

Triangulation and Simplification of Incomplete Height Maps with Restricted Quad Trees

Masterarbeit im Fach Informatik

vorgelegt von

Simon Mederer

geb. am 11. Mai 1996 in Nürnberg

angefertigt am

**Department Informatik
Lehrstuhl für Graphische Datenverarbeitung
Friedrich-Alexander-Universität Erlangen-Nürnberg**

Betreuer: M.Sc. Darius Rückert

Betreuer Hochschullehrer: Prof. Dr. Marc Stammerer

Beginn der Arbeit: 13. Juni 2019

Abgabe der Arbeit: 13. Dezember 2019

Kurzfassung

Diese Arbeit beschäftigt sich mit der Triangulierung einer potentiell lückenhaften Tiefenkarte. Hierfür werden die Tiefendaten zuerst vorverarbeitet, um Kanten zwischen Objekten zu erkennen und zu entfernen, die nicht miteinander verbunden sind. Im zweiten Schritt werden Restricted Quadtrees verwendet, um aus der bearbeiteten Tiefenkarte ein Mesh zu erzeugen, welches im Vergleich zur naiven Triangulierung Primitiven an detailarmen Stellen einspart. Zuletzt wird das erstellte Mesh reduziert. Dieser Schritt dient dem Entfernen derjenigen Vertices, die durch die Erstellungsregeln des Quadtree Teil der Triangulierung sind, jedoch keine relevanten Informationen im Mesh darstellen. Als zusätzlicher Teil der Dezimierung werden diverse Erweiterungen implementiert, durch welche die Vollständigkeit der Geometrie verbessert und die Fehlererzeugung insgesamt geringer gehalten wird.

Als Ergebnis zeigt sich, dass die Quadtree-basierte Triangulierung je nach gewähltem Threshold eine große Menge an Vertices einsparen und dabei einen geringen Fehler erzeugen kann. Werden genug Vertices eingespart, kann diese Art der Triangulierung trotz ihrer zusätzlichen Überprüfungen sogar schneller als der naive Ansatz sein.

Die vorgestellte Dezimierung erzeugt im Vergleich zu einer regulären Dezimierung mit gleicher grundlegender Errormetrik einen geringeren Fehler und sorgt für eine bessere Vollständigkeit der Geometrie. Dies liegt daran, dass sie diverse zusätzliche Erweiterungen beachtet. Aus diesem Grund ist ihre Laufzeit im Vergleich jedoch wesentlich erhöht. Wird sie allerdings auf einem durch Quadtrees erstellten Mesh ausgeführt, verbessert die geringere Menge an Vertices die Ausführungszeit stark.

Insgesamt ergibt also das Ausführen der Triangulierung via Quadtrees und die vorgestellte Dezimierung in Kombination ein sehr gutes Ergebnis, da somit sowohl eine gute Laufzeit als auch eine geringe Fehlererzeugung erreicht wird.

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Überblick	3
1.3 Beitrag dieser Arbeit	4
2 Verwandte Arbeiten	5
2.1 Triangulierung via Quadtrees	5
2.2 Mesh Simplifizierung	8
3 Vorverarbeitung	11
3.1 Occlusion Edges	11
3.1.1 Definition	11
3.1.2 Aspect Ratio als Grundlage	12
3.1.3 Vermeidung von Over-Detection	13
3.1.4 Erhöhte Erkennung in der Entfernung	14
3.1.5 Erkennung der Kanten	14
3.2 Hysteresis Threshold zur Erkennung von Randfällen	15
3.3 Glättung des Bildes	17
4 Triangulierung	19
4.1 Naive Triangulierung	19
4.2 Quadtree	21
4.2.1 Allgemeine Struktur	21
4.2.2 Einschränkung zu Restricted Quadtree	22
4.2.3 Errormetrik	25
4.2.4 Abhängigkeitsgraph	27
5 Reduzierung des Meshes	29
5.1 Quadratische Errormetrik	29
5.2 Ablauf der Dezimierung	31
5.3 Erweiterungen	33
5.3.1 Tiefenabhängiger Fehler	33
5.3.2 Verhindern der Invertierung von Dreiecken	33

INHALTSVERZEICHNIS

5.3.3	Einschränkung der Kontraktionspartner von Rand-Vertices	34
5.3.4	Dezimierung von nur parallelen Rand-Kanten	34
5.3.5	Kantenerhaltung im Mesh	35
5.3.6	Minimaler Innenwinkel der Dreiecke	36
5.3.7	Optimale Vertex-Position nach der Kontraktion einer Kante	36
6	Ergebnisse	39
6.1	Verglichene Meshes	39
6.2	Unvollständige Dezimierungen der RQTs	41
6.3	Vergleich der Ausführungszeit	42
6.3.1	Konstante Verfahren	42
6.3.2	Variable Verfahren	42
6.4	Vollständigkeit	45
6.4.1	Definition der Metrik	45
6.4.2	Vollständigkeit der Verfahren	45
6.5	Hausdorff Distanz	47
6.5.1	Definition der Metrik	47
6.5.2	Entstehende Hausdorff Distanz durch die Verfahren	49
7	Zusammenfassung und Ausblick	51
	Literaturverzeichnis	53

Abbildungsverzeichnis

2.1	Die ursprüngliche Triangulierung der Restricted Quadtrees	5
2.2	Eine verbesserte Triangulierung der Restricted Quadtrees	6
2.3	Das Bottom-up Verfahren zur Triangulierung der Quadtrees	6
2.4	Vereinigung von Dreiecken	7
2.5	Berechnung der von der Sicht abhängigen Metrik	8
2.6	Der Toleranzbereich durch die Hausdorff Distanz	9
2.7	Berechnung der spezialisierten Metrik am Rand des Meshes	10
3.1	Die Eingabebilder nach der Skalierung	12
3.2	Naive Triangulierung ohne Entfernung der Occlusion Edges	12
3.3	Aspect Ratio Berechnung	13
3.4	Erkannte Occlusion Edges bei simplem Threshold	15
3.5	Darstellung der Occlusion Edge Berechnung	15
3.6	Erkannte Occlusion Edges mit Hysteresis Threshold	16
3.7	Gauß-Filter Beispiel	17
4.1	Exemplarische Darstellung der naiven Triangulierung der Tiefendaten	19
4.2	Exemplarische Darstellung eines Quadtrees	21
4.3	Auswahl von Vertices bei einer Triangulierung via Quadtrees	22
4.4	Simple Triangulierungsarten eines Quadtrees	23
4.5	Erweiterung eines Quadtrees zu einem RQT	23
4.6	Verwendete Vertices zur Triangulierung eines Restricted Quadtrees	24
4.7	RQT Triangulierungsregel 2	24
4.8	RQT Triangulierungsregel 3	25
4.9	Komplette Triangulierung eines RQT	25
4.10	RQT Unterteilungsregeln	26
4.11	Abhängigkeitsgraph Regel 1	28
4.12	Abhängigkeitsgraph Regel 2	28
5.1	Eine simple Dezimierungsoperation	31
5.2	Invertierung eines Dreiecks bei der Dezimierung	33
5.3	Abbildung des Kollabierungsverbotes von Rand-Vertices nach innen	34
5.4	Abbildung des Kollabierungsverbotes von nicht-parallelen Rand-Kanten	35

ABBILDUNGSVERZEICHNIS

5.5	Änderungen durch Verhindern der Kontraktion von Rand-Vertices um Ecken	35
5.6	Änderungen durch Kantenerhaltung im Mesh	36
5.7	Änderungen durch das Erhalten eines minimalen Innenwinkels der Dreiecke	37
6.1	Beispiele für den Ablauf der Pipeline	39
6.2	Nicht ausreichende Dezimierungen nach RQT-Triangulierungen	41
6.3	Ergebnisse für die Zeit	43
6.4	Ergebnisse für h_n	46
6.5	Ergebnisse für h_c	46
6.6	Veranschaulichung der Hausdorff Distanz	48
6.7	Ergebnisse für die Hausdorff Distanz	49

Kapitel 1

Einleitung

1.1 Motivation

In der Computergraphik existieren schon seit längerem sogenannte "simultaneous localization and mapping"-Algorithmen (SLAM). Solche Anwendungen erstellen aus einer Folge von verschiedenen Kameraaufnahmen der gleichen Szene ein einzelnes, zusammengefügtes 3D-Modell. Moderne Methoden können zudem in Echtzeit angewandt werden, um eine solche 3D-Rekonstruktion einer Szene zu erstellen [16, 9]. Die meisten dieser Algorithmen stellen die Umgebung jedoch nur in Form einer Punktewolke dar und es wird keine tatsächliche Geometrie erzeugt. Dies limitiert die Möglichkeiten, für die die erstellte virtuelle Szene im weiteren Verlauf genutzt werden kann. Darglein et. al. [13] stellen hingegen FragmenFusion vor, welches eine SLAM-Pipeline ist, die in Echtzeit vollständige Dreiecksnetze von der betrachteten Szene erzeugt. Diese Masterarbeit beschäftigt sich damit, die Triangulierung der dort vorgestellten Pipeline um mehrere Schritte zu erweitern. Aus den aufgenommenen Tiefendaten einer Szene soll demnach schnell und effizient ein Mesh generiert werden, welches die Umgebung möglichst fehlerfrei darstellt und gleichzeitig nur eine geringe Zahl an Primitiven besitzt.

Für das Triangulieren solcher 2,5-dimensionalen Tiefenkarten werden häufig Quadtrees verwendet. Sie sind bereits lange als Mittel bekannt, um Terraindaten zu triangulieren [18, 11]. Des Weiteren wurden sie in der Vergangenheit bereits verwendet, um planare Flächen in Meshes umzuwandeln [7, 3]. Diese Arbeit kombiniert nun solche Ansätze und trianguliert beliebige Szenen mithilfe von Quadtrees, die von einem Tiefensensor aufgenommen werden. Entsprechend sind die zu triangulierenden Bilder zwar nicht planar, dennoch gibt es pro Pixel nur eine Messung. Zudem sind die Aufnahmen nicht gleich den bekannten Terraindaten, da an vielen Stellen fehlerhafte Messungen entstehen können, mit denen umgegangen werden muss.

Des Weiteren beschäftigt sich die Arbeit damit, das entstandene Mesh weiter zu minimieren, um alle überflüssigen Vertices zu entfernen, welche durch die Erstellungsregeln des Quadtrees entstehen. Somit ist das Gesamtergebnis ein reduziertes Mesh, welches die aufgenommene Szene darstellt. Eine solche Anwendung, die mithilfe eines Tiefenbildes eine Szene in ein Mesh umwandeln kann, ist zudem in vielen verschiedenen Fällen hilfreich: Durch den Zusatz der Dezimierung kann diese Arbeit eine relevante Erweiterung für die Kartographie und das Verarbeiten sehr großer Terraindaten sein. Zudem könnte beispielsweise auch die autonome Robotik davon profitieren, indem ein Roboter mit

KAPITEL 1. EINLEITUNG

Hilfe mehrerer Aufnahmen ein Modell seiner normalen Arbeitsumgebung abspeichert und weitere Berechnungen für Wegfindung oder andere Problemlösungen anstellt.

Der zusätzliche Fokus dieser Arbeit ist die Geschwindigkeit. Damit ist einerseits die Ausführungs geschwindigkeit der vorgestellten Methoden gemeint. Andererseits soll auch die Geschwindigkeit von hierauf aufbauenden Arbeiten verbessert werden, indem die Zahl der Primitiven verringert wird. Somit wird die Arbeit auch für Anwendungen mit Zeitdruck relevant. So könnte beispielsweise diese Arbeit innerhalb von FragmentFusion [13] verwendet werden, um in Augmented Reality Applikationen de tailliertere Meshes der Umgebung zu erzeugen und besser mit der betrachteten Szene zu interagieren. Ein Beispiel dafür wäre das passende Platzieren von virtuellen Objekten im Raum.

