

Kapitel drei beschreibt einige selbst entwickelte Herangehensweisen für die Dungeongenerierung, inklusive des finalen Ansatzes. Weiterhin werden wichtige Bemerkungen zur verwendeten Engine, dem Koordinatensystem, den Abstandsmaßen und dem Zufallsgenerator getroffen. Kapitel vier bis acht bilden den Kern der Arbeit und beschreiben die eigentlichen Schritte des Generierungsprozesses.

Kapitel vier erklärt die Erstellung von Voxelhöhlen auf der Basis von L-Systemen. Grundlagen hierfür sind die Konzepte Fraktal, L-System, Voxel und 3D-Bresenham-Linien-Algorithmus. Basierend darauf wird ein Verfahren entwickelt, um Turtle-Grafiken im Voxelraum zu zeichnen. Weiterhin werden zwei Algorithmen zur Nachbearbeitung der Voxelgrafiken ausgearbeitet.

Kapitel fünf beschreibt die Umwandlung von Voxelgrafiken in Dreiecksnetze und deren Darstellung. Grundlagen hierfür sind die Konzepte Sweepverfahren, Octree, Normalenberechnung, 3D-Texturen und das Lambertsche Beleuchtungsmodell. Es werden Verfahren entwickelt, um diese Umwandlung durchzuführen, passende Koordinaten der Eckpunkte zu berechnen, Normalen von äquivalenten Eckpunkten an Meshgrenzen zusammenzufassen und das Dreiecksnetz zu beleuchten.

Kapitel sechs beschreibt die Erstellung von Gängen und Räumen. Grundlagen hierfür sind die Konzepte kubische Hermite-Splines, Bogenlänge und das Bisektionsverfahren. Es werden Algorithmen zur Erstellung polygonaler Schläuche und polygonaler Adapter hergeleitet, das Platzieren von Objekten entlang Kurven erläutert und das Konzept der Subszene definiert.

Kapitel sieben beschreibt die Platzierung und das Verbinden von Gängen und Räumen mit und um die Voxelhöhle herum. Grundlagen hierfür sind die Konzepte Bézierkurven und konvexe Hülle. Es werden Algorithmen entwickelt, um diese Platzierung vorzunehmen und die Objekte korrekt zu verbinden.

Kapitel acht beschreibt Techniken zur Reduzierung des Renderaufwandes. Grundlagen hierfür sind die Konzepte Culling, Level of Detail und die Reduktion von Dreiecksnetzen. Es wird ein Verfahren entwickelt, um einen Sichtbarkeitsgraphen aufzubauen und die Sichtbarkeitseigenschaften der generierten Objekte zu bestimmen. Weiterhin wird ein eigener Reduktionsalgorithmus für möglichst regelmäßige Meshes aufgezeigt.

Kapitel neun beschreibt den strukturellen Aufbau des entwickelten Generatorprogramms. In *Kapitel zehn* werden Tests bezüglich verschiedener Generierungsparameter durchgeführt, eine Zusammenfassung gegeben und mögliche Punkte für weiterführende Arbeiten aufgeführt.

1.3. Voraussetzungen

Voraussetzung für das Verständnis der Arbeit sind fundierte Kenntnisse der Grundkonzepte der Informatik, wie dem binären Zahlensystem und der Bedeutung der Landauschen Symbole. Ebenfalls Voraussetzung ist das Verständnis der Grundkonzepte der Computergrafik,

1. Einleitung

wie Vektorrechnung und Dreiecksnetze, sowie gute Kenntnisse der Mathematik. Hilfreich sind weiterhin Kenntnisse in algorithmischer Geometrie, als auch vertiefte Kenntnisse der Computergrafik.

1.4. Ergebnisse in Kurzform

Es wurde eine Methode entwickelt, um dreidimensionale Dungeons generieren zu können. Es sind Generierungsverfahren für zerklüftete Höhlensysteme und gewundenen Gänge geschaffen worden. Für die Räume wurde ein Konzept überlegt, wie sich vordefinierte Szenen als Vorlagen für diese einbinden lassen. Der Realismusgrad der erzeugten Höhlen ließe sich durch passende prozedurale 3D-Texturen weiter steigern.

Beim Entwurf der Algorithmen wurde darauf geachtet, dass diese möglichst auch für andere Zwecke eingesetzt werden können. Es ist möglich, L-Systeme und Turtle-Grafiken im Voxelraum darzustellen und daraus ein Dreiecksnetz inklusive Reduktionsstufen zu berechnen. Die Methode zur Erstellung polygonaler Schläuche entlang von Kurven sowie das Konzept der Adapter zum Anbinden an andere Geometrien erlauben ebenfalls weitere Einsatzmöglichkeiten. Die entwickelten Verfahren sind effizient, zumeist handelt es sich um Linearzeitalgorithmen.

Sie wurden sowohl in einem Programm implementiert als auch getestet. Die Ergebnisse sind visuell ansprechend und die Generierungsmethoden auch in der Praxis effektiv. Für die Erstellung eines kompletten Dungeons wird auf handelsüblichen PCs weniger als eine Minute Rechenzeit benötigt.

2. Vor betrachtung

In diesem Kapitel erfolgt zuerst die Erklärung der Begriffe Höhle und Dungeon sowie das Aufführen einiger Beispiele hierfür. Danach wird der Begriff prozedurale Generierung näher betrachtet und es werden Beispiele dafür aus dem Bereich der Spielweltgenerierung aufgezeigt.

2.1. Höhlen und Dungeons

Höhlen seien im Folgenden analog zu den Ausführungen von [Tri68, S.6 ff.] beschrieben. Hier wird erläutert, dass die Auffassungen darüber, was unter dem Begriff Höhle zu verstehen ist, relativ subjektiv sind und weit auseinander gehen. [Tri68] beschreibt Höhlen, als „vom Menschen begehbar und durch Naturprozesse gebildete Hohlräume, die ganz oder teilweise von festen Gestein umgeben sind“.

Es wird zwischen primären und sekundären Höhlen unterschieden. Primäre Höhlen entstehen schon beim Prozess der Gesteinsbildung. Ein Beispiel sind Lavahöhlen, die entstehen, wenn erkaltende Lava oberflächlich erstarrt und sich tiefere Teile des Lavastroms weiterbewegen. Sekundäre Höhlen entstehen erst nach dem Prozess der Gesteinsbildung. Ein Beispiel hierfür sind Wasserhöhlen, deren Entstehung auf das erosive Wirken von Wasser zurückzuführen ist.

In [Duc11] wird eine Höhle als „natürlicher unterirdischer Hohlraum, der groß genug ist von Menschen betreten zu werden“ beschrieben. Hier werden Gletscherhöhlen explizit als Höhlen aufgeführt. Beispiele für Höhlen sind in Abbildung 2.1 dargestellt.

Dungeons bestehen meist aus einer oder mehreren Höhlen, Räumen und Gängen zur Verbindung dieser Bestandteile. Die Räume und Gänge sind dabei zumeist, wie auch die Höhle, unterirdisch angelegt. Es müssen nicht alle dieser Bestandteile vorhanden sein. Es gibt auch Dungeons, die nur aus Räumen und Gängen bestehen, als auch solche, die nur ein Höhlensystem bilden. Diese Ausarbeitung beschäftigt sich mit der Generierung von Dungeons, die aus einer Höhle sowie mehreren Gängen und Räumen aufgebaut sind. Die generierten Höhlen besitzen keine direkte Verbindung zur Außenwelt, sind also geschlossen und nur durch Gänge betretbar.

Dungeons finden in Spielen vielfach Verwendung, insbesondere in Rollenspielen mit Fantasyhintergrund. Sie werden von gefährlichen Monstern bevölkert, sind gespickt mit tödlichen Fallen und reich an wertvollen Schätzen. Einige Beispiele für Dungeons sind in Abbildung 2.2 aufgeführt. Dungeons können zu Fuß erkundet werden, aber auch fliegend und kletternd.

2.2. Prozedurale Generierung

Unter *prozeduraler Generierung*, auch als *prozedurale Synthese* bezeichnet, versteht man die automatische Generierung von Daten nach einer festgelegten Generierungsprozedur. Die Daten werden also in ihrer endgültigen Form nicht manuell erzeugt, sondern unter Vorgabe entsprechender Parameter mittels eines Algorithmus. Solche Daten sind oft medialer Natur,

2. Vorbetrachtung



Abbildung 2.1.: *Diverse in der Natur vorkommende Höhlen: v.l.n.r & v.o.n.u.*
 (a) Gletscherhöhle [Hak09], (b) Lavahöhle [Neu09], (c) die Mammuthöhle im Dachstein [Ste11], (d) Kreidehöhle in der Weißen Wüste [Mos06],
 (e) Grundriss der Crystal Cave in Wisconsin [Uni07]

2. Vor betrachtung

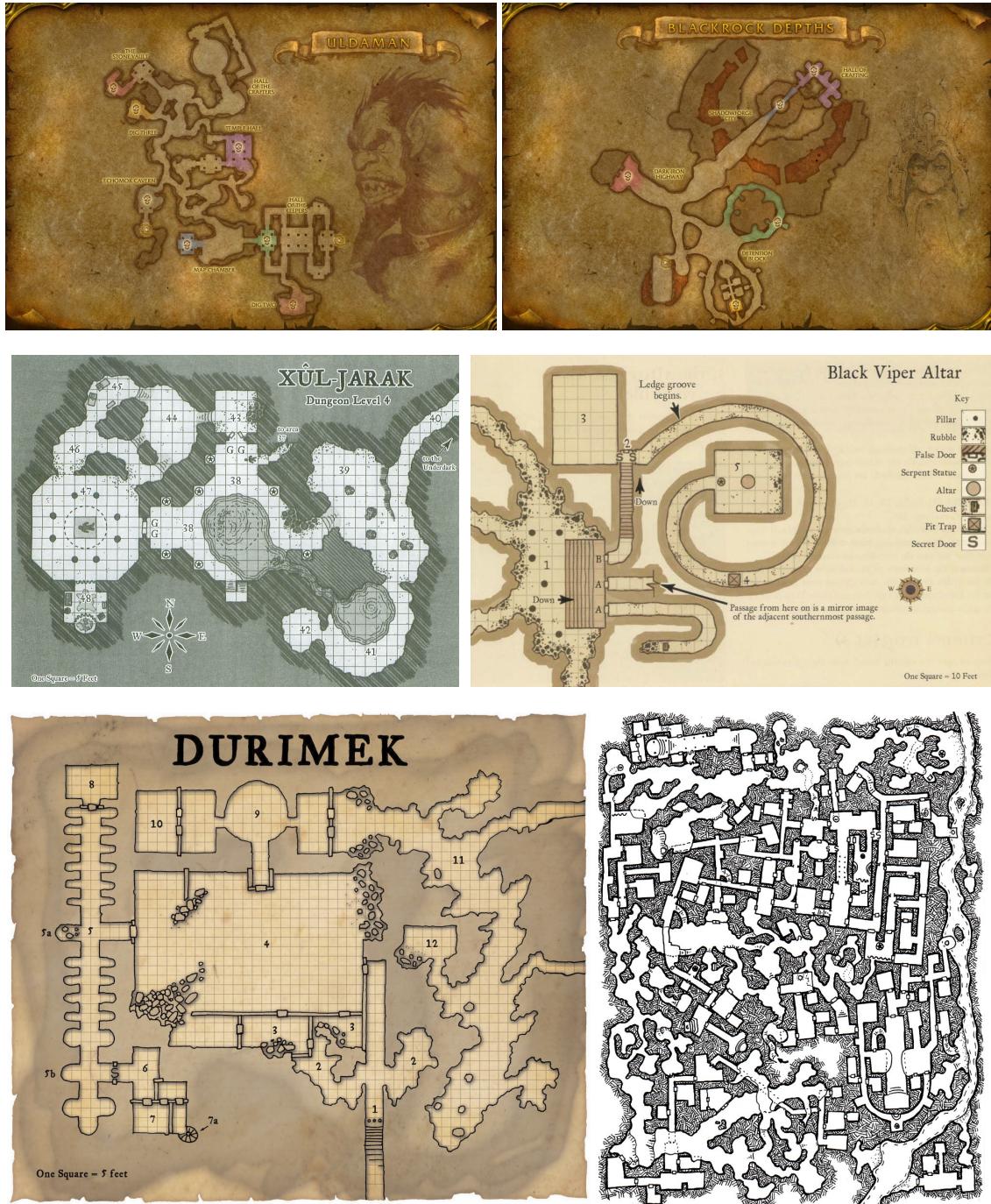


Abbildung 2.2.: *Diverse Dungeons*: v.l.n.r & v.o.n.u.

- (a)+(b) aus dem Computerspiel World of Warcraft [Bli12c],
- (c)[Per05, S.22]+(d)[GBD04, S.169]+(e)[Wil04, S.3] aus dem Pen-and-Paper-Rollenspiel Dungeons & Dragons,
- (f) Beispiel eines Megadungeons [Log10]

2. Vorbetrachtung

beispielsweise Texturen, Geometrien oder Klänge. Der Prozess ist meist deterministisch, allerdings können auch Zufallsparameter einbezogen werden.

2.2.1. Prozedurale Inhalte in Computerspielen

Prozedurale Inhalte sind in Computerspielen weit verbreitet. Es gibt zwei grundlegende Möglichkeiten zur Anwendung: die Daten werden bei Bedarf in Echtzeit berechnet oder sie sind vorberechnet. Die Vorberechnung kann in einem externen Generatorprogramm geschehen oder intern im Spiel selbst, z.B. beim Start des Programms.

Durch prozedurale Generierung kann Speicherplatz eingespart werden, da die Generierungsparameter i.A. deutlich weniger Speicher benötigen als die generierten Daten. Weiterhin ist eine hohe Vielfalt bezüglich dieser Daten erzielbar, die bei manueller Erstellung nicht möglich wäre.

Minecraft

Minecraft [Moj12] ist eine Adventure- und Aufbausimulation. Das Gelände ist dreidimensional aus Blöcken aufgebaut. Jeder Block besteht aus einem bestimmtem Material wie Fels, Erde, Sand, Wasser oder Luft. Das Gelände wird on-the-fly prozedural generiert, die theoretische Spielweltgröße ist riesig. Es wird immer nur das Gelände in der Nähe der Spielfigur erzeugt. Bei Änderungen am Gelände, beispielsweise durch Aktionen des Spielers, werden diese Änderungen gespeichert.

Der Basisansatz des Generierungsverfahrens gestaltet sich nach [Per11] folgendermaßen: Mittels eines 3D-Perlin-Noise wird die Dichte jedes Blocks bestimmt, wobei zusätzlich die Höhe des Wasserpegels einbezogen wird. Ist die Dichte kleiner null, handelt es sich bei dem Block um Luft, ist sie größer oder gleich null, handelt es sich um festes Material.

.kkrieger

.kkrieger ist ein 3D-Ego-Shooter von der Codergruppe Farbrausch/.thepronukkt [Far10]. Die Besonderheit ist, dass das Programm inklusive aller Daten nur 96 KiB groß ist. Die benötigte Daten werden direkt nach Programmstart prozedural generiert. Dazu gehören die 3D-Modelle, Texturen, Musik, Soundeffekte und der Aufbau der Spielwelt selbst.

Die Generierung funktioniert nach dem Prinzip modularer Synthesizer [Gie04]. Hierbei sind einzelne Generierungsbausteine, wie Perlin Noise oder Bump Mapping, über einen kreisfreien Graphen miteinander verknüpft. Die Ausgabe eines Generatorbausteins ist dabei die Eingabe des nächsten. Gespeichert werden nur diese Operatoren und ihre Verbindungen. Das Spiel kann mit diesen Informationen alle Daten prozedural generieren.

2.2.2. Roguelikes und prozedurale Dungeons

Roguelikes [Wik12] sind Computerrollenspiele, die auf dem Spielprinzip des Spiels „Rogue“ basieren. Hierzu gehört vor allem eine zufällig generierte Spielwelt, primär aufgebaut aus zweidimensionalen Dungeons, und das Hindurchkämpfen durch die diese Dungeons bewohnenden Monsterhorden. Die 2D-Dungeons in Roguelikes bestehen aus einzelnen Feldern, die in einem quadratischen Gitter angeordnet sind. Diese Felder sind als bestimmte Typen charakterisiert, beispielsweise als Räume, Gänge oder Wände.

Viele Computerrollenspiele basieren heute noch auf diesem Prinzip. Beispiele hierfür sind

2. Vorbetrachtung

Diablo II und Diablo III, für die Screenshots in Abbildung 2.3 gezeigt sind. Es gibt eine Vielzahl verschiedener Verfahren, um solche Dungeons zu erstellen.



Abbildung 2.3.: Bilder von Roguelikes: v.l.n.r. Screenshot von (a) Diablo II [Bli12a], (b) Diablo III [Bli12b]

Der folgende Algorithmus für die Generierung eines Dungeons zur Verwendung in Roguelikes wird in [Hug10] beschrieben:

1. Die Karte besteht aus quadratischen Feldern, initial alles Wände bzw. Felsen, also unpassierbares Gelände.
2. Teile die Karte durch ein Gitternetz auf, jedes Gitterstück beinhaltet einen Raum.
3. Ordne jedem Raum zu: Flag „verbunden“ (initial sind alle Räume nicht-verbunden), Array zum Speichern von Verbindungen zu Nachbarräumen.
4. Bestimme einen zufälligen Raum, markiere diesen als „verbunden“ und mache ihn zum aktuellen Raum.
5. Solange es nicht-verbundene Nachbarräume gibt:
 - Stelle eine Verbindung zu einem Nachbarraum her.
 - Markiere diesen als „verbunden“ und mache ihn zum aktuellen Raum.
6. Solange es nicht-verbundene Räume gibt:
 - Wähle einen nicht-verbundenen Raum.
 - Versuche zu einem zufälligen Nachbar zu verbinden, der schon „verbunden“ ist.
 - In diesen Fall markiere den aktuellen Raum als „verbunden“, ansonsten wähle nächsten nicht-verbundenen Raum aus.
7. Alle Räume sind nun verbunden.
8. Füge weitere, zufällige Verbindungen zwischen benachbarten Räumen hinzu.
9. Felder setzen: Freiräume
 - Zeichne Räume auf der Karte ein.
 - Zeichne Korridore zwischen miteinander verbundenen benachbarten Räumen.
10. Felder setzen: Türen
 - Suche nach Feldern mit 2 angrenzenden Wandfeldern, 1-2 angrenzenden Raumfeldern und 0-1 angrenzenden Korridorfeldern.
 - Wandle diese in Türen um.
11. Felder setzen: Treppen
 - Setze Treppe nach oben im ersten als „verbunden“ markierten Raum.
 - Setze Treppe nach unten im letzten als „verbunden“ markierten Raum.

2. Vorbetrachtung

Durch die gewählte Methode des Treppensetzens wird i.d.R. ein langer Laufweg erzielt, der durch viele Räume hindurchführt.

Höhlen per zellulärem Automat

In [Lar03] wird eine Methode für die Generierung von 2D-Höhlen beschrieben. Das Spielfeld besteht aus einzelnen quadratischen Feldern. Die Methode basiert auf zellulären Automaten:

1. Erstelle initiale Karte durch zufälliges Setzen der Felder nach folgender Verteilung:
 - 60% der Felder sind Gestein.
 - 40% der Felder sind Freiraum.
2. Wende die 4-5 Regel für zelluläre Automaten auf jedes Feld der Karte an (eventuell mehrfache Durchläufe):
 - Bei weniger als 4 angrenzenden Felsfeldern (in 8er Nachbarschaft): Feld wird zu Freiraum.
 - Bei mehr als 5 angrenzenden Felsfeldern: Feld wird zu Fels.
 - Ansonsten: Feld bleibt so belassen.
3. Verbinde die einzelnen Kavernen zu einer Gesamthöhle:
 - Bestimme die einzelnen Kavernen.
 - Zeichne von einem Freiraumfeld jeder Kaverne eine mäandrische Linie von Freiraumfeldern in Richtung Mitte, bis eine andere Kaverne getroffen wird.

DungeonMaker

Der DungeonMaker [Hen02b] ist eigenständiger Dungeongenerator zum Erstellen gitterbasierten 2D-Dungeons. Er ist beispielsweise für Pen-and-Paper-Rollenspiele oder zur Einbindung in Computerspiele nutzbar.

Der Generierungsprozess verwendet Mechaniken von virtuellem Leben, um Dungeons zu erstellen. Zuerst platziert der Nutzer einige Designelemente auf der Karte, z.B. Räume und Mauern. Nun generiert der DungeonMaker um diese vorplatzierten Elemente den Dungeon. Dies geschieht mit den Buildern, kleinen Agenten, die mittels KI gesteuert werden. Subklassen der Builder sind WallCrawler (bauen Wände), Tunneler (heben Tunnel aus) und Roomies (bauen Räume).

Ein Vorschlag für das Aufbrechen der zugrundeliegenden achsenparallelen Gitterstruktur ist in [Hen02a] aufgeführt: Die entstehende Karte wird in 3D gerendert und dabei verzerrt.

3. Aufbau der Generatorpipeline

Wie im vorherigen Kapitel dargelegt wurde, ist die Generierung von auf einem Gitterlayout basierenden 2D-Dungeons gut erforscht. Ziel ist es, realistische 3D-Dungeons ohne achsenparallele Gitterstruktur zu erstellen. Da Spiele i.A. mit Dreiecksnetzen als Basis für Geometrien arbeiten, sollen als Output eben solche plus eventuelle Zusatzdaten erzeugt werden. Das bedeutet, dass eine Konvertierung aller durch andere Modellierungsmethoden erzeugten Geometrien in Dreiecksnetze erforderlich ist.

3.1. Erste Ansätze

3.1.1. Hierarchische bausteinartige Generierung

Der erste Ansatz verwendet eine hierarchische Generierung. Ein Generator ist für das Levellayout zuständig. Er legt Lage, Eigenschaften und Verbindungen einzelner Levelbausteine fest. Die einzelnen Levelbausteine werden durch weitere Generatoren erstellt. Die Generierung der Bausteine erfolgt nach den bereits festgelegten Eigenschaften. Dazu gehören u.a. Öffnungen bzw. Verbindungen an definierten Stellen sowie die Vorgabe der Größe und des Typs des Bausteins, z.B. Raum, Gangstück oder Höhlenteil. Weitere Generatoren generieren die Feingometrien, also Teile der Levelbausteine, wie beispielsweise die Wand eines Raums, Tische oder Torbögen.

Zu den möglichen Probleme zählen die relativ regelmäßigen erzeugten Strukturen, die ähnlich zu den Roguelikes eine Art Kachel-Layout darstellen würden. Problematisch ist ebenfalls, die Höhlenstruktur realistisch und exakt in einen Bereich einzupassen, auch in Bezug auf Öffnungen und Verbindungen der Höhle zum Rest des Dungeons. Durch diese Problematik ist der Realismusgrad eventuell nicht hoch genug.

3.1.2. Übereinanderlegung und Vereinigung polygonaler Netze

Der nächste Ansatz orientiert sich bei der Generierung mehr an der Realität und damit an der Frage: Wie würde ein Dungeon entstehen? Zuerst erfolgt die Generierung einer zerklüfteten, gewundenen Höhle. Nach der Höhlenerstellung geschieht das Hereinbauen von Räumen und Gängen um und in die fertige Höhlenstruktur. Insgesamt sind folgende Schritte angedacht:

1. Schritt: Generierung der Höhle. Ein Beispiel für die zugrundeliegende Idee ist in Abbildung 3.1 gezeigt: Eine fraktalähnliche Struktur auf der Basis eines Lindenmayer-Systems (s. Kapitel 4.1) ergibt durch räumliche Expansion ein zerklüftetes Höhlensystem. Hierzu erfolgt zuerst die Erstellung eines fraktalartigen Höhlenskeletts mittels eines passenden L-Systems. Danach wird dieses Skelett zu einem Dreiecksnetz aufgeblasen und entstehende Überlappungen werden entfernt.

Problematisch ist das Entfernen der Selbstüberschneidung der Höhlenstruktur. Hierzu müssen alle Schnitte zwischen Dreiecken gefunden, die schneidenden Dreiecke zerlegt und danach sämtliche inneren Dreiecke entfernt werden. Dieser Prozess ist sehr aufwändig.

2. Schritt: Platzierung von Gängen und Räumen an passenden Stellen. Zuerst werden die

3. Aufbau der Generatorpipeline

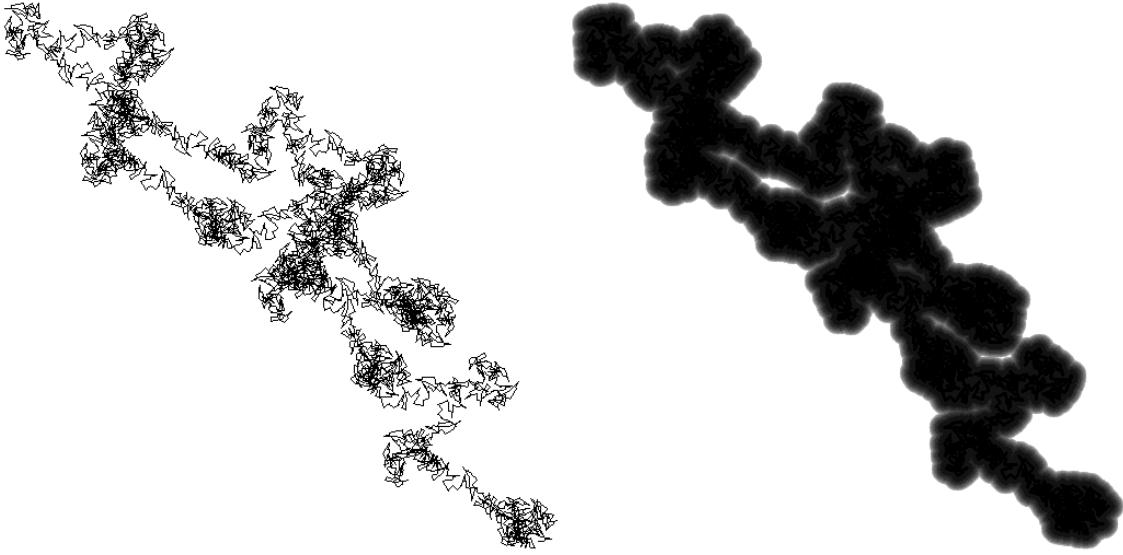


Abbildung 3.1.: *L-Systeme für Höhlenstrukturen*: v.l.n.r. (a) dargestellt ist die sechste Iteration eines zufällig erzeugten L-Systems G mit
 $G = \langle \{F, X, Y, +, -\}, FF, \{F \rightarrow FXF + YX - F-, X \rightarrow YFY+, Y \rightarrow -X - YXFF\} \rangle$ und $\alpha_{Gier} = 276^\circ$
(b) dieselbe Iteration desselben L-Systems, gezeichnet mit räumlicher Ausdehnung der Striche

passenden Stellen gesucht. Dann erfolgt die Auswahl oder auch Generierung der Gänge und Räume. Abschließend werden diese platziert, wobei sich die Objekte an den Verbindungsstellen überlappen.

3. Schritt: Verschmelzung. Die Höhle, Gänge und Räume werden zu einem Dreiecksnetz verschmolzen. Dies entspricht der Booleschen Vereinigung des Inneren der Ausgangsnetze.

Es ergibt sich ein ähnliches Problem wie in Schritt 1. Zusätzlich herausfordernd ist die Beibehaltung der passenden Texturierung beim Zerlegen von Raum- oder Gangdreiecken. Die Höhle kann per prozeduraler 3D-Textur texturiert werden, wofür keine expliziten Texturkoordinaten berechnet werden müssen. Bei Gängen und Räumen sind 2D-Texturen besser geeignet.

3.2. Iterative Erstellung mittels Subgeneratoren für einzelne Dungeonbestandteile

Das umgesetzte Verfahren enthält Elemente aus beiden Ansätzen. Die Generierungsme thode ist iterativ und der natürlichen Entstehung eines Dungeons folgend. Es werden Subgeneratoren für die Erstellung der einzelnen Geometrien verwendet, wobei die Teile des Dungeons so konstruiert werden, dass sie nahtlos aneinander passen. Das Verfahren besteht aus den folgenden Schritten:

1. Schritt: Es werden Turtle-Grafik-Zeichenanweisungen mittels eines Lindenmayer-Systems

3. Aufbau der Generatorpipeline

erstellt (s. Kapitel 4.1). Diese beschreiben die grundlegende Höhlenstruktur.

2. Schritt: Die Turtle-Grafik wird im Voxelraum gezeichnet (s. Kapitel 4.2). Voxel haben gegenüber der Verwendung eines Dreiecksnetzes (wie in Abschnitt 3.1.2) den Vorteil, dass die Überlagerungen der einzelnen Strukturen keine Konflikte verursachen.

3. Schritt: Es erfolgt die Erosion der Höhle im Voxelraum, um die natürliche Zerklüftung zu verstärken (s. Kapitel 4.3.1).

4. Schritt: Die Filterung schwebender Fragmente im Voxelraum wird durchgeführt (s. Kapitel 4.3.2). Hierbei werden solche Fragmente entfernt, da sie der natürlichen Gravitation widersprechen.

5. Schritt: Räume und Gänge werden in den Freiräumen um die Höhle hinzugefügt (s. Kapitel 6 und 7). Die Räume werden dabei über die Gänge untereinander und mit der Höhle verbunden. Dieser Schritt beinhaltet ebenfalls die Berechnung verschiedener Detailstufen und Sichtbarkeitsinformationen für die Gänge (s. Kapitel 8).

6. Schritt: Eine zweite Filterung schwebender Fragmente im Voxelraum wird durchgeführt, um die eventuell durch das Ausfräsen von Gangöffnungen entstandenen Fragmente zu entfernen.

7. Schritt: Die Umwandlung der Voxelhöhle in ein Dreiecksnetz bzw. mehrere Subnetze findet statt (s. Kapitel 5). Dieser Schritt beinhaltet ebenfalls die Berechnung verschiedener Detailstufen für die entstehenden Netze und der Sichtbarkeitsberechnungen für die Höhle (s. Kapitel 8).

Diese Umwandlung der Höhle in ein Dreiecksnetz erfolgt erst im letzten Schritt, da in Schritt 5 und 6 Änderungen an der Voxelhöhle vorgenommen werden. In Schritt 5 werden Öffnungen für Gänge hinzugefügt und in Schritt 6 dadurch entstandene schwelende Fragmente entfernt.

Auf die Erosion und das Filtern schwelender Fragmente kann bei der Erstellung eines Dungeons verzichtet werden. Wenn Schritt 6 durchgeführt werden soll, muss auch Schritt 4 durchgeführt werden, da sonst Räume in Fragmenten platziert werden können, die später entfernt werden.

3.3. Allgemeine Anmerkungen

3.3.1. Die Irrlicht-Engine

Das entwickelte Dungeongenerator-Programm nutzt die Irrlicht-Engine [Irr12] für die Darstellung, für die GUI, als Exporter und Importer, zur Speicherung von Dreiecksnetzen und für mathematische Hilfsroutinen. Die Vorteile der Irrlicht-Engine im Vergleich zu anderen Engines werden in [BFH⁺09] dargelegt. Zu nennen ist hier beispielsweise die große Anzahl an unterstützten Mesh- und Texturformaten.

Der wichtigste Grund für die Verwendung ist das Vorhandensein eines eigenen 3D-Szenenformats, dem .irr-Format, welches auf XML aufbaut und den Export sowie Import

3. Aufbau der Generatorpipeline

kompletter Szenen ermöglicht. Damit können erstellte Dungeons exportiert werden. Eine solche Szene kann dann als Spielwelt in einem Computerspiel importiert werden, wie es unter anderem in [Hön09] getan wird.

Die Szenen in Irrlicht werden als Szenengraph verwaltet. Der Aufbau dieses Graphen entspricht einer Baumstruktur. Jeder Knoten enthält eine Transformation und optional ein Szenenobjekt, wie z.B. ein zu zeichnendes 3D-Objekt. Die Berechnung der resultierenden Transformation für jedes Objekt geschieht durch Aufmultiplizierung aller Transformationsmatrizen von der Wurzel des Szenengraphen bis zum Objekt selbst.

Die Transformationen sind entweder direkt als 4×4 -Matrix M angebbar oder einzeln als 3D-Vektoren für Skalierung, Rotation und Translation. Die Rotationswinkel $(\alpha_x, \alpha_y, \alpha_z)$ sind im Gradmaß für die jeweilige Achse im mathematisch positiven Sinn angeben.

Der Aufbau von M entspricht dem aus Formel 3.1. Die Matrizen der Skalierung S , der Translation T und der Rotationen gemäß Eulerwinkeln R_x (Rotation um X-Achse), R_y (Rotation um Y-Achse) und R_z (Rotation um Z-Achse) seien dabei aus den entsprechenden 3D-Vektoren gebildet.

$$M = T \cdot R_z \cdot R_y \cdot R_x \cdot S \quad (3.1)$$

Dreiecksnetze

Die Ablage der Dreiecksnetze, die auch als Dreiecksmeshes oder Meshes bezeichnet werden, geschieht mittels einer Menge von Vertices und einer Menge von Dreiecken. Die *Vertices* werden in einem Array gespeichert. Jeder Vertex hat die Eigenschaften Position, Normale und Texturkoordinaten. Die *Dreiecke* sind durch ihre Eckpunkte definiert. In einem Array werden die Indices der Vertices abgespeichert, aus denen die Dreiecke bestehen. Jedes Dreieck wird aus drei aufeinanderfolgenden Einträgen gebildet.

Durch die Verwendung eines unsigned 16 Bit-Datentypes für die Vertexindices der Dreiecke ist die maximale Anzahl der Vertices je Netz auf 65536 begrenzt. Meshobjekte bestehen in Irrlicht aus einem oder mehreren Meshbuffern, die einzelne Dreiecksnetze darstellen. Die Begrenzung der Vertices gilt je Meshbuffer.

Koordinatensysteme

In vielen 3D-Modelling-Programmen und der meisten Literatur wird ein rechtshändiges Koordinatensystem mit Z als Höhenachse verwendet. Irrlicht verwendet ein linkshändiges Koordinatensystem mit Y als Höhenachse.¹ Diese Ausarbeitung und das Dungeongeneratorprogramm verwenden ebenfalls ein derartiges linkshändiges System, wodurch Umrechnungen innerhalb des Generatorprogramms vermieden werden können.

Falls eine Konvertierung erforderlich ist (beispielsweise beim Export in ausgewählte 3D-Formate), kann zur Umrechnung beider Koordinatensystem die Y- und Z-Achse vertauscht werden. In homogenen Koordinaten ist dies durch die Multiplikation mit Matrix M_k mög-

¹Die Idee hinter einem solchen Koordinatensystem ist es, das zweidimensionale XY-Koordinatensystem des Bildschirms um die Tiefenachse Z zu erweitert.

3. Aufbau der Generatorpipeline

lich:

$$M_k = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.2)$$

Unter Umständen muss nach dieser Umrechnung die Umlaufrichtung der Dreiecke geändert werden, damit die Flächennormalen in die korrekte Richtung zeigen.

3.3.2. Abstandsmaße

Für die weitere Betrachtung ist eine Erklärung der verwendeten Abstandsmaße relevant. Solange nichts anderes erwähnt wird, soll mit „Abstand“ der euklidische Abstand bezeichnet werden. Der euklidische Abstand zwischen zwei Punkten $\vec{x} = (x_1, x_2, \dots, x_n)$ und $\vec{y} = (y_1, y_2, \dots, y_n)$ im n-dimensionalen Raum ist wie folgt definiert:

$$D_e(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2} \quad (3.3)$$

Der Manhattenabstand zwischen zwei solchen Punkten im n-dimensionalen Raum ist folgendermaßen definiert:

$$D_m(\vec{x}, \vec{y}) = \sum_{i=1}^n |x_i - y_i| \quad (3.4)$$

3.3.3. Zufallsgenerator

Damit Teile des Dungeons zufällig erzeugt werden können und somit eine große Vielfalt von Form und Aufbau leicht zugänglich ist, müssen im Generatorprogramm Zufallszahlen erzeugt werden. Wichtig für diese ist eine optionale deterministische Generierungsmethode, damit jeder Dungeon aus wenigen Daten komplett erstellt werden kann.

Als Methode wird der *lineare Kongruenzgenerator* verwendet, der 1949 von D.H. Lehmer eingeführt wurde und unter anderem in [Knu98, S.10 ff.] beschrieben ist. Die Methode verwendet vier Parameter: den Modulus m , den Multiplikator a , den Inkrementor c und den Startwert x_0 , welcher auch als Seed bezeichnet wird. Die Sequenz der Zufallszahlen x_i berechnet sich daraus wie in Formel 3.5 angegeben, wobei mit x_0 begonnen wird. Durch feste Werte für a , c , m und x_0 ist die Sequenz genau definiert, also deterministisch. Durch zufällige Festlegung der Werte, i.A. des Startwerts x_0 , lassen sich variable Sequenzen erzeugen.

$$x_{n+1} = (a \cdot x_n + c) \bmod m \quad (3.5)$$

Sinnvoll ist es, die Parameter so zu wählen, dass die Periode der erzeugten Sequenz möglichst groß ist. Diese gibt an, nach wie vielen Zahlen sich die gesamte Sequenz wiederholt. Nach dem Satz von Knuth [Knu98, S.17] wird die maximale Periodenlänge m erreicht, wenn folgende drei Bedingungen zutreffen:

3. Aufbau der Generatorpipeline

1. c ist teilerfremd zu m
2. jeder Primfaktor vom m ist Teiler von $a - 1$
3. wenn m ein Vielfaches von 4 ist, dann auch $a - 1$

Beispielwerte hierfür sind die Parameter $a = 1103515245$, $c = 12345$ und $m = 32768$, die somit eine Periodenlänge von 32768 erzeugen.

Anzumerken ist, dass es sich bei den generierten Zahlen nicht um echte Zufallszahlen handelt, sondern um Pseudozufallszahlen, da sie nach festen Vorschriften berechnet werden. Echte Zufallszahlen sind mit heutiger Rechentechnik nicht erzeugbar.

4. Erstellung der Höhle im Voxelraum

Das Grundgedanke der Höhlenerstellung beruht darauf, dass eine (zufällige) fraktalähnliche Grundstruktur, chaotisch, aber trotzdem mit einer inhärenten Ordnung, durch das Zeichnen von Strichen mit einem bestimmten Radius so überlagert wird, dass eine Höhlenstruktur entsteht. Zuerst erfolgt hierzu die Erstellung einer Höhlenstruktur-Beschreibung mittels eines L-Systems. Diese Höhle wird dann im Voxelraum gezeichnet und danach nachbearbeitet, um sie realistischer zu gestalten.

4.1. Generierung von Lindenmayer-System-Fraktalen

Der Begriff *Fraktal* wurde von Benoît B. Mandelbrot entscheidend geprägt. Besonders hervorzuheben ist hier sein Werk „Die fraktale Geometrie der Natur“ [Man91], auf welchem die folgende Erklärung des Fraktalbegriffes basiert.

Um Fraktale zu definieren, benötigt man den Begriff der *Dimension*. Die *topologische Dimension* stellt stets eine ganze Zahl dar. Für Kurven ist sie gleich 1, für Flächen gleich 2 und für Körper gleich 3. Es gibt verschiedene allgemeinere Charakterisierungen der topologischen Dimension, eine gebräuchliche davon ist die Lebesgue'sche Überdeckungsdimension.

Nach [Man91, S.49] kann für selbstähnliche Figuren die *Hausdorff-Besicovitch-Dimension* mittels der *Ähnlichkeitsdimension* abgeschätzt werden. Diese sind für solche Fälle oftmals identisch, wobei die Ähnlichkeitsdimension einfacher zu ermitteln ist. Die Hausdorff-Besicovitch-Dimension sowie die Ähnlichkeitsdimension sind grundsätzlich mindestens so groß wie die topologische Dimension.

Kann eine Figur in N Teile aufgespalten werden und diese Teile mittels einer Ähnlichkeitsabbildung mit dem Quotienten r wieder dem Original entsprechen, so hat diese Figur die Ähnlichkeitsdimension D , wobei gilt:

$$D = -\frac{\log N}{\log r} \quad (4.1)$$

Ein Beispiel für eine solche Figur ist die Koch-Kurve, deren Konstruktion in Abbildung 4.1 dargestellt wird. Die Erstellung beginnt mit dem *Initiator*. Nun wird in jeder Iteration jede Strecke der vorhergehenden Iteration durch den *Generator* ersetzt. Der Generator besteht im Fall der Koch-Kurve aus vier Strecken, die gegenüber der zu ersetzenen Strecke um ein Drittel verkleinert wurden, und hat die Form der ersten Iteration aus Abbildung 4.1. Die eigentliche Koch-Kurve entsteht nach unendlich vielen solchen Iterationen. Als Ähnlichkeitsdimension der Kurve ergibt sich $D = -\log 4 / \log \frac{1}{3} \approx 1,26$. Es gibt eine Vielzahl weiterer solcher mittels Generatoren konstruierten selbstähnlichen Kurven.

Mit den Dimensionsbegriffen kann nun der Begriff des Fraktals definiert werden. Die exakte Definition nach [Man91, S.27] ist in Definition 1 wiedergegeben. Die Koch-Kurve ist somit ein Fraktal.

Definition 1 (Fraktal) Ein Fraktal ist eine Menge, deren Hausdorff-Besicovitch-Dimension echt die topologische Dimension übersteigt.

4. Erstellung der Höhle im Voxelraum

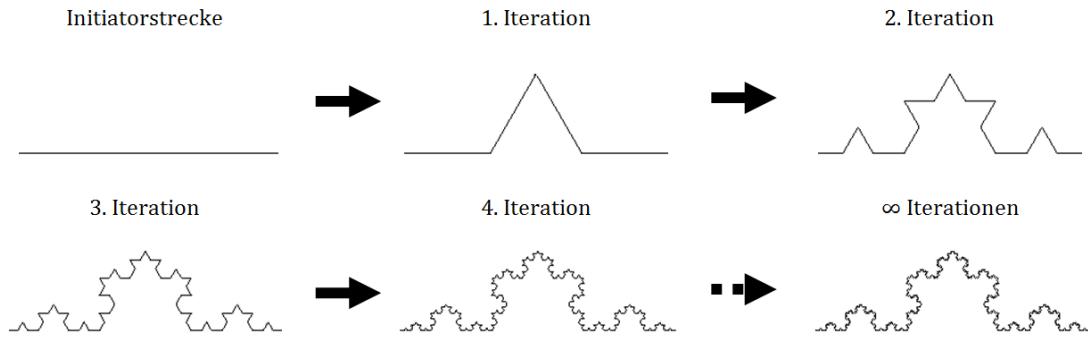


Abbildung 4.1.: Konstruktion der Koch-Kurve [Jab11]

4.1.1. Lindenmayer-Systeme

Lindenmayer-Systeme, oder auch *L-Systeme* genannt, wurden vom Biologen Aristid Lindenmayer entwickelt. Die Erläuterung der L-Systeme folgt in dieser Ausarbeitung im Wesentlichen den Ausführungen von [PL90], wo diese zur Generierung von Pflanzenstrukturen angewendet werden.

Ein L-System stellt eine Grammatik dar, bei der im Gegensatz zu formalen Chomsky-Grammatiken, die Ersetzungsregeln (Produktionen) gleichzeitig auf ein ganzes Wort angewendet werden, statt sequentiell. Ein kontextfreies L-System (OL-System) wird nach [PL90, S.4] durch die Definitionen 2 und 3 beschrieben. In dieser Arbeit finden ausschließlich deterministische kontextfreie L-Systeme Anwendung. Die Ableitung der Länge n von ω sei auch als n -te Iteration des L-Systems bezeichnet.

Definition 2 (Kontextfreies L-System) Sei V ein Alphabet, V^* die Menge aller Wörter über V und V^+ die Menge aller nichtleeren Wörter über V . Ein OL-System ist ein geordnetes Tripel $G = \langle V, \omega, P \rangle$, wobei V das Alphabet des Systems sei, $\omega \in V^+$ ein nichtleeres Wort, welches Axiom genannt wird, und $P \subset V \times V^*$ eine endliche Menge an Produktionen. Eine Produktion $(a, x) \in P$ wird geschrieben als $a \rightarrow x$. Das Zeichen a heißt Predecessor und das Wort $x \in V^*$ Successor dieser Produktion. Für jedes Zeichnen $a \in V$ soll mindestens ein Wort $x \in V^*$ existieren, mit $a \rightarrow x$. Wenn für einen Predecessor $a \in V$ keine Produktion angegeben ist, so wird angenommen, dass die Produktion $a \rightarrow a$ existiert. Ein OL-System ist deterministisch gdw. für jedes $a \in V$ genau ein Wort $x \in V^*$ mit $a \rightarrow x$ existiert.

Definition 3 (Ableitungen von ω) Sei $\mu = a_1 \cdots a_m$ ein beliebiges Wort über V . Das Wort $\nu = x_1 \cdots x_m \in V^*$ ist direkt abgeleitet von μ , notiert als $\mu \Rightarrow \nu$, gdw. $\forall i = 1, \dots, m : \exists a_i \rightarrow x_i$. Ein Wort ν wird durch G in einer Ableitung der Länge n generiert, wenn es eine Entwicklungssequenz von Wörtern $\mu_0, \mu_1, \dots, \mu_n$ gibt, so dass $\mu_0 = \omega$, $\mu_n = \nu$ und $\mu_0 \Rightarrow \mu_1 \Rightarrow \dots \Rightarrow \mu_n$.

Die Symbole der L-Systeme können nach [PL90, S.6] als Turtle-Grafik-Zeichenanweisungen interpretiert werden. Dabei stellen die Zeichen des Alphabets die einzelne Anweisungen für die „Turtle“ dar. Eine Folge dieser Anweisungen, notiert durch eine Ableitung von ω , wird nacheinander Zeichen für Zeichen abgearbeitet, wodurch die eigentliche Grafik entsteht. Die gebräuchlichsten Symbole und ihre Interpretationen sind beispielsweise in [PL90],

4. Erstellung der Höhle im Voxelraum

Symbol	Verwendet	Interpretation
F	F	Bewege vorwärts und zeichne Strich mit Radius r
+	+	Drehe um \overrightarrow{Oben} um α_{Gier} (nach links)
-	-	Drehe um \overrightarrow{Oben} um $-\alpha_{Gier}$ (nach rechts)
^	o	Drehe um \overrightarrow{Links} um $-\alpha_{Nick}$ (nach oben)
&	u	Drehe um \overrightarrow{Links} um α_{Nick} (nach unten)
\	g	Drehe um \overrightarrow{Vorn} um $-\alpha_{Roll}$ (gegen UZS, nach vorn blickend)
/	z	Drehe um \overrightarrow{Vorn} um α_{Roll} (im UZS, nach vorn blickend)
		Drehe um \overrightarrow{Oben} um 180°
\$	\$	Drehe die Turtle so, dass \overrightarrow{Oben} gleich $(0, 1, 0)$ ist (In [PL90]: so rotieren, dass \overrightarrow{Links} senkrecht zu $(0, 1, 0)$)
[[Speichere ζ_{lok} , \vec{P} und r auf dem Stack
]]	Lade ζ_{lok} , \vec{P} und r vom Stack
!	!	Reduziere Radius r

Tabelle 4.1.: Symbole für Turtle-Grafik-Zeichenanweisungen
v.l.n.r: Symbol nach [PL90], Symbol in dieser Arbeit, Interpretation des Symbols

[RS92] und [ESK97] beschrieben. Tabelle 4.1 zeigt die in dieser Arbeit verwendeten Befehle auf, wobei jeweils das Symbol nach [PL90, S.209] angeben ist und das Symbol, welches im Generatorprogramm genutzt wird. Die Abänderung erfolgt zur Vermeidung von XML-Sonderzeichen, da die Daten u.a. als XML-Dateien gespeichert werden. Symbol '\$' wurde in seiner Interpretation modifiziert, um eine Möglichkeit zu haben, die Turtle horizontal auszurichten und so einfacher ebene, begehbar Segmente innerhalb von Höhlen schaffen zu können. Für die Tabelle seien folgende Festlegungen getroffen: die „Turtle“ habe die Eigenschaften: lokales Koordinatensystem ζ_{lok} , mit den Achsen \overrightarrow{Vorn} , \overrightarrow{Oben} , \overrightarrow{Links} und der Position \vec{P} , sowie den Radius r . Die Winkel α_{Gier} , α_{Nick} und α_{Roll} seien als zusätzliche Parameter gegeben.

Weitere Zeichen mit anderen Interpretationen sind prinzipiell beliebig möglich. Ein vielfach erwähntes, aber nicht verwendetes Symbol ist für das Vorwärtsbewegen ohne zu Zeichnen zuständig. Hierauf wurde verzichtet, damit die entstehenden Strukturen in jedem Fall zusammenhängend sind. Alle nicht interpretierbaren Symbole werden beim Zeichnen grundsätzlich ignoriert, können aber zur Schaffung von Ersetzungsregeln verwendet werden.

Das generelle Konzept der L-Systeme ist das fortlaufende iterative Ersetzen von Objektteilen mittels einer Menge von Ersetzungsregeln, ausgehend von einem Initialobjekt. Ein Beispiel dafür ist die oben beschriebene Koch-Kurve, die ebenfalls durch solche Ersetzungen generiert wird. Diese lässt sich über das L-System

$G = \langle \{F, +, -\}, F, \{F \rightarrow F + F - -F + F\} \rangle$ mit $\alpha_{Gier} = 60^\circ$ beschreiben. In jeder neuen Ableitung wird die Strichlänge auf $\frac{1}{3}$ der Strichlänge der vorherigen Ableitung gesetzt.

Bei über Generatoren erstellten Fraktalen können die Ersetzungen mittels entsprechender Produktionsregeln dargestellt werden. Mittels entsprechender L-Systeme lassen sich also Fraktale beschreiben, wobei aber nicht jedes L-System zwangsläufig ein solches definiert.

4. Erstellung der Höhle im Voxelraum

4.1.2. Fraktalgenerator

Der Fraktalgenerator ist ein Generierungsbaustein des Dungeongenerator-Programms, welcher dazu dient, zufällige L-Systeme zu erstellen sowie Turtle-Grafik-Zeichenanweisungen aus gegebenen L-Systemen zu generieren. Zum einen lassen sich Parameter, wie die Winkel α_{Gier} , α_{Nick} , α_{Roll} , zufällig bestimmen, zum anderen können ein zufälliges Axiom sowie zufällige Produktionen für gegebene Symbole erzeugt werden.

Für die Generierung einer Folge von Zeichenanweisungen werden ein L-System und die gewünschte Länge der Ableitung n benötigt. Ausgehend vom Axiom als Startstring wird nun iteriert. In jeder Iterationsstufe wird ein neuer String erzeugt, in dem über den alten String gelaufen wird und die Ersetzungen mittels der Produktionen vorgenommen werden. Diese Iterationen werden insgesamt n mal durchgeführt. Die resultierende Folge von Zeichenanweisungen ist der zuletzt erzeugte String.

Anmerkung: Diese Zeichenanweisungen erzeugen auch im Fall von fraktalbeschreibenden L-Systemen keine echten Fraktale, da die Ableitungsfolge nicht unendlich lang ist.¹ Die resultierenden Strukturen sind aber i.d.R. fraktalähnlich, ähneln also in ihrer Form echten Fraktalen.

4.2. Turtle-Grafik im Voxelraum

Der Fraktalgenerator liefert Turtle-Grafik-Zeichenanweisungen, die nun im 3D-Raum gezeichnet werden müssen. Als Grundlage für den 3D-Raum werden Voxel verwendet. *Voxel* („volume element“) bilden das dreidimensionale Äquivalent zu den zweidimensionalen Pixeln („picture element“). Der Voxelraum ist nach [FvFH97, S.549] per gleichmäßigen Gitter in kleine identische Zellen aufgeteilt, die meistens kubischer Form sind. Jede dieser Zellen stellt einen Voxel dar. Abbildung 4.2 zeigt einen aus Voxeln aufgebauten Torus.

Die in dieser Arbeit verwendeten Voxel sollen ausschließlich kubischer Natur sein. Die Schreibweise $Voxel(x, y, z)$ bezeichne die Belegung des Voxels an Position (x, y, z) mit $x, y, z \in \mathbb{N}_0$. Der Voxelraum soll den Bereich aller Voxel in $[0, X_{vmax}] \times [0, Y_{vmax}] \times [0, Z_{vmax}]$ umfassen.

Über den Manhattenabstand (siehe Formel 3.4) lassen sich verschiedene Nachbarschaftsbeziehungen zwischen Voxeln definieren. Diese sind dabei an die gebräuchlichen Begriffe 4er-Nachbarschaft und 8er-Nachbarschaft aus der Bildverarbeitung angelehnt (vgl. [BB06, S.173-174]), die sich auf den zweidimensionalen Pixelraum beziehen.

Definition 4 (6er-Nachbarschaft) Bezuglich eines Voxel an Position \vec{a} liegt ein Voxel an Position \vec{b} in 6er-Nachbarschaft, wenn gilt: $D_m(\vec{a}, \vec{b}) = 1$ und $D_e(\vec{a}, \vec{b}) < 2$.

Definition 5 (18er-Nachbarschaft) Bezuglich eines Voxel an Position \vec{a} liegt ein Voxel an Position \vec{b} in 18er-Nachbarschaft, wenn gilt: $1 \leq D_m(\vec{a}, \vec{b}) \leq 2$ und $D_e(\vec{a}, \vec{b}) < 2$.

Definition 6 (26er-Nachbarschaft) Bezuglich eines Voxel an Position \vec{a} liegt ein Voxel an Position \vec{b} in 26er-Nachbarschaft, wenn gilt: $1 \leq D_m(\vec{a}, \vec{b}) \leq 3$ und $D_e(\vec{a}, \vec{b}) < 2$.

Für das Zeichnen der Höhle soll ein nicht gesetzter Voxel (0) äquivalent zu Felsgestein sein, welches gleichzeitig der Initialwert für alle Voxel ist. Ein gesetzter Voxel (1) bezeichnet den Freiraum der Höhle. Temporär sind andere Markierungen für bestimmte Algorithmen

¹Solche unendlich langen Ableitungsfolgen lassen sich verständlicherweise nicht generieren.

4. Erstellung der Höhle im Voxelraum

möglich, die entsprechend erwähnt werden. Bei der Implementierung wird die Größe des Voxelraums auf $512 \times 512 \times 512$ Einheiten festgelegt, welches 134.217.728 Voxeln entspricht. Zur Speicherung wird ein Byte je Voxel verwendet, wodurch hierfür insgesamt 128 MiB nötig sind. Da heutzutage oftmals mehrere GiB Arbeitsspeicher vorhanden sind, ist dies problemlos umsetzbar. Der gesamte Voxelraum wird als 3D-Array abgelegt.

Alternativ lässt sich ein Bit je Voxel mit acht Voxeln in einem Byte verwenden, wodurch für gleiche Voxelraum-Größe nur noch 16 MiB Speicherplatz benötigt werden würden. Diese Speicherungsvariante kann für größere Voxelräume relevant sein, da bei Verdopplung der Kantenlänge des Voxelraums der Speicherbedarf um das Achtfache steigt. Allerdings erfolgt der Zugriff auf einzelne Voxel hier per Binäroperatoren (binäres Und, binäres Oder), was unperformanter als der direkte Zugriff ist. Da für bestimmte verwendete Algorithmen andere Markierungen außer 0 und 1 nötig sind, ist hier ebenfalls weiterer Speicher nötig.

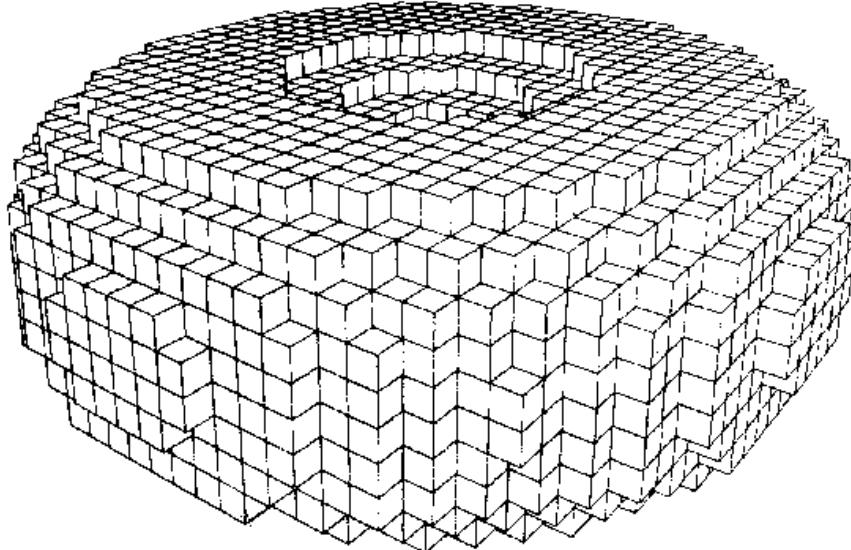


Abbildung 4.2.: Aus Voxeln aufgebauter Torus [FvFH97, S.549]

Die Turtle habe die in Kapitel 4.1.1 genannten Eigenschaften. Für das Zeichnen der Turtle-Grafik seien weiterhin als Parameter gegeben: die Winkel α_{Gier} , α_{Nick} , α_{Roll} , Radiusfaktor r_f und Radiusdekkrementor r_d . Der Stack für das Speichern der Turtle-Zustände sei initial leer.

4.2.1. Drehungen und Radiusänderungen

Für die Umsetzung der Rotationen (Symbole '+', '−', 'o', 'u', 'g', 'z') sei gefordert, dass die Vektoren des lokalen Koordinatensystems immer senkrecht aufeinander stehen und immer die Länge 1 besitzen. Dies erfordert auch bei vielen hintereinander ausgeführten Rotationen numerische Stabilität. Dazu erfolgt bei einer Drehung zuerst die Errechnung einer neuen Achse des lokalen Koordinatensystems. Die Konstruktion der zweiten sich verändernden Achse erfolgt per Kreuzprodukt. In jedem Schritt wird der entsprechende Vektor normalisiert.² Formel 4.2 gibt das Beispiel für die Drehung um Achse \overrightarrow{Oben} mit Winkel α_{Gier} an.

²Bei absolut exakter Berechnung könnte das Normalisieren weggelassen werden.

4. Erstellung der Höhle im Voxelraum

Die Achse \overrightarrow{Oben} bleibt bei dieser Drehung unverändert, die beiden anderen Achsen werden wie angegeben modifiziert.

$$\begin{aligned}\overrightarrow{Vorn_{neu}} &= \frac{\cos \alpha_{Gier} \cdot \overrightarrow{Vorn} + \sin \alpha_{Gier} \cdot \overrightarrow{Links}}{\left| \cos \alpha_{Gier} \cdot \overrightarrow{Vorn} + \sin \alpha_{Gier} \cdot \overrightarrow{Links} \right|} \\ \overrightarrow{Links_{neu}} &= \frac{\overrightarrow{Vorn_{neu}} \times \overrightarrow{Oben}}{\left| \overrightarrow{Vorn_{neu}} \times \overrightarrow{Oben} \right|}\end{aligned}\quad (4.2)$$

Bei der Ausrichtung von \overrightarrow{Oben} (Symbol '\$') nach $(0, 1, 0)$ erfolgt zuerst die Neubelegung von \overrightarrow{Oben} . Um die nachfolgende Berechnung numerisch stabil zu gestalten, wird nun bestimmt, welcher der beiden Vektoren \overrightarrow{Vorn} oder \overrightarrow{Links} den höchsten Betrag des Skalarprodukts mit \overrightarrow{Oben} aufweist, also möglichst weit senkrecht auf \overrightarrow{Oben} steht. Der entsprechende Vektor sei Vektor \vec{a} , der andere Vektor \vec{b} . Vektor \vec{a} wird nun für die Berechnung des neuen Werts für \vec{b} verwendet, die durch die Bildung des Kreuzprodukts mit \overrightarrow{Oben} erfolgt. Danach wird der neue Wert für \vec{a} ebenfalls per Kreuzprodukt ermittelt.

Für die Drehung um 180° (Symbol '|') brauchen nur \overrightarrow{Vorn} und \overrightarrow{Links} mit -1 multipliziert zu werden.

Die Reduzierung des Radius (Symbol '!') erfolgt nach Formel 4.3. Ziel ist es eine prozentuale (per r_f), als auch eine absolute (per r_d) Radiusreduzierung zu ermöglichen. Der Radius kann per Definition nicht kleiner als 0 werden. In der Implementierung des Generator-Programms ist ein minimaler Radius von 1 vorgesehen. Der Wert des Radius wird als Gleitkommazahl gespeichert und beim Zeichnen von Strichen auf eine ganze Zahl abgerundet.

$$r_{neu} = r_f \cdot r - r_d \quad (4.3)$$

4.2.2. Zeichnen von Strichen mit gegebenem Radius

Das Zeichnen der Striche erfordert die Aktualisierung der Position. Die neue Position berechnet sich wie in Formel 4.4 angegeben. Die Strichlänge sei L .

$$\overrightarrow{P_{neu}} = \vec{P} + L \cdot \overrightarrow{Vorn} \quad (4.4)$$

Weiterhin müssen die durch den Strich abgedeckten Voxel auf 1 gesetzt werden. Der erste Ansatz bestand darin, die gesamte Turtle-Grafik zuerst mittels des 3D-Bresenham-Algorithmus nur mit Strichdicke 1 zu zeichnen und dieses Skelett dann zur finalen Ausdehnung zu expandieren. Dazu wurde eine Prioritätswarteschlange als Datenstruktur für die expandierenden Voxel verwendet. Die Abarbeitungspriorität der Voxel entsprach (*gewünschter Radius - aktueller Abstand vom Skelett*). Problematisch war die Laufzeitkomplexität von $\mathcal{O}(n \log n)$, mit n als Anzahl zu zeichnender Voxel, und dem hohen Speicherverbrauch pro Voxel. Diese Methode hatte bereits für einfache Turtle-Grafiken mit wenigen Strichen eine Laufzeit von mehreren Minuten. Im Vergleich dazu schaffte es der finale Bresenham-Zylinder-Kugel-Algorithmus diese Grafiken in weniger als einer Sekunde zu zeichnen.

4. Erstellung der Höhle im Voxelraum

Exkurs: Bresenham-Linien-Algorithmus in 3D

Der *Bresenham-Linien-Algorithmus*, auch kurz Bresenham-Algorithmus oder Bresenham genannt, ist ein Algorithmus zum Zeichnen von Linien in zweidimensionalen Rastergrafiken, welcher 1965 von J. E. Bresenham in [Bre65] vorgestellt wurde. Der Algorithmus ist sehr schnell, da er prinzipiell nur mit Additionen und Entscheidungen auskommt und ausschließlich ganze Zahlen verwendet. Damit stellt er quasi den Standardalgorithmus zum Zeichnen von Linien dar.

Die Rasterung einer Linie von (x_1, y_1) nach (x_2, y_2) , mit $dx \geq dy$ (wobei $dx = x_2 - x_1$, $dy = y_2 - y_1$), nach dem Bresenham-Algorithmus wird in Abbildung 4.3 dargestellt. Zusätzliche Voraussetzungen sind $x_1 \leq x_2$ und $y_1 \leq y_2$. Für alle anderen Fälle wird der Algorithmus entsprechend angepasst. Begonnen wird in (x_1, y_1) . In jedem Zeichenschritt wird die Koordinate x um 1 erhöht. Für die Koordinate y gibt es zwei Möglichkeiten: entweder wird sie so belassen oder es findet ebenfalls eine Erhöhung um 1 statt. Die Idee besteht darin, dies anhand des Fehlers zu entscheiden, der entstehen würde, und dabei die Variante zu wählen, bei der der gezeichnete Pixel näher an der echten Linie liegt. Der Bresenham-Algorithmus führt hierzu eine Fehlervariable ein, anhand der entschieden wird. Diese Variable wird zusätzlich zur Positionsänderung je nach Entscheidung unterschiedlich aktualisiert.

Diese Fehlervariable sei e_y . Falls $e_y \geq 0$ wird die Y-Koordinate um 1 erhöht, ansonsten so gelassen. In [Bre65] wird gezeigt, dass wenn anfangs $e_y = 2 \cdot dy - dx$ gesetzt wird, die nachfolgende Aktualisierung dem Prinzip aus Formel 4.5 genügen muss.

$$e_y^{neu} = \begin{cases} e_y + 2 \cdot dy - 2 \cdot dx & \text{falls } e_y \geq 0 \\ e_y + 2 \cdot dy & \text{sonst} \end{cases} \quad (4.5)$$

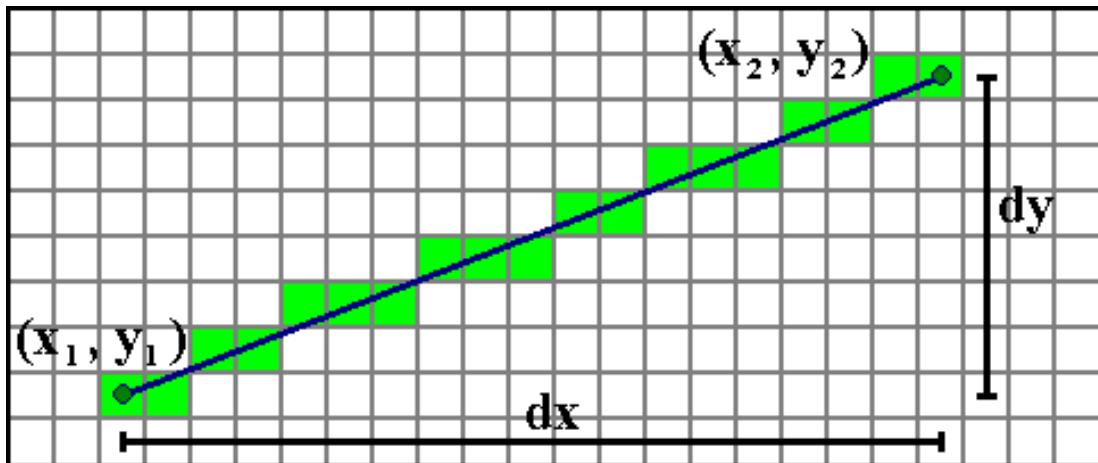


Abbildung 4.3.: Rasterung einer Linie nach dem Bresenham-Verfahren [Wik10]

Der Bresenham-Algorithmus lässt sich auf dreidimensionale Rastergrafiken, also Voxelgrafiken, erweitern. In [LC02] wird diese 3D-Variante beschrieben. Algorithmus 1 stellt eine leicht modifizierte Version dar, mit beliebiger Reihenfolge der Start- und Endpunkte. Dennoch muss gelten, dass $|dx| \geq |dy|$ und $|dx| \geq |dz|$, also ist x die Primärkoordinate. Für die Anwendung auf alle Fälle wird eine Funktion vorgeschaltet, die den passenden Fall mit x , y oder z als Primärkoordinate bestimmt und daraufhin den entsprechend angepassten

4. Erstellung der Höhle im Voxelraum

Algorithmus aufruft. Die Anpassung erfolgt durch den Rollentausch der Koordinaten x , y und z .

Der Unterschied zum originalen Bresenham-Algorithmus besteht darin, dass in jedem Zeichenschritt zusätzlich die Z-Koordinate modifiziert werden kann. Hierzu gibt es die Fehlervariable e_z , anhand der entschieden wird. Deren Aktualisierung findet äquivalent zu der von e_y statt.

Algorithmus 1 Bresenham-Linien-Algorithmus in 3D (C++ Listing)

Eingabe: /*Endpunkte (mit X als längster Achse)*/
 int xs,int ys,int zs,int xe,int ye,int ze
Ausgabe: gezeichnete Linie

```

1: /* Richtungen festlegen */
2: int xinc = 1; if(xe<xs) xinc = -1;
3: int yinc = 1; if(ye<ys) yinc = -1;
4: int zinc = 1; if(ze<zs) zinc = -1;
5: /* Berechne die delta-Werte */
6: int dx=abs(xe-xs); dy=abs(ye-ys); dz=abs(ze-zs);
7: /* Berechne die Initialwerte der Entscheidungsparameter */
8: int ey=2*dy-dx; ez=2*dz-dx;
9: /* Berechne Konstanten */
10: int twoDy=2*dy; twoDyDx=2*(dy-dx);
11: int twoDz=2*dz; twoDzDx=2*(dz-dx);
12: /* Setze Startposition */
13: int x=xs; y=ys; z=zs;
14: /* Annahme: dx>=dy, dx>=dz */
15: for (i=0; i<=dx; i++)
16: {
17:     output(x,y,z); /* Zeichne die aktuelle Position*/
18:     x+=xinc; /* Start der Berechnung der nächsten Koordinaten */
19:     if (ey>=0) {y+=yinc; ey+=twoDyDx;}
20:     else ey+=twoDy;
21:     if (ez>=0) {z+=zinc; ez+=twoDzDx;}
22:     else ez+=twoDz; /* Ende der Berechnung der nächsten Koordinaten */
23: }
```

Kugel und Kreiszylinder

Die finale Methode zum Zeichnen von Strichen mit gewünschtem Radius r im Voxelraum basiert auf dem Zeichnen von zwei grafischen Primitiven. Zuerst erfolgt das Zeichnen eines geraden Kreiszylinders mit Radius r . Danach wird eine abschließende Kugel mit gleichem Radius am Zylinderende gezeichnet, damit bei Drehungen nachfolgender Striche keine Spalten entstehen und eine weichere Überblendung bei Radiusreduzierungen erreicht wird.

Der Algorithmus für die Kugel ist vergleichsweise einfach umzusetzen. Es erfolgt das Ablauen des Würfels mit dem Mittelpunkt gleich dem Kugelmittelpunkt und der Seitenlänge gleich dem Kugeldurchmesser mittels einer dreifach geschachtelte Schleife, wobei jede Schachtelung für eine Dimension des Würfels zuständig ist. Im Inneren der Schleife

4. Erstellung der Höhle im Voxelraum

wird für jeden Voxel ein Test durchgeführt, ob der Abstand vom Mittelpunkt der Kugel höchstens so groß wie der Radius der Kugel ist. Wenn dieser Test positiv verläuft, wird der Voxel gezeichnet, also auf 1 gesetzt. Zur Optimierung werden bei allen Berechnungen und Tests quadratische Abstände verwendet und Rechenoperationen möglichst weit aus den Schleifenschachtelungen herausgezogen.

Das lokale Koordinatensystem ist durch die Turtle bekannt, die Vektoren sind normiert. Die Idee des Kreiszylinder-Algorithmus ist es, diese Vektoren zu nutzen, um den Zylinder aufzuspannen. \overrightarrow{Oben} und \overrightarrow{Links} multipliziert mit dem Zylindradius spannen die Grundfläche auf und die Zylinderachse verläuft von der Startposition zur Endposition des Striches, die beide ebenfalls bekannt sind.

Das Aufspannen der Vektoren geschieht mittels des 3D-Bresenham-Algorithmus. Für jede der drei Achsen wird ein Bresenham verwendet, der in einen Buffer zeichnet, statt direkt in den Voxelraum. Danach wird eine dreifach geschachtelte Schleife durchlaufen, die jeweils über alle Buffer-Elemente läuft, die Koordinaten zusammenaddiert und ggf. den entsprechenden Voxel zeichnet, falls er wirklich im Zylinder liegt. Das Verfahren ist in Algorithmus 2 dargestellt.

Durch die Spiegelung eines Viertels der Zylindergrundfläche lässt sich die Zahl nötiger Tests reduzieren. Allerdings wird die `ZeichneVoxel()`-Routine an Positionen mit \vec{P}_1 oder \vec{P}_2 gleich dem Nullvektor doppelt und mit \vec{P}_1 und \vec{P}_2 gleich dem Nullvektor sogar vierfach ausgeführt. Dies ist aber bei größeren Zylindradien vernachlässigbar, da der Anteil dieser Fälle im Verhältnis zur Gesamtvoxelzahl abnimmt. Ein Auffangen der Fälle vorm Zeichnen der Voxel ist i.d.R. nicht empfehlenswert, da bedingte Anweisungen vergleichsweise teure Operationen darstellen.

Algorithmus 2 Gerader Kreiszylinder im Voxelraum

Eingabe: Radius r , Startpunkt \vec{P}_s , Endpunkt \vec{P}_e , Lokales Koordinatensystem

Ausgabe: gezeichneter Zylinder

```

1:  $Buffer1 \leftarrow Bresenham3D(\vec{0}, r \cdot \overrightarrow{Oben})$                                  $\triangleright \vec{0}$  bezeichnet Nullvektor
2:  $Buffer2 \leftarrow Bresenham3D(\vec{0}, r \cdot \overrightarrow{Links})$ 
3:  $Buffer3 \leftarrow Bresenham3D(\vec{P}_s, \vec{P}_e)$ 
4: for all  $\vec{P}_1 \in Buffer1$  do
5:   for all  $\vec{P}_2 \in Buffer2$  do
6:     if  $(\vec{P}_1 + \vec{P}_2)^2 \leq r^2$  then
7:       for all  $\vec{P}_3 \in Buffer3$  do
8:          $ZeichneVoxel(\vec{P}_3 + \vec{P}_1 + \vec{P}_2)$ 
9:          $ZeichneVoxel(\vec{P}_3 - \vec{P}_1 - \vec{P}_2)$                                  $\triangleright$  Spiegelung
10:         $ZeichneVoxel(\vec{P}_3 + \vec{P}_1 - \vec{P}_2)$                                  $\triangleright$  Spiegelung
11:         $ZeichneVoxel(\vec{P}_3 - \vec{P}_1 + \vec{P}_2)$                                  $\triangleright$  Spiegelung
12:       end for
13:     end if
14:   end for
15: end for

```

Problematisch ist, dass es beim Anwenden von Algorithmus 2 unter der Verwendung einer `ZeichneVoxel()`-Routine, die genau einen Voxel an gegebener Position zeichnet, zur Lückenbildung im Zylinder kommen kann. Die Ursache liegt darin, dass der 3D-Bresenham-

4. Erstellung der Höhle im Voxelraum

Algorithmus beim Zeichnen von Linien die 26er-Nachbarschaftsbeziehung verwendet, äquivalent zum 2D-Bresenham, der die 8er-Nachbarschaft nutzt. Abbildung 4.4 zeigt die Entstehung von Lücken beim Zeichnen einer Fläche. Äquivalent kommt es bei der Kombination der 3D-Bresenhams zu einem volumetrischen Gebilde potentiell zu Lücken. Jede dieser Lücken besitzt aber grundsätzlich mindestens einen gesetzten Voxel in 6er-Nachbarschaft.

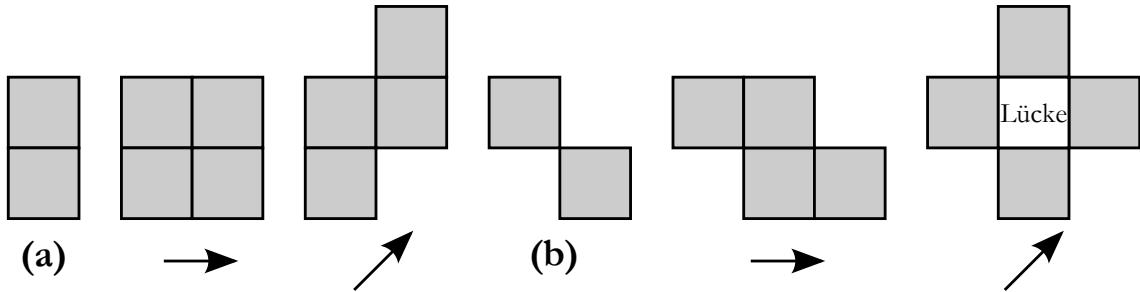


Abbildung 4.4.: *Entstehung von Lücken beim Zeichnen des Zylinders*: betrachtet wird 2D-Fall: Fig. (a) und (b) zeigen Teilausschnitte einer Linie. Werden deren Punkte als Startpunkte neuer Linien verwendet, so können je nach Linienrichtung Lücken entstehen.

Dadurch ergibt sich als mögliche Lösung, dass Voxelblöcke statt einzelner Voxel gezeichnet werden. Diese Blöcke sollten annähernd kugelförmig bezüglich eines Mittelpunktsvoxels sein, damit die Symmetrie des Zylinders gewahrt bleibt. Wenn man einen 6er-Nachbarschafts-Voxelblock zeichnet (Bezugsvoxel und alle Voxel in 6er-Nachbarschaft), so werden die Lücken innerhalb des Zylinders geschlossen. Allerdings können, wie in Abbildung 4.5 zu sehen ist, noch Scharten am Rand des Zylinders auftreten. Diese können durch Verwendung von 18er-Nachbarschafts-Voxelblöcken eliminiert werden. Durch das Verwenden dieser Voxelblöcke muss der Parameter r beim Aufruf von Algorithmus 2 um 1 verringert werden, um den korrekten resultierenden Radius zu erhalten.

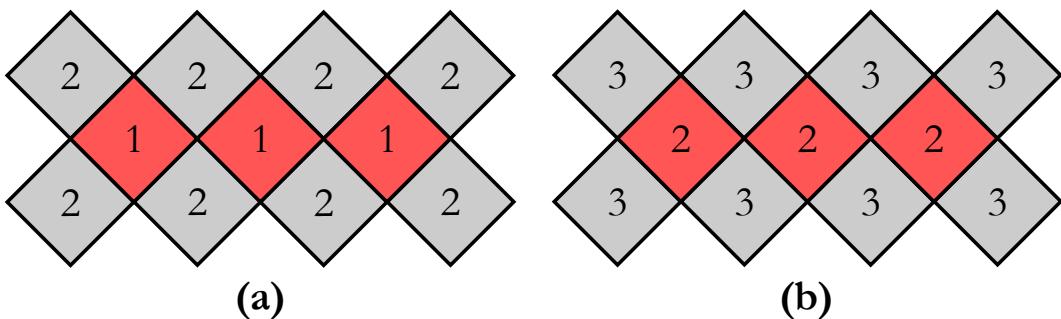


Abbildung 4.5.: *Scharten am Rand des Zylinders*: betrachtet wird Zylinderrand, die Zahlenwerte geben die relative Höhe an: Sind initial Lücken am Rand (Fig. (a)), rote Quadrate), so werden diese nicht per 6er-Nachbarschafts-Voxelblock geschlossen (Fig. (b)).

Eine Verbesserungsidee stellt das Schließen der Lücken schon in der Grundfläche dar, also vor dem eigentlichen Zeichnen des Zylinders. Dadurch können einzelne Voxel statt Blöcken gezeichnet werden, der Zylinder ist runder und damit exakter. Diese Methode ist je nach Aufwand der Modifikation möglicherweise langsamer, besonders bei kurzen Zylindern, bei

4. Erstellung der Höhle im Voxelraum

denen die Berechnung der Grundfläche einen relativ hohen Anteil am Gesamtaufwand hat. Zudem wären mehr Elemente in den Buffern, die durchlaufen werden müssten.

Anmerkung: Algorithmus 2 ist nicht auf gerade Kreiszylinder beschränkt. Wenn $(\vec{P}_e - \vec{P}_s)$ nicht senkrecht auf der Ebene, die durch \overrightarrow{Oben} und \overrightarrow{Links} aufgespannt wird, steht, so wird ein schiefer Zylinder gezeichnet. Durch Verwendung von Vektoren \overrightarrow{Oben} und \overrightarrow{Links} mit einer Länge ungleich 1, lassen sich Zylinder mit elliptischer Grundfläche zeichnen. Der Test in Zeile 6 des Algorithmus muss dabei wie in Formel 4.6 angepasst werden, wobei r nicht mehr für den Radius, sondern für einen Skalierungsfaktor steht.

$$\left(\frac{\vec{P}_1}{|\overrightarrow{Oben}|} + \frac{\vec{P}_2}{|\overrightarrow{Links}|} \right)^2 \leq r^2 \quad (4.6)$$

4.2.3. Abarbeitung der Zeichenanweisungen

Das Zeichen der Gesamtstruktur erfolgt in zwei Schritten. In beiden Schritten sind die Vektoren des lokalen Koordinatensystems der Turtle initial $\overrightarrow{Vorn} = (1, 0, 0)$, $\overrightarrow{Oben} = (0, 1, 0)$ und $\overrightarrow{Links} = (0, 0, 1)$.

Im ersten Schritt werden die Anweisungen zwar vollständig abgearbeitet, aber statt Striche wirklich zu zeichnen, wird nur die Ausdehnung der Gesamtstruktur bestimmt, um sie danach im vordefinierten Zeichenbereich, also dem Voxelraum, darstellen zu können.

Hierzu werden vor Abarbeitung der Anweisungen die Strichlänge $L = 1$ und die Startposition $\vec{P}_0 = (0, 0, 0)$ gesetzt. Eine Bounding Box wird erstellt, die anfangs nur die Startposition beinhaltet. Nach jeder Positionsänderung wird die Bounding Box aktualisiert, so dass diese die Gesamtstruktur umschließt. Am Ende von Schritt 1 erfolgt die Berechnung der neuen Daten. Die Startposition wird so gesetzt, dass der Mittelpunkt der Struktur im Mittelpunkt des Voxelraums liegt. Die Strichlänge wird so angepasst, dass die gesamte Struktur in den vordefinierten Voxelraum passt. Dies geschieht unter Berücksichtigung des maximalen Strichradius, der dem Startradius entspricht. Der Startradius selbst wird nicht modifiziert.

Im zweiten Schritt wird tatsächlich gezeichnet. Zuerst wird eine Kugel mit dem Startradius an der Startposition \vec{P}_0 zum Abschließen des ersten Striches gesetzt. Dann erfolgt die schrittweise Abarbeitung der Zeichenanweisungen unter Verwendung der im ersten Schritt bestimmten Werte für L und \vec{P}_0 .

Zur Optimierung werden direkt aufeinander folgende Striche zu einem Strich zusammengefasst. Dazu werden Folgen von 'F' und '[' kombiniert. Bei Symbol 'F' wird die aktuelle Position errechnet, bei Symbol '[' wird der aktuelle Zustand gespeichert. Der Strich wird erst gezeichnet, wenn er die komplette Länge erreicht hat. Dadurch werden Verbindungs kugeln zwischen den einzelnen Strichen und das Berechnen der Grundfläche für weitere Zylinder eingespart.

Für die Position der Turtle werden Gleitkommazahlen benutzt. Bei größeren Strukturen kann die Strichlänge kleine Werte annehmen, wodurch sich die Position nicht zwangsläufig ändert, wenn eine Zeichenanweisung 'F' vorliegt. Deswegen werden Striche nur dann gezeichnet, wenn, in Voxelkoordinaten umgerechnet, die neue Position ungleich der alten Position ist. Falls sich die neue Position von der alten Position in jeder Dimension um

4. Erstellung der Höhle im Voxelraum

höchstens einen Voxel unterscheidet, reicht es, nur eine Kugel zu zeichnen, statt Zylinder und Kugel.

Den Hauptanteil an der Laufzeitkomplexität des Verfahrens trägt im Normalfall der Zeichenalgorithmus für Striche, welcher auf der Laufzeitkomplexität für das Zeichnen von Kugel und Zylinder beruht. Er ist linear in der Anzahl der gezeichneten Voxel, wobei sich aber Striche untereinander überlagern können und so Voxel mehrfach gezeichnet werden. Mit einer Strichlänge von L und einem Startradius von r ergibt sich eine Laufzeitkomplexität pro Strich von $\mathcal{O}(L \cdot r^2)$ für den Zylinder plus $\mathcal{O}(r^3)$ für die Kugel, also $\mathcal{O}(L \cdot r^2 + r^3)$ insgesamt.

Mit n Strichen und m weiteren Anweisungen ergibt sich als Gesamlaufzeitkomplexität $\mathcal{O}(m + n \cdot L \cdot r^2 + n \cdot r^3)$.

4.3. Nachbearbeitung

Die nun gezeichnete Voxelgrafik wird nachbearbeitet, um die Struktur höhlenartiger und realistischer zu gestalten. Zu den Nachbearbeitungsschritten gehört zum einen die Erosion und zum anderen die Filterung schwiegender Fragmente. Für diese Schritte sind die Begriffe *Randvoxel* und *Randvoxel im erweiterten Sinn* relevant. Ein *Randvoxel* ist ein Voxel mit Belegung 0 mit mindestens einem Nachbarvoxel in 6er-Nachbarschaft, der die Belegung 1 besitzt. Ein *Randvoxel im erweiterten Sinn* ist ein Voxel mit Belegung 0 mit mindestens einem Nachbarvoxel in 18er-Nachbarschaft, der die Belegung 1 besitzt.

4.3.1. Erosion

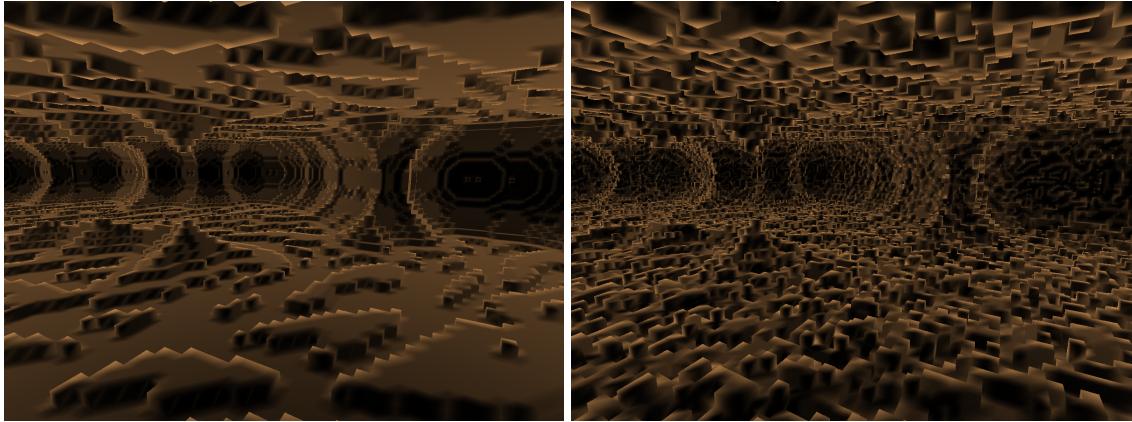


Abbildung 4.6.: *Erosion im Voxelraum*: v.l.n.r. (a) ohne Erosion, (b) mit Erosion (ein Erosionschritt mit $p_e = 0,5$)

Die Erosion bewirkt Zerklüftung an den Rändern der Grafik und ist mehrfach anwendbar. Die Auswirkungen sind in Abbildung 4.6 zu sehen. Erodiert werden grundsätzlich nur *Randvoxel* der Struktur.

Bei der Durchführung des Verfahrens wird jeder Randvoxel mit einer Wahrscheinlichkeit, die Erosionswahrscheinlichkeit p_e genannt wird, auf 1 gesetzt. Im ersten Schritt wird mittels einer Schleife über den kompletten Voxelraum gelaufen und jeder Randvoxel mit

4. Erstellung der Höhle im Voxelraum

der Wahrscheinlichkeit p_e mit 2 markiert. Im zweiten Schritt werden alle Voxel des Voxelraums mit Markierung 2 auf den Wert 1 gesetzt, also erodiert. Durch Aufteilung in zwei Schritte werden Konflikte vermieden, da bei nur einem Schritt die Nachbarn neu erodierter Voxel sofort wieder erodiert werden könnten. Dadurch könnte die Erosion bei schon einer Anwendung des Erosionsverfahrens prinzipiell beliebig weit voranschreiten, statt nur einen Voxel tief.

Der Erosionsprozess hat eine Laufzeitkomplexität von $\mathcal{O}(n)$, mit n als Anzahl aller Voxel im Voxelraum.

4.3.2. Filterung schwebender Fragmente

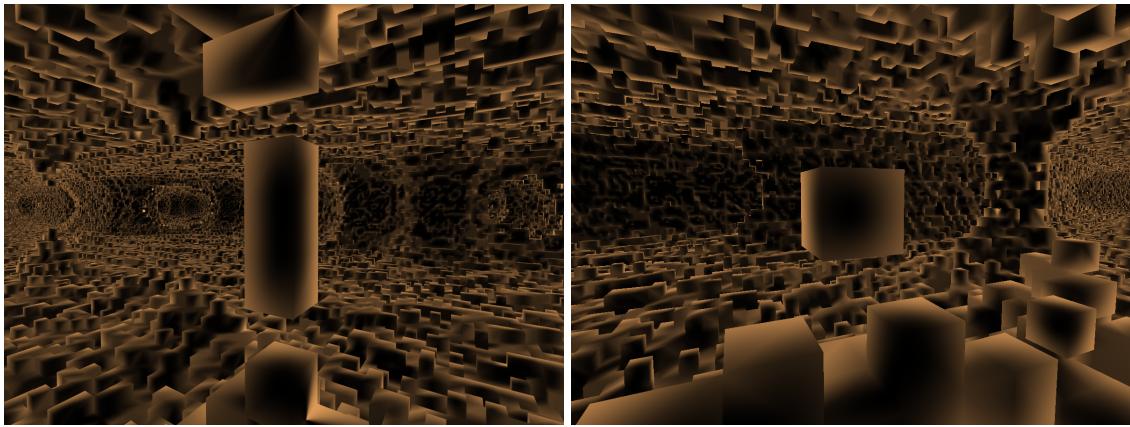


Abbildung 4.7.: Schwebende Fragmente im Voxelraum

Schwebende Fragmente entstehen bei der Umschließung von Gebieten durch gezeichnete Striche, bei der Erosion oder durch das Hineinschneiden von Gangöffnungen in die Voxelstruktur (siehe Kapitel 7.3). Abbildung 4.7 zeigt solche Fragmente. Sie sind i.d.R. für Höhlen unrealistisch, da sie dem Wirken der Gravitation widersprechen. Durch die Filterung werden diese Fragmente entfernt.

Es sei im Folgenden angenommen, dass jeder Voxel am Rand des Voxelraums die Belegung 0 habe. Um ein schwebendes Fragment zu charakterisieren, wird Definition 7 benötigt. Bei schwebenden Fragmenten handelt es sich um Zusammenhangskomponenten, die nicht der äußeren Zusammenhangskomponente entsprechen.

Definition 7 (Zusammenhangskomponente) Eine Zusammenhangskomponente besteht ausschließlich aus Voxeln mit Belegung 0 und jeder Voxel mit Belegung 0 ist Teil einer Zusammenhangskomponente. Ein Voxel gehört zu einer Zusammenhangskomponente M , wenn er einen Nachbarvoxel in 6er-Nachbarschaft besitzt, der zu M gehört. Ein Voxel kann nicht mehreren Zusammenhangskomponenten angehören.

Die äußere Zusammenhangskomponente ist die Zusammenhangskomponente, der der Voxel mit den Koordinaten $(0, 0, 0)$ angehört.

Der erste Ansatz des Filterungsalgorithmus verwendet eine Tiefensuche oder wahlweise Breitensuche über alle 0-Voxel von $(0, 0, 0)$ aus beginnend. Dabei wird der Voxelraum als Graph aufgefasst, mit den Voxeln als Knoten und Kanten zwischen Voxeln gleicher Belegung in 6er-Nachbarschaft. Alle gefundenen Voxel werden mit 2 markiert, wodurch genau

4. Erstellung der Höhle im Voxelraum

und ausschließlich die äußere Zusammenhangskomponente komplett mit 2 markiert wird. Im zweiten Schritt werden alle 0-Voxel zu 1-Voxeln abgeändert, da diese zu schwebenden Fragmenten gehören. Danach werden alle 2-Voxel wieder zu 0-Voxeln umgelabelt.

Problematisch ist, dass die äußere Zusammenhangskomponente eine meist hohe Voxelanzahl besitzt. Dies bedeutet einen hohen Speicherverbrauch, da im Stack (für die Tiefensuche) bzw. in der Queue (für die Breitensuche) die Position der Voxel gespeichert werden muss. Diese umfasst je nach Auflösung des Voxelraums mehrere Byte.³

Die zweite und implementierte Variante wird in Algorithmus 3 aufgezeigt. Gegeben sei der Voxelraum mit seiner Ausdehnung von $[0, X_{vmax}] \times [0, Y_{vmax}] \times [0, Z_{vmax}]$.

Schritt 1: Beginnend vom Rand des Voxelraums wird ein Randvoxel der äußeren Zusammenhangskomponente gesucht. Für einen schnellen Test wird nur geprüft, ob der aktuelle $\text{Voxel}(x, y, z) = 0$ und $\text{Voxel}(x + 1, y, z) = 1$ gilt, anstatt alle Nachbarn zu betrachten. Die Suche wird mit $x = 0$ begonnen und mit aufsteigenden x -Werten fortgesetzt. Der erste so gefundene Voxel muss zur äußeren Zusammenhangskomponente gehören.

Schritt 2: Von diesem Voxel aus wird eine Breitensuche über alle Randvoxel im erweiterten Sinn durchgeführt und alle gefundenen Voxel auf 2 gesetzt. Damit wird der komplette Rand der äußeren Zusammenhangskomponente, also die *äußere Hülle*, mit 2 markiert.

Schritt 3: Nun wird der restliche Voxelraum geprüft und erneut nach Randvoxeln gesucht. Dadurch, dass schwebende Fragmente im Inneren von 1er-Voxel-Gebieten liegen, kann der zu durchsuchende Bereich in jeder Dimension um einige Voxel eingeschränkt werden. Wenn ein Randvoxel gefunden wird, muss dieser zu einem schwebenden Fragment gehören. Nun wird eine Breitensuche über die komplette Zusammenhangskomponente durchgeführt und alle gefundenen Voxel werden auf 1 gesetzt, wodurch das schwebende Fragment entfernt wird.

Schritt 4: Zum Schluss werden alle Voxel mit Belegung 2 wieder auf 0 abgeändert.

Bei dieser Variante werden nur die Voxel des Rands der äußeren Zusammenhangskomponente sowie die Voxel der schwebenden Fragmente in die Queue geladen und dies einzeln Komponente für Komponente. Obwohl beide Varianten einen Komplexität bzgl. des Speicherverbrauchs von $\mathcal{O}(n)$ haben, wird der letztere Algorithmus in der Praxis i.A. weniger Speicher benötigen als der erstere. Beide Verfahren haben eine Laufzeitkomplexität von $\mathcal{O}(n)$.

4.3.3. Anmerkung zur Optimierung

Algorithmus 3 benötigt zur Markierung der äußeren Hülle mindestens einen 1 Voxel breiten Rand aus Voxeln mit Belegung 0. Zur Vermeidung von Sonderfällen an Rändern ist es sinnvoll, diesen Rand auf drei Voxel zu erweitern. Ein Voxel wird dabei für die Markierung der äußeren Hülle benötigt plus einen Voxel für die Nachbarschaft plus einen Voxel für die Prüfung, ob dieser Nachbarvoxel ebenfalls ein Randvoxel im erweiterten Sinn ist.

Dieser Rand ist ebenfalls ausreichend, damit die Erosion, die im nächsten Kapitel folgende Umwandlung sowie das Markieren von Andockstellen (Kapitel 7) ohne Randprüfungs-Sonderfälle arbeiten können. In anfänglichen Implementierungen wurde hierdurch ein Speedup von bis zu Faktor 5 erzielt.

³Bei der verwendeten Auflösung von 512 Voxeln Seitenlänge mindestens 27 Bit.