1.2. ÜBERBLICK

1.2 Überblick

In dieser Arbeit werden die Daten eines Tiefensensors ausgelesen und zu einem Mesh verarbeitet. Wichtig sind dabei Geschwindigkeit, eine möglichst fehlerfreie Darstellung der aufgenommenen Szene und das Vermeiden von unnötigen Primitiven im Mesh. Das Vorgehen dieser Arbeit ist in drei große Schritte unterteilt:

Kapitel 3 dreht sich um eine generelle Vorverarbeitung der eingehenden Tiefendaten als erster Schritt. Hierbei werden vor allem sogenannte Occlusion Edges entfernt. Diese sind direkte Verbindungen zwischen unabhängigen Objekten, welche bei einer Triangulierung der Daten ohne Vorverarbeitung für eine falsche Darstellung der aufgenommenen Szene sorgen würden. Im Folgenden wird das Bild gefiltert, um das in Tiefendaten stets vorhandene Rauschen der Messwerte abzuschwächen.

Der zweite Schritt wird in Kapitel 4 behandelt und ist das Triangulieren der bearbeiteten Daten. Anstelle einer naiven Triangulierung wird dabei ein Ansatz verwendet, welcher Quadtrees zu Hilfe nimmt. Hierbei wird das Eingabebild mit der aktuellen Triangulierung des Quadtree verglichen. Ist der Unterschied zu groß, also der Detailgrad des Bildes um einen gewissen Grad höher als die aktuelle Abdeckung des Quadtree, teilt sich der Quadtree an entsprechender Stelle, um dort mehr Details abzudecken. Um Quadtree vernünftig triangulieren zu können, wird zudem eine Erweiterung auf Restricted Quadtrees verwendet, welche auf einen fertigen Quadtree angewandt werden kann. Sobald dies getan ist, kann die gesamte Struktur mithilfe weniger Regeln komplett trianguliert werden.

Kapitel 5 behandelt den dritten und letzten Schritt, welcher eine Dezimierung des erstellten Meshes umfasst. Dies ist relevant, da der Aufbau eines Quadtree an diverse Regeln gebunden ist und es somit trotz der vermindernten Zahl von Vertices an detailarmen Stellen noch großes Potential zur Verringerung der Primitiven unter geringer Fehlerzeugung gibt. Zuerst wird der grundlegende Vorgang und die verwendete Errormetrik vorgestellt, daraufhin werden diverse Erweiterungen eingeführt, welche in Form von Einschränkungen bei der Dezimierung für einen geringeren Fehler und erhöhte Completeness des Ergebnismeshes sorgen.

In Kapitel 6 werden alle Vorgehensweisen miteinander verglichen und auf ihren Fehlergrad geprüft. Durch die Triangulierung via Quadtree wird zwar ein gewisser Fehler erzeugt, allerdings kann die Zahl der nötigen Vertices im Mesh im Vergleich zu einer naiven Triangulierung stark verringert werden. Für die Dezimierung wird der Ansatz dieser Arbeit mit einer bereits existenten Implementierung von OpenMesh [1] verglichen. Dies ist eine Open Source Bibliothek der Rheinisch-Westfälischen Technischen Hochschule Aachen, welche eine generische und effiziente Datenstruktur zum Darstellen und Manipulieren von Polygonnetzen implementiert. Es stellt sich heraus, dass OpenMesh zwar schneller in der Ausführung ist, jedoch durch das Fehlen einiger Erweiterungen aus Abschnitt 5.3 einen größeren Fehler in dem dezimierten Mesh verursacht. Insgesamt hängt die beste Kombination der Implementierungen davon ab, ob hohe Geschwindigkeit oder ein geringer Fehler wichtiger sind.

1.3 Beitrag dieser Arbeit

Die Beiträge dieser Arbeit sind folgende:

1. Es werden diverse bereits existente Vorgehensweisen miteinander verbunden und teilweise erweitert.
2. Die entstehenden Implementierungen werden in verschiedenen Kombinationen ausgeführt und evaluiert.
3. Die Verfahren werden zu einer Pipeline zusammengesetzt, welche Tiefenkarten als Eingabe erhält und Meshes mit einer niedrigen Zahl an Primitiven erzeugt.
4. Diese Pipeline wird als Open Source Code bereitgestellt. Dies geschieht, indem sie in das Saiga Framework [12] integriert wird.

Kapitel 2

Verwandte Arbeiten

Innerhalb der Masterarbeit werden vor allem zwei große Konzepte behandelt. Diese sind einerseits das Triangulieren einer 2,5-dimensionalen Tiefenkarte mithilfe von Quadtrees und andererseits das Reduzieren der Primitiven eines derart erstellten Meshes. Zu beiden Vorgehen gibt es bereits diverse Referenzwerke, aus welchen in den folgenden Abschnitten ausgewählte Beispiele vorgestellt werden.

2.1 Triangulierung via Quadtrees

Quadtrees werden bereits an diversen Stellen zum Triangulieren von Umgebungen genutzt. Sie werden häufig angewandt, wenn Höhenkarten, wie beispielsweise in der Kartographie, primitivensparend in Meshes mit unterschiedlichem level-of-detail umgewandelt werden sollen. Somit bestehen bereits einige Werke, welche sich mit ihnen auseinandersetzen:

Vorgestellt wurde der hierarchische Ansatz zur Triangulierung unter Verwendung von Restricted Quadtrees von Von Herzen et. al. [17]. Dabei entspricht ein Restricted Quadtree einem regulären Quadtree, bei welchem der Levelunterschied benachbarter Quads nicht größer als 1 sein darf. Diese Einschränkung sorgt dafür, dass der gesamte Quadtree ohne Probleme trianguliert werden kann. Die Art der Triangulierung ist in Abbildung 2.1 dargestellt: Ein Quad wird in acht Dreiecke geteilt, von welchen jeweils zwei eine Kante abdecken. Einzig wenn die entsprechende Kante an einem größeren Quad anliegt, werden die beiden Dreiecke zu einem vereint. Auf diese Weise kann der komplette Quadtree ohne Risse im Mesh trianguliert werden.

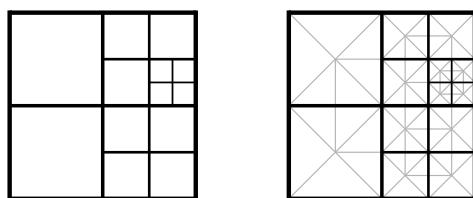


Abbildung 2.1: Die Triangulierung von Restricted Quadtrees nach Von Herzen et. al. [17].

Von Herzen et. al. geben selbst keinen ausführlichen Algorithmus an, allerdings bauen andere Veröffentlichungen weiter darauf auf: Sivan et. al. [15] verbessern den Algorithmus zur Triangulierung, indem sie die Dreiecke einer Kante im Quad ebenfalls vereinen, wenn das benachbarte

Quad die gleiche Größe besitzt. Dies ist in Abbildung 2.2 dargestellt und sorgt für eine geringere Zahl an Dreiecken, die zur Darstellung des Meshes nötig sind. Zudem stellen sie zwei effiziente Algorithmen zur Erstellung und Triangulierung eines Restricted Quadtrees vor – eine Variante arbeitet Top-down, die andere Bottom-up.

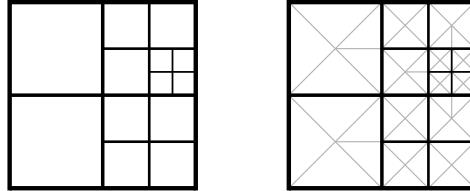


Abbildung 2.2: Die Triangulierung von Restricted Quadtrees nach Sivan et. al. [15].

Der Bottom-up-Ansatz funktioniert derart, dass zu Beginn das gesamte Eingabebild bis auf die tiefste Ebene im Quadtree unterteilt wird. Die entsprechende Größe der Quads entspricht 3×3 Pixeln. Daraufhin werden alle Quads miteinander vereint, die bestimmte Bedingungen erfüllen: Sie müssen den gleichen Elternknoten im Quadtree und selbst keine Kindknoten besitzen. Außerdem darf keiner dieser vier Geschwisterknoten einen Nachbarn haben, der auf ein tieferes Level unterteilt ist. Zudem muss der entstehende Fehler beim Vereinen der Quads nach einer gewählten Metrik unterhalb eines ebenfalls gewählten Thresholds liegen. Die verwendete Errormetrik von Sivan et. al. entspricht folgender: Soll ein Vertex durch das Vereinen von Quads nicht mehr für die Triangulierung verwendet werden, so muss sein vertikaler (und nicht kürzester) Abstand zum abgewandelten Mesh unterhalb eines Thresholds liegen. Abbildung 2.3 stellt dar, wie das Vereinen von vier Quads aussehen kann. Dabei entsprechend die grauen Vertices denen, die nach der Vereinigungsoperation nicht mehr für die Triangulierung verwendet werden. Der Fehler für Vertex B wird beispielsweise daran berechnet, wie weit sein Tiefenwert vom durchschnittlichen Tiefenwert der Vertices A und C entfernt ist. Das Problem dieser Fehlerberechnung besteht darin, dass sich der Fehler über mehrere Vereinigungsoperationen immer weiter abwandeln kann, da für jede Überprüfung nur die Vertices des Quads auf aktuellem Level, nicht jedoch die der vorherigen Level, verwendet werden.

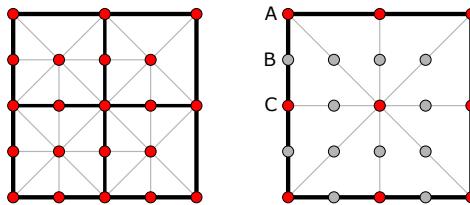


Abbildung 2.3: Das Bottom-up Verfahren von Sivan et. al. [15].

Die Top-down Alternative beginnt mit einem einzelnen Quad, das das gesamte Bild abdeckt. Ähnlich zum vorherigen Ansatz werden alle Vertices auf Level l ermittelt, die durch ein Unterteilen eines Quads auf Level $l - 1$ ebenfalls Teil der Triangulierung werden würden. Jeder dieser Vertices des Levels l wird zu diesem Zeitpunkt durch eine oder mehr Kanten zwischen Vertices des Levels $l - 1$ dargestellt. Überschreitet die vertikale Distanz zwischen der Kante und dem Vertex einen gewissen Threshold, wird das Quad in vier Teile geteilt. Auf diese Weise können zwar Details im Mesh

2.1. TRIANGULIERUNG VIA QUADTREES

übersehen werden, allerdings pflanzt sich durch das Top-down Vorgehen auch keine Fehlerentwicklung fort. Um einen Restricted Quadtree zu erhalten wird außerdem nach jeder Unterteilung eines Quads sichergestellt, dass seine Nachbarn nicht zu weit im Level entfernt sind. Ansonsten werden diese ebenfalls geteilt.

Lindstrom et. al. [8] gehen ebenfalls Bottom-up bei der Erstellung und Triangulierung der QuadTrees vor. Anstatt ganze Quads direkt zu vereinen, werden jedoch Paare von Dreiecken innerhalb eines Quads in zwei Schritten vereint. Zu Beginn ist das abzubildende Gitter genau wie bei Sivan et. al. [15] komplett aufgeteilt. Abbildung 2.4-b zeigt, wie der erste Schritt der Vereinigung aussieht: Es wird ein Paar an Dreiecken betrachtet, das eine kurze Seite miteinander teilt und bei dem beide Dreiecke an der gleichen Kante des Quads anliegen. Werden beispielsweise a_l und a_r aus der Abbildung vereint, wird der rot markierte Vertex in der Mitte der unteren Kante entfernt. Im zweiten Schritt aus Abbildung 2.4-c werden auf ähnliche Weise die benachbarten Dreiecke entlang zweier Kanten des Quads betrachtet, zum Beispiel e_l und e_r . In diesem Fall wird der Vertex in der Mitte des Quads bei einer Vereinigungsoperation entfernt. Damit im Mesh keine Risse entstehen, muss jedoch immer darauf geachtet werden, dass das nächste Dreiecks-Paar ebenfalls vereint wird, welches den gleichen Vertex enthält, der entfernt werden soll. In Abbildung 2.4-c sind beispielsweise die Paare (e_l, e_r) und (f_l, f_r) derart abhängig, in Abbildung 2.4-b haben alle Dreieckspaare Abhängigkeiten außerhalb des dargestellten Quads.