4. Erstellung der Höhle im Voxelraum

Algorithmus 3 Filterung schwebender Fragmente im Voxelraum

Eingabe: ungefilterter Voxelraum, mind. 1 Voxel breiter Rand des Voxelraums mit 0 belegt

Ausgabe: gefilterter Voxelraum

```

1: ▷ Schritt 1: Finde Startvoxel für äußere Hülle
2: for  $x = 0 \rightarrow X_{vmax} - 2$  do                                ▷ Suche mit aufsteigender X-Koordinate
3:   for all  $(y, z) \in [1, Y_{vmax} - 1] \times [1, Z_{vmax} - 1]$  do
4:     if  $Voxel(x, y, z) = 0 \wedge Voxel(x + 1, y, z) = 1$  then
5:       put  $(x, y, z)$  in Queue
6:        $E \leftarrow x$                                 ▷ niedrigste X-Koordinate der äußeren Hülle
7:       break                                ▷ Abbrechen der Suche
8:     end if
9:   end for
10: end for
11: ▷ Schritt 2: Markierung der äußeren Hülle
12: while Queue  $\neq$  leer do
13:   take  $\vec{a}$  from Queue
14:   for all  $\vec{b} \in 6\text{er Nachbarschaft of } \vec{a}$  do
15:     if  $\vec{b} \equiv \text{Randvoxel im erweiterten Sinn}$  then
16:       put  $\vec{b}$  in Queue
17:        $Voxel(\vec{b}) \leftarrow 2$                       ▷ Markieren des Voxels
18:     end if
19:   end for
20: end while
21: ▷ Schritt 3: Löschung freischwebender Komponenten
22: for all  $(x, y, z) \in [E + 2, X_{vmax} - 2] \times [2, Y_{vmax} - 2] \times [2, Z_{vmax} - 2]$  do
23:   if  $Voxel(x, y, z) = 0 \wedge Voxel(x + 1, y, z) = 1$  then
24:     put  $(x, y, z)$  in Queue
25:      $Voxel(x, y, z) \leftarrow 1$                       ▷ Löschen des Voxels
26:     while Queue  $\neq$  leer do
27:       take  $\vec{a}$  from Queue
28:       for all  $\vec{b} \in 6\text{er Nachbarschaft of } \vec{a}$  do
29:         if  $Voxel(\vec{b}) = 0$  then
30:           put  $\vec{b}$  in Queue
31:            $Voxel(\vec{b}) \leftarrow 1$                       ▷ Löschen des Voxels
32:         end if
33:       end for
34:     end while
35:   end if
36: end for
37: ▷ Schritt 4: Originalmarkierung der äußeren Hülle wiederherstellen
38: for all  $(x, y, z) \in [E, X_{vmax}] \times [0, Y_{vmax}] \times [0, Z_{vmax}]$  do
39:   if  $Voxel(x, y, z) = 2$  then
40:      $Voxel(x, y, z) \leftarrow 0$ 
41:   end if
42: end for

```

5. Umrechnung der Voxelhöhle in Dreiecksnetz

Die Höhlenstruktur liegt nun als Voxelzeichnung vor. Diese soll in ein entsprechendes Dreiecksnetz umgewandelt werden.

Jeder Voxel soll einem Würfel der Kantenlänge 1 entsprechen, wobei ein Voxel mit Position (x, y, z) den Würfel $[x, x + 1] \times [y, y + 1] \times [z, z + 1]$ widerspiegelt. Für das Dreiecksnetz sind nur die $0 \Leftrightarrow 1$ -Übergänge, also die Grenzen des Höhlenfreiraums zu festem Gestein, relevant. Jeder dieser Übergänge entspricht einer quadratischen Fläche, die durch zwei Dreiecke dargestellt werden kann. Jedes dieser Dreieckspaare benötigt vier Vertices, die allerdings mit anderen Dreieckspaaren geteilt werden können. Die Beziehung zwischen Voxeln und Vertices ist in Abbildung 5.1(a) aufgezeigt.

Gesucht ist ein Algorithmus mit der Aufgabe, aus allen Übergängen das resultierende Dreiecksnetz zu bilden. Dabei ist eine Aufteilung der Höhle in mehrere Subnetze notwendig, da die Irrlicht Engine auf maximal 65536 Vertices pro Meshbuffer begrenzt ist (siehe Kapitel 3.3.1). Die Aufteilung der Höhle in mehrere Subnetze ist ebenso sinnvoll, da sich in Abhängigkeit von Position und Ausrichtung des Betrachters so Teile der Höhle leichter ausblenden oder in niedrigeren Detailstufen darstellen lassen.

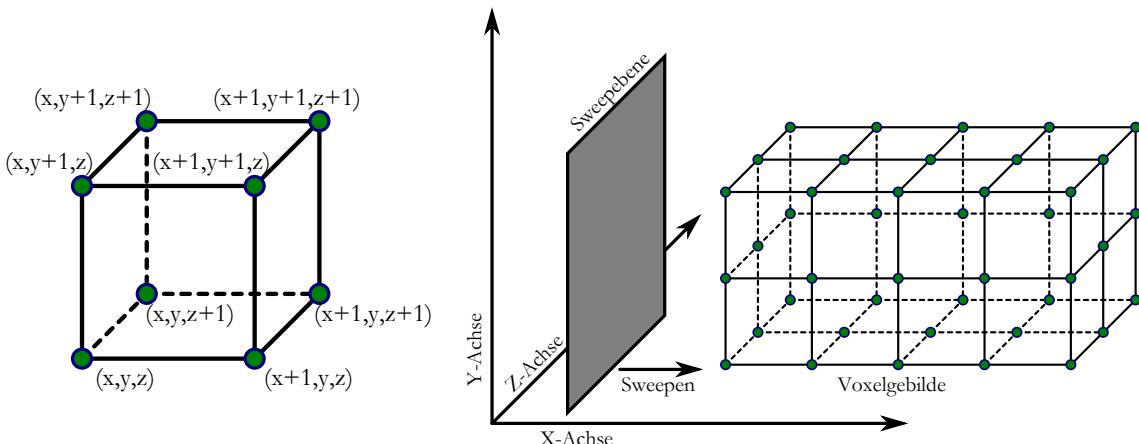


Abbildung 5.1.: Voxelkoordinaten zu Vertexkoordinaten: v.l.n.r.

- (a) einzelner Voxel mit Voxelkoordinaten (x, y, z) mit seinen acht begrenzenden Vertices (grün) und ihren Vertexkoordinaten
- (b) Umwandlung eines Voxelgebildes per Sweepverfahren

5.1. Umwandlungsalgorithmus

Sweep-Verfahren funktionieren nach dem Prinzip der Dimensionsreduzierung [Kle05, S.51 ff.]. Ein statisches d -dimensionales Problem wird in ein dynamisches $(d - 1)$ -dimensionales Problem umgewandelt. Eine räumliche Dimension wird so zu einer zeitlichen Dimension.

Im Fall des 3D-Raums ergibt sich eine *Sweepebene*. Die zeitliche Dimension ist das Sweepen entlang einer Achse. Sinn und Zweck von Sweep-Verfahren ist das Reduzieren von Lauf-

5. Umrechnung der Voxelhöhle in Dreiecksnetz

zeiten (da weniger Elementbeziehungen gleichzeitig betrachtet werden müssen) oder das Einsparen von Speicherplatz. Im Umwandlungsalgorithmus wird es zur Reduzierung des benötigten Speicherplatzes verwendet.

Ein *Octree* ist eine Struktur, die ein gegebenes quaderförmiges Raumgebiet in acht kleinere gleichgroße Raumgebiete aufteilt, die Oktanten genannt werden [Len04, S.237-238]. Die Aufteilung des Raumgebietes erfolgt jeweils mittig des Quaders durch drei senkrecht aufeinanderliegende Schnittebenen. Die Oktanten werden bei Bedarf auf die gleiche Weise weiter unterteilt. Die entstehende Baumstruktur bildet den Octree. Die Raumgebiete stellen die Knoten des Baums dar, die Kanten verlaufen vom jeweiligen Raumgebiet zu den zugehörigen Oktanten.

Der Octree wird zur Aufteilung des Netzes verwendet, so dass in jedem Teilnetz maximal V_{max} viele Vertices vorhanden sind. In der Implementierung wird $V_{max} = 65000$ gewählt, welches unter dem Maximum der Irrlicht-Engine je Meshbuffer liegt. Die Tiefe des erzeugten Octrees liegt in $\mathcal{O}(\log n)$ mit n als Anzahl aller Voxel.

Die klassische Methode zum Aufbau des Octrees ist die *Top-Down*-Variante. Begonnen wird mit dem gesamten Voxelraum als Startknoten. Für jeden Knoten erfolgt das Zählen der benötigten Vertices V mittels Algorithmus 4. Wenn $V > V_{max}$ ist, dann wird der entsprechende Knoten in acht gleichgroße Kindknoten aufgeteilt, mit denen äquivalent verfahren wird. Die Laufzeitkomplexität der Methode ist $\mathcal{O}(n \log n)$, da in jeder Ebene des Octrees schlimmstenfalls alle Voxel geprüft werden müssen.

Die schnellere Variante ist die *Bottom-Up* Methode. Hier wird zuerst der Octree soweit aufgebaut, bis jeder Knoten maximal so groß ist, dass seine *theoretische* maximale Vertexanzahl höchsten V_{max} beträgt. Für $V_{max} = 65000$ wäre dies bei einer Ausgangsgröße des Voxelraums von 512^3 eine Knotengröße von 32^3 , welches maximal $33^3 = 35937$ Vertices entsprechen würde. Eine Zählung mittels Algorithmus 4 findet nur für die unterste Ebene des Octrees, also die Blätter, statt. Nun werden von den Kindknoten aus bis zur Wurzel hin die Werte V der Knoten zusammenaddiert. Dabei berechnet sich V eines Elternknotens aus der Summe der V der Kindknoten:

$$V_{Eltern} = \sum_{i=0}^7 (V_{Kind\ i}) \quad (5.1)$$

Durch dieses Verfahren kann V eines Nichtblattknotens höher liegen als die tatsächlich Anzahl an Vertices, die für diesen Knoten benötigt werden würden. Keinesfalls liegt sie aber darunter. Die Werte V stellen somit obere Schranken dar. Dieses Verfahren hat eine Laufzeitkomplexität von $\mathcal{O}(n)$ und liegt damit im Punkt Geschwindigkeit vor der Top-Down-Variante.¹ In der Implementierung des Programms wird die Bottom-Up-Methode verwendet.

Algorithmus 4 nutzt für das Zählen der benötigten Vertices ein Sweepverfahren, um Speicherplatz zu sparen. Statt $\mathcal{O}((X_{max} - X_{min}) \cdot (Y_{max} - Y_{min}) \cdot (Z_{max} - Z_{min}))$ zusätzlichen Speicherplatz zu benötigen, wird nur $\mathcal{O}((Y_{max} - Y_{min}) \cdot (Z_{max} - Z_{min}))$ benötigt. Der Algorithmus durchläuft das entsprechende Voxelgebiet und testet auf $0 \Leftrightarrow 1$ Übergänge zwischen benachbarten Voxeln. Jeder dieser Übergänge benötigt vier Vertices an

¹Ausnahme: Octree hat nur eine Hierarchieebene. Hier liegt die Top-Down-Variante minimal vor der Bottom-Up-Variante.

5. Umrechnung der Voxelhöhle in Dreiecksnetz

Algorithmus 4 Umwandlung Voxelgebiet in Dreiecksnetz: Zählen der Vertices

Eingabe: Voxelraum, Grenzen des Voxelgebietes: $X_{min}, X_{max}, Y_{min}, Y_{max}, Z_{min}, Z_{max}$.

$S[a]$ und $S[b]$ sind mit 0 initialisiert.

Ausgabe: Anzahl benötigter Vertices: V

```

1:  $V \leftarrow 0$ 
2: for  $x = X_{min} \rightarrow X_{max}$  do                                ▷ Sweepen entlang der X-Achse
3:   for all  $(y, z) \in [Y_{min}, Y_{max}] \times [Z_{min}, Z_{max}]$  do
4:     if  $Voxel(x, y, z) = 1$  then
5:       if  $Voxel(x - 1, y, z) = 0$  then
6:          $S[a](y, z) \leftarrow 1$  ,  $S[a](y + 1, z) \leftarrow 1$ 
7:          $S[a](y, z + 1) \leftarrow 1$  ,  $S[a](y + 1, z + 1) \leftarrow 1$ 
8:       end if
9:       if  $Voxel(x + 1, y, z) = 0$  then
10:         $S[b](y, z) \leftarrow 1$  ,  $S[b](y + 1, z) \leftarrow 1$ 
11:         $S[b](y, z + 1) \leftarrow 1$  ,  $S[b](y + 1, z + 1) \leftarrow 1$ 
12:      end if
13:      if  $Voxel(x, y - 1, z) = 0$  then
14:         $S[a](y, z) \leftarrow 1$  ,  $S[a](y, z + 1) \leftarrow 1$ 
15:         $S[b](y, z) \leftarrow 1$  ,  $S[b](y, z + 1) \leftarrow 1$ 
16:      end if
17:      if  $Voxel(x, y + 1, z) = 0$  then
18:         $S[a](y + 1, z) \leftarrow 1$  ,  $S[a](y + 1, z + 1) \leftarrow 1$ 
19:         $S[b](y + 1, z) \leftarrow 1$  ,  $S[b](y + 1, z + 1) \leftarrow 1$ 
20:      end if
21:      if  $Voxel(x, y, z - 1) = 0$  then
22:         $S[a](y, z) \leftarrow 1$  ,  $S[a](y + 1, z) \leftarrow 1$ 
23:         $S[b](y, z) \leftarrow 1$  ,  $S[b](y + 1, z) \leftarrow 1$ 
24:      end if
25:      if  $Voxel(x, y, z + 1) = 0$  then
26:         $S[a](y, z + 1) \leftarrow 1$  ,  $S[a](y + 1, z + 1) \leftarrow 1$ 
27:         $S[b](y, z + 1) \leftarrow 1$  ,  $S[b](y + 1, z + 1) \leftarrow 1$ 
28:      end if
29:    end if
30:  end for
31:  for all  $(y, z) \in [Y_{min}, Y_{max} + 1] \times [Z_{min}, Z_{max} + 1]$  do
32:     $V \leftarrow V + S[a](y, z)$                                 ▷ Vertices zählen
33:     $S[a](y, z) \leftarrow 0$                                      ▷ Sweepalebene a löschen
34:  end for
35:   $a \leftrightarrow b$                                          ▷ Sweepalebene vertauschen
36:  end for                                                 ▷ Sweep abgeschlossen
37:  for all  $(y, z) \in [Y_{min}, Y_{max} + 1] \times [Z_{min}, Z_{max} + 1]$  do
38:     $V \leftarrow V + S[a](y, z)$                                 ▷ verbleibende Vertices zählen
39:  end for

```

5. Umrechnung der Voxelhöhle in Dreiecksnetz

entsprechender Position, um durch zwei Dreiecke dargestellt werden zu können. Diese Vertices werden in den beiden Sweep-Ebenen markiert. Bei jedem neuen Sweep-Schritt entlang der X-Achse wird die Anzahl der benötigten Vertices aus der älteren der beiden Sweep-Ebenen ausgelesen und zur Gesamtanzahl dazuaddiert. Das Sweepverfahren ist in Abbildung 5.1(b) skizziert.

Nach dem Aufbau des Octrees findet der Umwandlungsschritt statt. Der Octree wird nun Top-Down durchlaufen. Für jeden Knoten wird getestet, ob $V \leq V_{max}$, wobei die vorher ausgerechneten Werte für V verwendet werden. Falls der Test positiv verläuft, wird das zum Knoten gehörige Gebiet umgewandelt. Bei negativem Test wird stattdessen mit den Kindknoten weiter verfahren.

Zur Umwandlung eines Voxelgebietes in ein Dreiecksnetz wird Algorithmus 4 entsprechend angepasst. Statt den Einträgen der Sweep-Ebenen eine 1 zuzuweisen, wird zuerst überprüft, ob ein Eintrag vorliegt. Wenn nicht, wird ein neuer Vertex mit den entsprechenden Koordinaten (Sweep-Ebene a: $x_{res} = x$, Sweep-Ebene b: $x_{res} = x + 1, y$ und z werden übernommen) erstellt und ein Verweis auf diesen Vertex in der zugehörigen Sweep-Ebene gespeichert. Die Berechnung der finalen Koordinaten des Vertex findet wie in Kapitel 5.2 beschrieben statt.

Nach dem Testen der vier Sweep-Ebeneneinträge werden zwei Dreiecke erstellt, die die jeweilige Voxelgrenzfläche darstellen. Für die Eckpunkte der zwei Dreiecke werden die vier Vertices entsprechend der Einträge in den beiden Sweep-Ebenen verwendet. Danach wird der nächste der sechs Fälle abgearbeitet.

Das Zählen der Vertices entfällt in diesem Algorithmus. Der Umwandlungsprozess hat eine Laufzeitkomplexität von $\mathcal{O}(n)$ mit n als Anzahl aller Voxel.

5.2. Berechnung der Vertexkoordinaten

Die Berechnung der Vertexkoordinaten wird über eine Funktion realisiert, die aus Eingangs-Koordinaten Ausgangs-Koordinaten berechnet. Dabei sind die Eingangs-Koordinaten ganzzahlig, da es sich um Voxel-Eckpunkte handelt, und die Ausgangs-Koordinaten Gleitkommazahlen. Das Prinzip Eingangs-Koordinaten in Ausgangs-Koordinaten umzuwandeln entspricht dem von Raumverzerrungsmethoden. Andere oder auch zusätzliche Raumverzerrungsverfahren zur hier verwendeten Funktion sind denkbar.

Wichtig dabei ist, dass die Funktion deterministisch arbeitet, d.h. bei gleichen Eingangs-Koordinaten die gleichen Ausgangs-Koordinaten ausgibt. Dies ist relevant, damit die verschiedenen Subnetze der Höhle an den Rändern zusammenpassen, d.h. die korrespondierenden Randvertices müssen identische Koordinaten besitzen. Weiterhin können so Gänge exakt an die Höhle angedockt werden (siehe Kapitel 7.3).

Bei der Basisfunktion entsprechen die Ausgangs-Koordinaten genau den Eingangs-Koordinaten, wodurch die Würfelstruktur des Voxelraums erhalten bleibt. Eine weitere Anpassung der Koordinaten für zusätzlichen Realismus ist sinnvoll. Statt die Voxelstruktur nur durch die Würfelform wiederzugeben, ist es besser, diese Struktur zu verzerrn, so dass sie natürlicher wirkt. Als grundlegende Strategie werden die Vertices *verwackelt*, um dem Dreiecksnetz der Höhle eine zerklüftetere Struktur zu geben.

Beim Verwackeln wird der Eingangs-Vertex entlang aller drei Achsen per deterministischer Zufallsfunktion verschoben. Die Zufallsfunktion wird mit einem einzigartigen Seed

5. Umrechnung der Voxelhöhle in Dreiecksnetz

für jede mögliche Eingangsposition initialisiert. Jede Eingangsposition erhält so einzigartige Zufallswerte, die Funktion ist deterministisch.

Dann werden die Abweichungen von der Ausgabeposition zur Eingangsposition wie in Formel 5.2 berechnet. Die Formeln für Δ_y und Δ_z sind äquivalent. Hierbei stellt K die Verwacklungsstärke dar, die zur Erhaltung der Symmetrie für alle drei Achsen gleich ist. Die Funktion $Zufallswert[0, 1]$ liefert eine zufällige Gleitkommazahl im Intervall $[0, 1]$. Resultat für Δ_x , Δ_y und Δ_z sind Werte zwischen $-K$ und K .

$$\Delta_x = -K + 2 \cdot K \cdot Zufallswert[0, 1] \quad (5.2)$$

Es muss $K < 0,5$ gelten, damit keine Vertices aufeinanderliegen können, siehe hierzu Abbildung 5.2(a). Dies ist aber nicht ausreichend, da trotzdem Konflikte auftreten können, wie in (b) zu sehen ist. Hier würden sich beide Dreiecke der Fläche (grau) ineinander schieben. Dem Problem kann vorgebeugt werden, indem der Vertex nur so verschoben wird, dass der Manhattanabstand zur Ausgangsposition in jeder Ausgangsebene der Dreiecksflächen (XY, XZ, YZ) weniger als 0,5 beträgt, wie in (c) zu sehen ist. Es muss also gelten:

$$|\Delta_x| + |\Delta_y| < 0,5 \wedge |\Delta_x| + |\Delta_z| < 0,5 \wedge |\Delta_y| + |\Delta_z| < 0,5 \quad (5.3)$$

Der erste Lösungsansatz besteht in Begrenzung der Verwacklungsstärke auf $< 0,25$. Nachteil ist, dass Potential verschenkt wird, da Koordinaten so nur recht wenig abgeändert werden. Der neue Ansatz erlaubt weiterhin Verwacklungsstärken $< 0,5$. Hier wird nach der Berechnung von Δ_x , Δ_y und Δ_z getestet, ob ein Problemfall vorliegt, also der zulässige Manhattanabstand überschritten wurde. Wenn ja, dann wird dieser Konflikt aufgelöst.

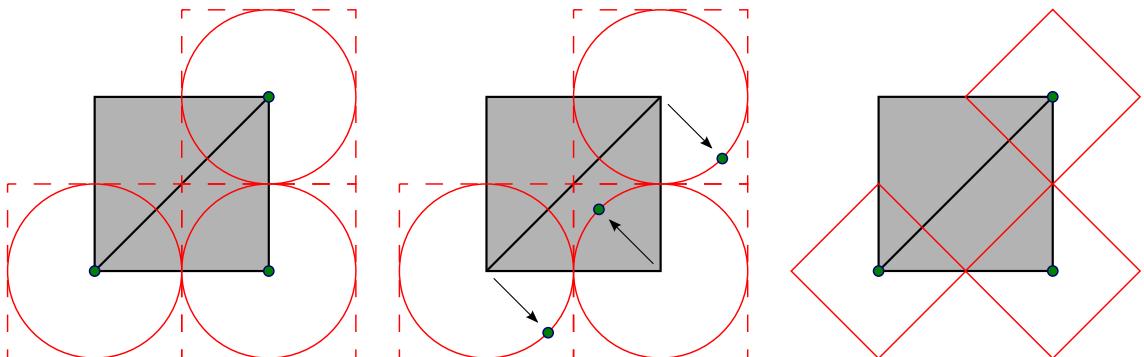


Abbildung 5.2.: Verwacklung der Vertexkoordinaten: v.l.n.r.

- (a) Änderung einzelner Koordinaten $< 0,5$ (Grenzen: Quadrate, rot gestrichelt), euklidischer Abstand $< 0,5$ (Kreise, rot)
- (b) beides kann zu Konflikten führen
- (c) keine Konflikte bei Manhattanabstand $< 0,5$ (Quadrate, rot)

Dies geschieht mittels *Clamping*.² Wenn der maximal zulässige Manhattanabstand M überschritten wird, dann beschneide die Koordinatenabweichung so, dass keine Überschreitung

²In der Computergrafik bezeichnet der Begriff „Clamping“ das Begrenzen von Werten auf einen bestimmten Bereich (vgl. Color Clamping in OpenGL).

5. Umrechnung der Voxelhöhle in Dreiecksnetz

von M mehr vorliegt. In der Implementierung wurde der Wert $M = 0,49$ gewählt. Für K muss gelten $0 \leq K \leq M$.

Es können grundsätzlich drei verschiedene Problemfälle auftreten, die unterschiedliche Clampingstrategien sinnvoll machen. Wichtig für die entwickelten Clampingstrategien ist es, die Ausgangswerte möglichst wenig und möglichst gleichmäßig zu verändern.

Fall 1: Es liegt eine Verletzung der Bedingung in allen drei Ebenen vor. Lösungsstrategie: Normalisiere alle Abweichungswerte ($\Delta_x, \Delta_y, \Delta_z$). Nutze dazu folgendes Verfahren:

$$\begin{aligned} n &= \max(|\Delta_x| + |\Delta_y|, |\Delta_x| + |\Delta_z|, |\Delta_y| + |\Delta_z|) \\ \Delta_x^{neu} &= \Delta_x \cdot M/n \\ \Delta_y^{neu} &= \Delta_y \cdot M/n \\ \Delta_z^{neu} &= \Delta_z \cdot M/n \end{aligned} \tag{5.4}$$

Fall 2: Es liegt eine Verletzung der Bedingung in genau zwei Ebenen vor. Lösungsstrategie: Clampe die gemeinsame Koordinate. Nutze dazu folgendes Verfahren (Beispiel für XY und XZ \rightarrow Clamping von Δ_x):

$$\begin{aligned} n &= \min(M - |\Delta_y|, M - |\Delta_z|) \\ \Delta_x^{neu} &= \begin{cases} n & \text{falls } \Delta_x > 0 \\ -n & \text{sonst} \end{cases} \end{aligned} \tag{5.5}$$

Fall 3: Es liegt eine Verletzung der Bedingung in exakt einer Ebene vor. Lösungsstrategie: Normalisiere beide zur Ebene gehörigen Abweichungswerte. Nutze dazu folgendes Verfahren (Beispiel für XY-Ebene):

$$\begin{aligned} n &= |\Delta_x| + |\Delta_y| \\ \Delta_x^{neu} &= \Delta_x \cdot M/n \\ \Delta_y^{neu} &= \Delta_y \cdot M/n \end{aligned} \tag{5.6}$$

Nach dem Clamping werden die resultierenden Ausgangskoordinaten ($x_{aus}, y_{aus}, z_{aus}$) aus den Eingangskoordinaten ($x_{ein}, y_{ein}, z_{ein}$) berechnet:

$$(x_{aus}, y_{aus}, z_{aus}) = (x_{ein} + \Delta_x, y_{ein} + \Delta_y, z_{ein} + \Delta_z) \tag{5.7}$$

5.2.1. Glättung des Dreiecksnetzes

Falls die Höhlenwände eine glattere, aber dennoch unregelmäßige Struktur besitzen sollen, wäre eine optionale Glättung des verwackelten Dreiecksnetzes von Vorteil. Die Überlegung besteht darin, die Verwacklungsrichtungen so zu gestalten, dass durch den Verwacklungsprozess diese Glättung erreicht wird.

Dazu wird die Nachbarschaft des jeweiligen Vertex betrachtet. Zu einem gegebenen Vertex sei ein anderer Vertex benachbart, wenn er mit diesem über eine Voxelkante verbunden ist. In Abbildung 5.4 sind solche Nachbarschaften aufgezeigt. Ein Vertex kann über maximal sechs Nachbarvertices verfügen, in jeder Achsenrichtung einen. Folgende Beobachtung lässt

5. Umrechnung der Voxelhöhle in Dreiecksnetz

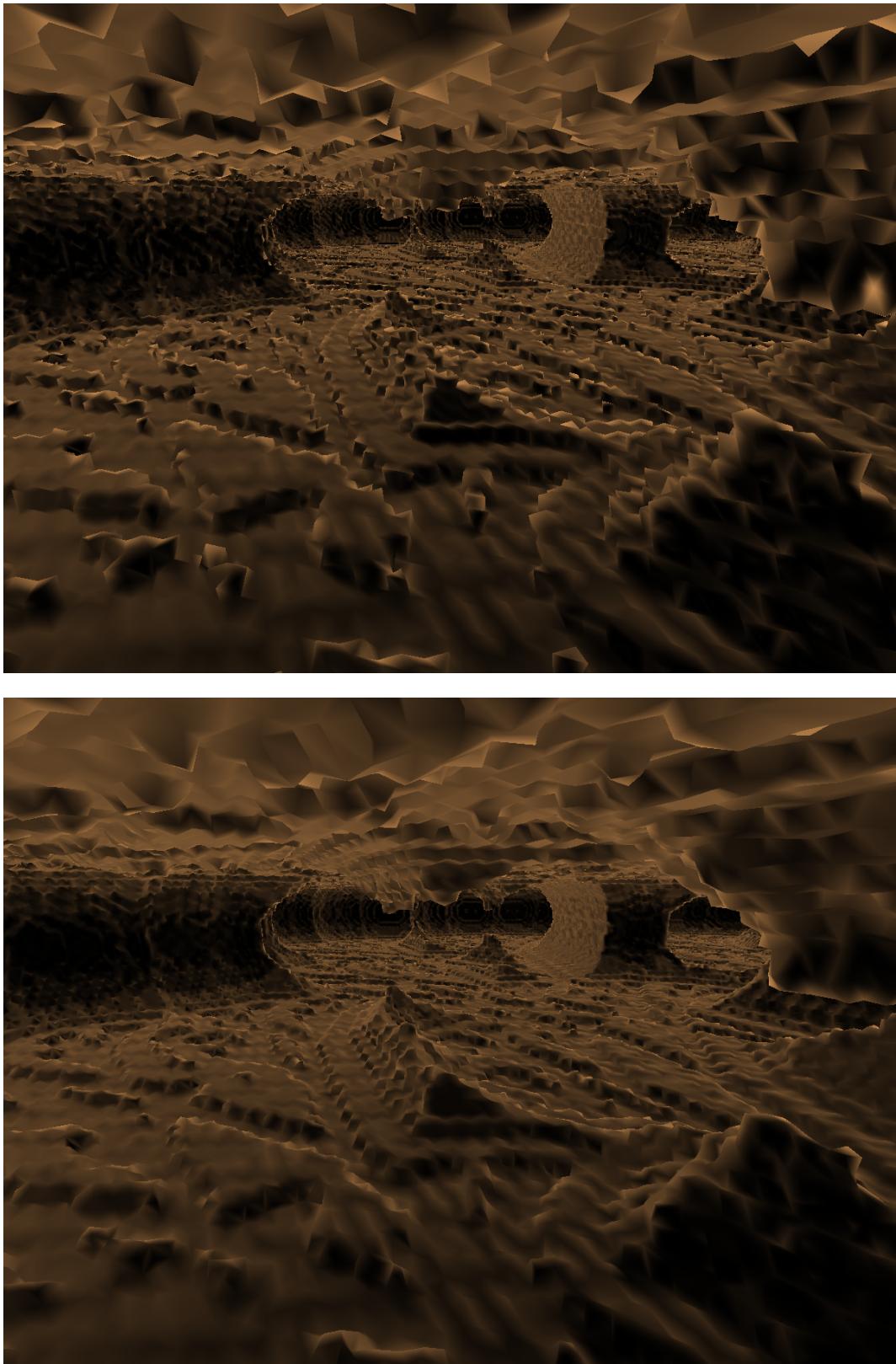


Abbildung 5.3.: *Modifizierung der Vertexkoordinaten: $K = 0,35$, v.o.n.u.* (a) Verwackeln ohne Glätten, (b) Verwackeln mit Glätten

5. Umrechnung der Voxelhöhle in Dreiecksnetz

sich treffen: für glattere Struktur muss ein Vertex i.d.R. in Richtung seiner Nachbarvertices bewegt werden. In entgegengesetzter Richtung würden Zacken entstehen.

Die Idee ist es, diese Kanten zu Nachbarvertices zu zählen und gegeneinander aufzuwiegen, um die finale Verwacklungsrichtung zu bestimmen. Durch Betrachtung der möglichen Fälle (Auswahl in Abbildung 5.4) wird ersichtlich, dass die Kanten dabei unterschiedlich gewichtetet werden müssen: Liegt die Kante zu einem Nachbarvertex inmitten einer Fläche, so zählt dies zweifach, liegt sie dagegen auf einer Ecke, zählt sie nur einfach. Wenn keine Kante existiert, so zählt dies nullfach.

Dadurch ergeben sich sechs Werte als *Kantengewichte*, für jede Achse jeweils in positive und negative Richtung: $W_{x+}, W_{x-}, W_{y+}, W_{y-}, W_{z+}, W_{z-}$

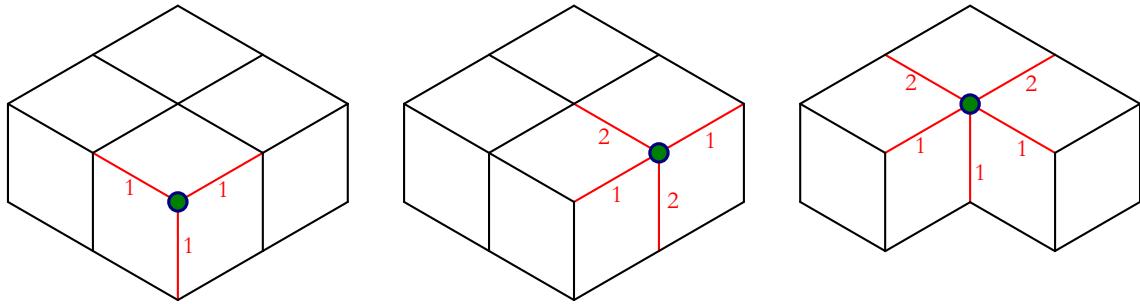


Abbildung 5.4.: *Wichtung der Kanten*: Vertices grün, Kanten zu Nachbarvertices rot, Zahlen geben Kantengewichte an

Für jede Achse werden die Kantengewichte für positive und negative Richtung gegeneinander aufgewogen, um die erlaubten Verwacklungsrichtungen zu bestimmen. Die Richtung mit dem höherem Gewicht ist erlaubt, bei gleichem Gewicht sind beide Richtungen erlaubt. Damit berechnen sich die *Verwacklungsrichtungsparameter* R_x , R_y und R_z wie in Formel 5.8 angeben (Beispiel für x , y und z äquivalent):

$$R_x = \begin{cases} 0 & \text{falls } W_{x+} = W_{x-} \\ 1 & \text{falls } W_{x+} > W_{x-} \\ -1 & \text{sonst} \end{cases} \quad (5.8)$$

Die Verwacklungsrichtungsparameter geben an, in welche Richtung die Koordinaten verwackelt werden dürfen. Der Verwacklungsschritt aus Formel 5.2 wird nun wie in Formel 5.9 angepasst (Beispiel für x , y und z äquivalent):

$$\Delta_x = \begin{cases} -K + 2 \cdot K \cdot \text{Zufallswert}[0, 1] & \text{falls } R_x = 0 \\ K \cdot \text{Zufallswert}[0, 1] & \text{falls } R_x > 0 \\ -K \cdot \text{Zufallswert}[0, 1] & \text{sonst} \end{cases} \quad (5.9)$$

Zur Optimierung wird eine Look-Up-Tabelle zu Speicherung von R_x , R_y und R_z verwendet. Für die Berechnung des Tabellenindex wird die Belegung $Voxel_i$ (0 oder 1) aller acht den Vertex umgebenden Voxel verwendet. Jeder Voxel repräsentiert dabei ein Bit des Index.

$$Index = \sum_{i=0}^7 (2^i \cdot Voxel_i) \quad (5.10)$$

5. Umrechnung der Voxelhöhle in Dreiecksnetz

Verfahren	Normalisieren	Wichtung für \vec{P}_1	W. für \vec{P}_2	W. für \vec{P}_3
Uniform	ja	1	1	1
per Winkel	ja	$\propto (\vec{P}_2 - \vec{P}_1, \vec{P}_3 - \vec{P}_1)$	\propto in \vec{P}_2	\propto in \vec{P}_3
per Flächeninhalt	nein	1	1	1

Tabelle 5.1.: Verschiedene Wichtungsmethoden von Normalen bei der Berechnung von Vertexnormalen aus Dreiecksnormalen

Der Aufbau der Tabelle erfolgt bei Initialisierung des Programms. Alle möglichen Indices von 0 bis 255 werden durchlaufen und dabei aus jedem Index die zugehörige Belegung der Voxel bestimmt. Für diese Voxelkonfiguration werden die Verwacklungsrichtungsparameter berechnet und in der Tabelle gespeichert.

Bei der Berechnung der Vertexkoordinaten werden die den Vertex umgebenden Voxel betrachtet und daraus der Index bestimmt. Der entsprechende Eintrag der Look-Up-Tabelle wird ausgelesen und die Werte werden angewendet.

Ein Vergleich der Resultate zwischen der Umwandlung mit Verwackeln ohne Glätten und der mit Verwackeln mit Glätten ist in Abbildung 5.3 zu sehen.

5.3. Berechnung der Normalen

Für die Beleuchtung der Höhle ist es erforderlich, die Normale in jedem Vertex zu kennen. Dabei wird jede Vertexnormale aus den Normalen der Dreiecke, zu denen dieser Vertex zugehörig ist, bestimmt.

Das Gesamtverfahren sieht wie folgt aus: Anfangs ist jede Vertexnormale mit dem Nullvektor initialisiert. Nun werden alle Dreiecke abgelaufen und jeweils die unnormalisierte Dreiecksnormale \vec{N}_d nach Formel 5.11 berechnet. Die Eckpunkte des entsprechenden Dreiecks seien $\vec{P}_1, \vec{P}_2, \vec{P}_3$.

$$\vec{N}_d = (\vec{P}_2 - \vec{P}_1) \times (\vec{P}_3 - \vec{P}_1) \quad (5.11)$$

Es gibt nun unterschiedliche Arten, diese Dreiecksnormalen für die Berechnung der Vertexnormalen einzubeziehen. Drei dieser Methoden werden in Tabelle 5.1 verglichen. Je nach Verfahren wird die Dreiecksnormale normalisiert. Daraufhin folgt die Wichtung der Normale für jeden der drei Eckpunkte und die Aufaddierung dieser gewichteten Normale zu der Normalen jedes Eckpunktes. Nach dem Verfahren besitzt jeder Vertex seine *Rohnormale*, die aus der Summe der gewichteten Dreiecksnormalen besteht.

Diese Rohnormalen werden am Ende des gesamten Berechnungsprozesses normalisiert.

Die *uniforme Wichtung* stellt die grundlegendste Methode dar. Die Normale eines Vertex berechnet sich aus dem Durchschnitt der Normalen der zugehörigen Dreiecke. Die normalisierte Normale jedes Dreiecks wird dabei mit Faktor 1 gewichtet.

Die *Wichtung per Winkel* wird beispielsweise von Irrlicht [Irr10] verwendet. Die Normale eines Vertex berechnet sich aus den normalisierten Normalen der zugehörigen Dreiecke, die

5. Umrechnung der Voxelhöhle in Dreiecksnetz

mit dem Winkel des jeweiligen Dreicks in diesem Vertex gewichtet werden. Die Winkel können per Kosinussatz berechnet werden.

Die *Wichtung per Flächeninhalt* wichtet die Normale jedes Dreiecks mit dem Flächeninhalt des Dreiecks. Die Länge der unnormalisierten Normale entspricht dem Flächeninhalt des Parallelogramms, welches von $(\vec{P}_2 - \vec{P}_1)$ und $(\vec{P}_3 - \vec{P}_1)$ aufgespannt wird. Dieser Flächeninhalt entspricht dem doppelten des Dreiecksflächeninhaltes. Die Normale braucht somit nicht normalisiert zu werden, da sie bereits korrekt gewichtet ist. Der Faktor 2 kann beibehalten werden, da er einen konstanten Faktor für jede Normale darstellt.

Durch den Wegfall der Normalisierung ist die Wichtung per Flächeninhalt die schnellste aller drei Varianten. Die Vorteile gegenüber den beiden anderen Methoden liegen dabei nicht nur in der schnelleren Berechnung: Degenerierte Dreiecke, wie Linien oder Punkte, ohne bestimmbare Normalen werden hierbei von vornherein nicht in die Vertexnormalen einbezogen, da der Flächeninhalt für diese Dreiecke null ist. Für die beiden anderen Varianten sei hier die Festlegung getroffen, dass der normalisierte Nullvektor wieder gleich dem Nullvektor sei.³ Da die unnormalisierte Dreiecksnormale jedes degenerierten Dreiecks dem Nullvektor entspricht, werden diese so auch nicht in die Vertexnormalen einbezogen.

Die Methode Wichtung per Winkel erscheint intuitiv, da Ecken mit größerem Winkel einen größeren Einfluss auf ein Dreieck haben. Bei der Wichtung per Flächeninhalt ist der Beitrag zur Normalen umso größer, je größer das Dreieck ist, was ebenfalls intuitiv erscheint.

Im Vergleich der Ergebnisse (siehe Abbildung 5.5) liefert die Wichtung per Flächeninhalt die gleichmäßigsten Normalen und die Wichtung per Winkel die weichsten Übergänge an Kanten. Insgesamt ähneln sich die Ergebnisse dennoch. Dies ist nicht verwunderlich: Die Voxelmeshes haben eine vergleichsweise regelmäßige Struktur. Für einen Mesh mit ausschließlich gleichgroßen gleichseitigen Dreiecken würden alle drei Methoden das gleiche Resultat liefern. Andere Methoden sind denkbar. [Slo91] beschreibt unter anderem die „Wichtung per inversen Flächeninhalt“ (Idee: kleine Dreiecke „wissen“ mehr über die lokale Oberfläche, als große Dreiecke mit weit auseinanderliegenden Eckpunkten) und eine „Methode der kleinsten Quadrate“ (nimm eine Ebene, um alle Vertices in der Nachbarschaft zu approximieren, und nutze deren Normale).

5.3.1. Normalen an Meshgrenzen

Die einzelnen Dreiecksnetze der Höhle teilen sich u.U. Vertices an ihren Grenzen. Diese Grenzvertices sind dann in mehreren Dreiecksnetzen vorhanden, besitzen aber identische Koordinaten. Es ist erforderlich, dass diese Vertices ebenfalls identische Normalen besitzen, da sie ein und den selben Vertex repräsentieren. Die entsprechenden Normalen müssen also zu einer einzigen Normale zusammengefasst werden.

Die Methode, um die Vertices zusammenzufassen, funktioniert wie folgt: Beim Umwandeln der Voxel in ein Dreiecksnetz werden bereits alle Grenzvertices markiert. Dazu wird getestet, ob sich der Vertex am Rand des umgewandelten Gebietes befindet und ob das benachbarte Netz an dieser Position ebenfalls einen Vertex generieren würde (dies kann durch Betrachtung der acht den Vertex umgebenden Voxel bestimmt werden). Wenn ja,

³Diese Variante ist auch in Irrlicht implementiert.

5. Umrechnung der Voxelhöhle in Dreiecksnetz

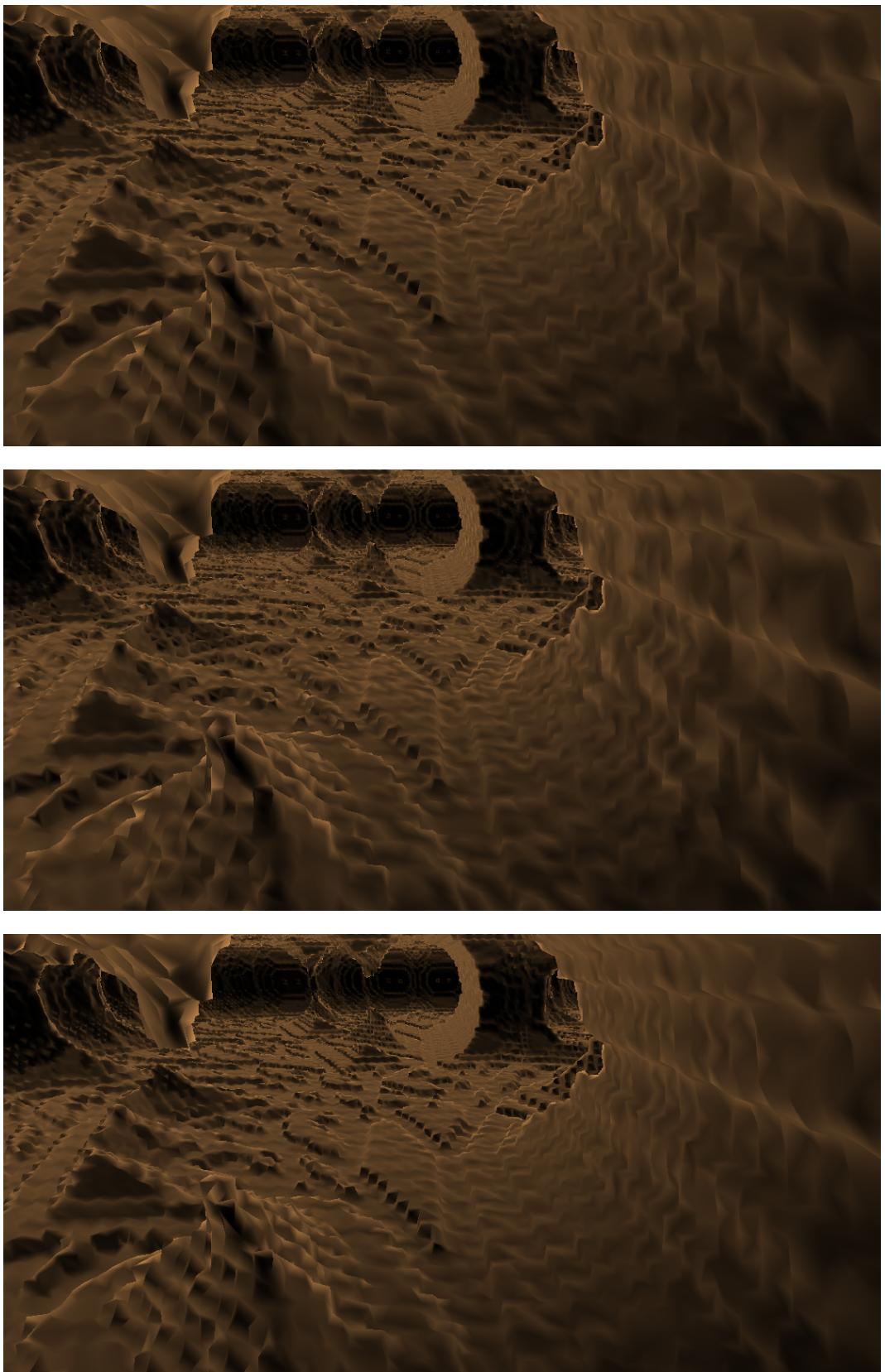


Abbildung 5.5.: Vergleich von Normalenberechnungsmethoden: v.o.n.u. (a) uniforme Wichtung, (b) Wichtung per Winkel, (c) Wichtung per Flächeninhalt

5. Umrechnung der Voxelhöhle in Dreiecksnetz

wird die X-Texturkoordinate $\vec{T}.x$ auf 1,0 gesetzt, ansonsten auf -1,0.⁴

Nach dem Umwandeln und der Berechnung der *Rohnormalen* für jeden Vertex (s.o.) werden alle Rohnormalen der Vertices mit $\vec{T}.x = 1,0$ in eine Tabelle geschrieben. Die Koordinaten (x, z) des jeweiligen Vertex definieren die Adresse in der Tabelle. Nach dem Eintragen werden alle Tabelleneinträge nach ihrer Y-Koordinate sortiert.

Nun wird jeder (x, z) -Eintrag der Tabelle durchlaufen und die Normalen von aufeinander folgenden Vertices gleicher Y-Koordinate zusammengefasst. Diese repräsentieren den gleichen Vertex, da sie identische (x, y, z) -Koordinaten haben. Beim Zusammenfassen werden alle Normalen zusammenaddiert und diese Summe dann den zugehörigen Vertices als neue Normale zugewiesen. Die Normalisierung aller Normalen findet zum Schluss des Prozesses statt.

Dadurch ergibt sich folgendes Gesamtverfahren:

1. Berechne alle Rohnormalen
2. Fasse die Rohnormalen aller Grenzvertices zusammen
3. Normalisiere alle Normalen

Die Laufzeitkomplexität der Normalenberechnung beträgt somit $\mathcal{O}(n \log y)$, mit n als Anzahl der Vertices und y als Anzahl der maximalen Anzahl der Tabelleneinträge. Das Sortierverfahren kostet hierbei die meiste Zeit, ohne Sortieren wäre die Laufzeitkomplexität linear in n . Als Verbesserungsmöglichkeit ließe sich statt einer (x, z) -Tabelle auch eine Hashtabelle verwenden. Nähere Ausführungen zur Wahl von Hashfunktionen sind in [SS06, S.398 ff.] und [CLRS10, S.264 ff.] gelistet. Die Herausforderung liegt hier darin, die Anzahl der Einträge der Tabelle nicht zu hoch zu wählen (Speicherplatz), aber auch keine zu hohe Anzahl an Vertices pro Tabelleneintrag zu generieren (Laufzeit).

5.4. Texturierung und Beleuchtung der Höhle

Die Höhle mittels einer 2D-Textur zu texturieren, gestaltet sich problematisch. Zum einen sind passende Texturkoordinaten schwierig zu berechnen, zum anderen würden viele Wiederholungen der verwendeten Textur auftreten oder aber sehr große Texturen nötig sein, also sehr specheraufwändige Megatexturen.

Viel sinnvoller ist die Verwendung prozeduraler 3D-Texturen. Eine 3D-Textur besitzt drei Texturkoordinaten statt wie im 2D-Fall nur zwei. Als Texturkoordinaten können im Fall der Höhle direkt die Positionsdaten (x, y, z) der einzelnen Punkte verwendet werden. Eine Texturfunktion weist einem solchen Texturkoordinaten-Tripel (x, y, z) einen konkreten Farbwert zu. Dieser Farbwert wird entsprechend der Beleuchtung modifiziert und dann dargestellt. Je nach Wahl der Funktion werden verschiedene Strukturen wie z.B. Marmor oder Holz erzeugt, Beispiele hierfür werden in [ESK97, S.213 ff.] beschrieben.

Kristalline Strukturen lassen sich in Verbindung mit der zerklüfteten unregelmäßigen Geometrie der Höhle durch passende Beleuchtung vergleichsweise einfach darstellen. Ausgegangen wird von der Grundformel 5.12 für diffuse Beleuchtung nach dem Lambertschen Gesetz [FvFH97, S.723-724], mit \vec{L} als Vektor in Richtung des einfallenden Lichtstrahls

⁴Da die Höhlenmeshes keine 2D-Texturen verwenden, können deren Texturkoordinaten für andere Zwecke genutzt werden.

5. Umrechnung der Voxelhöhle in Dreiecksnetz

und \vec{N} als Normale der Oberfläche. \vec{L} und \vec{N} seien normiert. C_{res} ist die resultierende Farbe, C_{basis} die gewünschte Basisfarbe mit $0 \cdot C_{basis}$ als schwarzer Farbe (unbeleuchtet).

$$C_{res} = \begin{cases} (\vec{L} \cdot \vec{N}) \cdot C_{basis} & \text{falls } (\vec{L} \cdot \vec{N}) \geq 0 \\ 0 \cdot C_{basis} & \text{sonst} \end{cases} \quad (5.12)$$

Der effektiv zulässige Winkel zwischen \vec{L} und \vec{N} für Formel 5.12 beträgt 0 bis 90 Grad. Durch die zerklüftete Struktur der Höhle ist dieser Winkel unter Verwendung von Vertexnormalen auch bei sichtbaren Oberflächenteilen oftmals größer. Mit Beleuchtung nach Formel 5.12 würde hier sehr früh zu schwarz überblendet werden. Eine feinere Abstufung der Beleuchtung würde entfallen. Statt zwischen \vec{L} und \vec{N} nur einen Winkel 0 bis 90 Grad zuzulassen, seien nun beliebige Winkel (also 0 bis 180 Grad) zulässig.

Als Lichtquelle wird eine Punktlichtquelle verwendet, die die Position der Kamera innehat. Somit zeigt \vec{L} vom entsprechendem Punkt der Oberfläche in Richtung Kamera. Die Normale wird invertiert, wodurch ein Verschlucken des Lichtes bei frontaler Betrachtung der jeweiligen Struktur erreicht wird. An den äußeren Rändern tritt jedoch das Licht hervor, welches schimmernde kristalline Strukturen, wie z.B. Eis, darstellt. Diese Modifikationen führen zu Formel 5.13. Ein Vergleich der Methoden ist in Abbildung 5.6 zu sehen.

$$C_{res} = \left(1 - (\vec{L} \cdot \vec{N})\right) \cdot \frac{C_{basis}}{2} \quad (5.13)$$

Für das Darstellen der Höhle im Dungeongenerator-Programm wird als Texturfunktion eine einheitliche Basisfarbe für alle (x, y, z) verwendet. Die Beleuchtung wird mittels der Methode aus Formel 5.13 durchgeführt. Diese Kombination erlaubt ein performantes Rendering der Höhlenstrukturen, da nur wenige Berechnungen notwendig sind, und ist einfach mittels Shadern implementierbar.

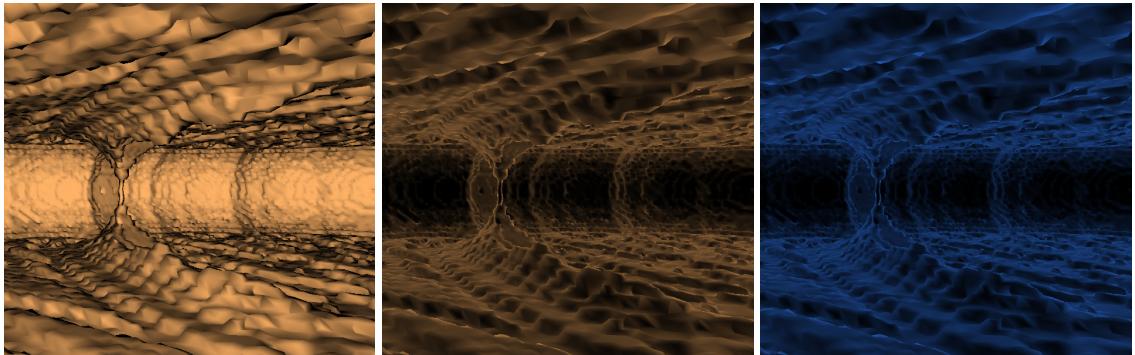


Abbildung 5.6.: Vergleich von Beleuchtungsmethoden: v.l.n.r. (a) lambertsche Beleuchtung, (b) Kristall braun (Bernstein), (c) Kristall blau (Eis)

6. Konzept der Gänge und Räume

In den beiden letzten Kapiteln wurde beschrieben, wie die Höhle für den Dungeon erstellt wird. Dieses Kapitel soll beschreiben, wie die Gänge und Räume als weitere Bestandteile des Dungeons konzipiert sind.

6.1. Gänge als polygonale Schläuche

Gänge dienen dazu, Räume untereinander und mit der Höhle zu verbinden. Ein Gang soll durch einen polygonalen Schlauch (aufgebaut aus Dreiecken) mit definierbarem Querschnitt wiedergegeben werden. Dieser Schlauch soll in seinem Verlauf einer fest vorgegebenen Kurve $\vec{P}(t)$, mit $t \in [0, 1]$, folgen.

Zur Konstruktion des Gangs seien n zweidimensionale Punkte $\vec{Q_0}, \dots, \vec{Q_{n-1}}$ gegeben, die den Querschnitt des Gangs repräsentieren. Diese Punkte seien um den Punkt $(0, 0)$ im Uhrzeigersinn herum angeordnet. Die Gangkurve $\vec{P}(t)$ habe die erste Ableitung $\vec{P}'(t)$. Für eine gleichmäßige Struktur des Gangs und eine gute Kontrolle über die Anzahl erzeugter Dreiecke ist es sinnvoll, den Abstand zwischen zwei aufeinanderfolgenden Gangsegmenten gleich groß zu gestalten. Dieser Abstand sei w .

Algorithmus 5 beschreibt die Konstruktion des Gangs. Für ein aktuelles t wird zunächst das lokale Koordinatensystem bestimmt, ähnlich dem Frenetschen Koordinatensystem. Der Unterschied ist die Einbeziehung des Höhenvektors $(0, 1, 0)$, der die Richtung „Oben“ für den Gang definiert. Das Frenetsche Koordinatensystem verwendet an dieser Stelle die zweite Ableitung der Kurve. Nun wird ein Gangsegment aus 3D-Punkten erzeugt und mittels Dreiecken mit dem vorherigen Gangsegment verbunden. Dabei werden immer zwei benachbarte Punkte des aktuellen Gangsegments mit den äquivalenten Punkten des letzten Gangsegments durch eine viereckige Fläche, bestehend aus zwei Dreiecken, verbunden (Zeilen 16-19 des Algorithmus 5). Danach erfolgt die Berechnung des nächsten t . Dieses soll so berechnet werden, dass der euklidische Abstand zwischen $\vec{P}(t_{neu})$ und $\vec{P}(t_{alt})$ genau w beträgt. So wird der Gang stückweise aufgebaut. Der Algorithmus terminiert bei $t = 1$. Die Bedingung des Abstands w muss für das letzte Segment nicht unbedingt eingehalten werden, da dieser hier typischerweise kleiner ist als für die anderen Segmente.

Von einer konstanten Anzahl von Gangsegmentpunkten und einer konstanten Laufzeit für das Finden des nächsten t ausgehend, besitzt Algorithmus 5 eine Laufzeitkomplexität von $\mathcal{O}(n)$, mit n als Anzahl der erstellten Segmente. Diese Anzahl ist sowohl vom Kurvenverlauf als auch von w abhängig.

Potentiell problematisch ist das Überschreiten des Limits für die Anzahl von Vertices je Meshbuffer (siehe Abschnitt 3.3.1). Um dieses Problem zu lösen, wird ein weiterer Test eingefügt. Nach der Vertex- und Dreiecksgenerierung wird geprüft, ob für das nächste Gangsegment noch genug Reserve vorhanden ist. Wenn nicht, wird ein neuer Meshbuffer angelegt und das aktuelle Gangsegment für den neuen Buffer ein zweites Mal erstellt. Danach wird wie gehabt mit dem Algorithmus fortgefahrene.

Zur Texturierung des Gangs wird für jeden Punkt des Querschnitts zusätzlich eine X-

6. Konzept der Gänge und Räume

Texturkoordinate angegeben. Die Y-Texturkoordinate wird entlang des Gangs entsprechend des Abstands zwischen $\overrightarrow{P(t_{neu})}$ und $\overrightarrow{P(t_{alt})}$ weitergezählt. Dabei ist die Erhöhung der Y-Texturkoordinate pro Abstand 1 angebbar.

Algorithmus 5 Erstellung eines polygonalen Schlauchs entlang Kurve $P(t)$, $t \in [0, 1]$

Eingabe: Kurve $P(t)$, Querschnitt des Schlauchs, Gangsegmentabstand w

Ausgabe: Dreiecksnetz des polygonalen Schlauchs

```

1:  $t \leftarrow 0$ 
2:  $\overrightarrow{Vorn} \leftarrow \left\| \overrightarrow{P'(t)} \right\|$                                  $\triangleright$  lokales Koordsys. berechnen
3:  $\overrightarrow{Links} \leftarrow \left\| \overrightarrow{Vorn} \times (0, 1, 0) \right\|$            $\triangleright \|\vec{v}\| \dots \vec{v}$  normalisiert
4:  $\overrightarrow{Oben} \leftarrow \overrightarrow{Links} \times \overrightarrow{Vorn}$ 
5:
6: for all  $\overrightarrow{Q_i} \in \text{Querschnitt}$  do                                          $\triangleright$  Vertices erzeugen
7:   create Vertex  $V_i$ 
8:    $\overrightarrow{V_i.Pos} \leftarrow \overrightarrow{P(t)} - \overrightarrow{Q_i.x} \cdot \overrightarrow{Links} + \overrightarrow{Q_i.y} \cdot \overrightarrow{Oben}$        $\triangleright$  Position berechnen
9:    $\overrightarrow{V_i.N} \leftarrow \left\| \overrightarrow{P(t)} - \overrightarrow{V_i.Pos} \right\|$                                 $\triangleright$  Normale berechnen
10: end for
11:
12: while  $t < 1$  do
13:    $t \leftarrow t_{neu}$  mit  $D_e(\overrightarrow{P(t)}, \overrightarrow{P(t_{neu})}) = w$                           $\triangleright$  nächstes t bestimmen
14:   calculate lokales Koordinatensystem                                                  $\triangleright$  s.o.
15:   create Vertices                                                                $\triangleright$  s.o.
16:   for  $i = 1 \rightarrow n - 1$  do
17:     create ViereckAusZweiDreiecken( $V_{i-1}, V_i, V_{i-1}^{letztes}, V_i^{letztes}$ )
18:   end for
19:   create ViereckAusZweiDreiecken( $V_{n-1}, V_0, V_{n-1}^{letztes}, V_0^{letztes}$ )
20: end while

```

6.1.1. Verlauf des Gangs per kubischem Hermite-Spline

Für die Form des Gangs wäre ein geschwungener Verlauf wünschenswert. Als Vorgabe sollen die Endpunkte und die Richtung des Gangs in diesen Punkten angegeben werden. Hierfür eignet sich ein dreidimensionaler kubischer Hermite-Spline als Kurve $\overrightarrow{P(t)}$. Dieser stellt ein Polynom dritter Ordnung dar und ermöglicht somit eine gewundene und abgerundete Struktur. Er ist durch die Vorgaben der Positionen und ersten Ableitungen der Endpunkte konstruierbar. Die ersten Ableitungen beschreiben die Richtung der Kurve im jeweiligen Punkt. Abbildung 6.1 zeigt einen solchen Hermite-Spline, Abbildung 6.2 zeigt einen per Hermite-Spline konstruierten Gang.

Die folgende Herleitung der Formeln für $\overrightarrow{P(t)}$ und $\overrightarrow{P'(t)}$ des kubischen Hermite-Splines ist im Wesentlichen an [Sal06, S.111 ff.] angelehnt. Die Formel für ein Polynom dritter Ordnung lautet:

$$\overrightarrow{P(t)} = \vec{a} \cdot t^3 + \vec{b} \cdot t^2 + \vec{c} \cdot t + \vec{d} \quad (6.1)$$

6. Konzept der Gänge und Räume

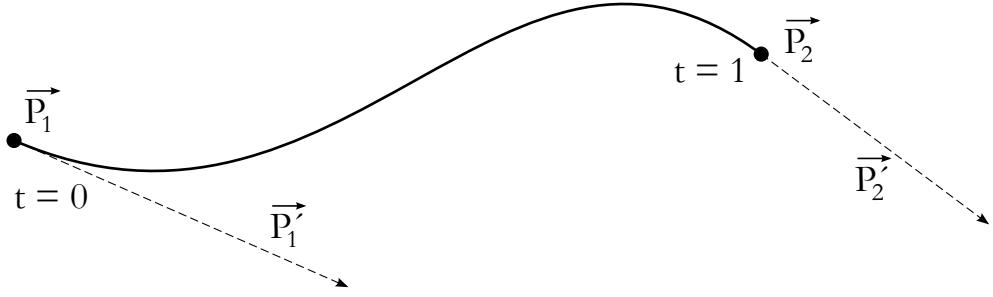


Abbildung 6.1.: *Kubischer Hermite-Spline*: mit Anfangs- und Endpunkt sowie den ersten Ableitungen in diesen Punkten (gestrichelt)

Daraus ergibt sich als erste Ableitung:

$$\overrightarrow{P'(t)} = 3 \cdot \vec{a} \cdot t^2 + 2 \cdot \vec{b} \cdot t + \vec{c} \quad (6.2)$$

Gegeben seien die zwei Punkte $\overrightarrow{P_1} = \overrightarrow{P(0)}$, $\overrightarrow{P_2} = \overrightarrow{P(1)}$ und die ersten Ableitungen in diesen Punkten $\overrightarrow{P'_1} = \overrightarrow{P'(0)}$, $\overrightarrow{P'_2} = \overrightarrow{P'(1)}$. Es gilt somit:

$$\begin{aligned} \overrightarrow{P_1} &= \vec{a} \cdot 0^3 + \vec{b} \cdot 0^2 + \vec{c} \cdot 0 + \vec{d} \\ \overrightarrow{P_2} &= \vec{a} \cdot 1^3 + \vec{b} \cdot 1^2 + \vec{c} \cdot 1 + \vec{d} \\ \overrightarrow{P'_1} &= 3 \cdot \vec{a} \cdot 0^2 + 2 \cdot \vec{b} \cdot 0 + \vec{c} \\ \overrightarrow{P'_2} &= 3 \cdot \vec{a} \cdot 1^2 + 2 \cdot \vec{b} \cdot 1 + \vec{c} \end{aligned} \quad (6.3)$$

Als Lösungen dieses Gleichungssystems ergeben sich:

$$\begin{aligned} \vec{a} &= 2 \cdot \overrightarrow{P_1} - 2 \cdot \overrightarrow{P_2} + \overrightarrow{P'_1} + \overrightarrow{P'_2} \\ \vec{b} &= -3 \cdot \overrightarrow{P_1} + 3 \cdot \overrightarrow{P_2} - 2 \cdot \overrightarrow{P'_1} - \overrightarrow{P'_2} \\ \vec{c} &= \overrightarrow{P'_1} \\ \vec{d} &= \overrightarrow{P_1} \end{aligned} \quad (6.4)$$

Man erhält eingesetzt und in Matrizenbeschreibweise umgeformt:

$$\overrightarrow{P(t)} = (t^3, t^2, t, 1) \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \overrightarrow{P_1} \\ \overrightarrow{P_2} \\ \overrightarrow{P'_1} \\ \overrightarrow{P'_2} \end{pmatrix} \quad (6.5)$$

$$\overrightarrow{P'(t)} = (t^3, t^2, t, 1) \begin{pmatrix} 0 & 0 & 0 & 0 \\ 6 & -6 & 3 & 3 \\ -6 & 6 & -4 & -2 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \overrightarrow{P_1} \\ \overrightarrow{P_2} \\ \overrightarrow{P'_1} \\ \overrightarrow{P'_2} \end{pmatrix} \quad (6.6)$$

Damit lassen sich für jedes t die zugehörige Position $\overrightarrow{P(t)}$ und die erste Ableitung $\overrightarrow{P'(t)}$

6. Konzept der Gänge und Räume

bestimmen. Bei der Konstruktion der Parameter werden im Programm die Positionen der Endpunkte \vec{P}_1 , \vec{P}_2 und die Richtungen des Gangs vom Endpunkt zum Ganginneren \vec{P}'_1 , $-\vec{P}'_2$ angegeben. Der Betrag der ersten Ableitung wirkt als Gewichtung für die Richtung des Gangs. Je höher diese ist, desto höher ist auch der Einfluss der Richtungsvorgabe.



Abbildung 6.2.: *Gänge per kubischem Hermite-Spline*: v.l.n.r. (a) Gang von innen, (b) Gang von außen. Verwendet wird eine Testtextur.

Äquidistante Unterteilung

Algorithmus 5 verwendet äquidistante Abstände w zwischen aufeinanderfolgenden Punkten $\vec{P}(t)$. Es stellt sich die Frage, wie diese Unterteilung zu bestimmen ist. Eine äquidistante Unterteilung von t wird i.A. keine solche Abstände liefern. Eine Approximation mittels der Bogenlänge s als euklidischem Abstand wäre denkbar. Die Kurve müsste also in gleichgroße Stücke bezüglich ihrer Bogenlänge aufgeteilt werden. Hierfür ist eine Reparametrisierung von $\vec{P}(t)$ nach $\vec{P}(s)$ nötig. Das Ablaufen der Kurve würde mittels gleichmäßiger Erhöhung des Parameters s erfolgen. Für die Reparametrisierung muss die Funktion $t(s)$ bekannt sein.

Die Bogenlänge einer Kurve $\vec{P}(u)$, mit $\vec{P}'(u)$ als erster Ableitung, zwischen den Punkten $\vec{P}(a)$ und $\vec{P}(b)$ berechnet sich wie folgt:

$$s(a, b) = \int_a^b \sqrt{\vec{P}'(u)^2} du \quad (6.7)$$

Verwendet man für $\vec{P}(u)$ ein Polynom dritter Ordnung, wie es beim kubischen Hermite-Spline der Fall ist, so erhält man für $\vec{P}'(u)$ eines zweiter Ordnung. Dieses wird durch die Quadrierung zu einem Polynom vierter Ordnung unter der Wurzel, mit k_i als den entsprechenden Koeffizienten:

$$s(a, b) = \int_a^b \sqrt{k_1 u^4 + k_2 u^3 + k_3 u^2 + k_4 u + k_5} du \quad (6.8)$$

6. Konzept der Gänge und Räume

Für $s(t)$ als Bogenlänge der Kurve zwischen $\overrightarrow{P(0)}$ und $\overrightarrow{P(t)}$ resultiert daraus:

$$s(t) = \int_0^t \sqrt{k_1 u^4 + k_2 u^3 + k_3 u^2 + k_4 u + k_5} du \quad (6.9)$$

Nun ergibt sich folgendes Problem: Löst man die Formel für $s(t)$ und die Umkehrfunktion $t(s)$ (beispielsweise mit einem Computeralgebrasystem), dann erhält man Formeln, die mehrere Seiten lang und äußerst komplex sind. Dieser Ansatz ist somit i.A. für kubische Polynome nicht realisierbar.

Die Alternative, welche im Generatorprogramm Anwendung findet, ist die Verwendung einer numerischen Lösung. Genutzt wird das Bisektionsverfahren, welches u.a. in [BF01, S.49-50] beschrieben wird. Das Verfahren halbiert ein gegebenes Intervall fortlaufend und verwendet dabei in der nächsten Iteration jeweils die Hälfte weiter, in der die Lösung liegen muss. Dies wird so lange fortgeführt, bis die gewünschte Genauigkeit für die Lösung erreicht ist. In der Praxis erweist sich diese Methode als sehr schnell und robust.

Gegeben sind t_{akt} , der gewünschte euklidische Abstand w_{ziel} zwischen $\overrightarrow{P(t_{akt})}$ und $\overrightarrow{P(t_{neu})}$ sowie die Genauigkeit ϵ . Gesucht ist t_{neu} .

Zuerst wird getestet, ob $D_e(\overrightarrow{P(t_{akt})}, \overrightarrow{P(1)}) \leq w_{ziel}$ gilt. Wenn dem so ist, dann ist $t_{neu} = 1$ das Rückgabergebnis. Der Rest des Verfahrens verläuft folgendermaßen:

1. Starte mit Intervall $[m, n] = [t_{akt}, 1]$
2. Setze $t_{neu} = m + \frac{(n-m)}{2}$
3. Berechne aktuellen Abstand $w_{akt} = D_e(\overrightarrow{P(t_{akt})}, \overrightarrow{P(t_{neu})})$
4. Teste, ob $w_{akt} \in [w_{ziel} - \epsilon, w_{ziel} + \epsilon]$
 - ja: Gewünschte Genauigkeit erreicht. Abbruch mit Ergebnis t_{neu} .
 - nein: Fahre fort.
5. Wenn $w_{akt} > w_{ziel}$, dann setze $n = t_{neu}$ (Wert zu groß → Weiterverwendung der unteren Intervallhälfte),
 - sonst setze $m = t_{neu}$ (Wert zu klein → Weiterverwendung der oberen Intervallhälfte).
6. Gehe zu 2.

Als Verbesserung ließe sich eine *adaptive Unterteilung* verwenden. Grundidee ist eine feinere Unterteilung an den Stellen, an denen sich die Kurvenrichtung stark ändert, um diese Abschnitte exakter wiedergeben zu können. Diese Stellen sind durch Betrachtung der zweiten Ableitung der Kurve bestimmbar. Charakteristisch ist ein hoher Betrag der zweiten Ableitung und eine Richtung dieser, die sich stark von der Richtung der ersten Ableitung unterscheidet.

6.1.2. Adapter

Der Querschnitt des Gangs hat i.A. eine andere Form als die Öffnungen von Räumen oder der Höhle, wodurch eine Anpassung der Geometrien notwendig ist. Die Geometrien der Öffnungen in Räumen und der Höhle seien als *Andockstellen* bezeichnet. Diese Andockstellen bilden exakt die Form der Öffnung ab. An den Enden des Gangs muss der Querschnitt so

6. Konzept der Gänge und Räume

angepasst werden, dass er zur Form der Andockstelle überblendet. Die dazu verwendete Geometrie sei *Adapter* genannt.

Das *Gangsegment* in $\overrightarrow{P(0)}$ bzw. $\overrightarrow{P(1)}$ sei gegeben durch eine Folge von Vertices und ein lokales Koordinatensystem. Für die Vertices sind Positionen, Normalen und Texturkoordinaten bekannt.

Die *Andockstelle* sei gegeben durch eine Folge von Vertices, einen Mittelpunkt $\overrightarrow{M_a}$ und eine normalisierte Normale $\overrightarrow{N_a}$. Die Vektoren $\overrightarrow{Links_a}$ und $\overrightarrow{Oben_a}$ des lokalen Koordinatensystems der Andockstelle werden berechnet. Die Normale der Andockstelle übernimmt dabei die Rolle von \overrightarrow{Vorn} . Für die Andockstellenvertices sind nur die Positionen, aber keine Texturkoordinaten oder Normalen bekannt.

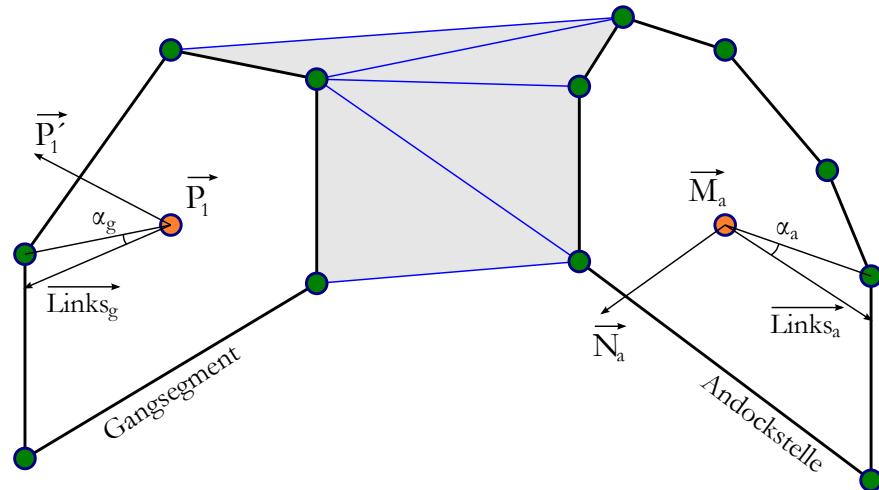


Abbildung 6.3.: *Prinzip des Adapters*: Verbindung zwischen ersten bzw. letztem Gangsegment und einer Andockstellengeometrie. Triangulierung (blau) erfolgt über Zuordnung der Vertices (grün) per Winkelähnlichkeit.

Das Prinzip der Adaptererstellung ist in Abbildung 6.3 dargestellt. Die Triangulation zwischen Gangsegment und Andockstelle erfolgt auf Basis der Winkelähnlichkeit ihrer Vertices. Zur späteren Zuordnung der Vertices wird hierfür jedem Vertex ein Winkel α zugewiesen. Die Winkelmessung findet mit den auf die jeweilige Ebene¹ projizierten Vektoren zwischen Vertex und Ebenenmittelpunkt statt. Der entsprechende Vektor \overrightarrow{Links} steht für 0° , gemessen wird im Uhrzeigersinn. Die Winkel für die Punkte des Gangsegments werden wie in Formel 6.10 berechnet. Für die Berechnung können die 2D-Koordinaten der Querschnittspunkte $\overrightarrow{Q_i}$ genutzt werden, aus denen die 3D-Punkte des Segments gebildet wurden.

$$\alpha = \text{atan2}(\overrightarrow{Q_i.y}, -\overrightarrow{Q_i.x}) \quad (6.10)$$

Die Winkel für die Punkte der Andockstelle werden wie folgt berechnet, mit $\overrightarrow{V_i.Pos}$ als 3D-Koordinaten der Andockstellenvertices:

$$\alpha = \text{atan2}\left(\left(\overrightarrow{V_i.Pos} - \overrightarrow{M_a}\right) \cdot \overrightarrow{Oben_a}, \left(\overrightarrow{V_i.Pos} - \overrightarrow{M_a}\right) \cdot \overrightarrow{Links_a}\right) \quad (6.11)$$

¹Gangsegmentebene: $\overrightarrow{P(0)}$ bzw. $\overrightarrow{P(1)}$ als Mittelpunkt und $\overrightarrow{P'(0)}$ bzw. $\overrightarrow{P'(1)}$ normalisiert als Normale.
Andockstellenebene: $\overrightarrow{M_a}$ als Mittelpunkt und $\overrightarrow{N_a}$ als Normale.

6. Konzept der Gänge und Räume

Für die Korrektheit der nachfolgenden Algorithmen sei gefordert, dass der Gang sowie die Andockstelle so konstruiert sind, dass sowohl die Gangsegmentvertices als auch die Andockstellenvertices zirkulär aufsteigende Winkel besitzen. Weiterhin ist gefordert, dass der auf die jeweilige Ebene projizierte Kantenzug, der alle Vertices ihrer Reihenfolge nach miteinander verbindet, den Mittelpunkt der Ebene umschließt.² Zur einfacheren Berechnung werden die Vertices zunächst so umsortiert, dass der kleinste Index auch den kleinsten Winkel enthält. Dies wird über eine Indexverschiebung realisiert, d.h. alle Elemente werden um eine bestimmte Anzahl von Positionen verschoben.

Der erste Ansatz verwendet nur die bestehenden Vertices zur Triangulation des Adapters. Zunächst werden die Normalen der Andockstellenvertices V_i berechnet. Diese entsprechen dem normalisierten, auf die Andockstellenebene projizierten Vektor zwischen Mittelpunkt und Vertex:

$$\overrightarrow{V_i \cdot N} = \frac{\left(\overrightarrow{M_a} - \overrightarrow{V_i \cdot Pos} \right) - \left(\overrightarrow{N_a} \cdot \left(\overrightarrow{M_a} - \overrightarrow{V_i \cdot Pos} \right) \right) \cdot \overrightarrow{N_a}}{\left| \left(\overrightarrow{M_a} - \overrightarrow{V_i \cdot Pos} \right) - \left(\overrightarrow{N_a} \cdot \left(\overrightarrow{M_a} - \overrightarrow{V_i \cdot Pos} \right) \right) \cdot \overrightarrow{N_a} \right|} \quad (6.12)$$

Danach erfolgt die Berechnung der Texturkoordinaten. Für die Y-Texturkoordinate wäre ein Weiterzählen dieser Koordinate entsprechend des Abstands zum letzten Gangsegment denkbar. Problematisch ist die X-Texturkoordinate. Abbildung 6.4 zeigt mögliche Probleme. Ein einfaches Übernehmen der X-Texturkoordinate per Winkelähnlichkeit (Fig. (a)) kann zu Verdrehungen der Textur führen. Interpoliert man die X-Texturkoordinaten (Fig. (b)), kann die Textur überdehnhen.

Die Lösung besteht in dem Einfügen zusätzlicher Stützvertices mit den entsprechenden Winkeln und X-Texturkoordinaten (Fig. (c)). Die Vertices zwischen den Stützvertices interpolieren diese Koordinaten.

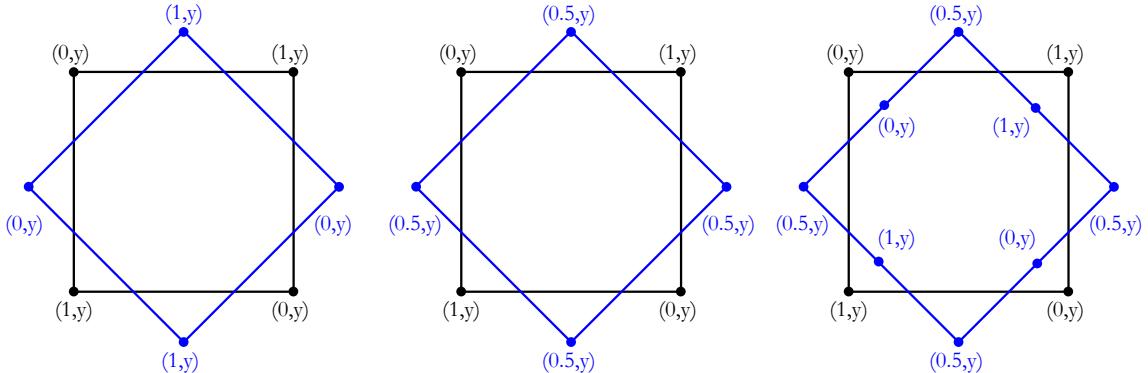


Abbildung 6.4.: Notwendigkeit von Stützvertices für Andockstellen: betrachtet wird Gangsegment (schwarz) und Andockstelle (blau): v.l.n.r.

- (a) Zuordnung der Texturkoordinaten per Winkelähnlichkeit führt zu Verzerrung, (b) per Interpolation zu Überdehnung
- (c) durch Stützvertices können die Texturkoordinaten korrekt interpoliert werden

²Dieser Kantenzug ist geschlossen, d.h. erster und letzter Vertex werden ebenfalls verbunden.

6. Konzept der Gänge und Räume

Finales Verfahren

Dies führt zu folgendem Verfahren: Erzeuge aus der Vertextmenge des Gangsegments und der Andockstelle die finalen Andockstellenvertices V'_i . Diese werden statt der ursprünglichen Vertices der Andockstelle für die Triangulation des Adapters verwendet. Zur Erzeugung der V'_i wird über die komplette Vertextmenge (Gangsegment und Andockstelle) per aufsteigendem Winkel gelaufen und es werden daraus die Vertices mit entsprechenden Parametern generiert.

Für die nachfolgenden Formeln werden folgende Notationen verwendet:

V_{av}	bezeichnet den Vertex der Andockstelle mit nächstkleinerem Winkel bezüglich des Winkels des aktuellen Vertex.
V_{an}	den mit nächstgrößerem Winkel.
V_{gv}	bezeichnet den Vertex des Gangsegments mit nächstkleinerem Winkel bezüglich des Winkels des aktuellen Vertex.
V_{gn}	den mit nächstgrößerem Winkel.
$TY_{Gangende}$	bezeichnet die Y-Texturkoordinate der Gangsegmentpunkte. ³
$TY_{IncPro1}$	gibt die Erhöhung der Y-Texturkoordinate pro Abstand 1 an.
$\Delta_{\prec}(\gamma_1, \gamma_2)$	bestimmt den resultierenden Winkelunterschied zwischen zwei Winkeln γ_1 und γ_2 . Dieser Winkelunterschied wird als Winkel zwischen 0° und 180° angegeben.
$\overrightarrow{V'_i.Pos}$	gibt die Position des Vertex an.
$\overrightarrow{V'_i.T}$	gibt die Texturkoordinaten des Vertex an.
$V'_i.Markierung$	dient zur Markierung, ob es sich um einen Textur-Stützvertex handelt (<i>Markierung = wahr</i>).
$V'_i.Korrespondenz$	gibt den winkelähnlichsten Vertex des Gangsegments an, welches der späteren Triangulierung dient.

Zur Berechnung der Y-Texturkoordinate $\overrightarrow{V'_i.T.y}$ wird entweder eine Addition oder Subtraktion verwendet, je nach Ende des Gangs. Sie berechnet sich über den Abstand des Andockstellenvertex zur Position auf dem Gangsegment mit identischem Winkel.

Der Interpolationsfaktor f sei im Folgenden stets so gewählt, dass die interpolierte Position den gleichen Winkel besitzt wie V_{akt} . Faktor f kann durch trigonometrische Betrachtungen bestimmt werden. Bei der Betrachtung der Vertices gibt es drei mögliche Fälle:

Fall 1: Der aktuelle Vertex V_{akt} ist ein Gangsegmentvertex. Vorgehen: Erzeuge einen neuen Stützvertex mit gleicher X-Texturkoordinate und gleichem Winkel.

$$\begin{aligned}
 \overrightarrow{V'_i.Pos} &= (1 - f) \cdot \overrightarrow{V_{av}.Pos} + f \cdot \overrightarrow{V_{an}.Pos} \\
 \overrightarrow{V'_i.T.x} &= \overrightarrow{V_{akt}.T.x} \\
 \overrightarrow{V'_i.T.y} &= TY_{Gangende} \pm \left| \overrightarrow{V'_i.Pos} - \overrightarrow{V_{akt}.Pos} \right| \cdot TY_{IncPro1}
 \end{aligned} \tag{6.13}$$

$V'_i.Markierung = \text{wahr}$
 $V'_i.Korrespondenz = V_{akt}$

³Diese ist für alle Punkte eines Gangsegments identisch.

6. Konzept der Gänge und Räume

Fall 2: Der aktuelle Vertex V_{akt} ist ein Andockstellenvertex. Vorgehen: Der Vertex wird weiterverwendet.

$$\begin{aligned} \overrightarrow{V'_i.Pos} &= \overrightarrow{V_{akt}.Pos} \\ \overrightarrow{\text{temp}} &= (1 - f) \cdot \overrightarrow{V_{gv}.Pos} + f \cdot \overrightarrow{V_{gn}.Pos} \\ \overrightarrow{V'_i.T.y} &= TY_{Gangende} \pm \left| \overrightarrow{V'_i.Pos} - \overrightarrow{\text{temp}} \right| \cdot TY_{IncPro1} \\ V'_i.\text{Markierung} &= \text{falsch} \\ V'_i.\text{Korrespondenz} &= \begin{cases} V_{gv} & \text{falls } \Delta_{\prec}(V_{akt.\alpha}, V_{gv.\alpha}) \leq \Delta_{\prec}(V_{akt.\alpha}, V_{gn.\alpha}) \\ V_{gn} & \text{sonst} \end{cases} \end{aligned} \quad (6.14)$$

Fall 3: Der aktuelle Vertex ist Andockstellenvertex V_{akt}^a und Gangsegmentvertex V_{akt}^g (Winkelgleichheit). Vorgehen: Der Andockstellenvertex übernimmt die X-Texturkoordinate des Gangsegmentvertex und wird als Stützvertex weiterverwendet.

$$\begin{aligned} \overrightarrow{V'_i.Pos} &= \overrightarrow{V_{akt}^a.Pos} \\ \overrightarrow{V'_i.T.x} &= \overrightarrow{V_{akt}^g.T.x} \\ \overrightarrow{V'_i.T.y} &= TY_{Gangende} \pm \left| \overrightarrow{V'_i.Pos} - \overrightarrow{V_{akt}^g.Pos} \right| \cdot TY_{IncPro1} \\ V'_i.\text{Markierung} &= \text{wahr} \\ V'_i.\text{Korrespondenz} &= V_{akt}^g \end{aligned} \quad (6.15)$$

Nun erfolgt die Berechnung der X-Texturkoordinaten der Nicht-Stützvertices der Andockstelle. Dazu wird die Texturkoordinaten eines solchen Vertex V'_k aus den beiden nächstliegenden Stützvertices V'_m und V'_n interpoliert. V'_m sei dabei der Stützvertex mit nächstkleinem Winkel und V'_n der Stützvertex mit nächstgrößtem Winkel bezüglich des Winkel von V'_k . Die Winkel werden dabei im Uhrzeigersinn zirkulär betrachtet. Der Interpolationsfaktor bestimmt sich aus den Längenverhältnissen der auf die Andockstellenebene projizierten Kantenzüge, die zum einen V'_m mit V'_k , als auch V'_m mit V'_n verbinden.

$$mit t = \frac{\sum_{i=m+1}^k \left| \left(\overrightarrow{V'_i.Pos} - \overrightarrow{V'_{i-1}.Pos} \right) - \left(\vec{N}_a \cdot \left(\overrightarrow{V'_i.Pos} - \overrightarrow{V'_{i-1}.Pos} \right) \right) \cdot \vec{N}_a \right|}{\sum_{i=m+1}^n \left| \left(\overrightarrow{V'_i.Pos} - \overrightarrow{V'_{i-1}.Pos} \right) - \left(\vec{N}_a \cdot \left(\overrightarrow{V'_i.Pos} - \overrightarrow{V'_{i-1}.Pos} \right) \right) \cdot \vec{N}_a \right|} \quad (6.16)$$

Die Normalen der V'_i werden äquivalent zu Formel 6.12 berechnet. Nach der Erzeugung dieser finalen Andockstellenvertices erfolgt die Generierung des Dreiecksnetzes des Adapters entsprechend Algorithmus 6. Die Triangulation nutzt, wie schon im ursprünglichen Ansatz und in Abbildung 6.3 aufgezeigt, die Winkelähnlichkeit der Vertices zur Zuordnung. Im Fall, dass der aktuelle Triangulationsschritt zwei Dreiecke generiert, wird die Diagonalachse so gewählt, dass sie die beiden Endpunkte mit dem kleineren Winkelunterschied verbindet.

6. Konzept der Gänge und Räume

Das Gesamtverfahren hat eine Laufzeitkomplexität von $\mathcal{O}(n + m)$, mit n als Anzahl der ursprünglichen Andockstellenvertices und m als Anzahl der Gangsegmentvertices. Einige Beispiele für fertige Adapter sind in Abbildung 6.5 zu sehen.

Algorithmus 6 Triangulierung eines Gangadapters

Eingabe: Andockstellenvertices $V'_0 \dots V'_{n-1}$, korrespondierende Gangsegmentvertices

Ausgabe: Dreiecksnetz des Gangadapters

```

1:  $K \leftarrow V'_0.Korrespondenz$ 
2: for  $i = 1 \rightarrow n - 1$  do
3:   if  $V'_i.Korrespondenz = K$  then            $\triangleright$  Korr. gleichbleibend: ein Dreieck einfügen
4:     create Dreieck( $V'_i, V'_{i-1}, K$ )
5:   else                                      $\triangleright$  Korr.-Wechsel: zwei Dreiecke einfügen
6:      $\beta_1 \leftarrow \Delta_{\prec} (V'_i.\alpha, K.\alpha)$            $\triangleright$  Diagonalachse bestimmen
7:      $\beta_2 \leftarrow \Delta_{\prec} (V'_{i-1}.\alpha, V'_i.Korrespondenz.\alpha)$ 
8:     if  $\beta_1 < \beta_2$  then
9:       create Dreieck( $V'_i, K, V'_{i-1}$ )
10:      create Dreieck( $V'_i, K, V'_i.Korrespondenz$ )
11:    else
12:      create Dreieck( $V'_{i-1}, V'_i.Korrespondenz, V'_i$ )
13:      create Dreieck( $V'_{i-1}, V'_i.Korrespondenz, K$ )
14:    end if
15:     $K \leftarrow V'_i.Korrespondenz$ 
16:  end if
17: end for
18: if  $V'_0.Korrespondenz = K$  then            $\triangleright$  Abarbeitung des verbleibenden Vertex
19:   create Dreieck( $V'_0, V'_{n-1}, K$ )
20: else
21:    $\beta_1 \leftarrow \Delta_{\prec} (V'_0.\alpha, K.\alpha)$            $\triangleright$  Diagonalachse bestimmen
22:    $\beta_2 \leftarrow \Delta_{\prec} (V'_{n-1}.\alpha, V'_0.Korrespondenz.\alpha)$ 
23:   if  $\beta_1 < \beta_2$  then
24:     create Dreieck( $V'_0, K, V'_{n-1}$ )
25:     create Dreieck( $V'_0, K, V'_0.Korrespondenz$ )
26:   else
27:     create Dreieck( $V'_{n-1}, V'_0.Korrespondenz, V'_0$ )
28:     create Dreieck( $V'_{n-1}, V'_0.Korrespondenz, K$ )
29:   end if
30: end if
```

Materialüberblendungen

Gang und Adapter bilden separate Geometrien. Ein zweites Paar an Texturkoordinaten für den Adapter wäre zum Textur-Blending zwischen beiden am Adapter angeschlossenen Geometrien sinnvoll. Die *Andockstellenseitigen-Vertices* erhalten als zweite Y-Texturkoordinate 1, 0 , die *Gangsegmentseitigen-Vertices* 0, 0. Dadurch ist ein Textur-Blending per zunehmendem Alphawert in Y-Richtung möglich.

6. Konzept der Gänge und Räume

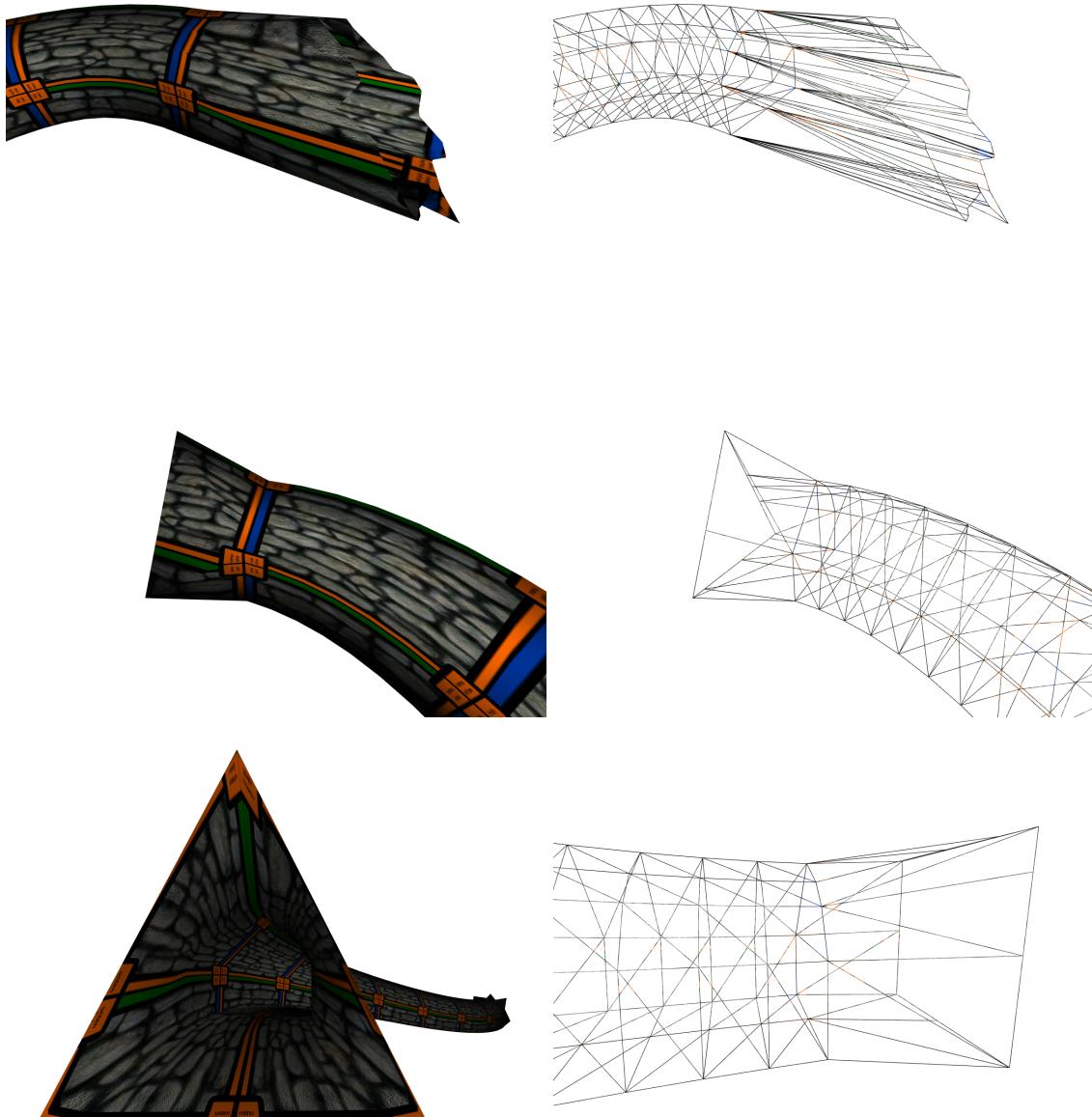


Abbildung 6.5.: *Adapter aus verschiedenen Andockstellen*: v.l.n.r. & v.o.n.u. (a)+(b) sternförmige Andockstelle, (c)+(d) Dreieck als Andockstelle, (e) Blick in einen Gang mit einem Adapter aus einer dreieckigen Andockstelle, (f) Viereck als Andockstelle

6. Konzept der Gänge und Räume

Allerdings unterstützt das für den Geometrieexport verwendete Wavefront OBJ-Format, wie auch viele andere 3D-Mesh-Formate, nur ein Paar Texturkoordinaten pro Vertex. Deswegen müssen die zweiten Texturkoordinaten beim Reimport der Daten neu erzeugt werden. Hierfür ist es erforderlich, die Vertices in Andockstellenseitige-Vertices und Gangsegmentseitige-Vertices unterteilen zu können.