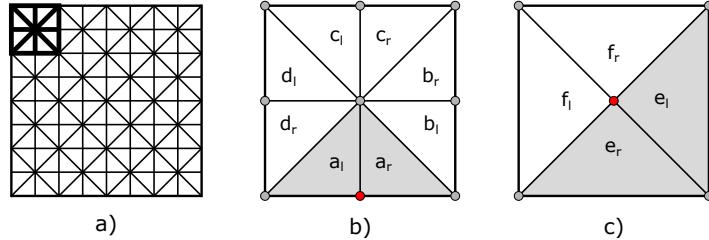


Abbildung 2.4: Die Unterteilung eines Quads nach Lindstrom et. al. [8] (a) und die beiden Schritte der Vereinigung von Dreiecken (b, c).

Lindstrom et. al. stellen zusätzlich einen Abhängigkeitsgraphen vor. Dieser zeigt für jeden Vertex an, welche weiteren Vertices nicht entfernt werden dürfen, um sie ohne Risse im Mesh darzustellen. Beispielsweise kann der rote Vertex in Abbildung 2.4-b nicht Teil der Triangulierung sein, wenn der rote Vertex in Abbildung 2.4-c nicht auch Teil dieser ist. Gleichermassen kann umgekehrt erkannt werden, dass der mittlere Knoten des Quads nicht entfernt werden kann, solange einer der Randknoten der Paare (a_l, a_r) , (b_l, b_r) , (c_l, c_r) oder (d_l, d_r) noch existieren. Diese Abhängigkeiten werden ebenfalls innerhalb der folgenden Arbeit verwendet und sind in den Abbildungen 4.11 und 4.12 für die Vertices der ersten beiden Level eines QuadTrees abgebildet.

Des Weiteren stellen Lindstrom et. al. eine Errormetrik vor, welche von der Sicht auf das Mesh abhängt. Diese ist dem Ansatz von Sivan et. al. ähnlich, allerdings wird der Fehler im Screen-space berechnet. Abbildung 2.5 zeigt die Berechnungen, ob Vertex v durch die Kante zwischen Vertex v_l und v_r dargestellt werden soll. Zuerst wird $v_m = (v_l + v_r)/2$ berechnet. Daraufhin wird $\delta_v = v - v_m$ berechnet und in den Screen-space transformiert. Liegt dessen Distanz daraufhin unterhalb eines Thresholds, kann v entfernt werden. Als Nachteil dieser Metrik ist jedoch zu nennen, dass sich auf

gleiche Weise wie in der zugrunde liegenden Metrik ein Fehler fortpflanzen und größer als der ursprünglich gewählte Threshold werden kann.

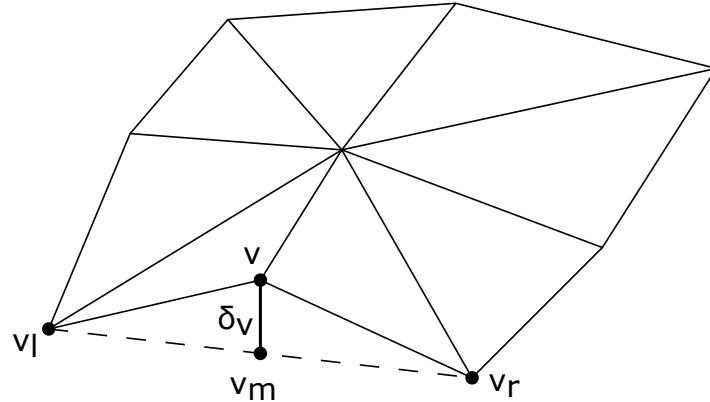


Abbildung 2.5: Die Berechnung des Fehlers nach Lindstrom et. al. [8].

2.2 Mesh Simplifizierung

Um die Anzahl der Primitiven in einem Mesh bei geringer Fehlererzeugung zu verringern gibt es verschiedene Werke, die sich jeweils in der Vorgehensweise und gewählten Metrik unterscheiden. Im Folgenden werden diverse Beispiele vorgestellt.

Eine globale Methode zur Dezimierung wird von Rossignac et. al. [14] vorgestellt. Diese führen das sogenannte Vertex Clustering ein, bei dem jedem Vertex ein Gewicht anhand der Größe anliegender Faces und der Oberflächenkrümmung in seiner Nähe zugeordnet wird. Daraufhin wird ein 3D-Gitter über das Mesh gelegt und alle Vertices eines Voxels werden auf den Vertex mit höchstem Gewicht geschoben. Der Algorithmus ist sehr schnell und kann auf jede Art von Meshes angewandt werden, allerdings hat er einen unvorhersehbaren Einfluss auf die Topologie und erzeugt bei starker Dezimierung einen hohen Fehler. Ebenso bleiben Details im Mesh nicht erhalten.

Eine andere Methode zur Dezimierung ist beispielsweise bei Borouchaki et. al. [2] zu finden. Diese erstellen einen sogenannten "Hausdorff Envelope", einen beidseitigen Toleranzbereich für das Originalmesh, der mithilfe der Hausdorff Distanz berechnet wird. Während den folgenden Simplifizierungs- und Optimierungsalgorithmen wird darauf geachtet, dass dieser Toleranzbereich nicht überschritten wird, um eine geometrische Ähnlichkeit zu bewahren. Der Toleranzbereich wird in Abbildung 2.6 gezeigt: Das Originalmesh wird als Σ dargestellt, δ ist der Radius einer bestimmten Hausdorff Distanz um einen Punkt P des Meshes. Wird für jeden Punkt des Meshes diese Hausdorff Distanz betrachtet, kann der entsprechende Toleranzbereich $\Sigma(\delta)$ abgegrenzt werden. Um den Verlauf des Meshes nicht zu stark abzuändern, darf durch eine Dezimierung zudem die Normale des Vertex nur begrenzt verändert werden.

Hussain et. al. [6] wenden eine Metrik an, die den geometrischen Fehler einer Kollabierung von Kanten mit der optischen Relevanz des entfernten Vertex in Verbindung bringt. Die optische Relevanz w_v eines Vertex v berechnet sich nach Gleichung 2.2.1, dabei entsprechen Δ dem Flächeninhalt und \vec{n} der Normalen eines Dreiecks, das an v anliegt. $\|k_v\|$ ist die euklidische Norm von k_v . Daraus ergibt

2.2. MESH SIMPLIFIZIERUNG

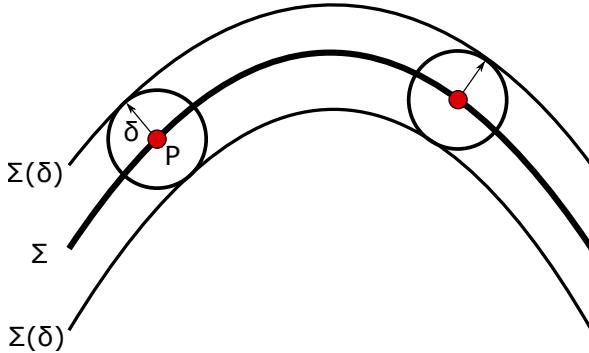


Abbildung 2.6: Die Erstellung eines Toleranzbereichs für die Dezimierung eines Meshes nach Borouchaki et. al. [2].

sich, dass w_v kleiner ist, je flacher die Umgebung von v ist.

$$w_v = 1 - \|k_v\|, \text{ wobei } k_v = \frac{\sum_i \Delta_i \vec{n}_i}{\sum_i \Delta_i} \quad (2.2.1)$$

Des Weiteren wird ein geometrischer Fehler Q_t für jedes Dreieck berechnet, welcher die Rotation und Flächenänderung des Dreiecks t vor und nach einer bestimmten Dezimierung vergleicht.

$$Q_t = l_t \cdot \theta_t, \text{ wobei } l_t = 0,5(\Delta_t + \Delta_{t'}) \quad (2.2.2)$$

t entspricht dem Dreieck vor, t' dem Dreieck nach der Dezimierung der aktuell betrachteten Kante. θ_t stellt den Winkel zwischen den Normalen von t und t' dar. Der geometrische Fehler Q_e für die Kontraktion einer Kante ist die Summe aller Q_t der davon betroffenen Dreiecke und wird in Gleichung 2.2.3 gezeigt.

$$Q_e = \sum_t Q_t \quad (2.2.3)$$

Da auf diese Weise am Rand eines Meshes leicht ein lokales Minimum entstehen würde, werden zwei Spezialfälle eingeführt: Würde ein Vertex am Rand des Meshes in das Innere des Meshes kollabiert werden, wird die Kontraktion verboten, um das Mesh weniger zu verformen. Liegen beide Vertices der betrachteten Kante am Rand des Meshes, wird ein Zusatzterm formuliert. Zur anschaulichen Erklärung wird auf Abbildung 2.7 verwiesen.

Man nehme an, die Berechnungen für die Kante e_{12} werden vollzogen. Um den entstehenden geometrischen Fehler der Kontraktion dieser Kante zu erhalten, wird $Q_{e_{12}}$ zusätzlich mit der Länge von e_{12} , dem Winkel Φ_{12} und einem konstanten Wert λ multipliziert. Das Ergebnis ist in Gleichung 2.2.4 zu sehen. Hierbei ist darauf zu verweisen, dass Φ_{12} nicht den tatsächlichen Winkel darstellt, sondern eine effizient zu berechnende Alternative ist. \hat{e} steht für die jeweilige normalisierte Kante. λ kann je nach Anwendungszweck derart gewählt werden, dass Kanten nach eigenem Wunsch verschieden stark erhalten bleiben.

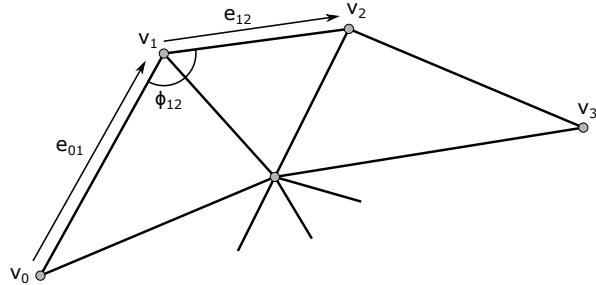


Abbildung 2.7: Die Strukturen am Rand eines Meshes, für die Hussain et. al. [6] eine spezielle Verfeinerung des geometrischen Fehlers berechnen.

$$Q_{e_{12}} = \lambda \Phi_{12} \|v_1 - v_2\| + \sum_t Q_t, \text{ wobei } \Phi_{12} = 1 - \hat{e}_{01} \cdot \hat{e}_{12} \quad (2.2.4)$$

Insgesamt wird für jeden Vertex zuerst die optisch Relevanz w_v berechnet. Daraufhin wird unter allen anliegenden Kanten jene gesucht, die den nach einer Dezimierung entstehenden geometrischen Fehler Q_e minimiert. Diese beiden Werte werden miteinander multipliziert, um einen gesamten Fehler für den Vertex und seine bestmögliche Kontraktion zu erhalten.

Dieses Verfahren sorgt dafür, dass optisch relevante Vertices und Features auch bei einer starken Dezimierung des Meshes erhalten bleiben.

Kapitel 3

Vorverarbeitung

In diesem Kapitel wird die Vorverarbeitung der Eingabedaten behandelt. Letztere sind Bilder, welche von einem Tiefensensor aufgenommen werden. Die Vorverarbeitung besteht aus dem Entfernen von Occlusion Edges und einer Glättung des Bildes. Occlusion Edges sollen entfernt werden, da sie die Kanten in dem Bild sind, welche unabhängige Objekte verbinden und somit bei deren Triangulierung für die Erstellung falscher Geometrie sorgen. Die Glättung hat den Sinn, die verrauschten Eingabedaten des Tiefensors zu vereinheitlichen und dadurch Errormetriken späterer Kapitel weniger fehleranfällig dagegen zu machen.

3.1 Occlusion Edges

3.1.1 Definition

Das größte Problem in den Tiefendaten sind die sogenannten Occlusion Edges. Diese sind Kanten, welche in der späteren Triangulierung zwischen an sich unabhängigen Objekten im Vorder- und Hintergrund auftauchen. Da diese Kanten in Wirklichkeit jedoch nicht existieren, wird durch Occlusion Edges falsche Geometrie erzeugt. Zur anschaulichen Darstellung ist in Abbildung 3.1 gezeigt, wie ein beispielhaftes Tiefenbild aussieht. Dabei entsprechen die roten Pixel in den Tiefendaten jenen, für welche der Tiefensensor keinen Abstand ermitteln konnte. An allen Stellen, bei denen die Tiefe eines Pixels im Vergleich zu Teilen der nahen Nachbarschaft stark unterschiedlich ist, soll keine Verbindung hergestellt werden. Ein Beispiel dafür ist der Übergang eines Tisches zur dahinter liegenden Wand. Wenn man die Tiefendaten allerdings in diesem Zustand direkt naiv trianguliert (wie später in Kapitel 4.1 beschrieben), erhält man ein Ergebnis, wie man es in Abbildung 3.2 sehen kann: Zwischen den Tischen und Wänden und auch an vielen anderen Orten sind einige Occlusion Edges zu sehen, welche einen falschen Eindruck der Szene vermitteln und für keine originalgetreue Darstellung dieser sorgen.

Am effektivsten kann man Occlusion Edges noch auf Bildebene erkennen und entfernen, da man dort noch nicht mit einer dreidimensionalen Mesh-Struktur arbeitet, sondern stattdessen mit pro-Pixel Vergleichen die Unterschiede der Tiefe in der Nachbarschaft eines Pixels messen kann.

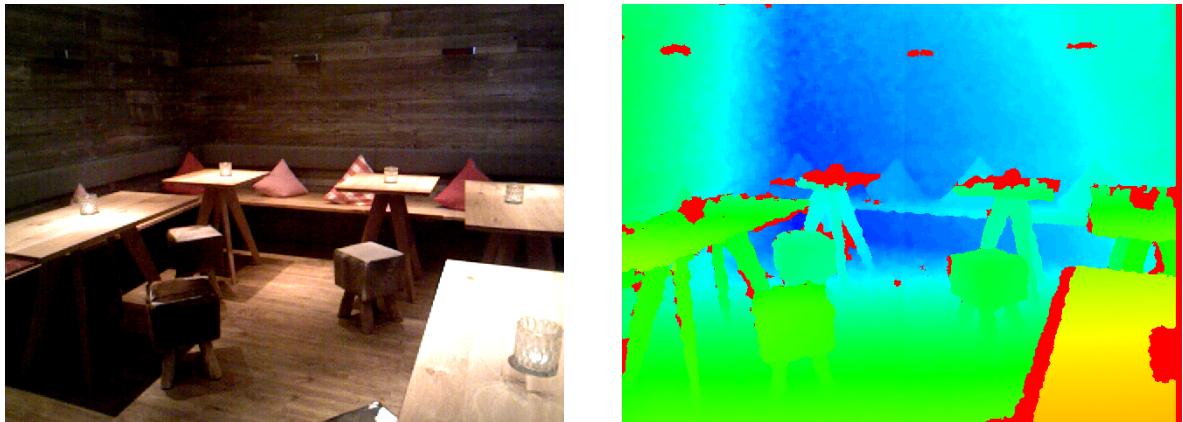


Abbildung 3.1: Das Farb- und Tiefenbild einer exemplarischen Szene.

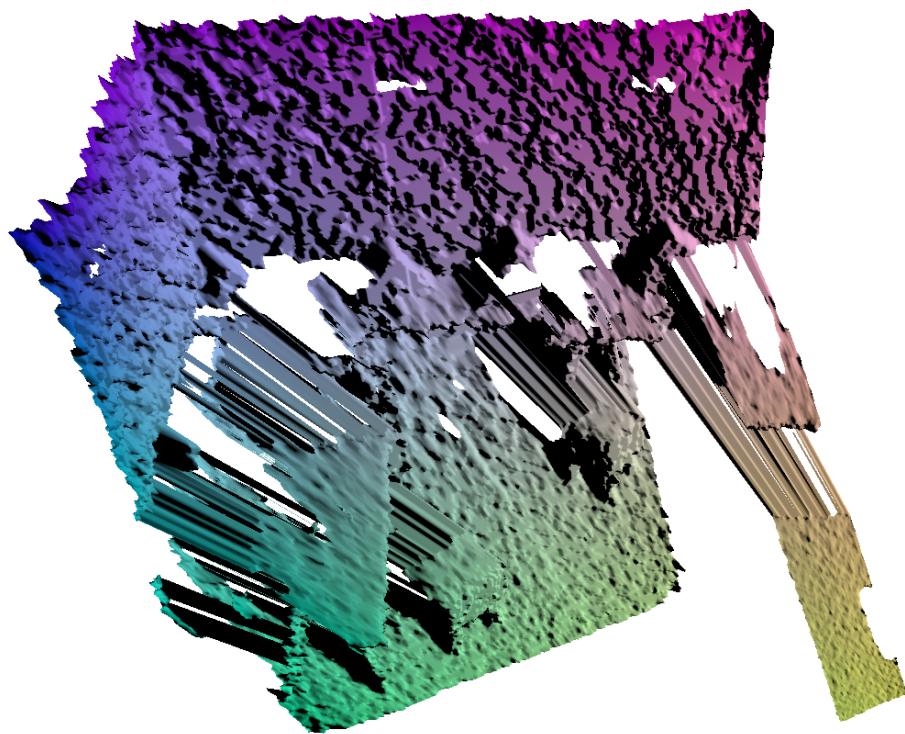


Abbildung 3.2: Das Ergebnis einer naiven Triangulierung, wenn keine Occlusion Edges entfernt werden.

3.1.2 Aspect Ratio als Grundlage

Die Grundlage für das Erkennen von Occlusion Edges bildet die Berechnung der aspect ratios ρ_t von Dreiecken, die über das Bild gelegt werden. Es gibt mehrere Varianten, ein aspect ratio für Dreiecke zu berechnen. In dieser Arbeit wird die Berechnungsform verwendet, bei der die längste Seite eines Dreiecks durch dessen kürzeste Seite geteilt wird. Somit erhalten gleichseitige Dreiecke einen Wert von 1, und umso spitzer das Dreieck ist, desto größer ist auch ρ_t . Da es entlang von Occlusion Edges

3.1. OCCLUSION EDGES

normalerweise einen vergleichsweise starken Sprung in der Tiefe gibt, erhalten alle Dreiecke, die diese abdecken, somit ein entsprechend großes ρ_t .

Das Verfahren zur Erkennung von Occlusion Edges orientiert sich an der Vorgehensweise von Hedman et. al. [5]. Es werden pro-Pixel Entscheidungen getroffen, wodurch keine großen Zusammenhänge über das Bild hinweg betrachtet werden müssen. Zunächst wird jedem Pixel ein aspect ratio ρ_p zugewiesen, welches sich aus der Betrachtung seiner acht Nachbarn ergibt. Diese werden in vier Quads der Größe 2×2 Pixel geteilt, so dass jedes Quad den aktuell behandelten Pixel an einer anderen Stelle enthält. Dies wird in Abbildung 3.3 dargestellt. Für jedes Quad ergibt sich daraufhin die Möglichkeit, zwei verschiedene Triangulierungen zu verwenden. Es wird die Variante gewählt, bei der das maximale ρ_t der beiden entstehenden Dreiecke geringer ist. Von diesen zwei Dreiecken wird das größere ρ_t für das aktuelle Quad gespeichert. Dem betrachteten Pixel wird daraufhin aus allen vier Quads das größte ρ_t als ρ_p zugeordnet.

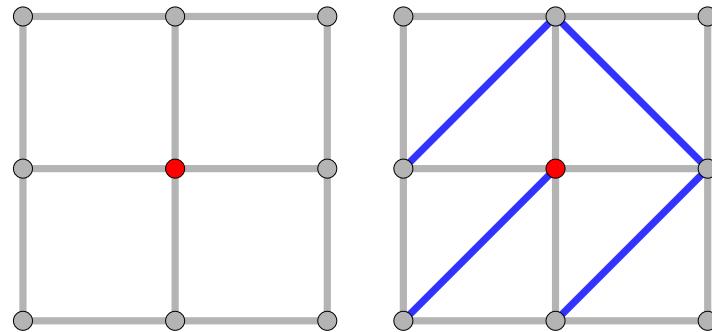


Abbildung 3.3: Die Verwendung der Nachbarn, wenn das aspect ratio eines Pixels berechnet wird.

Diese Abwandlung der Vorgehensweise von Hedman et. al. [5] sorgt dafür, dass häufiger Occlusion Edges erkannt und ρ_p im Allgemeinen größer wird. Ebenfalls werden die erkannten Edges breiter und das Gesamtergebnis enthält weniger Occlusion Edges. Insgesamt werden also mehr Pixel verworfen, um mit größerer Sicherheit mehr Occlusion Edges zu erkennen.

Während den Berechnungen kann es vorkommen, dass ein betrachtetes Pixel keine zugeordnete Tiefe hat, da der Tiefensor beim entsprechenden Bild nicht in der Lage war, den Wert zu bestimmen. Da in diesem Fall keine sichere Aussage über die Nachbarschaft gemacht werden kann, wird die Tiefe als unendlich angenommen, wodurch das zugehörige aspect ratio ebenfalls maximal wird.

Die weitere Vorgehensweise besteht darin, jeden Pixel mit einem Wert oberhalb eines selbst gewählten Thresholds τ als Teil einer Occlusion Edge zu sehen und zu entfernen. Dies allein sorgt bereits für gute Ergebnisse, jedoch fügen Hedman et. al. [5] noch zwei weitere Terme zu ρ_p hinzu. Diese werden in den folgenden Abschnitten 3.1.3 und 3.1.4 erläutert.

3.1.3 Vermeidung von Over-Detection

Hedman et. al. [5] nennen die Gefahr der over-detection, welche an solchen Stellen auftritt, die eher spitze Dreiecke erzeugen, welche allerdings immer noch Teil des Bildes sind. Beispielsweise treten an einer Tischfläche, welche mit einem flachen Betrachtungswinkel aufgenommen wird, wesentlich spitzere Dreiecke auf, als bei einem senkrechten Blick auf eine Wand. Dennoch sind diese spitzen

Dreiecke weiterhin Teil der echten Geometrie. Entsprechend ist an diesen Stellen auch das aspect ratio wesentlich erhöht. Um ein Verwerfen der Dreiecke zu verhindern, wird von Hedman et. al. der Term d_P eingeführt und mit dem aspect ratio multipliziert.

Innerhalb dieser Arbeit war es jedoch nicht möglich, eine offensichtliche over-detection derart nachzustellen, deshalb wird der Term d_P nicht verwendet.

3.1.4 Erhöhte Erkennung in der Entfernung

Die zweite Erweiterung sorgt für eine erhöhte Kantenerkennung in der Nähe und gleichermaßen für eine eingeschränktere Erkennung in der Entfernung zur Kamera. Dieser Zusatz wird hinzugefügt, weil die Informationen des Tiefensors mit größerer Entfernung ein immer größeres Rauschen enthalten und die dort gemessenen Daten somit unsicherer sind. Der in diesem Abschnitt einzuführende Term d_D verringert deswegen den Gesamtfehler, wenn der betrachtete Pixel in größerer Entfernung liegt und erhöht ihn, wenn der gemessene Wert nah an der Kamera ist. Seine Berechnung funktioniert folgendermaßen:

$$d_D = \min \left\{ 2; \max \left\{ 0,5; \frac{D}{D_{mean}} \right\} \right\} \quad (3.1.1)$$

Hierbei entspricht D der Disparität des aktuellen Pixels und D_{mean} dem Median der Disparitäten des gesamten Bildes, mithilfe dessen D normalisiert wird. Der Term wird auf das Intervall $[0,5; 2]$ beschränkt, um nicht zu extremen Einfluss zu nehmen. Insgesamt wird das durch d_D modulierte ρ_p also größer, wenn der aktuelle Pixel eine Disparität oberhalb des Medians hat, ansonsten kleiner. Dies entspricht dem genannten Wachstum in der näheren Hälfte aller Pixel und einer Verringerung in der entfernteren Hälfte.