Die Andockstellenseitigen-Vertices unterscheiden sich in ihrer bestehenden Y-Texturkoordinate vom ersten und letzten Vertex des Gangs. Die Gangsegmentseitigen-Vertices des Adapters tun dies dagegen nicht, da sie exakte Kopien der ersten bzw. letzten Gangvertices darstellen. Somit ist eine exakte Zuordnung möglich.

6.1.3. Objekte in Gängen

In Dungeons gibt es oft eine Vielzahl von Detailobjekten. Dies können verzierende Objekte wie z.B. Fackeln oder Statuen sein, aber auch interaktive Objekte wie Monster und Fallen. Gesucht ist eine Methode, Objekte entlang von Gängen zu platzieren.

Für das Platzieren von Objekten wird die Gangkurve $P(t)$ äquivalent zu Algorithmus 5 abgelaufen, nur ohne die Generierung von Vertices und Dreiecken. Stattdessen werden die Objekte postiert. Als Abstand zwischen zwei Gangsegmenten w wird exakt der gleiche Abstand wie bei der Ganggenerierung verwendet, um genau den gleichen Kurvenverlauf zu erhalten. In bestimmten, bei Bedarf zufälligen Intervallen werden nun die Objekte des gewünschten Typs platziert. Die Intervalle werden dabei in Vielfachen von Gangsegmenten gezählt.

Gegeben sei ein Ausgangsobjekt mit einer 2D-Position bezüglich des Gangquerschnitts von \vec{Q} , einer Ausgangsrotation von $(\beta_x, \beta_y, \beta_z)$ als Eulerwinkel und einer Skalierung von \vec{S} . Gesucht sind die resultierende Skalierung, die Position und die Rotation in Eulerwinkeln eines jeden auf dem Ausgangsobjekt basierenden Objekts O im Gang.

Die Skalierung bleibt unverändert und entspricht somit \vec{S} . Bei der Platzierung entlang des Gangs berechnen sich die Position $\overrightarrow{O.Pos}$ und die Rotation in Eulerwinkeln $\overrightarrow{O.Rot}$ wie in Formel 6.17 angegeben. \mathfrak{A}_M sei eine gegebene Umwandlungsfunktion für Rotationsmatrizen in Eulerwinkel $(\alpha_x, \alpha_y, \alpha_z)$, ihre Umkehrfunktion sei \mathfrak{A}_M^{-1} . R_{Kurve} sei die Rotationsmatrix, die einen Vektor $(0, 0, 1)$ nach $\overrightarrow{P'(t)}$ rotiere.

$$\begin{aligned}\overrightarrow{O.Pos} &= \overrightarrow{P(t)} - \vec{Q}.x \cdot \overrightarrow{Links} + \vec{Q}.y \cdot \overrightarrow{Oben} \\ \overrightarrow{O.Rot} &= \mathfrak{A}_M(R_{Kurve} \cdot R_{Objekt}) \\ &= \mathfrak{A}_M(R_{Kurve} \cdot \mathfrak{A}_M^{-1}(\beta_x, \beta_y, \beta_z))\end{aligned}\tag{6.17}$$

Bei der Einbindung von Detailobjekten ist die Verwendung von Token für die Repräsentation komplexerer Objekte, wie z.B. detailreiche animierte Meshes oder Partikelsysteme, sinnvoll. Die Token stellen sehr einfache Meshes mit wenigen Polygonen dar und dienen als Platzhalter. Nach dem Einladen in die jeweilige Anwendung kann das Token entsprechend ersetzt werden. Die Identifizierung des Tokens erfolgt über den Objektnamen und die Objekt-ID, die das Objekt im Szenengraphen besitzt (siehe Anhang A).

Die Vorteile der Token liegen sowohl in der schnelleren Darstellung im Generatorprogramm als auch in der Erzeugung eines geringeren Overheads beim Speichern der Szeneendatei, da ein solches Token weniger Parameter erfordert als z.B. das entsprechende

6. Konzept der Gänge und Räume

Partikelsystem. Im Dungeongenerator-Programm werden daher Detailobjekte grundsätzlich durch nichtanimierte Meshes repräsentiert, da sich diese optimal als Token eignen. Beispiele für Objekt-Token entlang von Gängen werden in Abbildung 6.6 dargestellt.

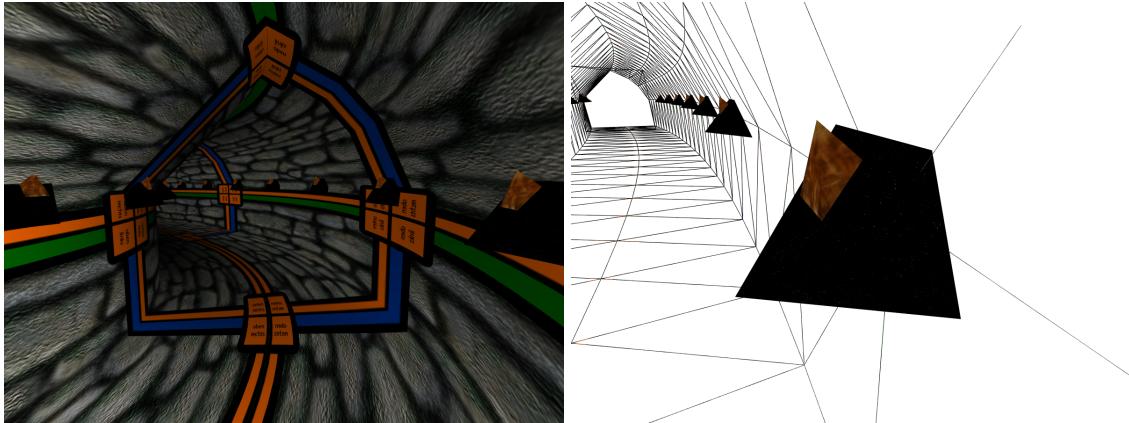


Abbildung 6.6.: *Objekte in Gängen*: v.l.n.r. (a) Objekt-Token entlang eines Gangs, (b) einzelnes Token in Nahansicht

6.2. Räume als Subszenen

Räume sollen die Integration eigener Umgebung erlauben und als fertig ausgestattete *Subszenen* in den Dungeon eingefügt werden können. Jede Subszene dient dabei als Bauvorlage für die im Dungeon platzierten Räume.

Diese Subszenen sind vorgefertigte Irrlicht-Szenen im .irr-Dateiformat. Sie sind fertig gestaltet, inklusive verzierenden und interaktiven Objekten. Es können so beliebige Szenen eingebunden werden. Prinzipiell kann auch eine Gangkreuzung oder ein extern modellierter Höhlenabschnitt als Raum definiert und verwendet werden.

Jeder Raum hat vier definierte *Andockstellen* für den Anschluss von Gängen, in Richtung Nord, Ost, Süd und West. Weiterhin gibt es für jede dieser Himmelsrichtungen zwei Geometrien: Die *Verschlussgeometrie* wird verwendet, wenn kein Gang angedockt wird. Ein Beispiel ist eine geschlossene Wand. Die *Andockgeometrie* wird verwendet, wenn ein Gang angedockt wird. Ein Beispiel hierfür ist ein Torbogen mit einer Falle am Boden.

Zur Generierung der Räume wird zuerst die Raumvorlage, also die Subszene, eingeladen und die vier Andockstellen werden ermittelt. Bei der konkreten Generierung des Dungeons werden Kopien der eingeladenen Subszenen erzeugt und mit gewünschter Transformation in den Szenengraphen eingefügt. Die Transformation wird ebenfalls auf die Andockstellen angewendet. Je nachdem in welchen Himmelsrichtungen Gänge angeschlossen werden, wird dort entweder die Verschlussgeometrie oder die Andockgeometrie entfernt. Diese Kopien stellen die finalen Räume dar. Ein Beispiel für einen Raum mit an die entsprechenden Andockstellen angeschlossenen Adapters ist in Abbildung 6.7 zu sehen.

6. Konzept der Gänge und Räume

6.2.1. Ermittlung der Andockstellen

Eine Andockstelle besteht, wie in Kapitel 6.1.2 erläutert, aus einer Folge von Vertices V_i , einem Mittelpunkt \vec{M}_a und der normalisierten Normale \vec{N}_a . Andockstellen werden aus Meshes mit einer genau definierten Randkurve generiert. Diese Meshes sind in der Subszene durch die Benennung „Nord_Andockstelle“, „Ost_Andockstelle“, „Sued_Andockstelle“ bzw. „West_Andockstelle“ definiert.

Die Randkurve besteht aus Vertices, die durch Kanten verbunden sind, die nur zu einem einzigen Dreieck gehören, d.h. Randkanten sind.⁴ \vec{M}_a sei definiert als der Nullpunkt des Meshes. Die Normale verlaufe in Richtung der Z-Achse des Meshes, wobei sowohl positive als auch negative Richtung erlaubt sind. Die finale Normalenrichtung wird so gewählt, dass die Normale von der Mitte der Subszene weg zeigt.

Für die Ermittlung der Andockstellenvertices sei der Mesh durch seine Vertexliste und Dreiecksliste gegeben, wie in Kapitel 3.3.1 erläutert. Zunächst wird die Adjazenzliste für alle Vertices aufgebaut. Diese gibt für jeden Vertex an, welche Vertices mit diesem über eine Kante verbunden sind. Für den Aufbau der Adjazenzliste wird die Dreiecksliste durchlaufen und es werden jeweils für alle drei Kanten eines Dreiecks die entsprechenden Einträge generiert. Falls ein Eintrag mehrfach vorkommt, weil eine Kante zu mehreren Dreiecken gehört, so wird dieser Eintrag als *mehrfa*ch markiert.

Nach dem Aufbau der Adjazenzliste wird ein Vertex V_i bestimmt, der mindestens einen Eintrag besitzt, welcher nicht als mehrfach markiert ist. Dieser Vertex ist ein Andockstellenvertex, also wird er zur Vertexmenge der Andockstelle hinzugefügt. Nun wird ein adjazenter Vertex zu V_i bestimmt, der zu diesem mit einer nicht-mehrfaichen Kante verbunden ist und damit fortgefahrene. So wird entlang der Vertices der Randkurve gehangelt, bis der Startvertex wieder erreicht ist. Um zu vermeiden, dass zwischen zwei Vertices hin- und hergesprungen wird, darf keine Kante mehrmals verwendet werden. Ergebnis ist die Folge der Andockstellenvertices.

Diese Andockstellenvertices sind nun entweder gegen oder im Uhrzeigersinn angeordnet. Falls sie gegen den Uhrzeigersinn angeordnet sind, wird die Reihenfolge umgekehrt. Zum Schluss werden die Andockstellenvertices, der Mittelpunkt und die Normale mittels der Transformation, mit welcher der Andockstellenmesh in die Subszene eingebunden wurde, transformiert. Die Andockstellenmeshes werden nach Ermittlung der Andockstellen aus der Subszene gelöscht.

Ausgehend von einem maximalen Knotengrad je Vertex von g , der Anzahl der Dreiecke des Andockstellenmeshs m und der Anzahl seiner Vertices n , liegt die Laufzeitkomplexität des implementierten Algorithmus in $\mathcal{O}(g^2 \cdot m + n)$. Der Aufbau der Adjazenzliste liegt dabei in $\mathcal{O}(g^2 \cdot m)$, wobei der Faktor g^2 aus der Prüfung auf mehrfache Einträge resultiert. Es wird trivial jeder neue Eintrag mit den vorhandenen verglichen. Hier ist Verbesserungspotential vorhanden. Die Ermittlung der Andockstellenvertices liegt in $\mathcal{O}(m + n)$ und die Transformation der Andockstelle in $\mathcal{O}(n)$.

Der Algorithmus ist im Gesamtkontext des Generierungsprogramms nicht sehr zeitintensiv, da er erstens relativ selten aufgerufen wird (nur beim erstmaligen Einlesen von Subszenen) und zweitens die Adaptermeshes i.d.R. vergleichsweise wenige Vertices und Dreiecke besitzen.

⁴Kanten gehören ansonsten i.d.R. zu genau zwei Dreiecken, die links und rechts der Kante liegen.

6. Konzept der Gänge und Räume

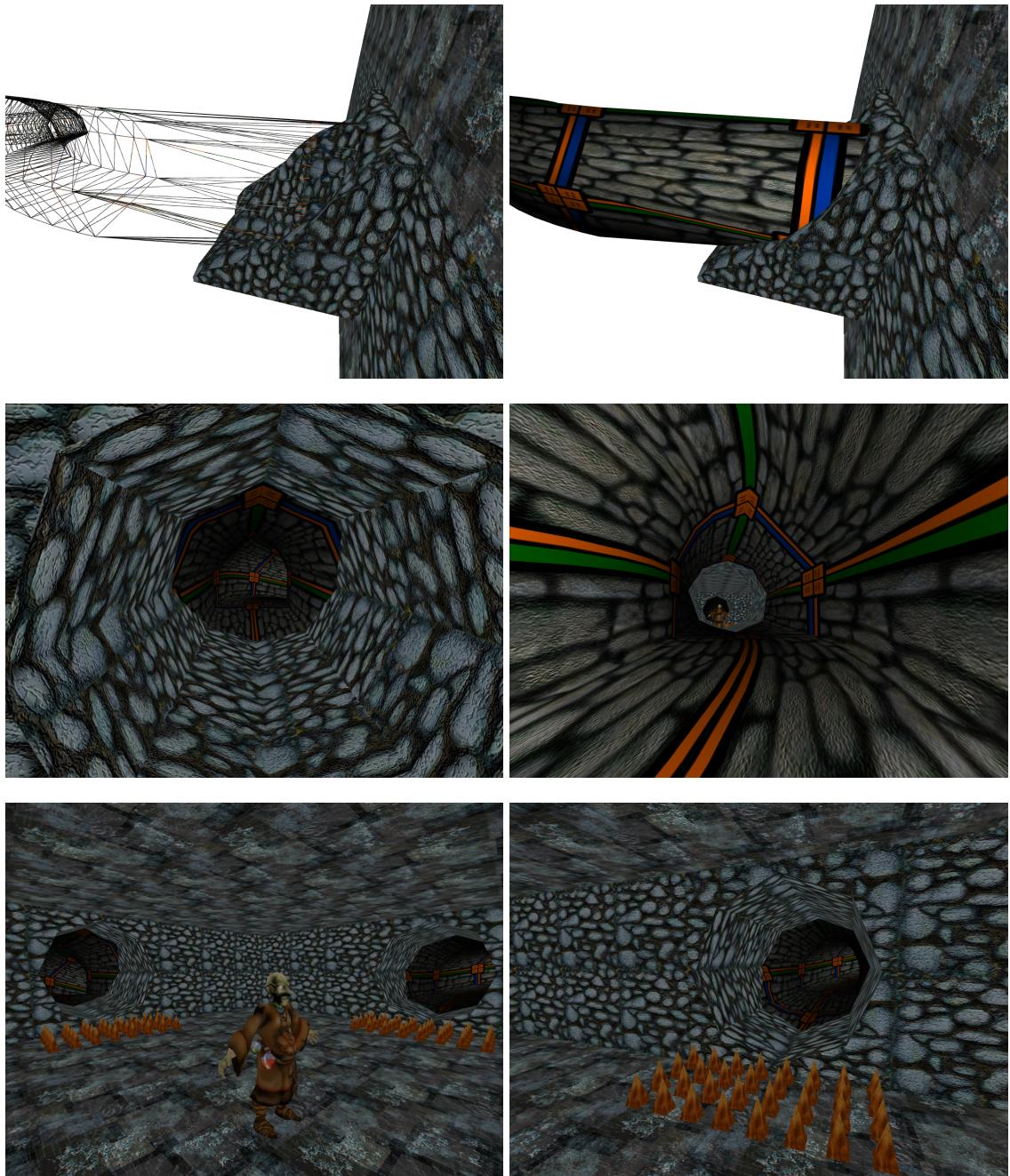


Abbildung 6.7.: Raum mit Adaptern: v.l.n.r. & v.o.n.u. (a) Drahtgitter des Adapters, (b) Adapter komplett, (c) Gang aus Raumsicht, (d) Raum aus Gangsicht, (e) Raum von innen, (f) gleiche Andockstelle wie bei (a)-(d) um 180° gedreht verwendet

7. Hinzufügen von Räumen und Gängen zum Dungeon

Aus der erzeugten Höhle soll die Konstruktion des Dungeons mittels Gängen und Räumen erfolgen. Diese sollen um die Höhle herum angeordnet sein und mit dieser verbunden werden. Das Verfahren dazu besteht aus vier Schritten.

Erster Schritt: Der Voxelraum wird in Sektoren unterteilt. Dazu ist die Erweiterung des bisherigen Voxelraums, in dem die Höhle gezeichnet werden kann, nötig. Dieser bisher betrachtete Voxelraum sei als *Normalvoxelraum* bezeichnet. Der Normalvoxelraum wird in jeder Richtung unendlich weit erweitert. Alle Voxel, die nicht im Normalvoxelraum liegen, seien grundsätzlich mit 0 belegt.

Die Aufteilung des Voxelraums erfolgt in $S_x \times S_y \times S_z$ Sektoren. Gefordert ist ein Zentralsektor, also eine ungerade Anzahl an Sektoren entlang jeder Dimension X , Y und Z .¹ Die Mitte des Zentralsektors soll dabei in der Mitte des Normalvoxelraums liegen. Es sollen mindestens zwei Sektoren Rand um den Normalvoxelraum bzgl. der X- und Z-Koordinate bestehen, damit um die Höhle herum genügend Platz für die Räume des Dungeons ist. Mit einer Sektorausdehung A entlang jeder Dimension und einer Ausdehnung des Normalvoxelraums von $V_x \times V_y \times V_z$ ergibt sich dadurch als Anzahl der Sektoren entlang einer Dimension (hier X , S_z äquivalent, bei S_y ohne Aufaddierung des Randes):

$$S_x = \begin{cases} \left\lceil \frac{V_x}{A} \right\rceil + 4 & \text{falls } \left\lceil \frac{V_x}{A} \right\rceil \text{ ungerade} \\ \left\lceil \frac{V_x}{A} \right\rceil + 5 & \text{sonst} \end{cases} \quad (7.1)$$

Ausgehend vom Zentralsektor $\left(\left\lfloor \frac{S_x}{2} \right\rfloor, \left\lfloor \frac{S_y}{2} \right\rfloor, \left\lfloor \frac{S_z}{2} \right\rfloor\right)$ berechnen sich die Grenzen eines Sektors (i, j, k) dadurch wie folgt (hier Dimension X/i , Y/j und Z/k äquivalent):

$$\begin{aligned} Min_x &= \left\lfloor \frac{V_x}{2} \right\rfloor + \left(i - \left\lfloor \frac{S_x}{2} \right\rfloor \right) \cdot A - \left\lfloor \frac{A}{2} \right\rfloor \\ Max_x &= Min_x + A - 1 \end{aligned} \quad (7.2)$$

Zwei Sektoren sind benachbart, wenn eine gemeinsame Grenzfläche entlang der X- oder Z-Achse existiert. Also hat jeder Sektor bis zu vier Nachbarsektoren: in positiver und negativer X- sowie positiver und negativer Z-Richtung. Nachbarsektoren in Y-Richtung existieren im Kontext der Dungeonkonstruktion nicht, da diese Achse die Höhenachse darstellt, über die keine Räume direkt benachbart sein sollen.

Nach der Aufteilung in Sektoren werden diese wie folgt charakterisiert:

- *Höhle:* Der Sektor enthält mindestens einen Voxel mit Belegung 1.
- *Höhlennachbar:* Der Sektor ist kein *Höhle*-Sektor, hat aber einen Nachbarsktor der mit *Höhle* charakterisiert ist.
- *Höhlennachbar2:* Der Sektor ist weder *Höhle*- noch *Höhlennachbar*-Sektor, aber ein Sektor in zwei Sektoren Abstand entlang der X- oder Z-Achse ist ein *Höhle*-Sektor.

¹Diese Anforderung beruht auf der Annahme, dass so mögliche zentrale Freiräume der Höhle besser erkannt und genutzt werden können.

7. Hinzufügen von Räumen und Gängen zum Dungeon

- *Frei*: Alle anderen Sektoren erhalten diese Charakterisierung.

Zweiter Schritt: Räume in Höhlennähe werden platziert und mit diesen verbunden. Dazu laufe über alle Sektoren. Wenn ein Sektor *Höhlnachbar* ist, so bestimme per Zufall eine beliebige Subszene. Die Möglichkeit zur Wahl keiner Subszene (die leere Subzene) existiert dabei ebenfalls. Falls eine nicht-leere Subszene bestimmt wurde, so platziere eine Instanz der Subszene als Raum im Sektor. Nun versuche von diesem Raum eine Verbindung per Gang zur Höhle herzustellen, wobei die Himmelsrichtungen in zufälliger Reihenfolge abgearbeitet werden. Falls die Verbindung nicht möglich ist, lösche den Raum wieder und markiere den Sektor mit *Frei*. Wenn die Verbindung möglich ist, markiere den aktuellen Sektor mit *Raum* und erstelle den Gang. Falls dieser Gang durch einen *Frei*-Sektor verläuft, so markiere diesen Sektor mit *Gang*. Nun erstelle, falls möglich, mit Wahrscheinlichkeit p_{gh} weitere Gänge vom Raum zur Höhle, wobei die übrigen Himmelsrichtungen verwendet werden.

Dieser Vorgang wird wiederholt, nur dieses Mal werden *Höhlnachbar2*-Sektoren verwendet. Durch diese Aufteilung in zwei Durchläufe werden Sektoren mit direkter Nachbarschaft zur Höhle priorisiert. Resultat dieses Schritts sind platzierte Räume, die direkt mit der Höhle durch Gänge verbunden sind.

Dritter Schritt: Räume neben anderen Räumen werden platziert und mit diesen verbunden. Dazu laufe über alle Sektoren und markiere alle diejenigen *Frei*-Sektoren, die Nachbarsektoren eines *Raum*-Sektors sind, mit *Raumnachbar*. Nun laufe erneut über alle Sektoren. Wenn ein Sektor mit *Raumnachbar* markiert ist, so wähle eine zufällige Subszene aus. Falls eine nicht-leere Subszene bestimmt wurde, so platziere eine Instanz der Subszene als Raum in diesem Sektor und markiere den Sektor mit *Raum*. Verbinde diesen Raum per Gang zu einem beliebigen, zufällig ausgewählten Raum in einem Nachbarsktor.

Die Markierung per *Raumnachbar* dient dazu, zu verhindern, dass gerade erstellte Räume gleich als potentielle Nachbarräume dienen können. Prinzipiell lässt sich dieser Schritt mehrfach durchführen, um ein sehr weit verzweigtes Netz aus Gängen und Räumen zu erhalten. In der Implementierung des Dungeongenerators wird nur ein Durchlauf verwendet. Resultat des Schritts sind platzierte Räume, die nur mit anderen Räumen, aber nicht direkt mit der Höhle verbunden sind. Dennoch sind alle Räume zumindest indirekt mit der Höhle verbunden.

Vierter Schritt: Weitere Gänge zwischen Räumen werden eingefügt. Laufe über alle Sektoren und prüfe, ob ein Sektor mit *Raum* markiert ist und der Sektor in positiver X-Richtung ebenfalls ein *Raum*-Sektor ist. Weiterhin überprüfe, ob noch kein Gang zwischen den Räumen der beiden Sektoren existiert. Bei Erfüllung der Bedingungen füge zwischen beiden Sektoren mit der Wahrscheinlichkeit p_{gr} einen Gang ein. Führe den gleichen Test in positiver Z-Richtung durch, auch hier wird entsprechend ein Gang eingefügt.

Durch diesen Schritt werden alle potentiellen Gangverbindungen zwischen den bestehenden Räumen des Dungeons getestet und per Zufall neue Gänge eingefügt. Resultat sind zusätzliche Gangverbindungen zwischen den Räumen des Dungeons. Dieser besitzt nun alle seine Gänge und Räume.

7. Hinzufügen von Räumen und Gängen zum Dungeon

7.1. Platzierung von Räumen und Verbindung durch Gänge

Nach Auswahl einer Subszene erfolgt die Platzierung des Raums im entsprechenden Sektor. Die Ausdehnung des Sektors sei $[Min_x, Max_x] \times [Min_y, Max_y] \times [Min_z, Max_z]$ mit einer Länge von A in jeder Dimension.

Zur Berechnung der Position des Raums wird von der Mitte des Sektors ausgegangen, damit in jeder Richtung möglichst viel Platz für die Gänge bleibt. Die Höhenkoordinate y wird per Zufall modifiziert, welches zu Formel 7.3 führt. Die Funktion Zufalls Wert liefert einen Zufalls Wert im angegebenen Intervall.

$$\begin{aligned}\overrightarrow{Pos.x} &= 0,5 \cdot (Min_x + Max_x) \\ \overrightarrow{Pos.y} &= Zufalls Wert[Min_y + 0,25 \cdot A, Max_y - 0,25 \cdot A] \\ \overrightarrow{Pos.z} &= 0,5 \cdot (Min_z + Max_z)\end{aligned}\quad (7.3)$$

Die Bestimmung der Rotation erfolgt ebenfalls per Zufallselement. Es findet nur eine Rotation um die Höhenachse Y statt, damit die Räume stets waagerecht bleiben.

$$\overrightarrow{Rot.y} = Zufalls Wert(0^\circ, 360^\circ) \quad (7.4)$$

Beim Verbinden per Gang wird dieser an derjenigen Andockstelle angedockt, die in Richtung des anderen Raums bzw. des entsprechenden Höhlenbereichs zeigt. Die Rotation des Raums erfordert hierzu eine Neubelegung der Himmelsrichtungen der Andockstellen. Das Anpassen dieser Himmelsrichtungen H wird folgendermaßen vorgenommen, mit $Nord \equiv 0 \equiv X \text{ positiv}, Ost \equiv 1 \equiv Z \text{ negativ}, Sued \equiv 2 \equiv X \text{ negativ}, West \equiv 3 \equiv Z \text{ positiv}$:

$$H_{neu} = \begin{cases} (H_{alt} + 1) \bmod 4 & \text{falls } 45^\circ < \overrightarrow{Rot.y} \leq 135^\circ \\ (H_{alt} + 2) \bmod 4 & \text{falls } 135^\circ < \overrightarrow{Rot.y} \leq 225^\circ \\ (H_{alt} + 3) \bmod 4 & \text{falls } 225^\circ < \overrightarrow{Rot.y} \leq 315^\circ \\ H_{alt} & \text{sonst} \end{cases} \quad (7.5)$$

Gänge aus Andockstellen

Die Erstellung eines Gangs erfolgt von einer Andockstelle 1 zu einer anderen Andockstelle 2. Die Normale von Andockstelle 1 sei $\overrightarrow{N_a}$, der Mittelpunkt $\overrightarrow{M_a}$ und der maximale Abstand eines Vertex der Andockstelle zur Ebene, mit $\overrightarrow{M_a}$ als Punkt der Ebene und $\overrightarrow{N_a}$ als Normale der Ebene, sei W_a . Die Breite des Gangs sei B . Ein Faktor g für die Stärke der Gangrichtung ist als Parameter im Generatorprogramm festlegbar. Für die Erstellung des Gangs werden die Parameter $\overrightarrow{P_1}$ und $\overrightarrow{P'_1}$ wie folgt bestimmt:

$$\begin{aligned}\overrightarrow{P_1} &= \overrightarrow{M_a} + \overrightarrow{N_a} \cdot (W_a + B) \\ \overrightarrow{P'_1} &= g \cdot A \cdot \overrightarrow{N_a}\end{aligned}\quad (7.6)$$

Die Einbeziehung von W_a in die Positions berechnung verhindert ein mögliches Ineinanderschieben von Adapter und Gang. Durch Aufaddieren von B wird genügend Platz für die Überblendung der Querschnitte des Adapters erzeugt. Die Einbeziehung der Sektorausdehnung A in die Ableitung des Gangsplines bewirkt eine Art Skalierung des Gangs mit dieser. $\overrightarrow{P_2}$ und $-\overrightarrow{P'_2}$ werden auf die gleiche Art und Weise von Andockstelle 2 bestimmt.

7. Hinzufügen von Räumen und Gängen zum Dungeon

Da die erstellten Andockstellen initial in Richtung Gang zeigen, wird für die Erstellung des Adapters an \vec{P}_2 des Gangs die Normale von Andockstelle 2 mit -1 multipliziert und die Reihenfolge der Andockstellenvertices invertiert, was einer Spiegelung der Andockstelle entspricht.

7.2. Test für Andocken an Höhle

Der Platzierungsalgorithmus für Räume und Gänge testet in Schritt 2, ob Verbindungen zur Höhle möglich sind. Als Bedingungen für das Andocken eines Gangs an die Höhle muss die lokale Struktur der Höhle den Anschluss eines Gangs ermöglichen. Auch darf der Gang nicht mit der Höhle, Räumen oder anderen Gängen kollidieren.

Als Hilfsmittel für die notwendigen Tests und den eigentlichen Andockprozess werden *Scankarten* verwendet. Diese ermöglichen den schnellen Zugriff auf die relevanten Informationen der umliegenden Höhlenteile. Das Prinzip einer Scankarte ist in Abbildung 7.1(a) dargestellt. Alle *Höhlenachbar-* und *Höhlenachbar2-Sektoren* scannen in allen vier Himmelsrichtungen den Voxelraum. Der Scan erfolgt zwei Sektoren tief, um in jedem Fall Voxel mit Belegung 1 zu finden. Der Bereich des eigenen Sektors wird ebenfalls in den Scan einbezogen, da die Scankarte auf Grund diverser Tests etwas breiter und höher als der eigentliche Sektor ist. Somit werden die entsprechenden Teile der Nachbarspektoren mitgescannt.

Der Scan bildet ein Tiefenprofil der Voxelhöhle bezogen auf die jeweilige Richtung. Jeder Eintrag der Scankarte berechnet sich wie in Formel 7.7 (Beispiel für Scan in positiver X-Richtung, andere Richtungen äquivalent) angegeben. $Offset_y$ und $Offset_z$ sind Offsets, die sich aus den Sektorgrenzen berechnen und die Abbildung der Scankartenkoordinaten auf Voxelkoordinaten ermöglichen. Falls kein entsprechendes k existiert, wird der Eintrag auf ∞ (bzw. $-\infty$, wenn Scan in negativer Richtung) gesetzt.

$$\begin{aligned} Scan_{x+}(i, j) = \min(k) : Voxel(k, j + Offset_y, i + Offset_z) = 1 \\ \wedge k \in [Min_x, Max_x + 2 \cdot A] \end{aligned} \quad (7.7)$$

Jeder Eintrag einer Scankarte besitzt ein Flag *valide*, welches angibt, ob der Eintrag noch gültig ist und verwendet werden kann. Anfangs sind alle Einträge valide.

Für den Andockversuch in Himmelsrichtung H wird zunächst ein zufälliger Startindex für die Scankarte von H bestimmt. Dann wird getestet, ob ein Andocken an dieser Position möglich ist. Wenn ja, wird angedockt, wenn nein, wird der Index weitergezählt. Dies wird solange fortgeführt, bis alle Indizes geprüft wurden oder eine gültige Andockposition gefunden wurde.

7.2.1. Filterung der Scankarten

Der Bereich der Höhle, an dem der Gang andockt, sollte keine zu starken Tiefenunterschiede besitzen, also möglichst eben sein. Gesucht ist somit ein Bereich auf der Scankarte mit möglichst kleinen Abweichungen der Einträge. Weiterhin müssen 1-Voxel, d.h. Teile der Höhle, in gesamten Andockbereich vorhanden sein. Nicht zulässig sind folglich Einträge mit dem Wert $-\infty$ oder ∞ .

Beim Test der Andockposition auf der Scankarte wird ein Bereich auf der Scankarte der Breite und Höhe $B + 2$ untersucht, also der Gangbreite mit einem Voxel Rand in jeder Richtung. Der Rand wird verwendet, um potentielle Kollisionen zwischen Adapter und

7. Hinzufügen von Räumen und Gängen zum Dungeon

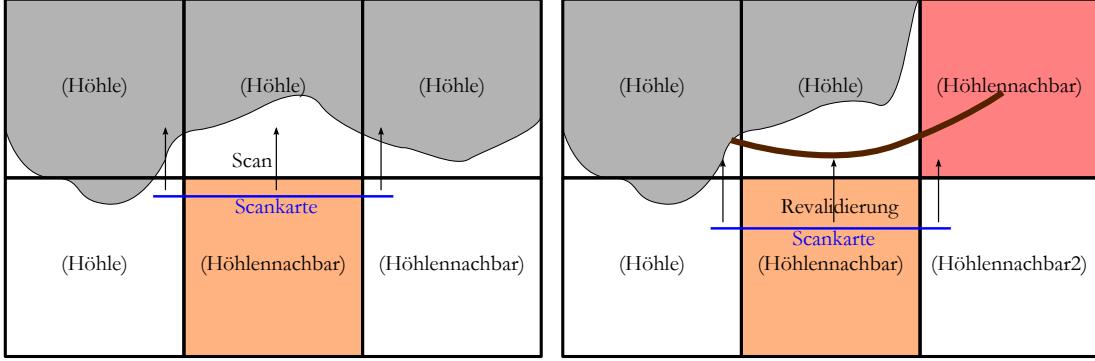


Abbildung 7.1.: Generierung und Revalidierung von Scankarten: v.l.n.r.

(a) Generierung: Gescannt wird von Höhlennachbar- und Höhlennachbar2-Sektoren in Richtung Höhlen-Sektoren. Scannender Sektor ist orange markiert.

(b) Revalidierung: Durch den Gang (braun) vom roten Sektor muss eine Scankarte (blau) vom orangenen Sektor revalidiert werden, da der Gang Teile der Höhle verdeckt.

Höhle zu vermeiden, der mögliche Gang würde den Bereich $[i + 1, i + B] \times [j + 1, j + B]$ belegen. Als maximale Tiefendifferenz für das Andocken wird die Gangbreite B verwendet, wodurch die Toleranz für den Test mit B skaliert. Jeder Eintrag der Scankarte im Bereich muss valide sein. Als Gültigkeitstest für eine Andockposition (i, j) ergibt sich somit:

$$Gültig(i, j) = \begin{cases} ja & \text{falls } Max_{scan} - Min_{scan} \leq B \\ & \wedge Min_{scan} > -\infty \wedge Max_{scan} < \infty \\ & \wedge \neg \exists Scan(a, b) \equiv invalide \\ nein & sonst \end{cases} \quad (7.8)$$

$$\text{mit } Min_{scan} = \min(Scan(a, b)) \wedge Max_{scan} = \max(Scan(a, b)) : \\ a \in [i, i + B + 1] \wedge b \in [j, j + B + 1]$$

Im Allgemeinen sind Gänge in Dungeons in Bodennähe mit der Höhle verbunden, andernfalls kann ein Begehen des Gangs unmöglich sein. Dies führt zu der Idee, den Scantest um den Test der Bodennähe zu erweitern. Der Boden wird durch eine große Tiefe auf der Scankarte im Vergleich zum Bereich darüber gekennzeichnet. Auf der Scankarte unterhalb des Andockbereiches muss also eine größere Tiefe vorliegen als im Andockbereich.

Der maximaler Abstand zum Boden sei R , die gewünschte Minimaltiefe für den Test sei F . Die Andockposition (i, j) sei gültig bzgl. des Abstands zum Boden wenn Formel 7.9 wahr ergibt. Getestet wird hierbei der Bereich direkt unter dem Gang. Min_{scan} und Max_{scan} seien die durch den obigen Test bereits ermittelten Werte. Das Gesamtverfahren ist in Abbildung 7.2 illustriert.

Test bei positiver Achsenrichtung :

$$\exists b \in [j - R, j - 1] (\forall a \in [i + 1, i + B] (Scan(a, b) - Max_{scan} \geq F)) \quad (7.9)$$

Test bei negativer Achsenrichtung :

$$\exists b \in [j - R, j - 1] (\forall a \in [i + 1, i + B] (Min_{scan} - Scan(a, b) \geq F))$$

7. Hinzufügen von Räumen und Gängen zum Dungeon

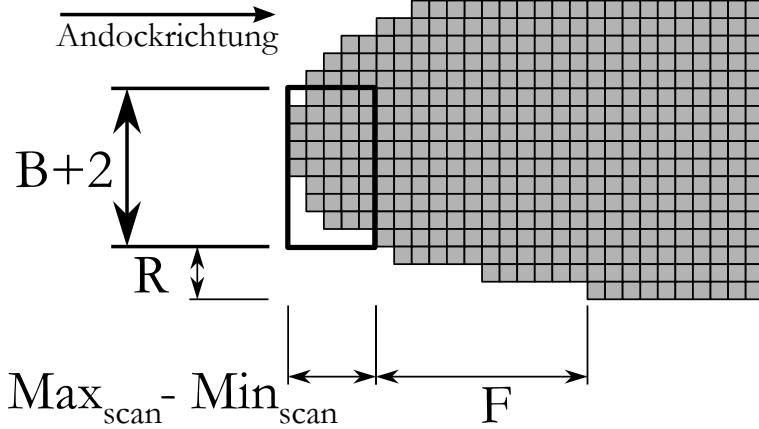


Abbildung 7.2.: *Bestimmung einer gültigen Andockposition*: gezeigt wird die Seitenansicht eines Höhlenteils: Im Bereich von $B+2$ wird getestet ob $\text{Max}_{\text{scan}} - \text{Min}_{\text{scan}}$ klein genug ist. R und F dienen dem Test, ob die Andockposition in Bodennähe ist.

Revalidierung von Scankarten-Einträgen

Neu erzeugte Gänge und Räume können Höhlenteile für andere Sektoren verdecken. Dies ist in Abbildung 7.1(b) dargestellt. Also müssen nach der Erzeugung neuer Gänge und Räume betroffene Scankarten überprüft und ggf. Teile mit *invalid* markiert werden.

Für den Test wird die Bounding Box des Gangs plus dem Sektor des erstellten Raums als Konfliktgebiet $[K_{\min X}, K_{\max X}] \times [K_{\min Y}, K_{\max Y}] \times [K_{\min Z}, K_{\max Z}]$ genutzt.

Die Erklärung erfolgt am Beispiel für Scankarten in positiver X-Richtung. Ein Sektor ist potentiell betroffen, wenn folgende Bedingungen zutreffen:

1. Überschneidung der Scankarte mit $[K_{\min Y}, K_{\max Y}] \times [K_{\min Z}, K_{\max Z}]$
2. Grenze des Sektors $\text{Min}_x \leq K_{\max X}$, also Überschneidung bzgl. der X-Koordinate

Für diese Sektoren werden die Scaneinträge überprüft. Ein Eintrag ist *invalid*, wenn er im oder hinter dem Konfliktgebiet liegt, d.h. durch das Konfliktgebiet verdeckt wird:

$$\text{Scan}_{x+}(i, j) \equiv \begin{cases} \text{invalid} & \text{falls } K_{\min X} \leq \text{Scan}_{x+}(i, j) \\ \text{valide} & \text{sonst} \end{cases} \quad (7.10)$$

Durch die Revalidierung der Scankarten werden potentielle Kollisionen der Gänge mit Räumen und anderen Gängen vermieden.

7.2.2. Test auf potentielle Kollision von Gang und Höhle

Da Gänge und Höhle potentiell kollidieren können, ist ein Kollisionstest Höhle \Leftrightarrow Gang vor dem Erstellen neuer Gänge erforderlich. Falls der Test positiv verläuft, soll kein Gang erstellt werden, da eine potentielle Kollision vorliegt. Ziel des Tests ist eine *false negative*-Rate von 0, also soll jede Kollision entdeckt werden. Für einen schnellen Test sei eine *false*

7. Hinzufügen von Räumen und Gängen zum Dungeon

positive-Rate > 0 erlaubt. Der Adapter kann durch seine Konstruktion nach Kapitel 7.3 nicht mit der Höhle kollidieren und muss daher nicht in den Test einbezogen werden.

Ein Gang soll in Himmelsrichtung H an die Höhle andocken. Im ersten Ansatz wird die Bounding Box des neuen Gangs genutzt und getestet, ob Elemente der H -Scankarte in die Bounding Box hineinreichen oder sogar davor liegen. In diesem Fall könnte der Gang in die Höhle ragen. Problematisch ist, dass die Bounding Box keine besonders exakte Approximation des Gangs darstellt. Potential für neue Gänge wird hier verschenkt, da die *false positive*-Rate zu hoch ist. Eine bessere Approximation wäre wünschenswert.

Wie in [Sal06, S.210, 425] beschrieben wird, lässt sich ein kubischer Hermite-Spline durch eine kubische Bézierkurve darstellen. Eine Bézierkurve ist nach [Sal06, S.178 ff.] durch Formel 7.11 gegeben. Die $B_{n,i}(t)$ werden Bernsteinpolynome genannt. Die Punkte \vec{Q}_i werden als Kontrollpunkte der Kurve bezeichnet.

$$\vec{Q}(t) = \sum_{i=0}^n \left(B_{n,i}(t) \cdot \vec{Q}_i \right) \quad \text{mit } B_{n,i}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad t \in [0, 1] \quad (7.11)$$

Die Bézierkurve für vier Punkte ergibt sich dadurch als:

$$\vec{Q}(t) = (1-t)^3 \vec{Q}_0 + 3t(1-t)^2 \vec{Q}_1 + 3t^2(1-t) \vec{Q}_2 + t^3 \vec{Q}_3 \quad (7.12)$$

Setzt man die Parameter $\vec{P}_1, \vec{P}_2, \vec{P}'_1, \vec{P}'_2$ des Hermite-Splines auf:

$$\vec{P}_1 = \vec{Q}_0, \quad \vec{P}_2 = \vec{Q}_3, \quad \vec{P}'_1 = 3(\vec{Q}_1 - \vec{Q}_0), \quad \vec{P}'_2 = 3(\vec{Q}_3 - \vec{Q}_2) \quad (7.13)$$

So erhält man eingesetzt in Formel 6.5:

$$\begin{aligned} \vec{P}(t) &= (2t^3 - 3t^2 + 1)\vec{Q}_0 + (-2t^3 + 3t^2)\vec{Q}_3 \\ &\quad + (t^3 - 2t^2 + t)3(\vec{Q}_1 - \vec{Q}_0) + (t^3 - t^2)3(\vec{Q}_3 - \vec{Q}_2) \\ &= (1-t)^3 \vec{Q}_0 + 3t(1-t)^2 \vec{Q}_1 + 3t^2(1-t) \vec{Q}_2 + t^3 \vec{Q}_3 \\ &= \vec{Q}(t) \end{aligned} \quad (7.14)$$

Somit ist die Gleichheit von Hermite-Spline und Bézierkurve gegeben. Stellt man Formel 7.13 um, so erhält man als Punkte $\vec{Q}_0, \vec{Q}_1, \vec{Q}_2, \vec{Q}_3$ der Bezierkurve:

$$\vec{Q}_0 = \vec{P}_1, \quad \vec{Q}_1 = \vec{P}_1 + \frac{\vec{P}'_1}{3}, \quad \vec{Q}_2 = \vec{P}_2 - \frac{\vec{P}'_2}{3}, \quad \vec{Q}_3 = \vec{P}_2 \quad (7.15)$$

Als Eigenschaften der Bernsteinpolynome nach [ESK96, S.134] gelten: Bernsteinpolynome im Intervall $[0, 1]$ sind eine Partition der 1 und stets positiv.

$$\begin{aligned} \forall t \in [0, 1] : \sum_{i=0}^n B_{n,i}(t) &= 1 \\ \forall t \in [0, 1] : B_{n,i}(t) &\geq 0 \end{aligned} \quad (7.16)$$

7. Hinzufügen von Räumen und Gängen zum Dungeon

Nach [O'R94, S.71-72] wird eine konvexe Hülle wie in Definition 8 definiert. Die Eigenschaften der k_i entsprechen genau denen der Bernsteinpolynome. Somit verläuft eine Bézierkurve bzw. der entsprechende Hermite-Spline innerhalb der konvexen Hülle der Punkte $\vec{Q}_0, \vec{Q}_1, \vec{Q}_2, \vec{Q}_3$. Diese Eigenschaft kann für den Kollisionstest genutzt werden. Wenn die Höhle nicht mit einer den Gang umschließenden Hülle kollidiert, dann auch nicht mit dem Gang selbst.

Definition 8 (Konvexe Hülle)

Eine Konvexitätskombination von Punkten $\vec{Q}_0, \vec{Q}_1, \vec{Q}_2, \dots, \vec{Q}_n$ ist definiert als:

$$\sum_{i=0}^n (k_i \cdot \vec{Q}_i) \quad \text{mit } \sum_{i=0}^n k_i = 1 \wedge \forall i : k_i \geq 0 \quad (7.17)$$

Die konvexe Hülle dieser Punkte ist die Menge aller ihrer Konvexitätskombinationen.

Schneller Kollisionstest

Ziel ist das schnelle Testen aller Scankarteneinträge auf die Kollision mit dem Gang. Die konvexe Hülle von $\vec{Q}_0, \vec{Q}_1, \vec{Q}_2, \vec{Q}_3$ umschließt nur den Gangspline, nicht den Gangs selbst. Also ist ein Hinzurechnen eines Abstands G zur konvexen Hülle notwendig. Dieser berechnet sich aus dem maximalen Gangradius plus Sicherheitsabstand nach Formel 7.18. Die Gangbreite sei B , der Sicherheitsabstand sei ϵ . Parameter ϵ sollte auf Grund der Koordinatenverwacklung der Höhlenvertices mindestens 0,5 betragen. In der Implementierung wurde $\epsilon = 1,0$ gewählt.

$$G = \sqrt{2} \cdot 0,5 \cdot B + \epsilon = \frac{1}{\sqrt{2}} \cdot B + \epsilon \quad (7.18)$$

Im Folgenden sei das Beispiel für das Andocken in positiver X-Richtung aus Raumsicht gegeben, d.h. der Gang verläuft von der Höhle aus in negativer X-Richtung. Das Prinzip des Tests wird in Abbildung 7.3 aufgezeigt. Es sei vorausgesetzt, dass $\vec{Q}_3.x < \vec{Q}_0.x$ und $\vec{Q}_2.x < \vec{Q}_0.x$ gilt, anderenfalls ist die Andockposition ungültig. $\vec{Q}_1.x < \vec{Q}_0.x$ gilt nach Konstruktion (siehe Abschnitt 7.3) immer und braucht nicht getestet zu werden. Für den Test wird eine Pyramide konstruiert, um die konvexe Hülle von $\vec{Q}_0, \vec{Q}_1, \vec{Q}_2, \vec{Q}_3$ zu umschließen. Die Ansteige $m_{z+}, m_{z-}, m_{y+}, m_{y-}$ aller vier Seitenflächen der Pyramide werden berechnet. Formel 7.19 gibt das Beispiel für m_{y+} an, die anderen Ansteige werden äquivalent ermittelt. Für das Berechnen der Ansteige werden die Richtungsvektoren $\vec{Q}_2 - \vec{Q}_0$ und $\vec{Q}_3 - \vec{Q}_0$ verwendet. Der Richtungsvektor $\vec{Q}_1 - \vec{Q}_0$ zeigt per Konstruktion in Richtung $(-1, 0, 0)$ und entspricht somit dem Anstieg $-\infty$.

$$\vec{r}_1 = \vec{Q}_2 - \vec{Q}_0, \vec{r}_2 = \vec{Q}_3 - \vec{Q}_0$$

$$m_{y+} = \begin{cases} \max(\vec{r}_1.y / \vec{r}_1.x, \vec{r}_2.y / \vec{r}_2.x) & \text{falls } \vec{r}_1.y > 0 \wedge \vec{r}_2.y > 0 \\ \vec{r}_1.x / \vec{r}_1.y & \text{falls } \vec{r}_1.y > 0 \wedge \vec{r}_2.y \leq 0 \\ \vec{r}_2.x / \vec{r}_2.y & \text{falls } \vec{r}_1.y \leq 0 \wedge \vec{r}_2.y > 0 \\ -\infty & \text{sonst} \end{cases} \quad (7.19)$$

Jeder Eintrag $Scan_{x+}(i, j)$ der Scankarte wird getestet, ob er innerhalb der auf die Scankarte projizierten Bounding Box $[K_{minY}, K_{maxY}] \times [K_{minZ}, K_{maxZ}]$ der konvexen Hülle, in

7. Hinzufügen von Räumen und Gängen zum Dungeon

jeder Dimension um G erweitert, liegt. Alle Scankarteneinträge, die darin liegen, werden nun in einem zweiten Test auf ihre Position bzgl. der Pyramide getestet. Je nach dem, ob diese Einträge links, rechts, oben oder unten vom Gangdockbereich liegen, wird die dortige Tiefe der Pyramide unter Verwendung der jeweiligen Anstiege berechnet.