3.1.5 Erkennung der Kanten

Werden nun die Teile aus den Kapiteln 3.1.2 und 3.1.4 zusammengesetzt, ergibt sich die Ungleichung

$$\rho_p \cdot d_D > \tau \quad (3.1.2)$$

Sobald ein Pixel den selbst gewählten Threshold τ überschreitet, wird er als Occlusion Edge erkannt und entsprechend entfernt. Für die innerhalb dieser Arbeit verwendeten Daten ergibt $\tau = 23$ einen guten Threshold, welcher im Weiteren verwendet wird.

Abbildung 3.4 stellt das Ergebnis dar, wenn auf diese Weise Occlusion Edges in dem Bild aus Abbildung 3.1 gefunden werden. Dabei zeigt sich, dass viele Pixel entlang der Occlusion Edges verworfen werden, jedoch gibt es einige Lücken, an denen die Occlusion Edges nicht erkannt werden.

3.2. HYSTERESIS THRESHOLD ZUR ERKENNUNG VON RANDFÄLLEN

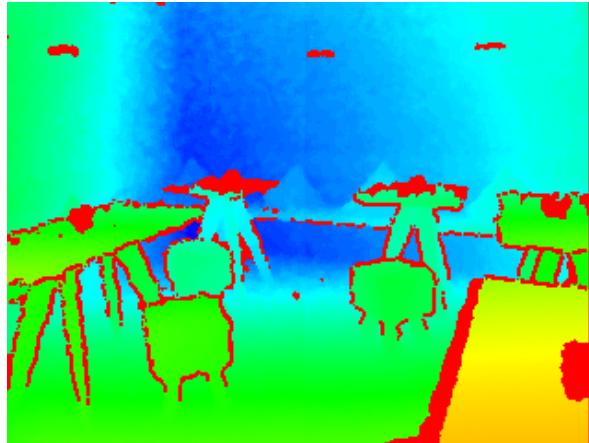


Abbildung 3.4: Das Ergebnis, wenn in Abbildung 3.1 auf bisher beschriebene Weise Occlusion Edges erkannt werden.

3.2 Hysteresis Threshold zur Erkennung von Randfällen

Ein potentielles Problem bei der Erkennung langerer Occlusion Edges ist, dass das gewählte τ nur knapp überschritten wird. Dadurch kann es sich ergeben, dass einige der Pixel einen Wert erhalten, welcher leicht unterhalb von τ liegt. Abbildung 3.5 zeigt, wie nah die Berechnungen von $\rho_p \cdot d_D$ dem Threshold in jedem Pixel des Beispielbildes kommen. Dabei sind alle Pixel mit einem Wert oberhalb des Thresholds vollständig weiß. Pixel, für die ursprünglich kein Tiefenwert gemessen werden konnte, sind schwarz. Da alle Pixel entlang einer Occlusion Edge einen Wert besitzen, der nah am Threshold liegt, kann ein Hysteresis Threshold genutzt werden, um sie zu erkennen und ebenfalls zu verwerfen.



Abbildung 3.5: Das optische Ergebnis, wenn für jeden Pixel der Abbildung 3.1 $\rho_p \cdot d_D$ berechnet wird.

Dieser funktioniert derart, dass nun zwei verschiedene Thresholds h_{min} und h_{max} festgelegt werden. Für jeden Pixel wird die gleiche Berechnung wie bereits in Abschnitt 3.1.5 durchgeführt. Liegt nun der Wert eines Pixels oberhalb von h_{max} , so wird dieser als sichere Kante betrachtet. Falls der Wert unterhalb von h_{min} liegt, bedeutet das, dass der Pixel keine Kante ist. Für den Fall, dass er jedoch

KAPITEL 3. VORVERARBEITUNG

zwischen h_{min} und h_{max} liegt, wird eine Entscheidung anhand seiner Nachbarn getroffen: Sofern einer oder mehrere der acht benachbarten Pixel eine sichere Kante sind, wird auch der aktuelle Pixel als solche gewertet, ansonsten als keine Kante. Entsprechend muss man das Bild potentiell mehrere Male durchlaufen, damit sich sichere Kanten propagieren können – jedoch sind nur simple Vergleiche vonnöten, weshalb die Iterationen nicht teuer sind.

Als Ergebnis kann man somit im Bild weitere, feinere Kanten erkennen, sofern sie mit den zuvor bereits erkannten Occlusion Edges verbunden sind. Außerdem sorgt der Hysteresis Threshold dafür, dass größtenteils erkannte Kanten noch besser erkannt werden und weniger Lücken in ihnen auftreten. In dieser Arbeit wird h_{max} auf 23 gesetzt, genau wie τ in Abschnitt 3.1.5. h_{min} wird auf 12 gesetzt. Abbildung 3.6 zeigt das Ergebnis bisher behandelte Bild, nachdem der Hysteresis Threshold darauf angewandt wurde.

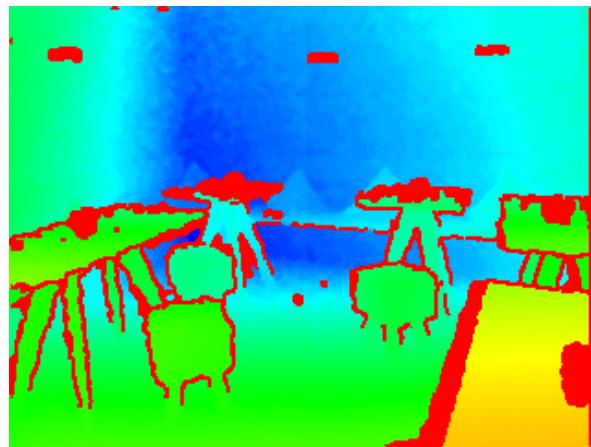


Abbildung 3.6: Das Bild aus Abbildung 3.4, nachdem ein Hysteresis Threshold angewandt wurde.

3.3. GLÄTTUNG DES BILDES

3.3 Glättung des Bildes

Der letzte Schritt der Vorverarbeitung besteht aus einer Glättung des Bildes. Der Grund dafür ist das bereits erwähnte Rauschen der Tiefendaten, speziell bei weiter entfernten Teilen einer Szene. Dieses Schwanken in den Aufnahmen sorgt beim späteren Einsatz von Errormetriken für Probleme. Um dem entgegenzuwirken, wird ein 9×9 Gauß-Filter über das Bild gelegt. Für eine möglichst schnelle Laufzeit wird die Separierbarkeit des Gauß-Filters ausgenutzt. Das bedeutet, dass der 2D-Filter in zwei 1D-Filter aufgeteilt wird, welche nacheinander auf das Eingabebild angewandt werden. Dies verringert die nötige Menge an Rechenoperationen pro Pixel von $O(n^2)$ auf $O(n)$, wobei n für den Durchmesser des Filters steht. Die beiden 1D-Filter werden nacheinander erst in x- und danach in y-Richtung auf das Eingabebild angewandt und sind abgesehen von ihrer Ausrichtung identisch:

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \quad (3.3.1)$$

x steht dabei für den Abstand des betrachteten Pixels zu dem Pixel, für den aktuell ein neuer Wert berechnet wird. σ entspricht der gewünschten Standardabweichung, welche in dieser Arbeit auf 1,2 gesetzt wird.

Das Filtern des Bildes wird erst an dieser Stelle in der Vorverarbeitung vorgenommen, da ansonsten potentiell über Occlusion Edges gefiltert werden könnte, wodurch deren Erkennung wiederum erschwert würde. Die erkannten und entfernten Occlusion Edges können nun zudem derart ausgenutzt werden, dass nicht über sie hinaus gefiltert wird. Wenn also in einem der beiden 1D-Filter ein verworfenes Pixel gefunden wird, dann werden die Gewichte für dieses und alle weiter entfernten Pixel auf 0 gesetzt.

Abbildung 3.7 stellt diese Art des Filterns exemplarisch für einen einzelnen Pixel dar. Rote Pixel enthalten dabei verworfene Werte, für blaue Pixel wird im entsprechenden Bild dargestellt, wie der Wert des aktuellen Passes berechnet wird. Die grünen Pfeile zeigen an, welche Pixel für die Berechnung der jeweiligen neuen Pixel verwendet werden. Das erste Bild stellt dar, wie für mehrere Pixel der Pass in x-Richtung berechnet wird. Im zweiten Bild werden alle somit erhaltenen Pixel-Werte verwendet, um in y-Richtung das Gesamtergebnis für ein einzelnes Pixel zu berechnen.

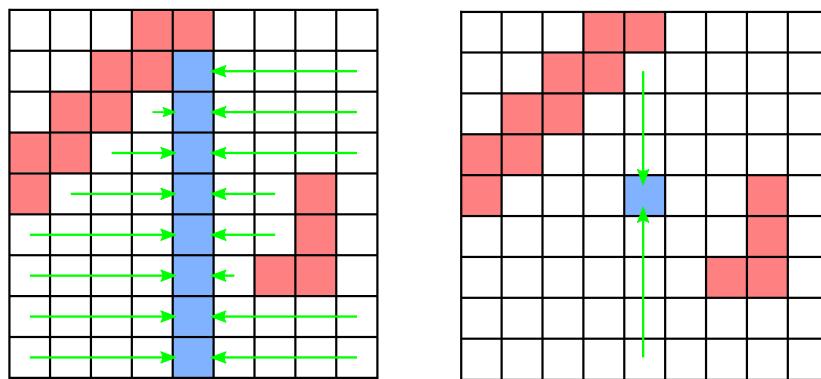


Abbildung 3.7: Die Anwendung des gesamten 9×9 Gauß-Filter für ein Pixel.

KAPITEL 3. VORVERARBEITUNG

Kapitel 4

Triangulierung

Der nächste Schritt in dieser Arbeit ist die Triangulierung der Tiefendaten. Abschnitt 4.1 zeigt eine naive Möglichkeit zu triangulieren. In Abschnitt 4.2 wird eine Triangulierung mittels Quadtrees beschrieben, welche in der Lage ist, eine große Menge an Vertices einzusparen. Der entstehende Fehler wird dadurch gering gehalten, dass nur an möglichst detailarmen Stellen Vertices verworfen werden. Dies ist vor allem praktisch, wenn das Mesh im Nachhinein noch weiter bearbeitet oder verwendet werden soll, da jegliche Arbeiten darauf entsprechend schneller möglich sind.

4.1 Naive Triangulierung

Als naive Triangulierung wird das Verfahren bezeichnet, welches das gesamte Bild in Quads der Größe 2×2 Pixel aufteilt. Diese Quads werden in zwei Dreiecke geteilt. Um zu entscheiden, welche der beiden möglichen Triangulierungen verwendet werden soll, werden die sich ergebenden Diagonalen betrachtet. Gewählt wird die Variante, welche die Länge der Diagonale minimiert, da dies im Normalfall für eine bessere Darstellung sorgt, beispielsweise entlang von Kanten jeglicher Art, die keine Occlusion Edges sind. Eine exemplarische naive Triangulierung ist in Abbildung 4.1 gezeigt.

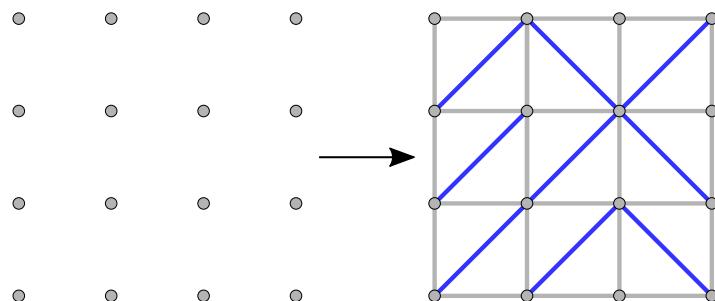


Abbildung 4.1: Eine exemplarische Darstellung der naiven Triangulierung.

Die naive Triangulierung zu nutzen hat den folgenden Vorteil: Dadurch, dass jeder einzelne Pixel verwendet wird, der nach der Vorverarbeitung noch verfügbar ist, wird ein minimaler Fehler in der Darstellung der Szene erreicht. Zwar ist das Erreichen des kleinst möglichen Fehlers wünschenswert, jedoch gibt es auch einen klaren Nachteil. Dieser besteht darin, dass das Nutzen jedes verfügbaren

KAPITEL 4. TRIANGULIERUNG

Pixels für eine extreme Menge an Primitiven im resultierenden Mesh sorgt. Bei einem Bild der Größe 320×240 , wie es in dieser Arbeit verwendet wird, könnten entsprechend bis zu 76.800 Vertices entstehen, selbst wenn ein Großteil des Bildes eine ebene Fläche zeigt. Dieses Übermaß an Primitiven ist vor allem aufgrund der dadurch sehr großen Verarbeitungszeit unpraktisch für jegliche Art der Weiterverarbeitung des Meshes. Zudem entspricht es in den meisten Fällen einem unnötigen Detailgrad.