Formel 7.20 gibt ein Beispiel für einen Punkt (i, j) auf der Scankarte an, der in positiver Z- und positiver Y-Richtung vom Andockbereich liegt. Parameter $i + Offset_z$ ergibt die Voxelkoordinate z des Scaneintrags, $j + Offset_y$ die Voxelkoordinate y . Die Position der Pyramidenspitze wird durch \vec{Q}_0 definiert. P_h ist die resultierende Tiefe der Pyramide. Durch die Einbeziehung von G resultiert ein Pyramidenstumpf als getestete Hülle.

$$\begin{aligned} P_{hy} &= m_{y+} \cdot (j + Offset_y - \vec{Q}_0.y - G) \\ P_{hz} &= m_{z+} \cdot (i + Offset_z - \vec{Q}_0.z - G) \\ P_h &= \vec{Q}_0.x + G + \min(P_{hy}, P_{hz}, 0) \end{aligned} \quad (7.20)$$

Diese Tiefe wird mit dem Wert auf der Scankarte verglichen. Ist der Scankartenwert kleiner, so liegt möglicherweise ein Teil der Höhle innerhalb des Pyramidenstumpfes. Dadurch ergibt sich eine potentielle Kollision und der Gang wird nicht erstellt.

$$Kollision(i, j) = \begin{cases} ja & Scan_{x+}(i, j) < P_h \\ nein & sonst \end{cases} \quad (7.21)$$

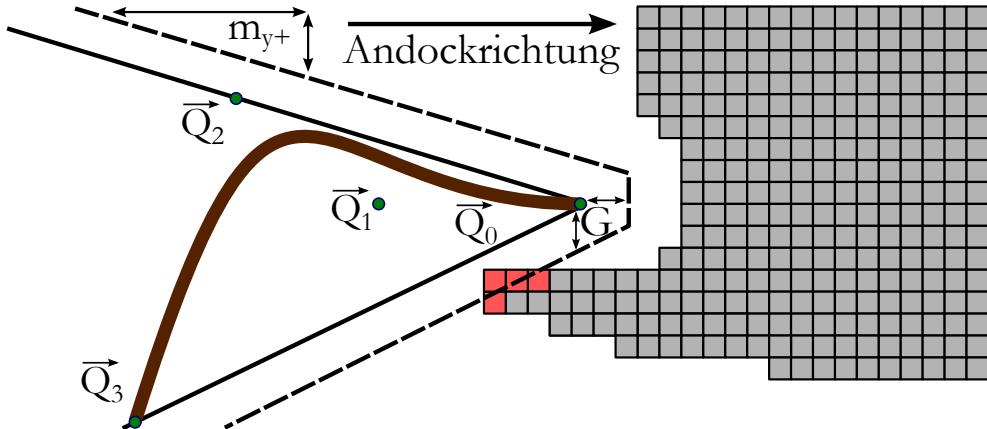


Abbildung 7.3.: *Test der konvexen Hülle per Pyramide:* gezeigt wird die Seitenansicht eines Höhlenteils: Um die konvexe Hülle der Kontrollpunkte $\vec{Q}_0, \vec{Q}_1, \vec{Q}_2, \vec{Q}_3$ des braunen Gangs wird eine Pyramide gelegt. Die roten Voxel liegen im Kollisionsgebiet.

Anmerkung: \vec{Q}_0 liegt ein Stück von der Höhle entfernt, da sich zwischen Gang und Höhle noch der Adapter befindet.

Als Verbesserungsmöglichkeit für einen exakteren Test könnte man, statt der Verwendung einer Pyramide, um die konvexe Hülle zu umschließen, die konvexe Hülle direkt für den Kollisionstest verwenden. Die verwendete konvexe Hülle der Kontrollpunkte lässt sich weiter verfeinern: Durch den de-Casteljau-Algorithmus [ESK96, S.136] können weitere Kontrollpunkte errechnet werden. Dieser lässt sich prinzipiell beliebig oft anwenden. Mit jeder Anwendungsstufe erhält man mehr Kontrollpunkte als in der vorherigen. Zudem liegen

7. Hinzufügen von Räumen und Gängen zum Dungeon

diese Kontrollpunkte immer näher am eigentlichen Splineverlauf. Somit approximiert die konvexe Hülle dieser neu errechneten Punkte die konvexe Hülle des Gangsplines genauer. Diese Verbesserungen sind allerdings mit höherem Rechenaufwand verbunden.

7.3. Andocken an Höhle

Beim Andocken des Gangs an die Höhle muss diese angepasst werden, um eine Öffnung für den Gang zu bieten. Des Weiteren ist eine Andockstelle für den Gang zu erstellen. Das Andockverfahren beschreibt Algorithmus 7 und ist in Abbildung 7.4 skizziert. Angegeben ist das Beispiel für das Andocken in positiver X-Richtung aus Raumsicht, die anderen Richtungen werden äquivalent behandelt.

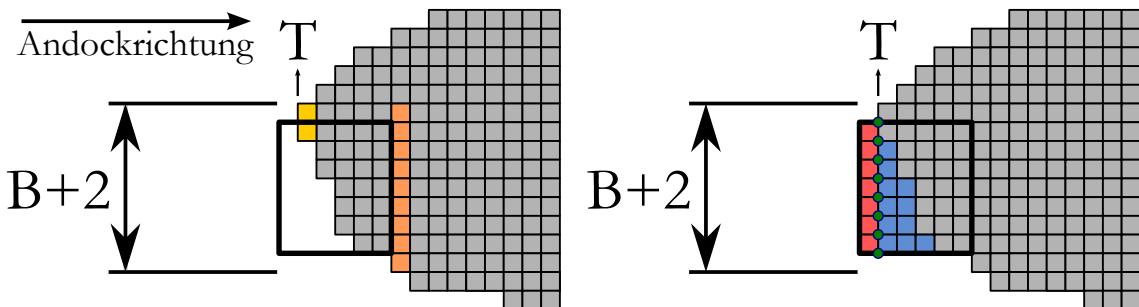


Abbildung 7.4.: *Andocken an Höhle*: gezeigt wird die Seitenansicht eines Höhlenteils: v.l.n.r
 (a) Höhle vor dem Andocken mit Min_{scan} (gelb) und Max_{scan} (orange),
 (b) Höhle nach dem Andocken mit Markierung 10 (rot), ausgefrästes Stück (blau) und Andockstellenvertices (grün)

Zuerst werden die Grenzen des zu ändernden Bereich des Voxelraums bestimmt. Diese liegen im Gangbereich und Min_{scan} , Max_{scan} , um einen Voxel verschoben.

Dann wird der Randbereich des Andockgebietes untersucht, welcher den zukünftigen Gang höhlenseitig umschließt. Die minimale X-Koordinate dieses Bereiches wird bestimmt. Diese Koordinate stellt eine Grenze T dar.² Alles das, was im Gangbereich weiter weg ist, wird „freigeschaufelt“ bzw. ausgefräst. Dieses Ausfräsen findet bis zur Höhlengrenze statt, welche durch die Scankarte bekannt ist. Es wird hier kein Bereich innerhalb der bestehenden Höhle verändert, um dortige Strukturen nicht zu entfernen.

Der nähere Bereich bzgl. der Grenze wird als Andockstelle im Voxelraum markiert, mit einer Belegung von 10.³ Dadurch sind im Gangbereich keine $0 \Leftrightarrow 1$ -Übergänge mehr vorhanden, wodurch dort keine Dreiecke gebildet werden (vgl. Kapitel 5). Es entsteht an dieser Stelle also eine Öffnung der Höhle.

Dann erfolgt die Bildung der Andockstelle. Deren Mittelpunkt liegt in der Mitte des Gangbereiches und auf Höhe der Ausfräsgrenze T . Die Normale zeigt in die negative X-Richtung, also in Richtung des Sektors, aus dem der Gang kommt. Die Vertices der Andockstelle werden im Uhrzeigersinn bezüglich der Normalenrichtung angeordnet und

²Dieser Wert T wird in der eigentlichen Implementierung schon mit Min_{scan} und Max_{scan} zusammen ermittelt.

³Prinzipiell sind andere Werte ungleich 0, 1 oder 2 möglich. Der Wert 10 ist aus diversen berechnungs-technischen Gründen vorteilhafter als niedrigere Werte.

7. Hinzufügen von Räumen und Gängen zum Dungeon

Algorithmus 7 Andocken Gang an Höhle, in positiver X-Richtung

Eingabe: Voxelraum, Scankarte, Min_{scan}, Max_{scan} ,
Andockbereich $[i, i + B + 1] \times [j, j + B + 1]$ mit Gangbreite B

Ausgabe: geändertes Voxelgebiet, Andockstelle für Gang

```

1:  $X_{min} \leftarrow Min_{scan} - 1$                                 ▷ Grenzen des zu ändernden Voxelgebietes
2:  $X_{max} \leftarrow Max_{scan} - 1$ 
3:  $Y_{min} \leftarrow j + Offset_y + 1$ 
4:  $Y_{max} \leftarrow Y_{min} + B - 1$                                 ▷ Bereich in Gangbreite
5:  $Z_{min} \leftarrow i + Offset_z + 1$ 
6:  $Z_{max} \leftarrow Z_{min} + B - 1$ 
7:
8:  $T \leftarrow \infty$                                               ▷ Suche nach nahestem Gangrandstück
9: for all  $(a, b) \in [i, i + B + 1] \times [j, j + B + 1]$  do
10:   if  $(a = i) \vee (b = j) \vee (a = i + B + 1) \vee (b = j + B + 1)$  then
11:     if  $Scan_{x+}(a, b) < T$  then
12:        $T \leftarrow Scan_{x+}(a, b)$ 
13:     end if
14:   end if
15: end for
16:                                              ▷ Voxelhöhle für Andocken anpassen
17: for all  $(x, y, z) \in [X_{min}, X_{max}] \times [Y_{min}, Y_{max}] \times [Z_{min}, Z_{max}]$  do
18:   if  $x < T$  then
19:      $Voxel(x, y, z) \leftarrow 10$                                      ▷ Bereich markieren
20:   else if  $x < Scan_{x+}(z - Offset_z, y - Offset_y)$  then
21:      $Voxel(x, y, z) \leftarrow 1$                                      ▷ Bereich ausfräsen
22:   end if
23: end for
24:
25:  $\vec{N}_a \leftarrow (-1, 0, 0)$                                      ▷ Andockstelle erstellen
26:  $M_a \leftarrow (T, (Y_{min} + Y_{max} + 1)/2, (Z_{min} + Z_{max} + 1)/2)$     ▷ +1 durch Voxel zu Vertex
27: for  $z = (Z_{max} + 1) \rightarrow Z_{min}$  do
28:   createAndAdd  $Vertex(T, Y_{min}, z)$ 
29: end for
30: for  $y = (Y_{min} + 1) \rightarrow Y_{max}$  do
31:   createAndAdd  $Vertex(T, y, Z_{min})$ 
32: end for
33: for  $z = Z_{min} \rightarrow (Z_{max} + 1)$  do
34:   createAndAdd  $Vertex(T, Y_{max} + 1, z)$ 
35: end for
36: for  $y = Y_{max} \rightarrow (Y_{min} + 1)$  do
37:   createAndAdd  $Vertex(T, y, Z_{max} + 1)$ 
38: end for

```

7. Hinzufügen von Räumen und Gängen zum Dungeon

umschliessen genau die entstandene Öffnung. Ihre Koordinaten werden nach derselben deterministischen Methode berechnet wie die Vertexkoordinaten der Höhle (vgl. Kapitel 5.2). Somit werden Gang und Höhle lückenlos miteinander verbunden. Ein an die Höhle angedockter Gang ist in Abbildung 7.5 zu sehen.

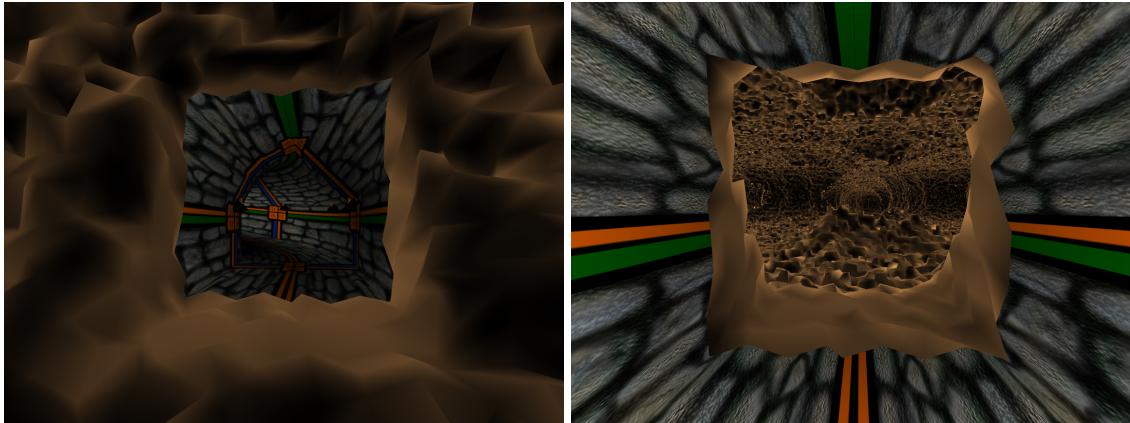


Abbildung 7.5.: *An Höhle angedockter Gang*: v.l.n.r. (a) aus Höhlensicht, (b) aus Gangsicht

Die Laufzeitkomplexität des gesamten Andockprozesses inklusive Andocktest wird im Wesentlichen von den Tests bestimmt. Im Worst Case müssen alle n Positionen auf der Scankarte überprüft werden, wobei der Test jeder Position durch den Kollisionstest in $\mathcal{O}(n)$ liegt. Daraus resultiert eine Gesamtkomplexität für die Laufzeit von $\mathcal{O}(n^2)$ bzw. $\mathcal{O}((A + B + R)^2 \cdot (A + B)^2)$, da die Scankarte eine Ausdehnung von $[A + B + 1] \times [A + B + 1 + R]$ besitzt.

7.4. Grenzen des Verfahrens

Die Kollision von Gängen bei Raum-zu-Raum-Verbindungen ist i.d.R. unproblematisch, da der Gang nur zwei benachbarte Sektoren verbindet. Allerdings sollten nicht zu große Ableitungsfaktoren g verwendet werden, damit der Gang nicht über die Sektorgrenzen hinausragt bzw. in Räume hinein. Das Gleiche gilt für die Ableitungsfaktoren g bei der Raum-zu-Höhle-Verbindung. Da nur der Bereich der Scankarte getestet wird, also der Sektorausdehnung plus Rand, können bei zu ausladenden Gängen eventuell Kollisionen nicht erkannt werden.

Des Weiteren sollte das Verhältnis von Sektorausdehnung und Raumgröße passend gewählt werden. Die Sektorausdehnung muss stets größer als die maximale Raumausdehnung sein, da sonst Kollisionen zwischen Räumen auftreten können.

7. Hinzufügen von Räumen und Gängen zum Dungeon

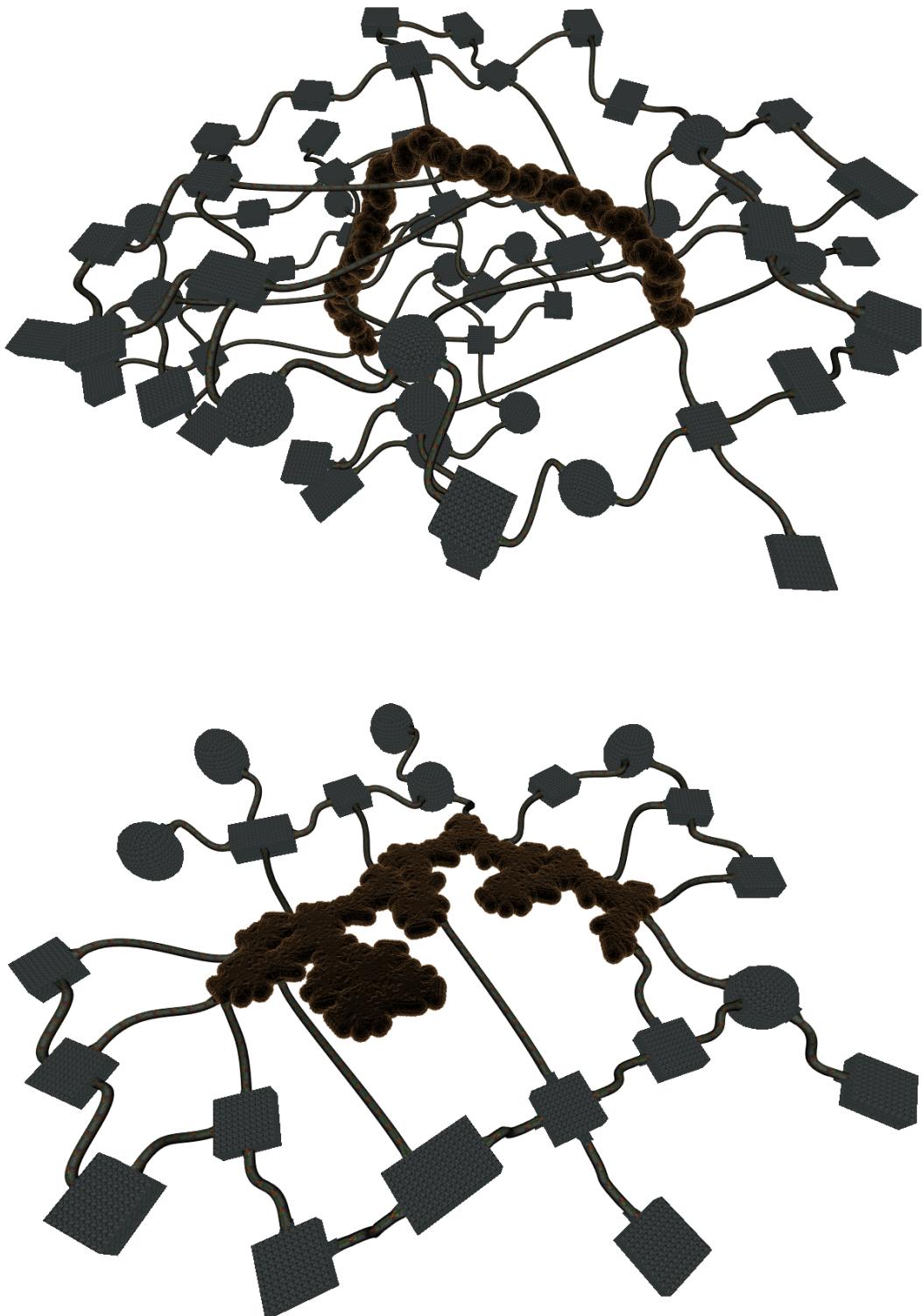


Abbildung 7.6.: *Beispieldungeons von außen:* Gezeigt sind durch das beschriebene Verfahren erstellte Dungeons. In Spielen wären solche Dungeons nur von innen erkundbar.

8. Methoden zur Reduzierung des Renderaufwandes

Ein genereller Konflikt in der Computergrafik ist der Widerspruch zwischen möglichst hoher Detailfülle und möglichst schneller Darstellung. Diese Darstellung muss beispielsweise in Computerspielen in Echtzeit erfolgen, also mit mindestens 30 Bildern pro Sekunde. Für das Erreichen dieses Ziels sind bestimmte Techniken notwendig. So sollen nicht sichtbare Objekte nach Möglichkeit von vornherein vom Zeichnen ausgeschlossen werden. Sichtbare Objekte sollten in desto niedrigerer Detailstufen dargestellt werden, je weiter weg sie sich von der Kamera befinden. Die Verwendung verschiedener Detailstufen stellt das Prinzip des *Level of Detail* dar. Das Weglassen von nicht sichtbaren Objekten ist das Prinzip des *Culling*. Arten des Culling sind beispielweise *Frustum Culling* (Ausschluß von Objekten außerhalb des Sichtbereiches der Kamera) und *Occlusion Culling* (Ausschluß von verdeckten Objekten).

8.1. Sichtbarkeitsgraph

Für die Entfernungs- und Sichtbarkeitstest ist es von Vorteil, einen *Sichtbarkeitsgraphen* zu haben. Dieser enthält die Informationen, welches Objekt mit welchem anderen Objekt, räumlich betrachtet, direkt verbunden ist und von welchem Objekt zu welchem anderen gesehen werden kann. Die Objekte im Sichtbarkeitsgraphen des Dungeons sind: jedes Subnetz der Höhle, alle Gänge und alle Räume.

Es wird bestimmt, wie die Objekte miteinander verbunden sind und welche Sichtbarkeits-eigenschaften sie besitzen:

Für jeden *Raum* speichere, in welcher Himmelsrichtung welcher Gang angeschlossen ist.¹ Der Aufbau des Raums, z.B. von welcher Andockstelle zu welcher eine Sichtverbindung existiert, ist extern zu ermitteln, da die zugrundeliegenden Subszenen importiert werden.

Für jeden *Gang* speichere, an welchem Ende welcher Raum bzw. die Höhle angeschlossen ist. Weiterhin speichere \vec{P}_1 , \vec{P}_2 , \vec{P}'_1 und $-\vec{P}'_2$, um zu wissen, wo der Gang endet und in welche Richtung er dort zeigt. Zusätzlich wird getestet, ob man von einem Ende des Gangs zum anderen sehen kann.

Für jedes *Höhlensubnetz* speichere die Grenzen X_{min} , X_{max} , Y_{min} , Y_{max} , Z_{min} und Z_{max} des zugrundeliegenden Voxelgebietes. Zusätzlich wird überprüft, in welcher der sechs Koordinatenachsenrichtungen ($X+, X-, Y+, Y-, Z+, Z-$) man von diesem Subnetz andere Subnetze sehen kann, also ob dort ein anderes Subnetz mit direkter Verbindung existiert.

Importiert man diese Informationen in die gewünschte Anwendung, so lässt sich ein Sichtbarkeitsgraph beispielsweise wie folgt erzeugen: Für die Knoten des Graphen werden die Verbindungsstellen zwischen direkt miteinander verbundenen Objekten verwendet. Zum Beispiel würde eine Andockstelle Gang zu Raum einen Knoten repräsentieren. Die Kanten werden durch die Objekte oder deren Teile gebildet. Beispielsweise würde ein Gang eine

¹Es werden die ursprünglichen Himmelsrichtungen der Subszene verwendet.

8. Methoden zur Reduzierung des Renderaufwandes

Kante darstellen, ein Raum entspricht bis zu sechs Kanten (zwischen jedem Paar von An-dockstellen eine Kante). Als Kantengewicht wird der euklidische Abstand zwischen beiden Knoten genutzt. Falls man von einer Verbindungsstelle nicht zur anderen sehen kann, dann wird keine Kante eingefügt. Wenn man also von einem Ende eines Gangs nicht zur anderen Ende sehen kann, dann wird der Gang nicht als Kante verwendet.

Der Sichtbarkeitsgraph kann in einem Spiel oder einer anderer Anwendung per Breitensuche oder Dijkstra-Algorithmus (dieser wird beispielsweise in [CLRS10, S.670 ff.] beschrieben) traversiert werden, um zu testen, wie weit ein Objekt vom Objekt, in dem sich die Kamera befindet, effektiv entfernt ist bzw. ob es überhaupt sichtbar ist (dies ist nur der Fall, wenn es im Graphen erreichbar ist).

8.1.1. Sichtbarkeitsinformationen der Höhlensubnetze

Das Verfahren für die Sichtbarkeitsinformationen der Höhlensubnetze ist vergleichsweise einfach umsetzbar. Es wird für jede Achsenrichtung getestet, ob an irgendeiner Position auf beiden Seiten der Grenzfläche ein Voxel mit Belegung 1, also ein Freiraumvoxel, existiert. Falls ja, würde die Höhle über die Grenzfläche hinweg verlaufen und man könnte entlang der Achse von dem einen Subnetz in das andere schauen.

Formel 8.1 beschreibt diesen Test. Der Parameter Vis gibt für jede Achsenrichtung an, ob andere Höhlensubnetze sichtbar sind. Wenn der Wert *wahr* beträgt, ist eine solche Sichtmöglichkeit vorhanden.

$$\begin{aligned}
 Vis_{X+} &= \exists(y, z) \in [Y_{min}, Y_{max}] \times [Z_{min}, Z_{max}] : \\
 &\quad Voxel(X_{max}, y, z) = 1 \wedge Voxel(X_{max} + 1, y, z) = 1 \\
 Vis_{X-} &= \exists(y, z) \in [Y_{min}, Y_{max}] \times [Z_{min}, Z_{max}] : \\
 &\quad Voxel(X_{min}, y, z) = 1 \wedge Voxel(X_{min} - 1, y, z) = 1 \\
 Vis_{Y+} &= \exists(x, z) \in [X_{min}, X_{max}] \times [Z_{min}, Z_{max}] : \\
 &\quad Voxel(x, Y_{max}, z) = 1 \wedge Voxel(x, Y_{max} + 1, z) = 1 \\
 Vis_{Y-} &= \exists(x, z) \in [X_{min}, X_{max}] \times [Z_{min}, Z_{max}] : \\
 &\quad Voxel(x, Y_{min}, z) = 1 \wedge Voxel(x, Y_{min} - 1, z) = 1 \\
 Vis_{Z+} &= \exists(x, y) \in [X_{min}, X_{max}] \times [Y_{min}, Y_{max}] : \\
 &\quad Voxel(x, y, Z_{max}) = 1 \wedge Voxel(x, y, Z_{max} + 1) = 1 \\
 Vis_{Z-} &= \exists(x, y) \in [X_{min}, X_{max}] \times [Y_{min}, Y_{max}] : \\
 &\quad Voxel(x, y, Z_{min}) = 1 \wedge Voxel(x, y, Z_{min} - 1) = 1
 \end{aligned} \tag{8.1}$$

8.1.2. Sichtbarkeitsinformationen der Gänge

Es soll geprüft werden, ob man von einem Ende des Gangs zum anderen Ende sehen kann. Dazu wird erst einmal ein beliebiges Stück eines polygonalen Schlauchs betrachtet. Gegeben seien drei Segmente R_1 , R_2 und R_3 des Schlauchs.

Die Überlegung ist folgende: Um R_1 und R_3 wird eine Bounding Box gelegt, welche im Folgenden als *Viewing Volume* bezeichnet wird. Wenn die Querschnittsfläche von R_2 dieses Viewing Volume nicht schneidet, dann kann kein Lichtstrahl von R_1 nach R_3 gelangen. Somit könnte man nicht von R_1 nach R_3 sehen und dieser Abschnitt des polygonalen Schlauchs wäre blickdicht. Das Prinzip ist in Abbildung 8.1 dargestellt.

8. Methoden zur Reduzierung des Renderaufwandes

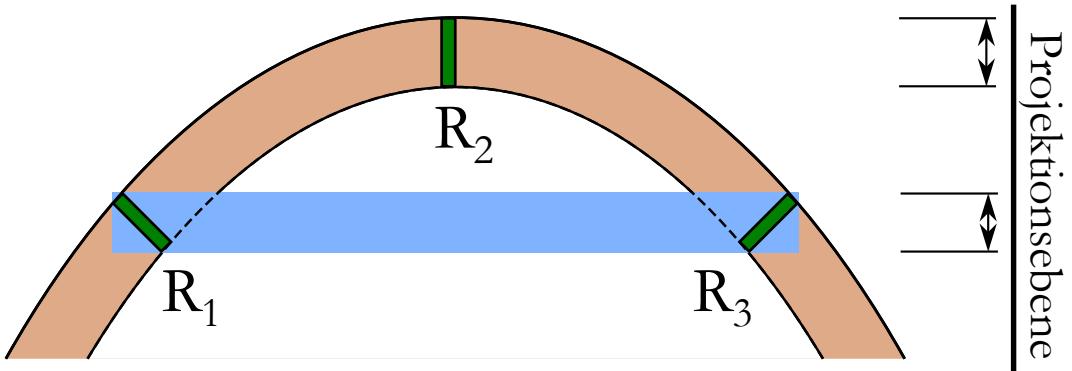


Abbildung 8.1.: *Blickdichtigkeitstest für Gänge:* Zu sehen sind die drei Segmente des Schlauchs R_1 , R_2 , R_3 (grün) und das Viewing Volume (blau). Der eigentliche Kollisionstest kann mit den auf eine Ebene projizierten Elementen durchgeführt werden.

Diesen Test kann man nun für alle Segmente des Gangs plus die beiden Adapter durchführen. Problematisch ist die Laufzeitkomplexität bei insgesamt n Segmenten von $\mathcal{O}(n^3)$ (durch $\mathcal{O}(n^2)$ Viewing Volumes mit je $\mathcal{O}(n)$ Tests).² Für alle Segmente ist das Verfahren dadurch in der Praxis nicht anwendbar.

Dies führt zu der Idee, die Tests auf „interessante“ Segmente, im folgenden *Region of Interest* genannt, zu beschränken. Als Regions of Interest sollen beide Ankerstellen und die Segmente des Gangs für $\overrightarrow{P(0)}$ und $\overrightarrow{P(1)}$ gelten.

Hinzu kommen Segmente, deren Abstand zur Achse $\overrightarrow{P(0)P(1)}$ möglichst groß ist, da dort die Wahrscheinlichkeit für eine Blickverdeckung am höchsten ist. Gesucht werden somit die Segmente an $P(t)$ für die D_{Achse} nach Formel 8.2 ein lokales Maximum darstellt. Für eine einfachere Berechnung wird der quadratischen Abstand $(D_{Achse})^2$ zur Bestimmung der Maxima verwendet. Die dadurch ermittelten Maxima sind identisch mit denen des einfachen Abstands.

$$D_{Achse} = \left| \overrightarrow{P(t)} - \overrightarrow{P(0)} - \left(\overrightarrow{N_{Achse}} \cdot (\overrightarrow{P(t)} - \overrightarrow{P(0)}) \right) \cdot \overrightarrow{N_{Achse}} \right|$$

mit $\overrightarrow{N_{Achse}} = \frac{\overrightarrow{P(1)} - \overrightarrow{P(0)}}{\|\overrightarrow{P(1)} - \overrightarrow{P(0)}\|}$ (8.2)

Die Anzahl der lokalen Maxima ist maximal 2. *Beweis:* Der quadratischer Abstand zur Achse ist ein Polynom 6. Ordnung in t . Zur Bestimmung der Extrema wird erste Ableitung gleich 0 gesetzt, welche ein Polynom 5. Ordnung darstellt. Gesucht werden also die Nullstellen eines Polynoms 5. Ordnung, wovon es maximal 5 geben kann. Es gibt somit maximal 5 Extrema. Davon sind mindestens 2 lokale Minima (quadr. Abstand = 0): der Anfangspunkt $\overrightarrow{P(0)}$ und der Endpunkt $\overrightarrow{P(1)}$.

Zwischen 2 bestehenden Maxima muss stets ein Minimum liegen, da die Kurve stetig ist. Wenn 2 lokale Maxima vorliegen, dann muss folglich dazwischen ein lokales Minimum existieren, d.h. 5 lokale Extrema würden vorliegen. Mehr Maxima kann es nicht geben, da

²Ausgegangen wird von einer konstanten Laufzeit für den eigentlichen Schnitttest.

8. Methoden zur Reduzierung des Renderaufwandes

dann mehr als 5 Extrema existieren würden. Folgerung: Es gibt maximal 2 lokale Maxima. *q.e.d.*

Da die exakten Nullstellen eines Polynoms 5. Ordnung schwer zu bestimmen sind, soll die Ermittlung der lokalen Maxima mit einer numerischen Lösung erfolgen. Beim der Erstellung des polygonalen Schlauchs wird während des Abtastens der Kurve der Abstand von $\overrightarrow{P(t)}$ zur Achse überwacht. Liegt der Abstand im letzten und folgenden Schritt niedriger als im aktuellen, wurde ein lokales Maximum gefunden.

Insgesamt gibt es also maximal sechs Regions of Interest, was eine konstante Maximalanzahl nötiger Tests bedeutet. Die Laufzeitkomplexität des Tests für n Segmente liegt nun bei $\mathcal{O}(1)$.

Durchführung des Tests mittels projizierten Bounding Boxes

Die Regions of Interest R_1, R_2, \dots, R_n seien nach ihrer Reihenfolge entlang des Gangs angeordnet. Zuerst erfolgt die Konstruktion der Viewing Volumes aus allen Paaren (R_i, R_j) mit $i + 1 < j$, also mit mindestens einer Region of Interest dazwischen. Dann werden für jedes Viewing Volume aus (R_i, R_j) alle Regions of Interest R_k mit $i < k < j$ getestet. Der eigentliche Schnitttest wird mit projizierten Bounding Boxes durchgeführt.

Eine Region of Interest sei gegeben durch die Position $\overrightarrow{P_R}$, die Vektoren des lokalen Koordinatensystems $\overrightarrow{\text{Links}_R}$ und $\overrightarrow{\text{Oben}_R}$ und die Bounding Box $[R\text{Min}_x, R\text{Max}_x] \times [R\text{Min}_y, R\text{Max}_y]$ des Segmentquerschnitts. Diese Bounding Box ist auf das 2D-Koordinatenystem mit $-\overrightarrow{\text{Links}_R}$ als X-Achse, $\overrightarrow{\text{Oben}_R}$ als Y-Achse und $\overrightarrow{P_R}$ als Nullpunkt bezogen.

Ein Viewing Volume wird durch seinen Bezugspunkt $\overrightarrow{P_V}$, die Vektoren des lokalen Koordinatensystems $\overrightarrow{\text{Links}_V}$ und $\overrightarrow{\text{Oben}_V}$ und seine Bounding Box $[V\text{Min}_x, V\text{Max}_x] \times [V\text{Min}_y, V\text{Max}_y]$ gekennzeichnet. Diese Bounding Box ist auf das 2D-Koordinatenystem mit $-\overrightarrow{\text{Links}_V}$ als X-Achse, $\overrightarrow{\text{Oben}_V}$ als Y-Achse und $\overrightarrow{P_V}$ als Nullpunkt bezogen.

Zur Konstruktion des Viewing Volumes aus R_i und R_j wird zuerst das lokale Koordinaten- system nach Formel 8.3 ermittelt. Der Normalenvektor der Viewing Volume-Ebene verläuft entlang der Achse $\overrightarrow{P_{Ri}P_{Rj}}$. Bezugspunkt sei $\overrightarrow{P_V} = \overrightarrow{P_{Ri}}$.

$$\begin{aligned} \overrightarrow{\text{Links}_V} &= \frac{\overrightarrow{\text{vorn}} \times (0, 1, 0)}{|\overrightarrow{\text{vorn}} \times (0, 1, 0)|}, \quad \overrightarrow{\text{Oben}_V} = \overrightarrow{\text{Links}_V} \times \overrightarrow{\text{vorn}} \\ \text{mit } \overrightarrow{\text{vorn}} &= \frac{\overrightarrow{P_{Rj}} - \overrightarrow{P_{Ri}}}{|\overrightarrow{P_{Rj}} - \overrightarrow{P_{Ri}}|} \end{aligned} \tag{8.3}$$

Nun werden die beiden Bounding Boxes der Region of Interests R_i und R_j in die Ebene des Viewing Volumes projiziert. Die generelle Projektion einer Bounding Box einer Region of Interest R erfolgt durch die Projektion aller vier Bounding Box-Eckpunkte auf die Ebene und das Bilden einer neuen Bounding Box um diese Projektionen. Formel 8.4 gibt das Beispiel für den unteren linken Eckpunkt an. Dessen Projektion sei als $\overrightarrow{Q_{proj}}$ bezeichnet. Die anderen Eckpunkte werden äquivalent behandelt.

$$\begin{aligned} \overrightarrow{\text{temp}} &= \overrightarrow{P_R} - \overrightarrow{P_V} - R\text{Min}_x \cdot \overrightarrow{\text{Links}_R} + R\text{Min}_y \cdot \overrightarrow{\text{Oben}_R} \\ \overrightarrow{Q_{proj}} &= (-\overrightarrow{\text{temp}} \cdot \overrightarrow{\text{Links}_V}, \overrightarrow{\text{temp}} \cdot \overrightarrow{\text{Oben}_V}) \end{aligned} \tag{8.4}$$

8. Methoden zur Reduzierung des Renderaufwandes

Bei den Bounding Boxes von R_i und R_j kann dabei der Term $\overrightarrow{P_R} - \overrightarrow{P_V}$ aus Zeile 1 weggelassen werden, da dieser in Normalenrichtung der Ebene verläuft und somit bei der Projektion entfällt. Die Viewing Volume-Bounding Box $[VMin_x, VMax_x] \times [VMin_y, VMax_y]$ wird durch eine Bounding Box um alle acht projizierten Eckpunkte gebildet.

Für den Test wird nun die Region of Interest R_k auf die Ebene des Viewing Volumes projiziert und der Schnitttest nach Formel 8.5 durchgeführt. Die projizierte Bounding Box der Region of Interest sei $[R_kMin_x^{pr}, R_kMax_x^{pr}] \times [R_kMin_y^{pr}, R_kMax_y^{pr}]$.

Wenn sich die Bounding Boxes überschneiden ($Vis_G = \text{wahr}$), ist der Gang möglicherweise nicht blickdicht. Ansonsten liegt *definitive Blickdichtigkeit* vor, d.h. man kann in keinem Fall von einem Ende des Gangs durch den Gang zum anderen sehen.

$$Vis_G = (R_kMin_x^{pr} < VMax_x) \wedge (VMin_x < R_kMax_x^{pr}) \wedge (R_kMin_y^{pr} < VMax_y) \wedge (VMin_y < R_kMax_y^{pr}) \quad (8.5)$$

Der Vorteil der Projektion liegt in einem schnelleren Test. Wenn ein Viewing Volume einmal erstellt ist, können alle gewünschten Regions of Interest per zweidimensionalen Überschneidungstest überprüft werden, anstatt dreidimensional, wie es ohne Projektion der Fall wäre. In Abbildung 8.2 ist ein Vergleich zwischen einem blickdichten und einem nicht blickdichten Gang aufgezeigt. Für das Bestimmen dieser Informationen wurde der obige Test verwendet.

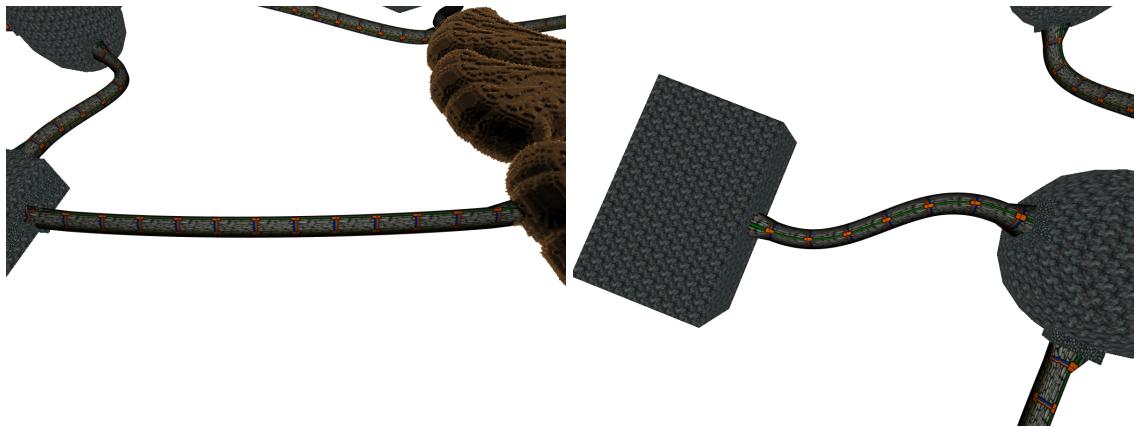


Abbildung 8.2.: *Blickdichtigkeit von Gängen:* v.l.n.r. (a) nicht blickdicht, (b) blickdicht

8.2. Reduktion von Dreiecksnetzen

Für die Generierung verschiedener Detailstufen von aus Dreiecksnetzen aufgebauten Objekten werden deren Netze reduziert. Wichtig bei dieser Reduktion ist zum einen, dass das reduzierte Objekt möglichst weit dem Original entspricht, und zum anderen, dass es möglichst einfach gestaltet ist, d.h. möglichst wenig Dreiecke und Vertices besitzt.

Für die *Räume* bleibt die Reduktion dem Raumgestalter überlassen. Es erfolgt für diese keine Reduktion im Generator.

8. Methoden zur Reduzierung des Renderaufwandes

Für die *Gänge* ermöglicht die schlauchartige Struktur eine effiziente Reduktion. Für Reduktionsstufe i wird der Gang erneut erstellt, mit gleichem Gangsegmentabstand w wie im Original, aber nur jedes 2^i -te Segment wird wirklich eingefügt. Zusätzlich werden alle lokalen Maxima eingefügt, damit die Blickdichtigkeitsberechnung für alle reduzierten Versionen ebenso gilt. Resultat ist eine geringere Anzahl benötigter Vertices und Dreiecke bei prinzipiell gleichem Gangverlauf. Diese Anzahl wird bei jeder zusätzlichen Reduktionsstufe halbiert.

Für die *Höhle* muss ein anderer Algorithmus eingesetzt werden. Ziel ist die Generierung mehrerer Detailgrade für jedes Subnetz der Höhle. Wichtig ist, dass die Meshes aller Detailstufen (inkl. Original) an den Rändern zusammenpassen. D.h. Position und Normalen müssen an diesen Stellen exakt gleich sein, damit beliebige Detailstufen in Kombination verwendet werden können. Für jede Detailstufe und alle möglichen Randbeziehungen je eine extra Variante zu generieren wäre deutlich zu specheraufwändig, da bis zu 26 Nachbarmeshes möglich sind (wobei 6 davon eine Grenzfläche teilen, also eine große Zahl gemeinsamer Grenzvertices besitzen können).

8.2.1. Reduktion der Höhlensubnetze

Die Topologie eines Meshs kann nach [LRC⁺03, S.14-17] über den Genus und die Mannigfaltigkeit beschrieben werden. Der *Genus* ist die Anzahl von Löchern im Mesh. Eine Kugel hat einen Genus von 0, ein Torus einen von 1. In *mannigfältigen* Meshes hat jede Kante exakt zwei benachbarte Dreiecke und jedes Dreieck teilt seine Kanten mit exakt drei benachbarten Dreiecken.

Topologie-erhaltende Algorithmen bewahren die Mannigfaltigkeit und den Genus in jedem Schritt. Manche Algorithmen sind *topologie-tolerant*, sie ignorieren nicht-mannigfaltige Topologie und verhalten sich ansonsten topologie-erhaltend.

Topologie-modifizierende Algorithmen können die Mannigfaltigkeit und den Genus von Meshes verändern, was üblicherweise schlechtere visuelle Ergebnisse zur Folge hat.

Wünschenswert wäre ein topologie-erhaltender bzw. topologie-toleranter Algorithmus, damit die Höhlenstruktur besser erhalten bleibt. Da die Öffnungen für Gänge und die Grenzen zu anderen Subnetzen im Gegensatz zum Rest des Meshes nicht-mannigfaltig sind (die Grenzkanten haben nur ein benachbartes Dreieck), muss bei Verwendung eines topologie-erhaltenden Algorithmus dieser zu einem topologie-toleranten Algorithmus modifiziert werden.

Viele Reduktionsalgorithmen verwenden folgendes Prinzip: Der Reduktionsprozess setzt sich aus der Durchführung einer meist größeren Anzahl von Anwendungen lokaler Reduktionsoperatoren zusammen. Die Auswahl der Region, an der der Operator angewendet wird, erfolgt i.d.R. anhand einer Fehlermetrik. Die Region mit dem kleinsten zu erwartenden Fehler wird als nächstes verwendet. [LRC⁺03, S.21 ff.] beschreibt folgende lokalen Reduktionsoperatoren:

Edge Collapse: Als Reduktionsmethode wird die Kontraktion einer Kante verwendet (siehe Abbildung 8.3). Die zwei Endpunkte einer Kante werden zu einem Vertex verschmolzen. Die Position des resultierenden Vertex kann nach verschiedenen Methoden berechnet werden, z.B. als Mitte zwischen beiden Ausgangsvertices. Beide Endpunkte der Kante dürfen nicht mehr als zwei gemeinsame Nachbarn haben, da sonst nicht-mannigfaltige Strukturen entstehen (siehe Abbildung 8.4).

8. Methoden zur Reduzierung des Renderaufwandes

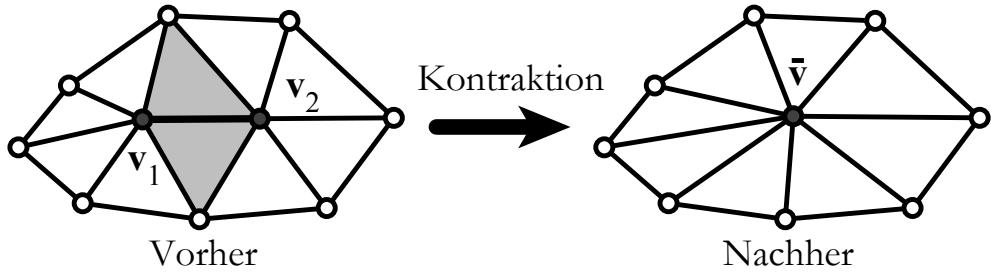


Abbildung 8.3.: Edge Collapse [GH97]

Vertex-Pair Collapse: Die Reduktionsmethode funktioniert äquivalent zu Edge Collapse, doch hier brauchen die verschmelzenden Vertices nicht zwangsläufig durch eine Kante verbunden zu sein.³

Triangle Collapse: Alle drei Vertices eines Dreiecks werden zu einem einzigen Vertex verschmolzen. Die Methode ist äquivalent zu zwei hintereinander ausgeführten Edge Collapses.

Cell Collapse: Der Raum wird mittels eines gleichmäßigen Gitters in Zellen unterteilt. Alle Vertices, die zusammen in einer Zelle sind, werden zu einem Vertex verschmolzen.

Vertex Removal: Ein Vertex wird entfernt. Das entstehende Loch des Meshes wird retrianguliert und dadurch wieder geschlossen.

Polygon Merging: Benachbarte, möglichst weit koplanare Polygone werden zu einem größeren Polygon verschmolzen. Dieses wird später trianguliert.

General Geometric Replacement: Eine Menge von adjazenten Dreiecken wird durch eine andere Menge von adjazenten Dreiecken ersetzt, so dass die Begrenzung die gleiche bleibt. Die neue Menge von Dreiecken besitzt dabei weniger Dreiecke als die alte.