Um dieses Problem zu lösen wird, im weiteren Verlauf des Kapitels auf eine alternative Form der Triangulierung eingegangen, welche Quadtrees als Grundlage nutzt. Dadurch soll die Menge an Vertices auf glatten, detailarmen Flächen minimiert und an detailreichen Stellen weiterhin hoch gehalten werden. Dies sorgt für einen immer noch sehr geringen Fehler in der Darstellung der Szene und gleichzeitig für eine starke Minimierung der Primitiven im Mesh.

4.2. QUADTREES

4.2 Quadtrees

4.2.1 Allgemeine Struktur

Quadtrees sind eine Baumstruktur, die sich vor allem zur Unterteilung von zweidimensionalen Eingabedaten eignet. Jeder Knoten hat bis zu vier weitere Kinder. Im Falle dieser Arbeit kann die Quadtree-Struktur derart genutzt werden, dass jeder Knoten einen bestimmten, quadratischen Teil des Eingabebildes abdeckt. Falls der abgedeckte Bereich einen zu hohen Detailgrad aufweist, wird er in vier weitere Quadrate geteilt, welche den entsprechenden Kindknoten zugeordnet werden. Wenn man diese Unterteilungen fortführt, erhält man einen Baum, der abhängig vom Detailgrad im Bild verschieden stark unterteilt ist. Die in dieser Arbeit verwendete Metrik zur Erfassung des Detailgrades wird in Kapitel 4.2.3 näher erklärt.

Um jeden Knoten in vier gleich große Kinder unterteilen zu können, muss das Eingabebild eine Größe von $(2^n + 1) \times (2^n + 1)$ haben. Die Bereiche der Kindknoten überschneiden sich daraufhin an genau einer Reihe beziehungsweise Spalte an Pixeln und haben eine Größe von $(2^{n-1} + 1) \times (2^{n-1} + 1)$. Hat die Eingabe eine unpassende Größe, kann der Wurzelknoten des Quadtrees auch so gewählt werden, dass sein Bereich über den Bildrand hinaus ragt. Es muss nur die Errormetrik derart angepasst werden, dass sie damit umgehen kann. Abbildung 4.2 stellt einen beispielhaft unterteilten Quadtree auf einem quadratischen Bild dar.

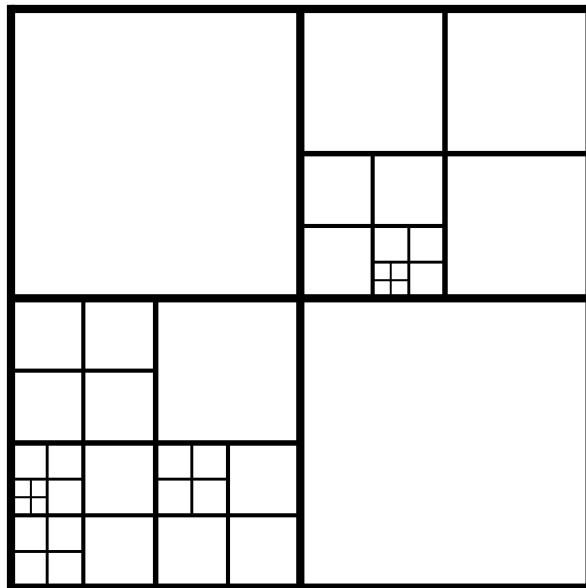


Abbildung 4.2: Ein Quadtree, wie er auf einem Eingabebild aussehen kann.

Während ein solcher Quadtree aufgebaut wird, können bereits alle für die folgende Triangulierung relevanten Vertices erkannt und gespeichert werden. Auf diese Weise ist es möglich, keine explizite Baumstruktur in der Implementierung nutzen zu müssen. In dieser Arbeit wird derart vorgegangen. Das hat zur Folge, dass lediglich Vertices ausgewählt werden müssen, die in der späteren Triangulierung verwendet werden sollen. Im weiteren Verlauf wird dieses Speichern von Vertices auch "zur Triangulierung hinzufügen" genannt.

Für die Triangulierung werden grundsätzlich immer die vier Vertices in den Ecken des Quadtrees verwendet. Des weiteren werden durch jede Unterteilung eines Quads bis zu fünf weitere Vertices festgelegt, welche Teil der späteren Triangulierung sein müssen. Abbildung 4.3 zeigt für die ersten drei Level eines Quadtrees, welche Vertices jeweils durch eine Unterteilung erkannt und zur Triangulierung hinzugefügt werden müssen.

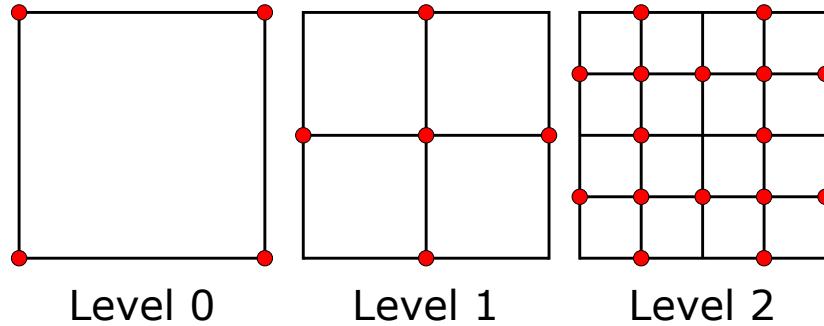


Abbildung 4.3: Die Vertices eines Quadtrees, welche für die folgende Triangulierung ausgewählt werden müssen, sofern das zugehörige Quad geteilt wird.

Eine solche Auswahl an Vertices kann daraufhin zur Triangulierung eines Bildes genutzt werden, wobei man jedes Quad einzeln betrachten und bearbeiten kann. Eine simple Art des Triangulierens besteht darin, jedes Quad in zwei Dreiecke zu teilen. Das Problem dieser Herangehensweise ist jedoch, dass ein Unterschied im Level benachbarter Quads im Quadtree für Risse im entstehenden Mesh sorgen würde. Ein Beispiel ist im ersten Bild der Abbildung 4.4 zu sehen, wobei das Mesh an den grauen Vertices Risse aufweist.

Als Alternative dazu bietet es sich an, jedes Quad derart zu triangulieren, dass alle Vertices von benachbarten Quads mit einbezogen werden. Hierfür wird ein Beispiel im zweiten Bild der Abbildung 4.4 dargestellt. Dieses Vorgehen sorgt dafür, dass die Triangulierung von größeren Quads, welche viele anliegende Nachbarn auf tieferem Level haben, kompliziert wird und viele spitze Dreiecke erzeugt.

Um sowohl das Problem der Risse als auch das der unregelmäßigen Dreiecke innerhalb des Meshes nach der Triangulierung zu lösen, werden die normalen Quadtrees zu sogenannten Restricted Quadtrees erweitert. Diese werden in Abschnitt 4.2.2 genauer beschrieben.

4.2.2 Einschränkung zu Restricted Quadtrees

Ein Restricted Quadtree (RQT) ist ein Quadtree, für den eine einzelne Zusatzregel gilt: Benachbarte Quads dürfen sich in ihrem Level um nicht mehr als 1 unterscheiden. Ein RQT kann sich von einem simplen Quadtree zudem derart unterscheiden, dass ein einzelner Teil eines Quads tiefer unterteilt wird, ohne dass der Rest des Quads von Änderungen betroffen ist. Diese Unterschiede zwischen einem regulären und Restricted Quadtree sind in Abbildung 4.5 dargestellt.

Um einen RQT zu erstellen, gibt es verschiedene Möglichkeiten. Diese Arbeit verwendet den Ansatz aus der Veröffentlichung von Renato Pajarola [10]. Dieser stellt einen Abhängigkeitsgraphen vor, mithilfe dessen ein RQT implizit und ohne besondere Quadtree-Strukturen gebaut werden kann.

4.2. QUADTREES

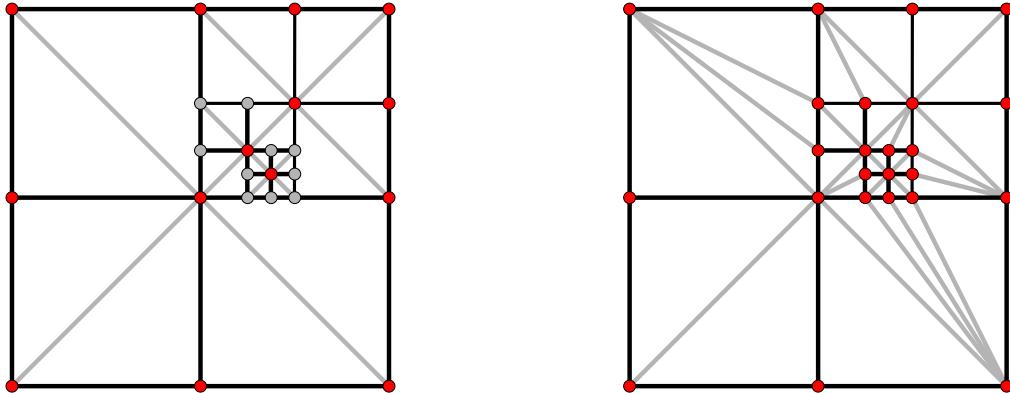


Abbildung 4.4: Mögliche naive Triangulierungen eines Quadtrees.

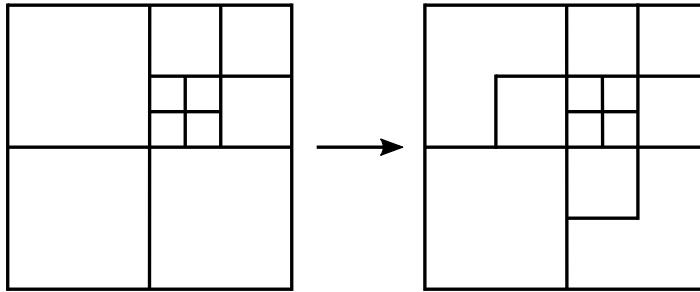


Abbildung 4.5: Die Erweiterung eines regulären Quadrees zu einem Restricted Quadtree.

Dieser Abhängigkeitsgraph enthält für jeden einzelnen Vertex eines Quadtrees Referenzen auf bestimmte andere Vertices. Während ein normaler Quadtree erstellt wird, müssen für jeden ausgewählten Vertex all seine Abhängigkeiten ebenfalls zur Triangulierung hinzugefügt werden. Wird dies getan, sind nach der Erstellung des Quadtrees alle Vertices für die Triangulierung ausgewählt, die für den zugehörigen RQT benötigt werden. Auf das genaue Aussehen und die Prinzipien des Abhängigkeitsgraphen wird in Kapitel 4.2.4 tiefer eingegangen.

Eine Besonderheit der Vertexauswahl im RQT ist, dass auf diese Weise der Vertex in der Mitte eines Quads für die Triangulierung ausgewählt werden kann. Dieser wird im weiteren Verlauf der zentrale Vertex eines Quads genannt. Ist ein solcher zentraler Vertex eines Quads für die Triangulierung ausgewählt, sorgt dies noch nicht dafür, dass das Quad unterteilt werden muss. Dieser Vertex wird im Folgenden dafür verwendet, Quads unterschiedlichen Levels nahtlos zu verbinden.

Abbildung 4.6 stellt dar, welche Vertices beim Erweitern eines regulären Quadrees zu einem RQT mithilfe des Abhängigkeitsgraphen dafür ausgewählt werden, während der anschließenden Triangulierung genutzt zu werden.

Ist ein RQT fertig erstellt, so kann dieser mit wenigen Regeln vollständig trianguliert werden. Dabei kann man pro Quad vorgehen, ohne die globale Aufteilung der Quads zu beachten. Die Regeln werden im Folgenden aufgelistet:

1. Wenn kein Quad außer dem auf Level 0 existiert, kann dieses in beliebiger Richtung trianguliert werden.

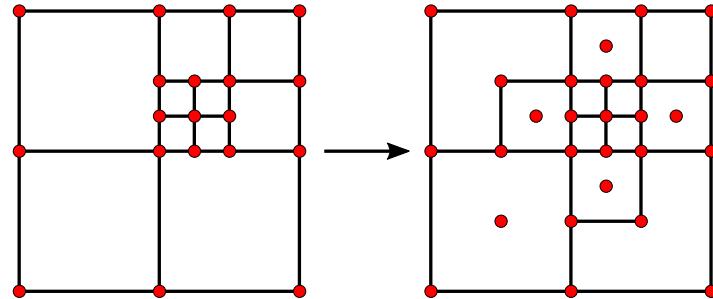


Abbildung 4.6: Die Vertices, welche nach der Erweiterung eines Quadtrees zu einem RQT von der folgenden Triangulierung genutzt werden müssen.

2. Wurde der zentrale Vertex eines Quads für die Triangulierung ausgesucht, so wird das entsprechende Quad in vier gleiche Dreiecke geteilt. Die Regel trifft auch zu, wenn das Quad des betrachteten zentralen Vertex weiter unterteilt wurde. Abbildung 4.7 zeigt dieses Vorgehen, wobei nur Vertices dargestellt werden, die für das Auslösen der Regel relevant sind.
3. Sind entlang des Randes eines Quads weitere Knoten Teil der Triangulierung, die durch Nachbarn auf einem tieferen Level hinzugefügt wurden, so muss der zentrale Vertex zusätzlich zu Regel 2 noch mit diesen Knoten verbunden werden. Ein Beispiel ist in Abbildung 4.8 zu sehen.

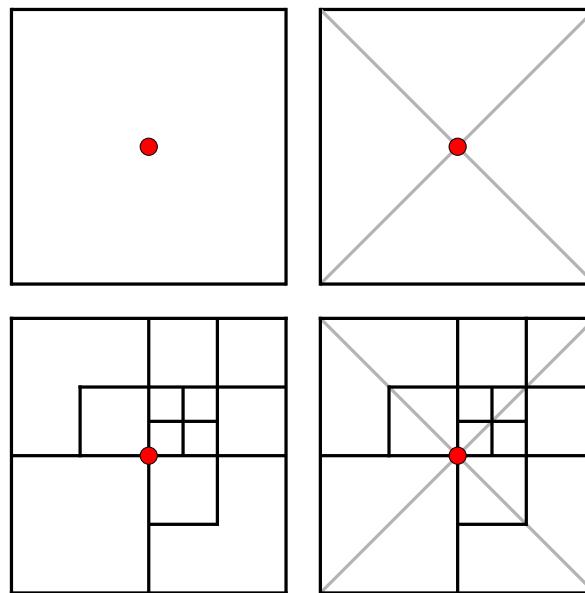


Abbildung 4.7: Triangulierungsregel 2 eines RQT.

Der Abhängigkeitsgraph aus Abschnitt 4.2.4 stellt sicher, dass jegliche Knoten für die Ausführung aller Regeln auch tatsächlich Teil der Triangulierung sind. In Abbildung 4.9 ist ein beispielhafter RQT zu sehen, wie er komplett trianguliert wird.

4.2. QUADTREES

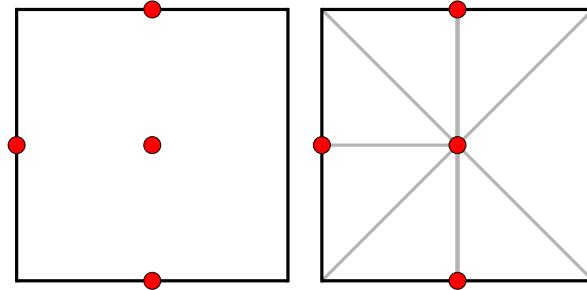


Abbildung 4.8: Triangulierungsregel 3 eines RQT.

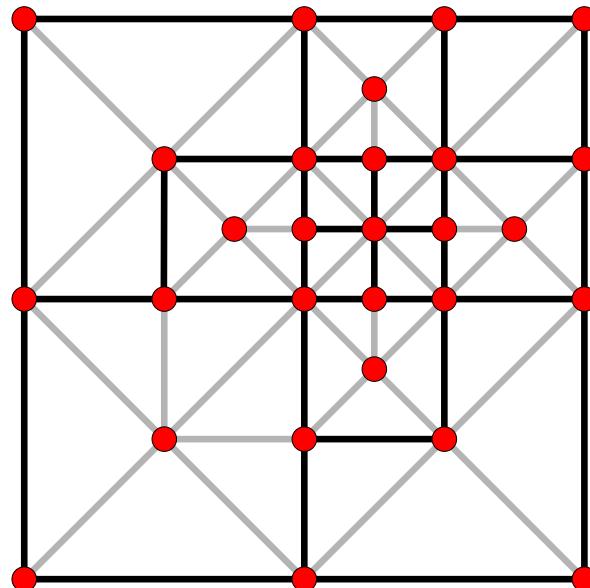


Abbildung 4.9: Triangulierung des exemplarischen RQT aus Abbildung 4.6.

4.2.3 Errormetrik

Um zu entscheiden, ob ein Quad unterteilt werden soll, wird es zuerst mit einer simplen Metrik betrachtet und in einen von vier Fällen eingeteilt. Diese Fälle werden in Abbildung 4.10 dargestellt, wobei der blaue Bereich dem behandelten Bild und das schwarze Quadrat dem Umfang des gesamten Quadtrees entspricht. Das rote Quadrat stellt außerdem jenes Quad des Quadtrees dar, welches in den entsprechenden Fall einzuordnen ist.

1. Das Quad liegt außerhalb des Eingabebildes beziehungsweise deckt maximal eine Reihe von Pixeln des Bildes ab. Dieser Fall tritt nur auf, wenn das Eingabebild nicht genau $(2^n + 1) \times (2^n + 1)$ Pixel groß ist.
2. Das Quad deckt nur verworfene Pixel ab. In diesem Fall darf entsprechend auch der Rand des Quads nur verworfene Werte enthalten.
3. Das Quad liegt nur zum Teil auf dem Eingabebild. Dieser Fall kann nur vorkommen, wenn das Eingabebild keine Größe von $(2^n + 1) \times (2^n + 1)$ besitzt.

4. Das Quad liegt vollständig auf dem Eingabebild.

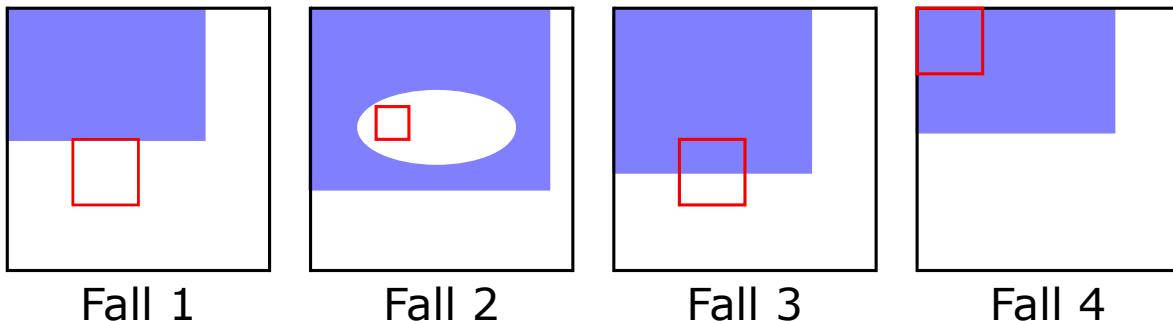


Abbildung 4.10: Die vier Fälle, in welche ein Quad eingeteilt werden kann.

Im Fall 1 kann das komplette Quad ignoriert und die Unterteilung gestoppt werden. Das entsprechende Quad deckt keine Geometrie ab, für dessen Triangulierung es gebraucht wird. Sind am Rand noch verwendbare Pixel enthalten, so können diese von dem Nachbar behandelt werden, der sich diese Reihe an Pixeln mit dem behandelten Quad teilt. Liegt das Quad jedoch innerhalb des Bildes, kann es auch potentiell Fall 2 zugeordnet werden. Hier kann aus gleichem Grund die weitere Unterteilung abgebrochen werden. Als zusätzliche Bedingung darf jedoch selbst der Rand des Quads keine nutzbaren Messwerte abdecken, weil dieser bereits alle Vertices enthalten könnte, welche für die Triangulierung des Quads auf entsprechendem Level nötig sind. Ist dies der Fall, würde die folgende Triangulierung das Quad fälschlicherweise triangulieren. Der Grund dafür ist, dass innerhalb dieser Arbeit keine explizite Datenstruktur genutzt, sondern nur mithilfe der für die Triangulierung ausgewählten Vertices gearbeitet wird. Im Fall 3 kann das Quad sofort und ohne weitere Fehlerüberprüfung unterteilt werden, da die ausgewählten Vertices des aktuellen Levels nicht ausreichen, um den kompletten Rand zur Triangulierung hinzuzufügen. Im Fall 4 muss das Quad mithilfe einer Errormetrik überprüft werden, um seinen Detailgrad zu bestimmen und über eine Unterteilung des Quads zu entscheiden.

Die in dieser Arbeit verwendete Metrik berechnet für jeden Pixel eines Quads einen Fehler und unterteilt es, sofern bereits an einem einzigen Pixel ein selbst gewählter Threshold überschritten wird. Zur Ermittlung des Fehlers pro Pixel wird die Triangulierung des Quads betrachtet, welche sich ohne einen ausgewählten zentralen Vertex ergeben würde, und es wird der Abstand von allen Pixeln zu diesen Dreiecken berechnet. Durch diese Vereinfachung ist die Metrik simpler und es müssen weniger Überprüfungen gemacht werden, da statt zwischen zwei und acht Dreiecken immer nur zwei verwendet werden. Da nur gleich viele oder weniger Dreiecke verwendet werden, als eigentlich Teil der Triangulierung wären, kann im Regelfall davon ausgegangen werden, dass der gemessene Fehler gleich oder größer ist. In diesem Fall ergibt sich entsprechend eine etwas genauere Unterteilung, welche einer zu groben Unterteilung vorzuziehen ist.

Für jedes Pixel wird der Abstand zu dem spezifischen Dreieck betrachtet, das ihn abdeckt. Der Abstand ist die kürzeste Distanz zwischen diesem Pixel und dem Dreieck. Dieser Abstand $dist$ wird noch weiter modifiziert, um den von der Metrik beschriebenen Error zu erhalten:

4.2. QUADTREES

$$error = \frac{dist}{depth^2} \quad (4.2.1)$$

$depth$ entspricht dem Abstand des für den Pixel gemessenen Tiefenwertes zum Tiefensensor, der das Bild aufgenommen hat. Dieser Wert wird als Nenner in der Gleichung verwendet, um dem Rauschen der Tiefendaten in der Entfernung entgegenzuwirken: Je weiter entfernt die aufgenommene Szene ist, desto stärker ist das Rauschen und desto stärker wird der Gesamtfehler eingeschränkt.

Als einzige Sonderregel werden Quads sofort unterteilt, sobald ein enthaltener Pixel einen kaputten oder entfernten Tiefenwert enthält. Der Grund dafür ist, dass die komplette Szene dargestellt werden soll, ohne für manche Pixel Messungen zu erfinden oder zu entfernen. Würde man also ein Quad mit einem entfernten Pixel triangulieren, würde man für dieses Pixel Geometrie erfinden, die nicht unbedingt existiert – beispielsweise für die in Kapitel 3 entfernten Occlusion Edges. Diese Sonderregel würde zwar dafür sorgen, dass die Regeln 1, 2 und 3 der vorangehenden simplen Überprüfung ebenfalls abgebildet werden, allerdings sorgt das komplett Ignorieren eines Quads für einen geringeren Rechenaufwand.