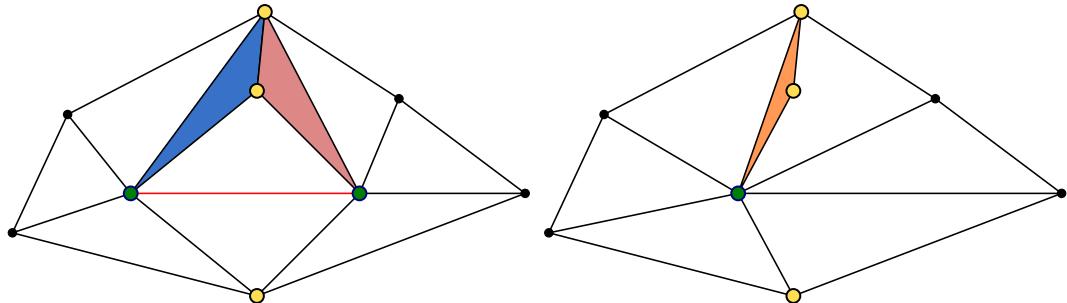


Abbildung 8.4.: Verlust der Mannigfaltigkeit bei mehr als zwei gemeinsamen Nachbarn:
betrachten wird ein Ausschnitt eines Dreiecksnetzes: v.l.n.r.

(a) Gezeigt ist die Ausgangstriangulation. Die zu kontrahierende Kante ist rot dargestellt. Beide Endpunkte der Kante (grün) haben drei gemeinsame Nachbarpunkte (gelb).

(b) Durch den Edge Collapse verschmelzen das rote und blaue Dreieck zum orangenen Dreieck. Die Mannigfaltigkeit ist verloren gegangen, da nun zwei Kanten dieses Dreiecks zu keinen weiteren Dreiecken gehören.

Im Vergleich lassen sich die Collapse-Operatoren am einfachsten implementieren. Cell Col-

³Diese Beschreibung folgt [GH97]. [LRC⁺03] nennt diesen Operator explizit für zwei nicht-verbundene Vertices.

8. Methoden zur Reduzierung des Renderaufwandes

lapse stellt das einfachste Verfahren dar, ist aber, wie auch Vertex-Pair Collapse, ein topologie-modifizierender Reduktionsschritt. Edge Collapse und Triangle Collapse sind topologie-erhaltend. Edge Collapse und Vertex-Pair Collapse lassen sich besser steuern als Triangle Collapse, da sie pro Schritt kleinere Änderungen vornehmen.

Möglichkeiten für Fehlermetriken werden in [LRC⁺03, S.47 ff.] genannt. Beispiele dafür sind: die Abstände zwischen vom Reduktionsoperator betroffenen Vertices, die Abstände vom betroffenen Vertex und der durch seine Umgebung gelegte Fläche, die Normalenänderungen im betroffenen Gebiet.

Die lokalen Reduktionsoperatoren lassen sich ggf. in Verbindung mit entsprechenden Fehlermetriken zu konkreten Algorithmen, wie sie in [LRC⁺03, S.121 ff.] beschrieben werden, kombinieren. *Vertex Clustering*, *Vertex Decimation* und *Quadric Error Metrics* stellen dabei drei der wichtigsten Verfahren dar.

Vertex Clustering, wie es von Rossignac und Borrel in [RB92] beschrieben wurde, verwendet die Methode Cell Collapse zur Reduktion. Zuerst wird jedem Vertex eine Wichtigkeit bzw. Wichtung zugeordnet. Vertices großer Flächen oder an ausgeprägten Krümmungen werden stärker gewichtet. Dann wird ein 3D-Gitter über das zu reduzierende Objekt gelegt und dieses so in Zellen aufgeteilt. Alle Vertices in einer Zelle werden zum am höchsten gewichteten Vertex kollabiert, d.h. der wichtigste Vertex repräsentiert nun alle Vertices der Zelle. Degenerierte Dreiecke werden entfernt. Der Algorithmus ist einer der schnellsten bekannten Reduktionsalgorithmen. Nachteil ist vor allem, dass der Algorithmus stark topologie-modifizierend ist.

Vertex Decimation wird in [SZL92] beschrieben und nutzt den Operator Vertex Removal. Der Algorithmus verwendet mehrere Durchläufe. In jedem Durchlauf werden Vertices nach bestimmten Kriterien entfernt und danach die entstandenen Löcher retrianguliert. Als Hauptkriterium wird die Distanz des zu entfernenden Vertex zur Ebene, die durch die benachbarten Vertices gelegt wird, genommen. Ist diese unter einem Grenzwert ϵ , so wird der Vertex entfernt. Das ϵ wird in jedem Durchlauf erhöht. Der Algorithmus terminiert beim Erreichen des gewünschten Reduktionslevels. Er erweist sich in der Praxis als recht schnell. Nachteil ist die schlechte Steuerbarkeit über den ϵ -Wert. Ist dieser zu groß, wird das Modell zu stark vereinfacht. Bei zu kleinen Werten werden zu viele Durchläufe benötigt, wodurch der Algorithmus langsam wird. Im Kontext der Höhlenmeshes ist anzumerken, dass durch ihre zerklüftete Struktur ein ähnlicher Distanz-Wert für die meisten Vertices vorliegt. Weiterhin würde der Algorithmus lokal stark ausgeprägte Voxelkanten hervorheben und eher weichere Kanten verschwinden lassen. Dadurch ist diese Methode hierfür ungeeignet.

Quadric Error Metrics wurde von Garland und Heckbert 1997 in [GH97] vorgestellt. Der Algorithmus basiert auf Vertex-Pair Collapses. Für die Kontraktion zugelassene Vertex-Paare sind alle Paare, die über eine Kante verbunden sind, sowie alle Paare, deren Abstand zueinander einen spezifizierten Grenzwert unterschreitet. Als Fehlermetrik wird die Quadric Error Metric benutzt. Jedem Vertex ist eine Quadrik Q zugeordnet, die kumulativ speichert, welche Verzerrungen durch diesen Vertex und seine bisherigen Collapses verursacht werden. Die Quadriken werden am Anfang für jeden Vertex berechnet. Bei jedem Collapse von Vertex a und Vertex b zu Vertex c ergibt sich $Q_c = Q_a + Q_b$. Die Vertex-Paare (a, b)

8. Methoden zur Reduzierung des Renderaufwandes

werden in einer Prioritätswarteschlange, sortiert nach steigendem Quadric Error $Q_a + Q_b$, gespeichert und abgearbeitet. Der Algorithmus terminiert, wenn die gewünschte Anzahl an Dreiecken erreicht wurde. Das Verfahren erzeugt qualitativ sehr hochwertige Ergebnisse und die erzeugte Dreiecksanzahl lässt sich exakt steuern. Allerdings hat die Abarbeitung der Vertex-Paare durch die Verwendung einer Prioritätswarteschlange eine Laufzeitkomplexität von $\mathcal{O}(n \log n)$ (n als Anzahl der Vertices), was bei einer hohen Vertexanzahl schnell zu vergleichsweise hohen Laufzeiten führt.

Reduktionsalgorithmus

Der implementierte Ansatz verwendet Edge Collapse als Reduktionsoperator. Als Fehlermetrik wird in jeder Reduktionstufe jeder Vertex höchstens einmal kollabiert. Dies bedeutet: Wenn von einer Kante mindestens ein Vertex schon kollabiert ist, dann erfolgt keine Kontraktion. Diese Metrik sei als *Single Contraction* bezeichnet. Vertices an Meshgrenzen werden nicht kollabiert, damit die Öffnungen für Gänge und Grenzen zu anderen Höhlenmeshes erhalten bleiben.

Durch die auf Grund der Voxelbasiertheit sehr regelmäßige Struktur des Dreiecksnetzes, wirkt diese Methode ähnlich dem Cell Collapse bzw. der Vertex Clustering-Methode, ist aber topologie-erhaltend. Zusätzlich wird eine Glättung des Meshs erreicht, statt Kanten der Voxel weiter hervorzuheben.

Das Verfahren ist in Algorithmus 8 aufgezeigt und wird für jedes Subnetz der Höhle separat angewendet. Das Subnetz sei gegeben durch seine Vertex- und Dreiecksmenge wie in Kapitel 3.3.1 erläutert. Die Grenzvertices haben eine X-Texturkoordinate von 1 (siehe Kapitel 5.3), andere von -1 . Äquivalent sollen Vertices an Gangöffnungen eine Y-Texturkoordinate von 1 haben, andere von -1 ⁴.

Vorm Start des Algorithmus erfolgt der Aufbau der Adjazenzliste. Dazu laufe über alle Dreiecke und füge für die drei Eckpunkte die Adjazenzen hinzu. Es wird nur eine Umlaufrichtung betrachtet, damit keine doppelten Adjazenzen auftreten. Aus Dreieck $\vec{P_1}, \vec{P_2}, \vec{P_3}$ ergeben sich als Adjazenzen: für $\vec{P_1}$ ist $\vec{P_2}$ adjazent, für $\vec{P_2}$ ist $\vec{P_3}$ adjazent und für $\vec{P_3}$ ist $\vec{P_1}$ adjazent.

Der Algorithmus unterscheidet zwischen Originalvertices V_i und Vertices V'_i , die dem reduzierten Netz angehören. Jeder V_i erhält einen Verweis Adr auf einen Vertex aus dem reduzierten Netz und eine Markierung $Flag$, die der Markierung von Nachbarschaften dient. Jeder V'_i erhält eine Markierung $Flag'$, ebenfalls für die Markierung von Nachbarschaften. Alle Adr -, $Flag$ - und $Flag'$ -Werte sind anfangs mit -1 initialisiert. Alle Vertices besitzen Indices ≥ 0 , d.h. solange für einen Vertex V_i $Adr(i) < 0$ gilt, ist dieser Vertex noch nicht kontrahiert, anderenfalls trägt Adr die neue Adresse des kontrahierten Vertex. $Flag$ und $Flag'$ sind durch diese Markierung anfangs keiner Nachbarschaft zugeordnet.

Der Algorithmus muss topologie-tolerant sein, d.h. Grenzvertices und Gangöffnungsvertices müssen erhalten bleiben. Weiterhin dürfen nur Originalvertices (also noch nicht verschmolzene Vertices) kontrahiert werden. Somit lässt sich folgender Test durchführen: Addiere die X-Texturkoordinaten und Y-Texturkoordinaten sowie die Adr -Werte beider Vertices einer Kante. Wenn die Summe -6 ergibt, sind alle diese Bedingungen erfüllt, bei -5 oder mehr

⁴Dies lässt sich bei der Umwandlung der Voxel in das Dreiecksnetz einfach bewerkstelligen: Vertices mit solchen Nachbarvoxeln, die Belegung 10 haben, sind Gangöffnungsvertices (vgl. Kapitel 7.3).

8. Methoden zur Reduzierung des Renderaufwandes

Algorithmus 8 Reduktion eines Dreicksnetzes, per Edge Collapse und Single Contraction

Eingabe: Dreicksnetz aus Dreiecken (i, j, k) und Vertices V_i , Adjazenzliste aller Vertices. Adr , $Flag$ und $Flag'$ sind mit -1 initialisiert.

Ausgabe: reduziertes Dreicksnetz aus Dreiecken (i, j, k) und Vertices V'_i

```

1: for all  $(i, j) \in$  Kanten aller Originaldreiecke do
2:   if  $Adr(i) + Adr(k) + \overrightarrow{V_i.T}.x + \overrightarrow{V_j.T}.x + \overrightarrow{V_i.T}.y + \overrightarrow{V_j.T}.y < -5.5$  then
3:     for all  $k \in$  Adjazenzliste( $i$ ) do
4:       if  $Adr(k) < 0$  then
5:          $Flag(k) \leftarrow i$                                  $\triangleright$  Originalvertex markieren
6:       else
7:          $r \leftarrow Adr(k)$                              $\triangleright$  kontrahierten Vertex markieren
8:          $Flag'(r) \leftarrow i$ 
9:       end if
10:      end for
11:       $doppelt \leftarrow 0$ 
12:      for all  $k \in$  Adjazenzliste( $j$ ) do
13:        if  $Adr(k) < 0$  then
14:          if  $Flag(k) = i$  then  $doppelt \leftarrow doppelt + 1$ 
15:        else
16:           $r \leftarrow Adr(k)$ 
17:          if  $Flag'(r) = i$  then  $doppelt \leftarrow doppelt + 1$ 
18:        end if
19:      end for
20:      if  $doppelt < 3$  then                       $\triangleright$  Bedingung für Erhaltung der Mannigfaltigkeit
21:        create Vertex  $V'_r$                           $\triangleright$  Durchführung des Edge Collapse
22:         $\overrightarrow{V'_r.Pos} \leftarrow (\overrightarrow{V_i.Pos} + \overrightarrow{V_j.Pos})/2$ 
23:         $\overrightarrow{V'_r.T} \leftarrow (-1, -1)$                  $\triangleright$  kein Gangöffnungs- oder Grenzvertex
24:         $Adr(i) \leftarrow r$ 
25:         $Adr(j) \leftarrow r$ 
26:      end if
27:    end if
28:  end for
29:  for all  $V_i \in$  Originalvertices do            $\triangleright$  Nicht kontrahierte Vertices übernehmen
30:    if  $Adr(i) < 0$  then
31:      create Vertex  $V'_r$ 
32:       $\overrightarrow{V'_r.Pos} \leftarrow \overrightarrow{V_i.Pos}$ 
33:       $\overrightarrow{V'_r.T} \leftarrow \overrightarrow{V_i.T}$ 
34:       $\overrightarrow{V'_r.N} \leftarrow \overrightarrow{V_i.N}$ 
35:       $Adr(i) \leftarrow r$ 
36:    end if
37:  end for
38:  for all  $(i, j, k) \in$  Originaldreiecke do       $\triangleright$  Nicht degenerierte Dreiecke übernehmen
39:    if  $Adr(i) \neq Adr(j) \wedge Adr(i) \neq Adr(k) \wedge Adr(j) \neq Adr(k)$  then
40:      create Dreieck'( $Adr(i), Adr(j), Adr(k)$ )
41:    end if
42:  end for

```

8. Methoden zur Reduzierung des Renderaufwandes

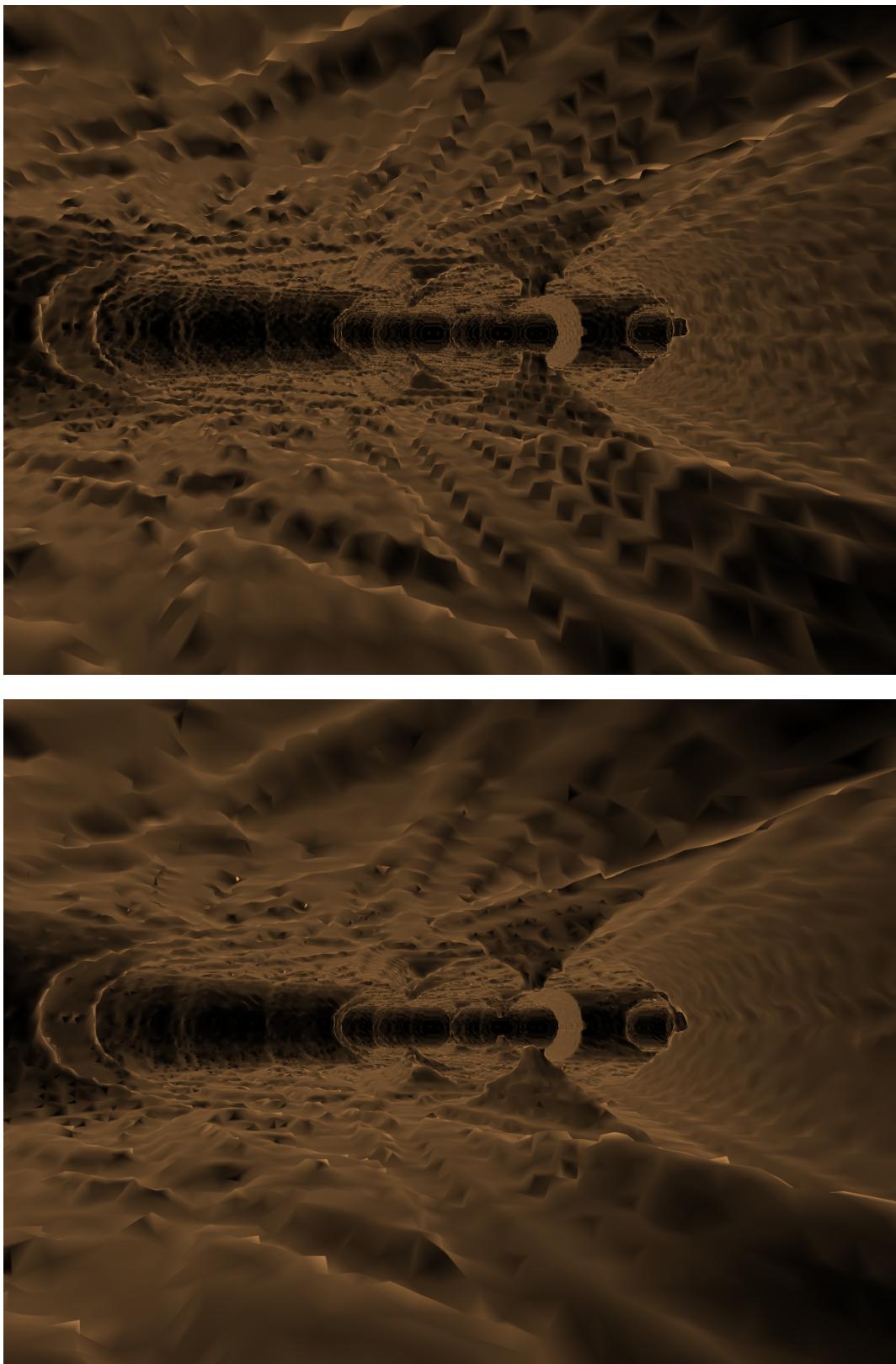


Abbildung 8.5.: *Reduktion der Höhlennetze:* v.o.n.u. (a) Originaldetailstufe, (b) eine Reduktionsstufe

8. Methoden zur Reduzierung des Renderaufwandes

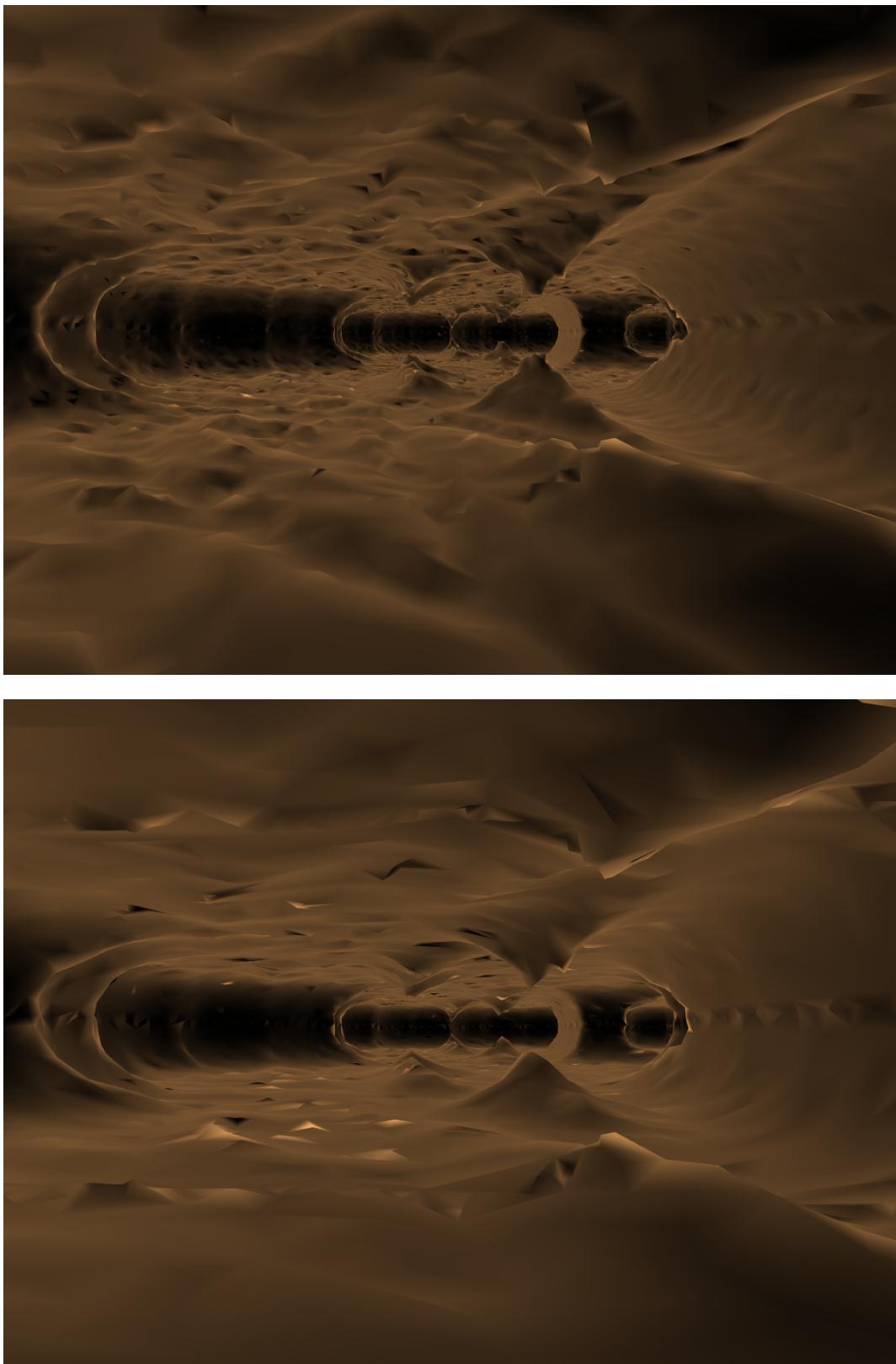


Abbildung 8.6.: *Reduktion der Höhlennetze ff.: v.o.n.u. (a) zwei Reduktionsstufen, (b) vier Reduktionsstufen*

8. Methoden zur Reduzierung des Renderaufwandes

nicht. Weiterhin muss getestet werden, ob maximal zwei gemeinsame Nachbarn vorliegen, damit die Mannigfaltigkeit des Meshs erhalten bleibt. Dazu werden erst für einen Vertex der Kante alle Nachbarn markiert (per *Flag* bzw. *Flag'*). Dann werden vom anderen Vertex aus alle entsprechenden Markierungen in der Nachbarschaft gezählt. Diese Zahl ist die Anzahl der gemeinsamen Nachbarn. Die Laufzeitkomplexität dieser Tests ist linear in der Anzahl der geprüften Vertices, bei einem trivialen Test aller Nachbarschaftspaare wäre diese quadratisch.

Wenn auch dieser Test positiv verläuft, so wird die Kante kontrahiert. Als Position des neuen Vertex wird die Mitte der Kante verwendet. Die Texturkoordinaten bleiben identisch zu denen der Ausgangsvertices und entsprechen somit $(-1, -1)$. Nachdem alle möglichen Kontraktionen durchgeführt wurden, werden alle nicht kontrahierten Vertices für den neuen Mesh übernommen. Danach werden alle diejenigen Dreiecke, die nicht zu Linien oder Punkten degeneriert wurden, ebenfalls übernommen.

Nach dem Durchlauf des Algorithmus müssen die Normalen berechnet werden. Dazu wird das Verfahren aus 5.3 verwendet. Allerdings bleiben die Normalen der Grenzvertices unverändert, da diese für die Meshes aller Detailstufen gleich bleiben müssen.

Die Anzahl der Dreiecke der Höhlenmeshes liegen konstruktionsbedingt in $\mathcal{O}(n)$, mit n als Anzahl der Vertices des Meshs. Der Algorithmus besitzt inklusive des Aufbaus der Adjazenzliste und Berechnung der Normalen somit eine Laufzeitkomplexität von $\mathcal{O}(g \cdot n)$, mit g als maximalem Knotengrad je Vertex. Der Faktor g resultiert aus der Prüfung auf mehrfache Nachbarn. In Abbildung 8.5 und 8.6 sind Ergebnisse des Verfahrens gezeigt. In der ersten Reduktionsstufe wird die Anzahl der Dreiecke und Vertices nahezu halbiert. Der Reduktionsfaktor sinkt in den folgenden Reduktionsstufen langsam ab, da die nicht kontrahierten Grenz- und Gangöffnungsvertices einen immer höheren Anteil an der Gesamtmenge der Vertices einnehmen.

Zu den Problemen des Verfahrens gehört das verstärkte Hervortreten der Meshgrenzen mit steigenden Reduktionsstufen, da diese keiner Reduktion unterliegen. Weiterhin kann es, falls keine Filterung schwelender Fragmente durchgeführt wurde, dazu kommen, dass kleinere solcher Fragmente zu Tetraedern kollabieren. Diese werden im nächsten Reduktionsschritt zu zwei aufeinanderliegenden Dreiecken kontrahiert und verschwinden im Schritt darauf vollständig. Für diesen Sonderfall ist die Topologieerhaltung lokal nicht gewährleistet. Ebenfalls kann es bei sehr vielen Reduktionsstufen zu einer starken Verformung des Höhlenmeshes kommen. Räume und Gänge, die an vielen Seiten von der Höhle umschlossen sind, können so möglicherweise mit dieser kollidieren. Im Allgemeinen wird die Anzahl der Reduktionsstufen jedoch nicht so hoch sein, dass dieser Fall auftritt. Das größte Problem stellen Mesh Foldover dar.

Mesh Foldover

Mesh Foldover werden in [LRC⁺03, S.22] beschrieben und stellen das ineinanderschieben benachbarter Dreiecke dar. Dies kann bei den Reduktionsoperatoren Edge Collapse, Vertex-Pair Collapse und Triangle Collapse auftreten. [LRC⁺03] beschreibt die folgende Methode zur Verhinderung: Es wird getestet, ob es Normalen mit mehr als 90° Änderung durch den Collapse gibt. Wenn ja, dann liegt ein potentieller Foldover vor und der Collapse wird nicht durchgeführt. Allerdings ist dies kein generelles Kriterium. [ZTS05] berichtet von einem günstigen Grenzwert, der für diese Normalenänderung gefunden werden muss. Dort

8. Methoden zur Reduzierung des Renderaufwandes

wird $\frac{\pi}{3}$ verwendet.

Bei relativ gleichmäßigen Meshes mit eher lokal flachen Oberflächen funktioniert diese Methode des Foldover-Verhinderns recht gut. Ein Foldover ist hier i.d.R. mit hohen Normalenänderungen verbunden, da das entsprechende Dreieck „umklappt“. Bei stark unebenen, zerklüfteten Meshes kann aber eine kleine Normalenänderung schon einen Foldover bedeuten. Dies ist in Abbildung 8.7 dargestellt.

Das Problem ist: Wenn der Grenzwert zu niedrig ist, werden kaum Reduktionsoperationen durchgeführt. Wenn er zu hoch liegt, entstehen Mesh Foldover. Im schlimmsten Fall können mit dieser Methodik keine Reduktionsoperationen durchgeführt werden, ohne dass es zu Mesh Foldovers kommt, und zwar in dem Fall, wenn die Operation mit der niedrigsten Normalenänderung schon einen Mesh Foldover bewirkt.

Eine sichere Methode diese Foldover zu verhindern, besteht darin, alle durch die Kontraktion betroffenen Dreiecke zu testen, ob sie danach andere Dreiecke schneiden würden. Dies hätte aber beträchtlichen Rechenaufwand zur Folge.

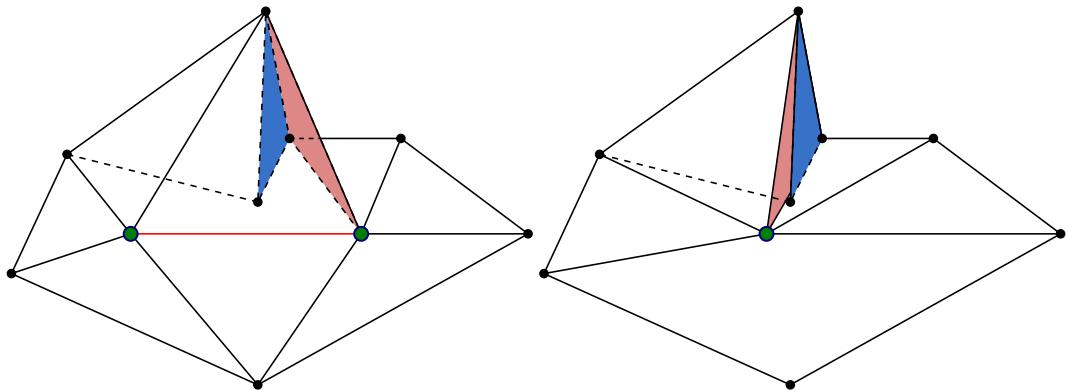


Abbildung 8.7.: *Mesh Foldover:* v.l.n.r.

- (a) Gezeigt ist die Ausgangstriangulation. Die zu kontrahierende Kante ist rot dargestellt.
- (b) Nach dem Edge Collapse gibt es einen Konflikt zwischen dem roten und blauen Dreieck. Die Normale des blauen Dreiecks wurde durch den Collapse nicht verändert, die des roten Dreiecks deutlich unter 90° . Durch horizontale Kontraktion der Skizze lassen sich Fälle mit Mesh Foldover bei beliebig kleiner Normalenänderung erzeugen.

Durch solche Foldover kommt es oft zu starken Beleuchtungsunterschieden. Bei der Verwendung von zweidimensionalen Texturen kommt es ebenfalls zu Texturunterschieden, bei dreidimensionalen Texturen nicht. Foldover können bei bestimmten Höhlenfarben und -texturierungen (speziell: Eis) vom Betrachter als weitere kristalline Strukturen wahrgenommen werden.

Mesh Foldover entstehen insbesondere bei sehr stark zerklüfteten Meshes, d.h. bei Verwendung von Erosion, und nach mehreren Reduktionsstufen, da sich der Fehler aufsummiert. Da die stark reduzierten Reduktionsstufen i.d.R. in größerer Entfernung angezeigt werden, ist das Problem dadurch etwas abgemindert. In Abbildung 8.6(b) sind solche Foldover an den lokalen Beleuchtungsunterschieden gut erkennbar.

9. Programmstruktur

Abbildung 9.1 zeigt den prinzipiellen Programmaufbau per Klassendiagramm nach UML 2.4.1 [OMG11a][OMG11b]. Abgebildet sind die wichtigsten Klassen mit ihren grundsätzlichen Beziehungen zueinander. Auf Angabe der Attribute und Operationen wurde aus Platzgründen verzichtet. Der Aufbau des Programms gliedert sich grundsätzlich in zwei Teile: zum einen die Klassen für die Ansteuerung, Eingabe und Ausgabe, zum anderen die Generatorklassen plus generierte Objekte.

CEventReceiver empfängt Inputs und leitet diese an die anderen Objekte weiter. *CGUI* definiert die Anordnung der GUI-Elemente. *COptionen* steuert die einzelnen Klassen an, zusätzlich werden hier sowohl Einstellung als auch Ergebnisse geladen und gespeichert. In dieser Klasse ist zugleich der Pipeline-Ablauf definiert. *CSzene* stellt sowohl die GUI als auch die Dungeonszene dar. Hierzu sind Methoden vorhanden, um die Dungeonszene aus den einzelnen Teilen zusammenzusetzen. Die *main*-Klasse enthält neben Initialisierungen eine Programmschleife zum Betrieb der Irrlicht-Engine.

CFraktalGenerator erstellt L-Systeme und deren Ableitungen, welche die Turtle-Grafik-Zeichenanweisungen darstellen. *CVoxelRaum* verwaltet den Voxelraum, zeichnet die Turtle-Grafik-Zeichenanweisungen und führt Erosion sowie Filterung schwebender Fragmente durch. *CDreiecksMesh* verwaltet die Höhlensubnetze, führt die Umwandlung Voxel zu Dreiecksnetz durch, berechnet Sichtbarkeitsinformationen der Höhle und führt die Reduktion der Höhlensubnetze durch. *CArchitekt* verwaltet Vorlagen für Räume, Detailobjekte sowie die Grundkonfiguration der Gänge und konstruiert aus einer gegebenen Voxelhöhle durch das Platzieren von Gängen und Räumen den Dungeon. Bei der Erstellung von Andockstellen an der Höhle wird auf Vertexkoordinaten-Berechnungsroutinen von *CDreiecksMesh* zurückgegriffen. Die Räume *SDungeonRaum* tragen einen Verweis auf die zugrundeliegende Subszene *CSubSzene*, welche die Geometrie des Raums enthält. Jeder Gang ist eine Instanz von *CSpline*. Für deren Erstellung werden von *CArchitekt* aus Gangparameter, ermittelte Andockstellen sowie Informationen aus Detailobjektvorlagen *SSplineDetailobjektVorlage* an *CSpline* übergeben. Hier erfolgt die Generierung der Ganggeometrie und Adaptergeometrien sowie die Platzierung der Detailobjekte als *SSplineDetailobjekt*. Sichtbarkeitsinformationen der Gänge und Gangreduktionsstufen werden ebenfalls in *CSpline* berechnet.

Es existieren zwei separate Zufallsgeneratoren vom Typ *CZufallsGenerator*: einer ist für *CDreiecksMesh* zuständig und wird dort für das Verwackeln der Eckpunkte eingesetzt, der andere liefert Zufallswerte für die restlichen Klassen.

9.1. Anmerkung zum Rendering

Die generierten Dreiecksnetze der Höhle sowie die der Gänge mit ihren Adaptoren sind bezüglich der Positionen ihrer Vertices bereits korrekt zueinander ausgerichtet. Das bedeutet, dass sie untransformiert in den Szenenraphen eingebunden werden können. Beim Rendering per Shader entfallen somit die Transformationen zur Berechnung der resultierenden Vertexpositionen und Normalen. Es braucht nur die Projektionstransformation von 3D-Raumkoordinaten zu 2D-Bildschirmkoordinaten durchgeführt zu werden. Durch das Weglassen der übrigen Transformationen ist eine höhere Performanz erzielbar.

9. Programmstruktur

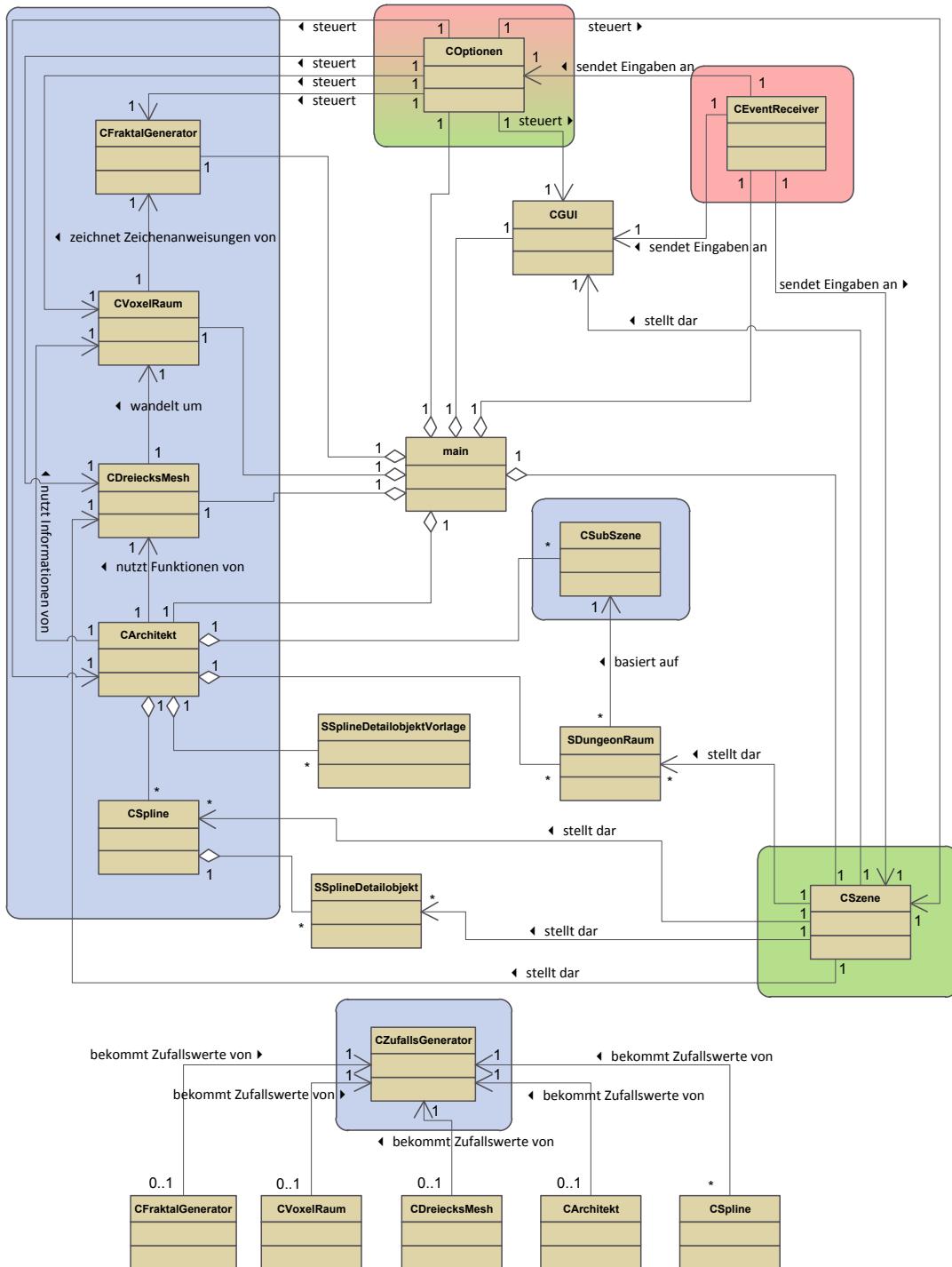


Abbildung 9.1.: *Programmaufbau - UML Klassendiagramm*: Dargestellt ist der prinzipielle Aufbau des Dungeongenerator-Programms. Auf die Darstellung von Klassen der Irrlicht-Engine wurde verzichtet. Der Inputteil ist *rot* markiert, die eigentlichen Generatoren *blau* und der Outputteil *grün*.

10. Auswertung

Dieses Kapitel beschäftigt sich mit konkreten Laufzeittests, Anmerkungen zu Parametern und weiteren Problemen. Danach wird eine Zusammenfassung getätigkt und ein Ausblick für fortsetzende Arbeiten gegeben.

10.1. Untersuchungen zu Parametern und Laufzeiten

Tabelle 10.1 zeigt einen Überblick über die durchgeführten Laufzeittests. Die Ergebnisse wurden für jeden Dungeon über 500 Messungen gemittelt. Die Abbildungen der verwendeten Dungeons finden sich in Anhang B, die zugehörigen XML-Konfigurationsdateien sind auf dem beiliegenden Datenträger mitgeliefert, siehe Anhang C.

Die aufgeführten Dungeons benötigen jeweils weniger als eine halbe Minute zur Erstellung. Tendenziell der größte Posten für die Laufzeit ist das Zeichnen der Voxelhöhle. Die Laufzeit hierfür ist stark abhängig von der Anzahl der Zeichenanweisungen, insbesondere Symbol 'F', sowie dem Strichradius. Beispiel B.1 hat sehr viele Zeichenanweisungen bei einem Strichradius von 14, und Beispiel B.5 hat sehr wenige Zeichenanweisungen, aber einen Strichradius von 32. Die Erosion ist algorithmisch vergleichsweise invariant gegenüber der Struktur des Voxelgebildes und benötigt so unabhängig vom Dungeon eine annähernd konstante Laufzeit. Filterung (schwebender Fragmente) 2 braucht ähnliche Zeiten für alle Dungeons. Es sind hier i.A. nur wenige, kleine schwebende Fragmente zu entfernen. Die vorhandenen Laufzeitunterschiede sind daher primär auf die unterschiedliche Ausdehnung der äußeren Hülle zurückzuführen. Bei Filterung 1 schwankt die Laufzeit stärker. Beispiel B.5 besitzt ein sehr großes schwebendes Fragment im Inneren, daher wird signifikant mehr Rechenzeit benötigt.

Der Aufbau der Sektor- und Scankarten hängt insbesondere von der Anzahl der zu erstellenden Scankarten ab. Beispiel B.4 stellt einen sehr flachen Dungeon dar, wodurch weniger Scankarten erstellt werden müssen. Dadurch ist die benötigte Laufzeit hier geringer. Die Laufzeit der Generierung von Gängen und Räumen ist nicht prinzipiell primär von der Anzahl der generierten Gänge abhängig. Falls starke Restriktionen bezüglich der Andockpositionen für die Gänge an der Höhle vorliegen, können die Andocktests den Primäranteil der Laufzeit bilden. Beispiel B.5 ist ein solcher Fall. Hier wurde per Bodenabstandstest festgelegt, dass die Gänge nur im unteren Bereich an der Höhle andocken sollen. Dadurch werden die meisten Andockpositionen verworfen. Wie sich zeigt, ist der Blickdichtigkeitstest für die Gänge recht effektiv, im Schnitt wurden über 84 Prozent aller Gänge als definitiv blickdicht erkannt.

Die Laufzeiten der Umwandlung und Reduktion der Höhlennetze skalieren in etwa linear der Anzahl der Vertices und Dreiecke. Die restlichen Berechnungen nehmen einen nahezu konstanten Anteil an der Gesamlaufzeit ein.

Beim Entwurf von Dungeons ist es sinnvoll, zuerst die Höhle zu erstellen und zu verfeinern. Erst dann, wenn die Höhle die gewünschte Struktur aufweist, sollten die Parameter für die Generierung der Gänge und Räume entsprechend angepasst werden.

Bei der Konstruktion von Höhlen zeigt sich, dass L-Systeme der Form $G = \langle \{\dots\}, F, \{F \rightarrow (F, +, -, o, u, g, z)^*\} \rangle$ relativ oft zu realistisch anmutenden Höhlen führen (siehe hierzu Beispiel B.3 und B.4). Die Winkel sollten dabei möglichst unregelmäßig gewählt

10. Auswertung

Dungeon	B.1	B.2	B.3	B.4	B.5
Erstellung Zeichenanweisungen	39	32	27	26	28
Anzahl Zeichenanweisungen	826992	181989	39061	21845	1367
Zeichnen	17219	1778	4239	3146	7164
Erosion	1586	1585	1579	1576	keine
Filterung 1	590	625	555	590	3300
Sektoren- und Scankartenaufbau	1337	1317	1766	820	2465
Generierung Räume und Gänge	1799	1830	2994	1156	2285
Generierung Gänge (inkl. LOD)	908	1026	1795	739	293
Platzierung Detailobjekte	23	24	42	18	5
Rest (Räume, Andocken, u.a.)	868	780	1157	399	1987
Anzahl Räume	35	42	66	27	16
Anzahl Gänge	49	56	97	40	16
davon definitiv blickdicht	44	44	84	39	11
Filterung 2	588	625	556	589	733
Umwandlung Voxelhöhle	1890	2285	1911	2151	5002
davon Normalenberechnung	286	464	316	434	824
Anzahl Vertices	400724	683354	441594	621232	1315602
Anzahl Dreiecke	822748	1390848	892120	1274544	2596588
Anzahl Höhlensubnetze	13	36	36	28	56
Reduktion Höhlennetze Stufe 1	458	813	586	732	1399
Anzahl Vertices	226202	387130	253981	353852	730974
Anzahl Dreiecke	473704	798400	516894	739784	1427332
Reduktion Höhlennetze Stufe 2	314	568	434	515	901
Anzahl Vertices	129322	221471	149665	206312	400090
Anzahl Dreiecke	279944	467082	308262	444704	765564
Reduktion Höhlennetze Stufe 3	229	429	346	392	613
Anzahl Vertices	79594	136054	95790	130910	225497
Anzahl Dreiecke	180488	296248	200512	293900	416378
Reduktion Höhlennetze Stufe 4	189	358	302	331	462
Anzahl Vertices	54817	93064	68611	93689	134969
Anzahl Dreiecke	130934	210268	146154	219458	235322
Restliche Berechnungen	517	528	519	524	552
Gesamt	26755	12773	15814	12548	24904

Tabelle 10.1.: Laufzeiten und weitere Daten für die Generierung von Dungeons

Laufzeiten sind grün unterlegt und jeweils in Millisekunden angegeben.

Gemessen auf: *Prozessor:* AMD Phenom II X4 945, 3 Ghz ,

RAM: 3,25 GiB Dual Channel DDR2-800 ,

Betriebssystem: Microsoft Windows 7 Professional 64 Bit mit Service Pack 1

10. Auswertung

werden, d.h. keine Vielfachen von 90° . Bei Verwendung von nur '+', '-' als Symbole für Drehungen entstehen ebene Höhlen. Bei zusätzlicher Verwendung von 'o', 'u', 'g', 'z' ist das Höhlenskelett in alle Dimensionen ausgedehnt. Symbol '\$' kann dabei helfen, eine sonst sehr unebene Höhle ebener zu gestalten (vgl. Beispiel B.1). Der Einsatz von Erosion ist nicht zwingend erforderlich, sondern von der Art der gewünschten Höhle abhängig. Gleiches gilt für die Filterung schwebender Fragmente.

Bei der Konstruktion des restlichen Dungeons sollte zunächst die Wahl der Sektogröße abhängig von der Höhlenbeschaffenheit durchgeführt werden. Falls z.B. mittlere Freiräume ausgenutzt werden sollen (vgl. Beispiel B.2), muss die Sektogröße entsprechend klein gewählt werden. Die Skalierung der Subszenen ist dabei entsprechend anzupassen, damit diese gut in die Sektoren passen. Als Faktor g für die Stärke der Gansplineableitungen zeigen Werte zwischen 0,5 und 1,5 gute Resultate. Parameter R und F für den Abstandstest zum Boden sollten nach Abhängigkeit von der Höhlenstruktur angepasst werden, bei größeren Zeichenradien oder dem Anwenden von Erosion sind u.U. höhere Werte sinnvoll.

10.2. Weitere Probleme



Abbildung 10.1.: *Lücken bei Raumadaptersn*: Hintergrundfarbe ist Weiß. Zu sehen ist ein schmaler weißer Spalt zwischen Adapter und umschließender Raumgeometrie.

An Adaptersn mit Anschluss an Räumen kann es zu feinen Lücken zwischen Raum und Adapter kommen, siehe Abbildung 10.1. Diese Lücken sind bei kontrastreichem Hintergrund zu erkennen. Blendet man die Raumgeometrie aus und löscht den dem Adapter zugrundeliegenden Andockstellenmesh nicht, so sind keine Lücken zwischen Adapter und Andockstellenmesh sichtbar.¹ Die Lücken bestehen also schon zwischen Andockstellenmesh und der umschließenden Subszenengeometrie. Sie sind somit bereits in der importierten .irr-Szene enthalten. Ungenauigkeiten beim Zusammenbau der jeweiligen Szenen im genutzten Szeneneditor Ambiera IrrEdit konnten durch Betrachtung der exportierten Daten ausgeschlossen werden. Ursache sind demnach Ungenauigkeiten beim Modelling und Export der

¹Bei ausgeschaltetem Antialiasing. Bei aktiviertem Antialiasing können dennoch Artefakte auftreten. Ursache: Die für diese Arbeit erstellten Subszenen haben in ihren Materialeigenschaften das Flag „Antialiasing“ nicht aktiviert, da die verwendete IrrEdit-Version dies nicht unterstützt. Daher wird das Antialiasing nicht auf sie angewendet, sondern nur auf den Adapter. Dies führt zu Artefakten an den gemeinsamen Kanten.

10. Auswertung

3D-Daten aus dem verwendeten 3D-Modelling-Progamm Autodesk 3DS Max 2010.² Eine mögliche Lösung wäre die Schaffung eines Plugins zur Modellierung sowie Export von Andockstellen und umgebender Geometrie für das jeweilig benutzte 3D-Modelling-Programm.

10.3. Zusammenfassung und Bewertung

Das entwickelte Verfahren zur Erstellung von Dungeons folgt dem natürlichen Entstehungsprozess: Zuerst findet die Generierung der Höhle statt, danach erfolgt das Hinzufügen von Gängen sowie Räumen.

Das Vorgehen für die Höhlenerstellung basiert auf Lindenmayer-Systemen. Als Erstes erfolgt die Generierung von Turtle-Grafik-Zeichenanweisungen auf der Basis eines L-Systems. Die L-Systeme zur Vorgabe der Höhlengrundstrukturen ermöglichen hierbei realistisch anmutende Höhlensysteme. Dann findet das Zeichnen der Struktur im Voxelraum statt. Danach wird eine Nachbearbeitung per Erosion und dem Herausfiltern schwiebender Fragmente durchgeführt, um die entstandenen Strukturen realitätsnaher zu gestalten.

Unter Ausnutzung der Voxel-Informationen werden Räume um die Höhle herum platziert und Gänge zwischen Räumen und der Höhle verlegt. Die Räume werden als Kopien vorgefertigter Szenen angelegt und ermöglichen so die Einbindung eigener Dungeonteile. Die Gänge sind polygonale Schläuche, die dem Verlauf kubischer Hermite-Splines folgen. Hierdurch sind gewundene Gangstrukturen möglich.

Die Voxelhöhle wird danach in ein Dreiecksnetz umgewandelt, welches aus mehren Subnetzen besteht. Bei der Umwandlung werden die Koordinaten der Eckpunkte verwackelt. Der Verwacklungsprozess kann zur Glättung der entstehenden Netze eingesetzt werden. Die entstehenden Strukturen ergeben zerklüftete Höhlen. Diese ließen sich in Verbindung mit prozeduralen Texturen realistischer gestalten.

Für die Verwendung von Techniken des Culling und des Level of Detail werden Informationen zum Aufbau eines Sichtbarkeitsgraphen generiert. Diese geben an, welche Dungeonteile mit welchen anderen wie verknüpft sind und welche Sichtbarkeitseigenschaften die Dungeonteile besitzen. Die Dreiecksnetze der Gänge und der Höhle werden für die Erstellung niedrigerer Detailstufen reduziert. Die Reduktion der Gangnetze ist qualitativ hochwertig, die der Höhlnetze noch nicht. Hier verhindern insbesondere Mesh Foldover bei zunehmenden Reduktionsstufen hohe Darstellungsqualität.

Die generierten Dungeons sind visuell ansprechend und lassen sich in Computerspielen verwenden. Die Höhlen sind realen Höhlen ähnlich, wie ein Vergleich der Lavahöhle aus Abbildung 2.1(b) und den erodierten Höhlen aus Abbildung B.6 zeigt. Andere Höhlenarten lassen sich durch Modifizierung von L-Systemen, Erosion und Texturierung bilden. Die erstellten Gänge sind geschwungen und für Dungeons sehr gut geeignet. Die entworfenen und implementierten Algorithmen sind effizient in Laufzeit und Speicherbedarf. Dungeons lassen sich i.d.R. in weniger als einer Minute komplett erstellen.

Die fertig erstellten Dungeons lassen sich exportieren. Allerdings sind die exportierten Wavefront OBJ-Meshes sehr speicherplatzintensiv, da sie als Text und nicht binär gespeichert werden. Damit benötigt auch der Gesamtdungeon viel Speicherplatz zur Ablage. Als Lösung ergibt sich hier zum einen die Verwendung eines anderen Exporters (binäres Format statt Text, z.B. 3DS³). Auch könnte der Dungeongenerator als Programmbibliothek

²Die Andockstellenmeshes und die umschließende Subszenegeometrie wurden größtenteils aus den gleichen Ausgangsobjekten mittels unterschiedlich verknüpften CSG-Operationen erstellt.

³Allerdings besitzt Irrlicht keinen 3DS-Exporter.

10. Auswertung

thek verwendet werden. Der gesamte Dungeon ließe sich dann bei Bedarf aus einer XML-Konfigurationsdatei deterministisch generieren, wodurch der benötigte Speicher zur Ablage der Dungeons extrem reduziert wäre.⁴ Die Generierungsgeschwindigkeit der Algorithmen reicht aus, um diese Methode praktikabel zu verwenden.

10.4. Ausblick

Die Parameter der L-Systeme, insbesondere die Winkel, könnte man beim Zeichnen zufällig variieren, um weitere Möglichkeiten zur Vorgabe von Höhlenstrukturen schaffen. Die Erosion im Voxelraum lässt sich besser steuerbar und realistischer gestalten, indem die Erosionswahrscheinlichkeit für jede Richtung (oben, unten, ...) einzeln festlegbar ist.

Eine bessere Aufteilung bei der Zerlegung der Höhle in Subnetze ist wünschenswert. Statt eines Octrees wäre ein Verfahren, welches die Struktur der Höhle besser einbezieht, geeigneter. So wären weniger Grenzvertices nötig und eine bessere Aufteilung für den Sichtbarkeitsgraphen möglich. Die Sektoraufteilung für die Raumplatzierung lässt sich ebenfalls adaptiv gestalten. Die Aufteilung in Sektoren würde nicht per Raumgitter erfolgen, sondern auch hier die Struktur der Höhle einbeziehen.

Die Heuristik zum Finden geeigneter Andockpositionen kann weiter verbessert werden. Beispielsweise ließe sich zusätzlich der Bereich links und rechts der Andockposition betrachten, statt nur der Bereich in Richtung Boden. Beim Andockvorgang von Gängen an die Höhle lässt sich der Querschnitt exakter ausfräsen. Statt der Verwendung eines quadratischen Andockbereiches könnte man hier den Gangquerschnitt per Voxeln abbilden. Weiterhin sinnvoll ist das Zulassen beliebiger Andockrichtungen, nicht nur entlang der X- bzw. Z-Achse.

Prozedurale Räume und prozedurale Detailobjekte sind ebenfalls denkbar. Die Räume könnten ähnlich dem Verfahren zur Generierung prozeduraler Gebäude aus [Jan07] ebenfalls mit L-Systemen erstellt werden. Auch eine „beliebige“ Anzahl von Andockstellen für Räume wäre möglich. Die Platzierung von Detailobjekten in Höhlen, beispielsweise Statuen an Gangmündungen oder Monster und Fallen auf relativ ebenen Flächen, ist ein weiterer Fortführungspunkt. Dafür ließen sich die Informationen aus dem Voxelraum bzw. für Gangmündungen zusätzlich die direkten Splineparameter nutzen. Denkbar wären Parameterangaben wie: Objekt mit Ausdehnung $X \times Y \times Z$ soll N Voxel über dem Boden platziert werden, mit einer Toleranz von M Voxeln.

Statt nur die Höhlenvertices zu verwackeln, könnte auch der Querschnitt des Gangs optional verwackelt werden. Auch das Ausprobieren und Vergleichen anderer Umwandlungstechniken von Voxeln in Dreicksnetze, wie beispielsweise des Marching Cubes-Algorithmus [LC87], ist denkbar. Andere Zufallsgeneratoren, separat für jeden Teil des Dungeongenerators, könnten getestet werden.

Die Texturierung der Höhle durch prozedurale 3D-Texturen sollte weiter betrachtet werden. Die Feinstruktur der Höhle lässt sich beispielsweise durch prozedurale 3D-Displacement-Texturen verbessern. Zusätzliche Untersuchungen bezüglich Reduktionstechniken für Dreiecksnetze sind notwendig, besonders um mit wenig Rechenaufwand Mesh Foldover verhindern zu können.

⁴I.d.R. werden statt über 100 MiB beim OBJ-Export nur noch ~15 KiB benötigt.

A. Export

Der Export des Dungeons erfolgt als (a) .irr-Datei für die komplette Dungeon-Szene, (b) Wavefront OBJ-Dateien für die erstellten Höhlen- und Ganggeometrien sowie (c) Zusatzinformationen zum Aufbau des Sichtbarkeitsgraphen als XML-Datei. Zur weiteren Verwendung der exportierten Daten müssen sich alle im Folgenden genannten Dateien sowie die in den genutzten Subszenen enthaltenen Texturen und Meshes im gleichen Ordner befinden und auch so weitergegeben werden.

Die Benennung der .irr-Szenendatei erfolgt als: „*[Dungeonname].irr*“.

Der Aufbau des Szenographen der exportierten Dungeon-Szene wird im Folgenden dargestellt. Die Benennung und ID's zum Wiederauffinden jedes Dungeonbestandteils sind dabei geschrieben als (*Benennung, ID*). Eine ID von -1 bedeutet, dass keine explizite ID vergeben wird, eine ID-Angabe von 0++ bedeutet aufsteigende ID's, beginnend mit 0.

- (*Dungeon,-1*) ⇒ kompletter Dungeon
 - (*Raeume,-1*) ⇒ alle Räume
 - * (*[gewählte Benennung],0++*) ⇒ nacheinander alle Räume mit ihrer Bezeichnung und ihren ID's
 - (*GangDetailstufe,0++*) ⇒ alle vorhandenen Detailstufen der Gänge
 - * (*Gang,0++*) ⇒ nacheinander alle Gänge mit ihren ID's
 - (*GangAdapter,-1*) ⇒ alle vorhandenen Gangadapter
 - * (*Adapter0,0++*) ⇒ nacheinander alle Adapter an Ende $\overrightarrow{P(0)}$ des Gangs mit der zum Gang gehörigen ID
 - * (*Adapter1,0++*) ⇒ nacheinander alle Adapter an Ende $\overrightarrow{P(1)}$ des Gangs mit der zum Gang gehörigen ID
 - (*Detailobjekte,-1*) ⇒ alle vorhandenen Detailobjekte
 - * (*DetailobjektAnGang,0++*) ⇒ nacheinander alle Detailobjekte des Gangs mit der zum Gang gehörigen ID
 - (*[gewählte Benennung],0++*) ⇒ nacheinander alle Detailobjekte mit ihrer Benennung, die ID's werden von $\overrightarrow{P(0)}$ bis $\overrightarrow{P(1)}$ aufsteigend gezählt
 - (*HoehlenDetailstufe,0++*) ⇒ alle vorhandenen Detailstufen der Höhle
 - * (*Hoehlenteil,0++*) ⇒ nacheinander alle Subnetze der Höhle mit ihren ID's

Die Benennung der exportierten Höhlengeometrien erfolgt als:

„*[Dungeonname]_HoehlenMesh_[SubnetzID]_Detailstufe_[DetailstufenNummer].obj*“.

Die Benennung der exportierten Gangmeshes erfolgt als:

„*[Dungeonname]_Gang_[GangID]_Detailstufe_[DetailstufenNummer].obj*“.

Die Benennung der exportierten Adaptermeshes erfolgt als:

„*[Dungeonname]_Gang_[GangID]_Adapter_[0 bzw. 1].obj*“.

Zu den OBJ-Meshes zugehörig ist die ebenfalls erzeugte Datei „*DungeonBasisMaterial.mtl*“, die ein Standardmaterial enthält.

Die Benennung der Datei für die Zusatzinformationen erfolgt als:

„*[Dungeonname]_Zusatzinformationen.xml*“.

Der Aufbau der Zusatzinformationen ist Folgender:

A. Export

```

<DungeonInformationen>
  <Raueme>
    <!--Aufzählung aller Räume:-->
    <Raum ID="[ID des Raums]">
      <GangIDs Gang0Nord="[ID des angeschlossenen Gangs, bzw. -1 für keinen]">
        Gang1Ost="[...]" Gang2Sued="[...]" Gang3West="[...]" />
    </Raum>
  </Raueme>

  <Gaenge GangBreite="[Breite der Gänge in Voxeln]">
    <!--Aufzählung aller Gänge:-->
    <Gang ID="[ID des Gangs]">
      DefinitivBlickdicht="[0 oder 1]">
      <RaumIDs RaumAndockstelle0="[ID des angeschlossenen
        Raums, bzw. -1 für Höhle]" RaumAndockstelle1="[...]" />
      <Position0 X="[...]" Y="[...]" Z="[...]" />
      <Position1 X="[...]" Y="[...]" Z="[...]" />
      <Ableitung0 X="[...]" Y="[...]" Z="[...]" />
      <Ableitung1 X="[...]" Y="[...]" Z="[...]" />
    </Gang>
  </Gaenge>

  <Hoehle>
    <!--Aufzählung aller Höhlensubnetze:-->
    <HoehlenTeil ID="[ID des Subnetzes]">
      <VoxelMin X="[Min X des zugehörigen Voxelgebietes]"
        Y="[...]" Z="[...]" />
      <VoxelMax X="[...]" Y="[...]" Z="[...]" />
      <BlickdichtNegativeRichtung X="[kann nicht nach X-negativ
        gesehen werden? 0 oder 1]" Y="[...]" Z="[...]" />
      <BlickdichtPositiveRichtung X="[...]" Y="[...]" Z="[...]" />
    </HoehlenTeil>
  </Hoehle>
</DungeonInformationen>

```

Weitere Exportmöglichkeiten

Die Höhle lässt sich auch einzeln in einer OBJ-Datei mit frei wählbarem Dateinamen exportieren. Alle Höhlensubmeshes werden dabei in diese Datei geschrieben. Die Datei „*DungeonBasisMaterial.mtl*“ wird hier ebenso erzeugt und ist zur OBJ-Datei zugehörig.

Die aus L-Systemen erzeugten Turtle-Grafik-Zeichenanweisungen lassen sich ebenfalls exportieren. Als Dateiformat wird XML verwendet. Die zum Zeichnen relevanten Parameter werden ebenfalls gespeichert. Der Dateiaufbau gestaltet sich wie folgt:

```

<LSystemAbleitungen>
  <LSystemParameter>
    <Winkel WinkelGier="[...]" WinkelNick="[...]" WinkelRoll="[...]" />
    <Radius StartRadius="[...]" RadiusFaktor="[...]"
      RadiusDekrementor="[...]" />
  </LSystemParameter>

  <!-- alle generierten Ableitungen, beginnend mit Iteration 0:-->
  <Iteration Nummer="Iterationsstufe">
    String="[Generierter String]" />
  </LSystemAbleitungen>

```

B. Weitere Beispiele für Dungeons und Höhlen

Im Folgenden finden sich einige Beispiele für Dungeons, die mittels des Dungeongenerator-Programms generiert wurden. Links ist jeweils die Höhle abgebildet, rechts der fertige Dungeon. Zu jeder Höhle ist das zugrundeliegende L-System angegeben. Anschließend finden sich einige Screenshots aus dem Inneren von Dungeons.

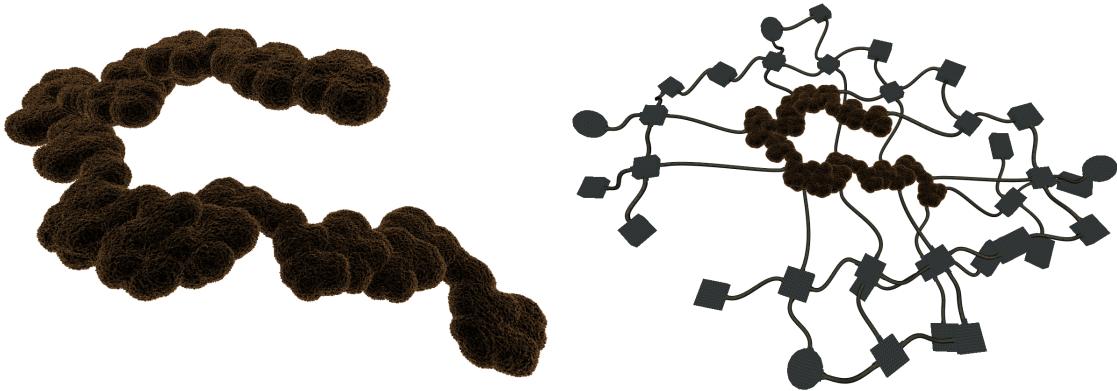


Abbildung B.1.: *Dungeon Beispiel 1:* $G = \langle \{F, X, Y, +, -, o, \$\}, YYFYF,$
 $\{F \rightarrow F - YX - X ---, X \rightarrow F\$F++F - X, Y \rightarrow oYX --XX ++\} \rangle$
mit $\alpha_{Gier} = 250^\circ$, $\alpha_{Nick} = 1^\circ$, Startradius = 14, 8. Iteration

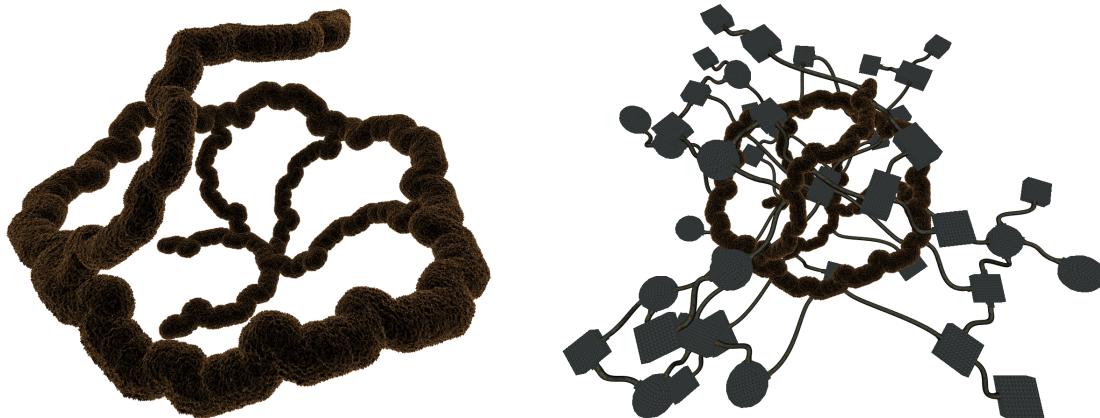


Abbildung B.2.: *Dungeon Beispiel 2:* $G = \langle \{F, +, o, u, g, z, [,], !\}, F + [! + uFoF]gFz + [! + uFoxxxxxF]ggFzz + [! + uF]gFz + F + [! + oFuF]zF,$
 $\{F \rightarrow F - F + FF + F - F\} \rangle$
mit $\alpha_{Gier} = 60^\circ$, $\alpha_{Nick} = 45^\circ$, $\alpha_{Roll} = 20^\circ$, $r_f = 1$, $r_d = 4$,
Startradius = 16, 5. Iteration

B. Weitere Beispiele für Dungeons und Höhlen



Abbildung B.3.: *Dungeon Beispiel 3*: $G = \langle \{F, +, -, o, u, g, z\}, F, \{F \rightarrow F + F o F g - F u z F\} \rangle$ mit $\alpha_{Gier} = 285^\circ$, $\alpha_{Nick} = 255^\circ$, $\alpha_{Roll} = 255^\circ$, Startradius = 13, 6. Iteration

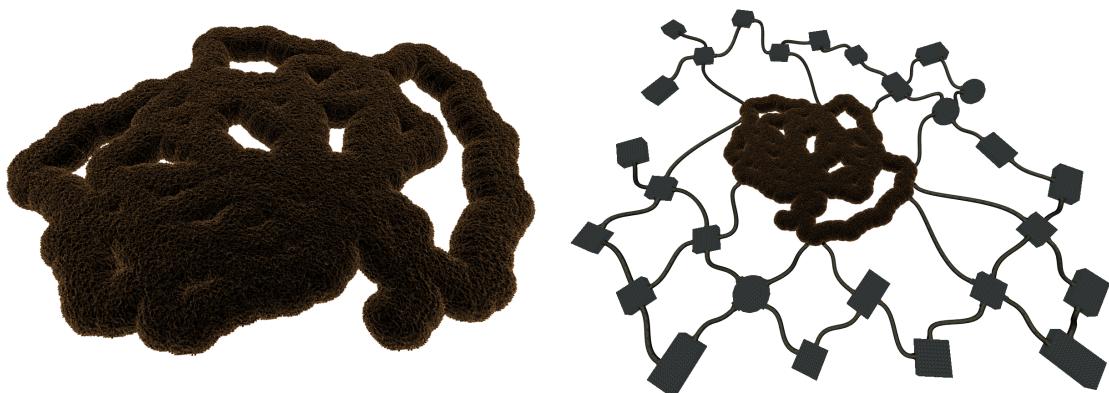


Abbildung B.4.: *Dungeon Beispiel 4*: $G = \langle \{F, +\}, F, \{F \rightarrow F + F F F\} \rangle$ mit $\alpha_{Gier} = 68^\circ$, Startradius = 16, 7. Iteration



Abbildung B.5.: *Dungeon Beispiel 5*: $G = \langle \{F, W, X, Y, +, o, [,]\}, W, \{W \rightarrow W + [X], X \rightarrow Y Y Y Y Y Y Y Y, Y \rightarrow F o F o F o F o\} \rangle$ mit $\alpha_{Gier} = 10^\circ$, $\alpha_{Nick} = 10^\circ$, Startradius = 32, 20. Iteration

B. Weitere Beispiele für Dungeons und Höhlen

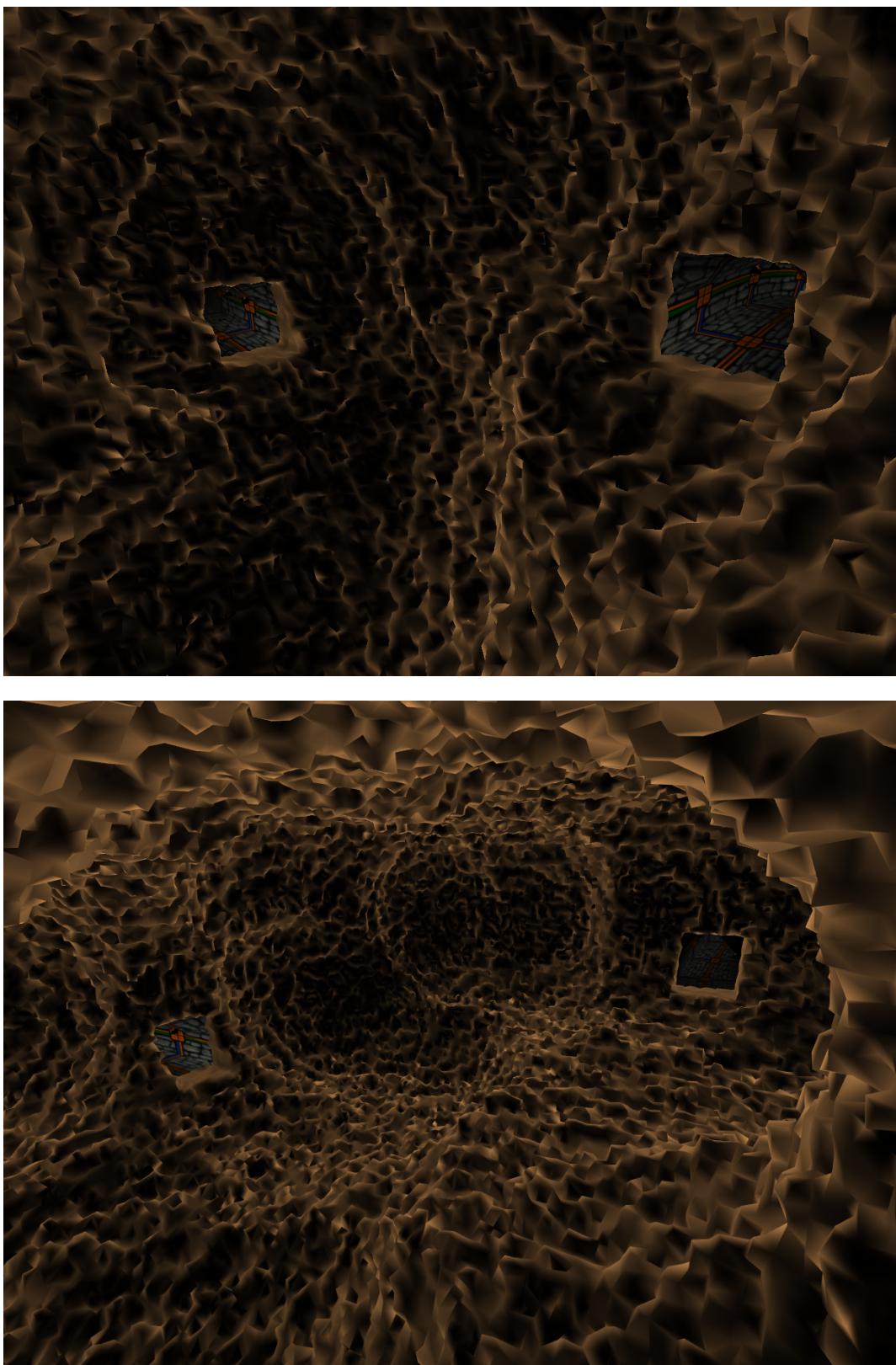


Abbildung B.6.: *Dungeons von innen*: beide Höhlen unter Einfluss von Erosion, Umwandlung per Verwackeln und Glätten

B. Weitere Beispiele für Dungeons und Höhlen

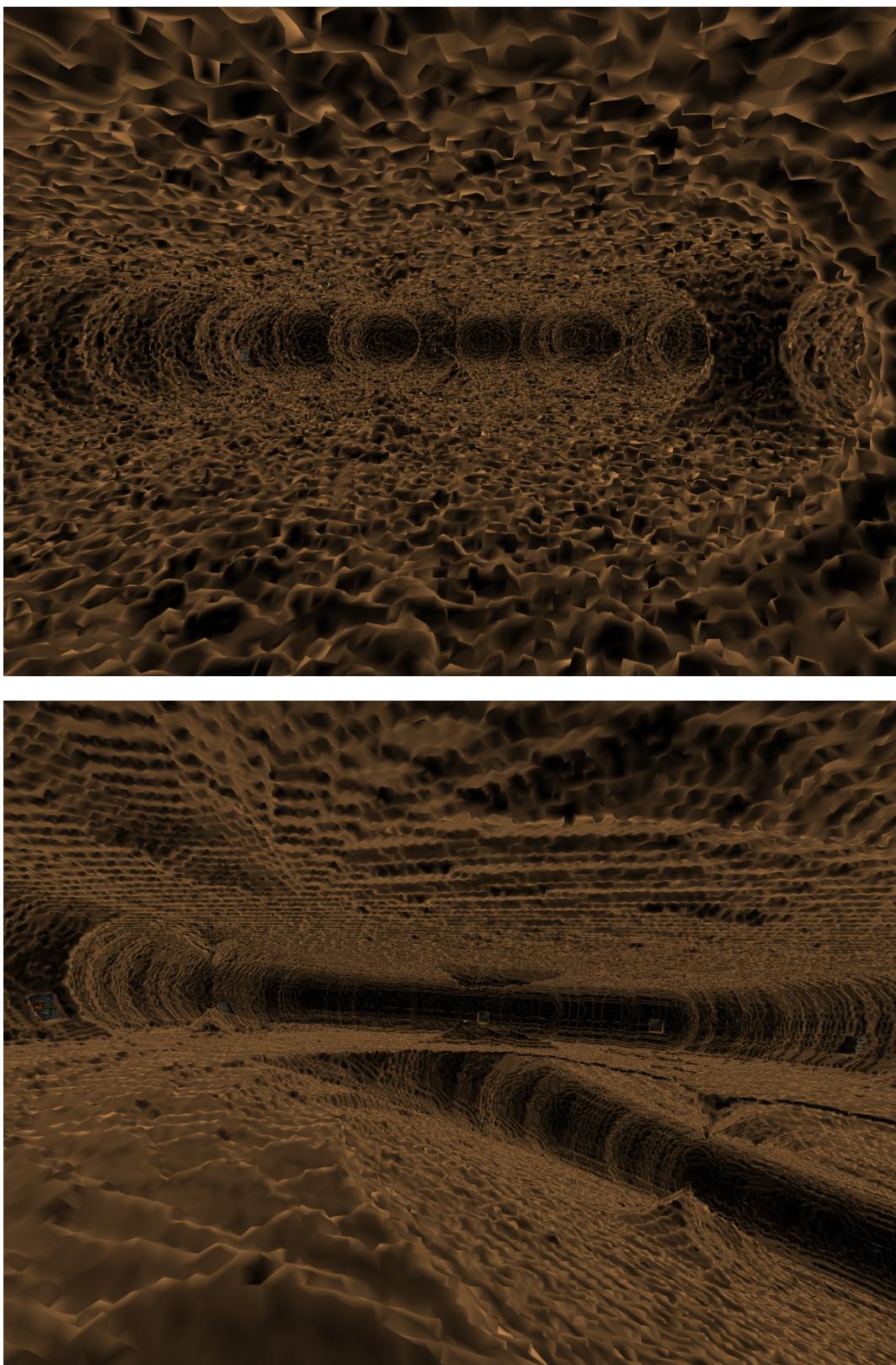


Abbildung B.7.: *Dungeons von innen ff. v.o.n.u.* (a) Höhle unter Einfluss von Erosion, (b) Höhle ohne Einfluss von Erosion, beide: Verwackeln und Glätten

C. CD-Inhalt

Inhalt des beiliegenden Datenträgers:

- Ausarbeitung als digitale Version:
Datei „\MaxHoenig_Masterarbeit.pdf“
- Anleitung für das beiliegende Programm:
Datei „\Anleitung_Dungeongenerator.pdf“
- Anleitung zum Kompilieren des Programms:
Datei „\Anleitung_Kompilierung.pdf“
- Übersicht über die Namenskonventionen, die innerhalb des Programms verwendet wurden:
Datei „\Namenskonventionen.pdf“
- Programmcode in C++ als Projekt für Microsoft Visual Studio 2010 mit Service Pack 1:
Ordner „\Programmcode“
- Für Windows x86 kompiliertes Programm (benötigt OpenGL):
Ordner „\Dungeongenerator“
- Irrlicht, Version 1.7.3 (verwendete Version) [Irr12]:
Ordner „\Extern\irrlicht-1.7.3“
- Szeneneditor IrrEdit (ermöglicht die Erstellung von .irr-Szenen), Version 1.5 [Amb12]:¹
Ordner „\Extern\irrEdit-1.5“
- Modellings der für die Subszenen verwendeten Basisgeometrien, Andockgeometrien, Verschlussgeometrien und Andockstellen als Autodesk 3DS Max 2010 .max-Dateien:
Ordner „\Modellings“

Beispieldateien

Beispieldateien für Höhlen, Dungeons und Höhlenfarben befinden sich im Verzeichnis „\Dungeongenerator“.

Beispiele für höhlenartige L-Systeme und bekannte L-Systeme:

- Abbildung 3.1: „LSystem_Hoehle_Flach1.xml“
- Abbildung 7.6 unten: „LSystem_Hoehle_Flach2.xml“
- Abbildung 7.6 oben: „LSystem_Hoehle_Tief1.xml“
- viele der Abbildungen, die eine Höhle von innen zeigen: „LSystem_Baum.xml“ [RS92, S.25] (stellt von außen betrachtet eine Baumstruktur dar, eignet sich aber von innen betrachtet auch als Höhle)

¹Bugs in Irredit 1.5: 1.) Das Einlesen von Meshes, die Meshbuffer mit mehr als ~ 32k Vertices erzeugen, resultiert in inkorrekten Meshes. Vermutete Ursache: Verwendung von *signed short* statt *unsigned short* als Datentyp für die Vertex-Indices. 2.) Das Ändern von Materialeigenschaften kann die Texturkoordinaten eines Objekts auf falsche Werte setzen.

C. CD-Inhalt

- Drachenkurve: „LSystem_Drachenkurve.xml“ [RS92, S.11]
- Waben: „LSystem_Waben.xml“ [Uni11]
- Koch-Kurve: „LSystem_Koch-Kurve.xml“
- Kochsche Schneeflocke: „LSystem_Kochsche-Schneeflocke.xml“

Beispiele für Dungeons:

- Abbildung B.1: Datei „Dungeon_Aufsteigend.xml“
- Abbildung B.2: Datei „Dungeon_Ring.xml“
- Abbildung B.3: Datei „Dungeon_Tief.xml“
- Abbildung B.4: Datei „Dungeon_Weit.xml“
- Abbildung B.5: Datei „Dungeon_Doppelkugel.xml“
- Dungeon aus Abbildung 7.6 oben, etwas modifiziert: „Dungeon_Verdreht.xml“
- Abbildung B.7(b): „Dungeon_Bizarr.xml“

Farben für Höhlen:

- Bernstein : „Farbe_Bernstein.xml“
- Eis : „Farbe_Eis.xml“
- Rubin : „Farbe_Rubin.xml“
- Smaragd : „Farbe_Smaragd.xml“
- Amethyst : „Farbe_Amethyst.xml“

Abbildungsverzeichnis

2.1. Diverse in der Natur vorkommende Höhlen	5
2.2. Diverse Dungeons	6
2.3. Bilder von Roguelikes	8
3.1. L-Systeme für Höhlenstrukturen	11
4.1. Konstruktion der Koch-Kurve	17
4.2. Aus Voxeln aufgebauter Torus	20
4.3. Rasterung einer Linie nach dem Bresenham-Verfahren	22
4.4. Entstehung von Lücken beim Zeichnen des Zylinders	25
4.5. Scharten am Rand des Zylinders	25
4.6. Erosion im Voxelraum	27
4.7. Schwebende Fragmente im Voxelraum	28
5.1. Voxelkoordinaten zu Vertexkoordinaten	31
5.2. Verwacklung der Vertexkoordinaten	35
5.3. Modifizierung der Vertexkoordinaten	37
5.4. Wichtung der Kanten	38
5.5. Vergleich von Normalenberechnungsmethoden	41
5.6. Vergleich von Beleuchtungsmethoden	43
6.1. Kubischer Hermite-Spline	46
6.2. Gänge per kubischem Hermite-Spline	47
6.3. Prinzip des Adapters	49
6.4. Notwendigkeit von Stützvertices für Andockstellen	50
6.5. Adapter aus verschiedenen Andockstellen	54
6.6. Objekte in Gängen	56
6.7. Raum mit Adaptern	58
7.1. Generierung und Revalidierung von Scankarten	63
7.2. Bestimmung einer gültigen Andockposition	64
7.3. Test der konvexen Hülle per Pyramide	67
7.4. Andocken an Höhle	68
7.5. An Höhle angedockter Gang	70
7.6. Beispieldungeons von außen	71
8.1. Blickdichtigkeitstest für Gänge	74
8.2. Blickdichtigkeit von Gängen	76
8.3. Edge Collapse	78
8.4. Verlust der Mannigfaltigkeit bei mehr als zwei gemeinsamen Nachbarn	78
8.5. Reduktion der Höhlennetze	82
8.6. Reduktion der Höhlennetze ff.	83
8.7. Mesh Foldover	85
9.1. Programmaufbau - UML Klassendiagramm	87

Abbildungsverzeichnis

10.1. Lücken bei Raumadaptionen	90
B.1. Dungeon Beispiel 1	95
B.2. Dungeon Beispiel 2	95
B.3. Dungeon Beispiel 3	96
B.4. Dungeon Beispiel 4	96
B.5. Dungeon Beispiel 5	96
B.6. Dungeons von innen	97
B.7. Dungeons von innen ff.	98

Literaturverzeichnis

- [Amb12] AMBIERA: *irrEdit*. Website. <http://www.ambiera.com/irredit/>. Version: 2012, Abruf: 18. Juni 2012
- [BB06] BURGER, Wilhelm ; BURGE, Mark J.: *Digitale Bildverarbeitung*. Springer-Verlag, 2006. – ISBN 3–540–30940–3. – 2. Auflage
- [BF01] BURDEN, Richard L. ; FAIRES, J. D.: *Numerical Analysis*. Brooks/Cole, 2001. – ISBN 0–534–38216–9. – 7. Auflage
- [BFH⁺09] BOLZ, Phillip ; FISCHER, Stephan ; HÖNIG, Maximilian ; SCHENK, Sebastian ; SCHULZ, Daniel ; URBANNEK, Maik: *Vergleich und Bewertung von 3D-Toolkits hinsichtlich ihrer Eignung für VR-Inszenierungen*, Martin-Luther-Universität Halle-Wittenberg, Projektarbeit, 2009
- [Bli12a] BLIZZARD ENTERTAINMENT: *Diablo II*. Computerspiel. <http://eu.blizzard.com/de-de/games/d2/>. Version: 2012, Abruf: 18. Juni 2012. – Erstveröffentlichung: 2000
- [Bli12b] BLIZZARD ENTERTAINMENT: *Diablo III*. Computerspiel. <http://eu.blizzard.com/de-de/games/d3/>. Version: 2012, Abruf: 18. Juni 2012. – Erstveröffentlichung: 2012
- [Bli12c] BLIZZARD ENTERTAINMENT: *World of Warcraft*. Computerspiel. <http://eu.blizzard.com/de-de/games/wow/>. Version: 2012, Abruf: 18. Juni 2012. – Erstveröffentlichung: 2004
- [Bre65] BRESENHAM, J.E.: Algorithm for computer control of a digital plotter. In: *IBM Systems Journal* 4 (1965), Nr. 1, S. 25–30
- [CLRS10] CORMEN, Th. H. ; LEISERSON, Ch. E. ; RIVEST, R. ; STEIN, C.: *Algorithmen - Eine Einführung*. Oldenbourg Verlag, 2010. – ISBN 978–3–486–59002–9. – 3. Auflage
- [Duc11] DUCKECK, Jochen: *Klassifikation von Höhlen*. Website. <http://www.showcaves.com/german/explain/Speleology/Classification.html>. Version: Dezember 2011, Abruf: 18. Juni 2012
- [ESK96] ENCARNAÇÃO, José ; STRASSER, Wolfgang ; KLEIN, Reinhard: *Graphische Datenverarbeitung 1*. R. Oldenbourg Verlag, 1996. – ISBN 3–486–23223–1. – 4. Auflage
- [ESK97] ENCARNAÇÃO, José ; STRASSER, Wolfgang ; KLEIN, Reinhard: *Graphische Datenverarbeitung 2*. R. Oldenbourg Verlag, 1997. – ISBN 3–486–23469–2. – 4. Auflage
- [Far10] FARBRAUSCH: *Farbrausch*. Website. <http://www.farb-rausch.com/>. Version: 2010, Abruf: 18. Juni 2012

Literaturverzeichnis

- [FvFH97] FOLEY, James D. ; VAN DAM, Andries ; FEINER, Steven K. ; HUGHES, John F.: *Computer Graphics: Principles and Practice*. Addison-Wesley, 1997. – ISBN 0-201-84840-6
- [GBD04] GREENWOOD, Ed ; BOYD, Eric L. ; DRADER, Darrin: *Dungeons & Dragons Campaign Supplement - Forgotten Realms: Serpent Kingdoms*. Wizards of the Coast Inc., 2004. – ISBN 0-7869-3277-5
- [GH97] GARLAND, Michael ; HECKBERT, Paul S.: Surface simplification using quadric error metrics. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* ACM Press/Addison-Wesley Publishing Co., 1997, S. 209–216
- [Gie04] GIESEN, Fabian: *.kkrieger Postmortem*. Website. http://web.archive.org/web/20050216145754/http://game-face.de/article.php3?id_article=132. Version: Oktober 2004, Abruf: 18. Juni 2012
- [Hak09] HAKEL, Benedikt: *Gletscherhöhle*. Website. <http://www.fotocommunity.de/pc/pc/display/19475695>. Version: Dezember 2009, Abruf: 18. Juni 2012
- [Hen02a] HENNINGSEN, Dr. P.: *How We Envision Using the DungeonMaker*. Website. http://dungeonnmaker.sourceforge.net/DM2_Manual/manVision.html. Version: 2002, Abruf: 18. Juni 2012
- [Hen02b] HENNINGSEN, Dr. P.: *Manual for the DungeonMaker ver 2.0*. Website. http://dungeonnmaker.sourceforge.net/DM2_Manual/index.html. Version: 2002, Abruf: 18. Juni 2012
- [Hön09] HÖNIG, Maximilian: *Umsetzung eines 3D-Action-Adventures mit der Open Source Engine Irrlicht*, Martin-Luther-Universität Halle-Wittenberg, Bachelorarbeit, 2009
- [Hug10] HUGHES, Mark D.: *Roguelike Dungeon Generation*. Website. http://kuoi.com/~kamikaze/GameDesign/art07_rogue_dungeon.php. Version: 2010, Abruf: 18. Juni 2012
- [Irr10] IRRLICHT TEAM: *Irrlicht Engine Class Reference - IMeshManipulator*. Website. http://irrlicht.sourceforge.net/docu/classirr_1_1scene_1_1_i_mesh_manipulator.html. Version: Januar 2010, Abruf: 18. Juni 2012
- [Irr12] IRRLICHT TEAM: *Irrlicht Engine Homepage*. Website. <http://irrlicht.sourceforge.net/>. Version: 2012, Abruf: 18. Juni 2012
- [Jab11] JABLONSKI, Adrian J.: *Fraktale Geometrie*. Website. <http://quadsoft.org/fraktale/>. Version: Juli 2011, Abruf: 18. Juni 2012
- [Jan07] JANUSCH, Sven: *Konzeption und Realisierung eines prozeduralen Ansatzes zur Erzeugung von Gebäuden*, Hochschule Darmstadt, Masterarbeit, 2007
- [Kle05] KLEIN, Rolf: *Algorithmische Geometrie*. Springer-Verlag, 2005. – ISBN 3-540-20956-5. – 2. Auflage

Literaturverzeichnis

- [Knu98] KNUTH, Donald E.: *The Art of Computer Programming - Volume 2 / Semi-numerical Algorithms.* Addison-Wesley, 1998. – ISBN 0-201-89684-2. – 3. Auflage
- [Lar03] LAROSE, Dana: *Using A Cellular Automata Style Rule To Create A Cave System.* Website. http://pixelenvy.ca/wa/ca_cave.html. Version: Februar 2003, Abruf: 18. Juni 2012
- [LC87] LORENSEN, William E. ; CLINE, Harvey E.: Marching cubes: A high resolution 3D surface construction algorithm. In: *ACM Siggraph Computer Graphics* 21 (1987), Nr. 4, S. 163–169
- [LC02] LIU, X.-W. ; CHENG, K.: Three-dimensional extension of Bresenham's algorithm and its application in straight-line interpolation. In: *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture* Bd. 216, Sage Publications, 2002, S. 459–463
- [Len04] LENGYEL, Eric: *Mathematics for 3D Game Programming and Computer Graphics.* Charles River Media, 2004. – ISBN 978-1-58450-277-7, 1-58450-277-0. – 2. Auflage
- [Log10] LOGAN: *Fantastic Worlds - Fantastische Welten: Megadungeons Teil 1.* Website. <http://cimorra.blogspot.de/2010/12/philosophie-megadungeons.html>. Version: Dezember 2010, Abruf: 18. Juni 2012
- [LRC⁺03] LUEBKE, David ; REDDY, Martin ; COHEN, Jonathan D. ; VARSHNEY, Amitabh ; WATSON, Benjamin ; HUEBNER, Robert: *Level of Detail for 3D Graphics.* Morgan Kaufmann Publishers, 2003. – ISBN 978-1-55860-838-2, 1-55860-838-9
- [Man91] MANDELBROT, Benoît B.: *Die fraktale Geometrie der Natur.* Birkhäuser Verlag, 1991. – ISBN 3-7643-2646-8
- [Moj12] MOJANG: *Minecraft.* Computerspiel. <http://www.minecraft.net/>. Version: 2012, Abruf: 18. Juni 2012. – Erstveröffentlichung: 2009
- [Mos06] MOSER, Dionys: *My Cave.* Website. <https://naturfotografen-forum.de/o43935-My+Cave>. Version: September 2006, Abruf: 18. Juni 2012
- [Neu09] NEUMANN, Dirk: *Höhlenbildung.* Website. http://www.planet-wissen.de/natur_technik/hoehlen/hoehlenforschung/wie_hoehlen_entstehen.jsp. Version: Juni 2009, Abruf: 18. Juni 2012
- [OMG11a] OMG OBJEKT MANAGEMENT GROUP: *OMG Unfified Modeling Language Infrastructure.* Spezifikation. <http://www.omg.org/spec/UML/2.4.1/Infrastructure>. Version: August 2011, Abruf: 18. Juni 2012
- [OMG11b] OMG OBJEKT MANAGEMENT GROUP: *OMG Unfified Modeling Language Superstructure.* Spezifikation. <http://www.omg.org/spec/UML/2.4.1/Superstructure>. Version: August 2011, Abruf: 18. Juni 2012
- [O'R94] O'ROURKE, Joseph: *Computational Geometry in C.* Cambridge University Press, 1994. – ISBN 0-521-44592-2

Literaturverzeichnis

- [Per05] PERKINS, Christopher: *Dungeons & Dragons Campaign Accessory - Forgotten Realms: Sons of Gruumsh*. Wizards of the Coast Inc., 2005. – ISBN 978-0-7869-3698-4, 0-7869-3698-3
- [Per11] PERSSON, Markus: *The Word of Notch: Terrain generation, Part 1*. Website. <http://notch.tumblr.com/post/3746989361/terrain-generation-part-1>. Version: März 2011, Abruf: 18. Juni 2012
- [PL90] PRUSINKIEWICZ, Przemyslaw ; LINDENMAYER, Aristid: *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990. – ISBN 3-540-97297-8, 0-387-97297-8
- [RB92] ROSSIGNAC, Jarek R. ; BORRELL, Paul: *Multi-resolution 3D approximations for rendering complex scenes*. IBM Research Division, TJ Watson Research Center, 1992
- [RS92] ROZENBERG, Grzegorz ; SALOMAA, Arto: *Lindenmayer Systems*. Springer-Verlag, 1992. – ISBN 3-540-55320-7, 0-387-55320-7
- [Sal06] SALOMON, David: *Curves and Surfaces for Computer Graphics*. Springer-Verlag New York Inc, 2006. – ISBN 978-0-387-24196-8, 0-387-24196-5, 0-387-28452-4
- [Slo91] SLOAN, Kenneth: *Generating Surface Normals*. Website. <http://steve.hollasch.net/cgindex/geometry/surfnorm.html>. Version: 1991, Abruf: 18. Juni 2012
- [SS06] SAAKE, Gunter ; SATTLER, Kai-Uwe: *Algorithmen und Datenstrukturen*. dpunkt.verlag, 2006. – ISBN 3-89864-385-9. – 3. Auflage
- [Ste11] STEININGER, Johann: *Dachstein Höhlenwelt: Mammuthöhle Paläotraun*. Website. http://www.foto360.at/vr_panorama_fotos_dyn/fullscreen/vollbild_qtvr_panorama_photo.php?id=616. Version: Mai 2011, Abruf: 18. Juni 2012
- [SZL92] SCHROEDER, William J. ; ZARGE, Jonathan A. ; LORENSEN, William E.: Decimation of triangle meshes. In: *ACM Siggraph Computer Graphics* 26 (1992), Nr. 2, S. 65–70
- [Tri68] TRIMMEL, Hubert: *Höhlenkunde*. Friedr. Vieweg & Sohn GmbH, 1968
- [Uni07] UNIVERSITY OF WISCONSIN: *Crystal Cave*. Website. <http://www.uwec.edu/jolhm/Cave2007/TeamC/TeamC/Home.html>. Version: 2007, Abruf: 18. Juni 2012
- [Uni11] UNIVERSITÄT BAYREUTH: *Hexagonal kolam*. Website. http://jsxgraph.uni-bayreuth.de/wiki/index.php?title=Hexagonal_kolam&oldid=5201. Version: Juni 2011, Abruf: 18. Juni 2012
- [Wik10] WIKIMEDIA COMMONS: *Bresenham-Algorithmus: rasterisierte Linie, im Vergleich zum Verlauf der Fehlervariablen*. Website. <http://de.wikipedia.org/w/index.php?title=Datei:BresenhamLine2.png&filetimestamp=20101017164957>. Version: Oktober 2010, Abruf: 18. Juni 2012

Literaturverzeichnis

- [Wik12] WIKIPEDIA: *Roguelike*. Website. <http://en.wikipedia.org/wiki/Roguelike>. Version: Mai 2012, Abruf: 18. Juni 2012
- [Wil04] WILLIAMS, Skip: *Dungeons & Dragons Web Enhancement - Races of Stone: Raiders of the High Citadel*. Wizards of the Coast Inc., 2004 <http://www.wizards.com/default.asp?x=dnd/we/20040813a>. – Abruf: 24. Juni 2012
- [ZTS05] ZHU, J. ; TANAKA, T. ; SAITO, Y.: *A New Mesh Simplification Algorithm for the Application of Surface Sculpture*, Tokyo Institute of Technology, Forschungsbericht, 2005