4.2.4 Abhängigkeitsgraph

Um einen Quadtree zu einem Restricted Quadtree zu erweitern, verwendet diese Arbeit eine Vorgehensweise, wie sie auch von Renato Pajarola [10] vorgestellt wird. Dabei wird ein Abhängigkeitsgraph verwendet, welcher jedem einzelnen Vertex bis zu zwei anderen Vertices als Abhängigkeit zuordnet. Während ein Quadtree erstellt wird, werden alle Vertices zur Triangulierung hinzugefügt, die nötig sind, um ihn darzustellen. Zusätzlich müssen währenddessen noch jegliche Abhängigkeiten dieser ausgewählten Vertices zur Menge der für die Triangulierung verwendeten Vertices hinzugefügt werden. Dieser Vorgang muss iteriert werden, da auch die neu hinzugefügten Vertices Abhängigkeiten haben können, welche noch nicht Teil der Triangulierung sind. Sobald dieser Schritt abgeschlossen ist, erfüllt der Quadtree alle nötigen Eigenschaften für einen RQT und kann entsprechend trianguliert werden.

Da der Graph simplen Regeln folgt und immer die gleiche Struktur hat, kann er automatisch für Quadtrees verschiedener Größen generiert werden. Zudem ist es möglich, nur einen Abhängigkeitsgraphen für mehrere Quadtrees gleicher Größe zu erstellen – dies ist vor allem praktisch, wenn mehrere Bilder der gleichen Größe nacheinander bearbeitet werden sollen.

Die Regeln, nach denen man einen Abhängigkeitsgraphen erstellen kann, werden im Folgenden genannt:

1. Zentrale Vertices eines Quads sind abhängig von den Eck-Vertices desselben Quads, die nicht bereits Teil dessen Triangulierung sind. Diese Regel wird in Abbildung 4.11 demonstriert.
2. Vertices auf dem Rand eines oder mehrerer Quads sind abhängig von den zentralen Vertices der entsprechenden Quads. Diese Regel wird in Abbildung 4.12 demonstriert.

Ein großer Vorteil des Abhängigkeitsgraphen ist, dass keine spezielle Struktur für Quadtrees implementiert werden muss, sondern lediglich die entsprechenden neuen Vertices zur zuvor festgelegten Auswahl an Vertices für das Triangulieren des Quadrees hinzugefügt werden müssen. Im

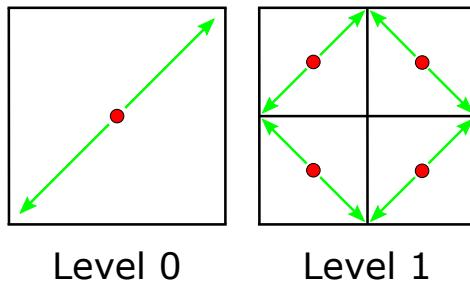


Abbildung 4.11: Regel 1 zum Erstellen eines Abhängigkeitsgraphen.

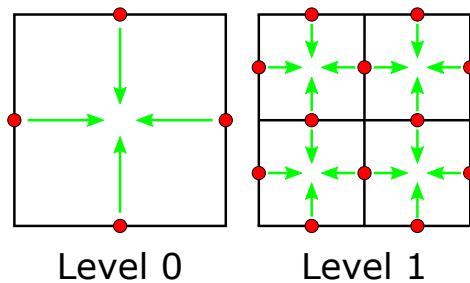


Abbildung 4.12: Regel 2 zum Erstellen eines Abhängigkeitsgraphen.

Ausgleich kann es allerdings vorkommen, dass ein Quad bereits fertig unterteilt wurde und durch Abhängigkeiten im Nachhinein weitere Vertices innerhalb des Quads ausgewählt werden. Unter diesen Umständen könnte die sich ergebende, feinere Triangulierung wieder den Error-Threshold der Quadtree Metrik überschreiten. Im Normalfall sorgt eine genauere Triangulierung allerdings auch für eine geringere Abweichungen von der Originalszene. Aufgrund dessen wird nicht weiter auf das mögliche Wachstum des Errors durch eine tiefere Unterteilung eines Quads eingegangen.

Kapitel 5

Reduzierung des Meshes

Dieses Kapitel beschäftigt sich mit einer weiteren Verringerung der Primitiven innerhalb des erstellten Meshes. Der Quadtree ist dafür gedacht, die detailarmen Stellen im bearbeiteten Tiefenbild mit weniger Primitiven darzustellen. Gleichzeitig ist er allerdings an bestimmte strukturelle Regeln gebunden. Dies sorgt dafür, dass er potentiell auch an solchen Bereichen Vertices erstellt, die keine besonderen Details darstellen. Ein Beispiel sind verworfene Pixel im Bild, welche ein Teilen der abdeckenden Quads auf die tiefste Ebene erzwingen. Diese weitere Reduzierung des RQT-Meshes entfernt vor allem solche Primitiven.

Innerhalb des Kapitels wird zuerst die Metrik genannt, nach der der Fehler für das Entfernen von Vertices berechnet wird. Des Weiteren wird auf den Ablauf der in dieser Arbeit verwendeten Dezimierung eingegangen. Zudem werden diverse Erweiterungen vorgestellt, welche die Darstellung der ursprünglichen Szene nach der Reduzierung originalgetreuer halten sollen.

5.1 Quadratische Errormetrik

Für das Reduzieren eines Meshes muss zuerst eine Errormetrik festgelegt werden, anhand derer man jegliche Änderungen bewerten kann. Es wird folgendermaßen vorgegangen: Zuerst werden zwei Vertices betrachtet, welche potentiell vereint werden sollen. Als neuer, kombinierter Vertex wird einer der beiden betrachteten Knoten gewählt. Der Fehler ergibt sich aus dem quadrierten und summierten Abstand dieses neuen Vertex zu all den Ebenen der Faces, die von den beiden ursprünglichen Vertices repräsentiert werden. Falls die Vertices zusammengefügt werden, stellt der neue Vertex daraufhin die gesamten Faces dieser beiden Vertices dar. Auf diesem Weg wird das Zusammenfassen vieler Vertices in einen immer teurer und ein Vertex repräsentiert bei der Fehlerberechnung alle Faces, die er im reduzierten Mesh darstellt, anstelle davon einen Mittelwert zu verwenden.

Um die dargestellten Faces effizient für jeden einzelnen Vertex zu speichern, wird die Vorgehensweise von Garland et. al. [4] angewandt: Jede Ebene eines Faces kann in einer 4×4 Matrix K_f dargestellt werden. Die Gleichung 5.1.1 zeigt die verwendete Darstellung für Ebenen, welche wie in Gleichung 5.1.2 zu der entsprechenden Matrix umgewandelt werden kann. Diese Matrix kann im Folgenden wiederum wie in Gleichung 5.1.3 mit einem beliebigen Punkt $v = [v_x \ v_y \ v_z \ 1]^T$ multipliziert werden, um den quadrierten Abstand d_{vK}^2 des Punktes zur Ebene zu erhalten. Der große

KAPITEL 5. REDUZIERUNG DES MESHES

Vorteil dieser Darstellung besteht aus dem Fakt, dass mehrere solcher Matrizen addiert und weiterhin auf gleiche Weise verwendet werden können. Das Ergebnis der Multiplikation aus Gleichung 5.1.3 ist die Summe aller quadrierten Abstände des Punktes zu den in der Matrix dargestellten Ebenen. Auf diese Weise kann der Speicherverbrauch im Verlauf der Dezimierung gleichbleibend gering gehalten werden, da für jeden Vertex nur eine einzige 4×4 Matrix angelegt wird und keine Liste an Faces verwaltet werden muss.

$$ax + by + cz + d = 0 \quad , \text{ wobei } a^2 + b^2 + c^2 = 1 \quad (5.1.1)$$

$$K = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix} \quad (5.1.2)$$

$$d_{vK}^2 = v^T K v \quad (5.1.3)$$

Um jedem Vertex eines Meshes eine solche Errormatrix zuzuordnen, wird zuerst die repräsentierende Matrix K_f eines jeden Faces ausgerechnet, bevor die Dezimierung beginnt. Als Zusatz wird jede Matrix einmal zu Beginn mit der Fläche ihres Faces multipliziert, um größere Faces stärker zu gewichten – diese Erweiterung wird von OpenMesh [1] übernommen. Daraufhin werden die Matrizen K_f auf die jeweiligen Matrizen K_v der drei anliegenden Vertices addiert. Jeder Vertex stellt also zu Beginn alle Faces dar, von denen er ein Teil ist. Werden zwei Vertices daraufhin vereint, müssen für die Fehlerberechnung im Folgenden nur ihre jeweiligen Errormatrizen addiert werden.

Hierbei lässt sich auch ein Nachteil der Darstellung aller Faces als Matrix erkennen: Die Errormatrix eines Faces taucht in drei verschiedenen Vertices auf und ist nach einer Addition mit anderen Matrizen nicht mehr unterscheidbar. Entsprechend kann die Ebene eines Faces nach ein paar Dezimierungen bis zu dreifach in einem einzelnen Vertex vorhanden sein. Da dies jedoch auch bedeutet, dass dieser eine Vertex das komplette Face darstellen muss, wird dies in Kauf genommen.

5.2. ABLAUF DER DEZIMIERUNG

5.2 Ablauf der Dezimierung

Wie im vorherigen Abschnitt bereits angesprochen, werden mehrere Vertices zu einem einzelnen vereint, um ein Mesh zu dezimieren. Innerhalb dieser Arbeit wird sich dabei auf Paare von Vertices beschränkt. Zudem werden nur Vertices als potentielle Dezimierungspartner betrachtet, wenn sie von einer Kante verbunden werden. Somit werden pro Dezimierungsoperation zwei Faces und ein Vertex entfernt. Dies wird in Abbildung 5.1 dargestellt: Eine Dezimierung von Vertex A auf Vertex B entfernt ersteren und die zwei grau markierten Flächen. Der Grund für die Einschränkung auf nur direkt benachbarte Vertex-Paare ist, dass das Verbinden von unzusammenhängenden Vertices unweigerlich zum Erfinden von neuer Geometrie und somit für ein schnelleres Abweichen vom Originalmesh sorgt. Diese Art der Abweichung würde zudem nur zum Teil von der quadratischen Errormetrik erfasst werden, da diese einerseits nur die kompletten Ebenen der betroffenen Faces betrachtet und andererseits nur auf den Fehler an dem neu entstehenden Vertex achtet. Ein Fehler, der dadurch zustande kommt, dass die anliegenden Faces nach der Dezimierung neue Geometrie darstellen oder alte Geometrie verwerfen, wird nicht erfasst.

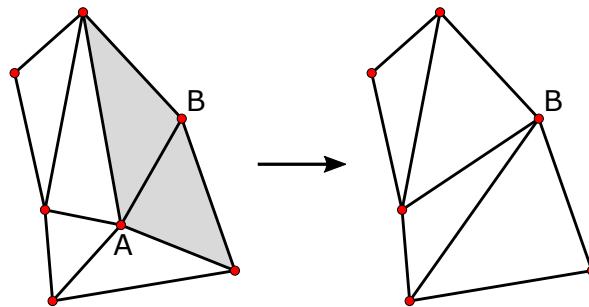


Abbildung 5.1: Eine simple Dezimierungsoperation.

Der Ablauf der Dezimierung sieht wie folgt aus:

1. Es werden die fundamentalen Errormatrizen K_f wie in Abschnitt 5.1 für jedes einzelne Face erstellt.
2. Jedem Vertex des Meshes wird die Summe aller K_f der anliegenden Faces als Errormatrix K_v zugeordnet.
3. Für jeden Vertex wird aus seinen Nachbarn der beste Dezimierungspartner bestimmt. Dies geschieht, indem für jedes Paar die beiden K_v addiert und mit der Position des entsprechenden Partners nach Gleichung 5.1.3 multipliziert werden. Der gewählte Dezimierungspartner ist jener Nachbar, mit welchem sich der geringste Fehler ergibt.
4. Alle Vertices werden anhand dessen sortiert, wie groß der entstehende Error wäre, wenn sie auf ihren optimalen Partner dezimiert werden würden.
5. Der Vertex mit dem niedrigsten Fehler wird aus der Liste genommen und dezimiert, sofern der Fehler unter einem selbst gewählten Threshold liegt. Ist dieser Threshold überschritten, wird die Dezimierung abgebrochen.

KAPITEL 5. REDUZIERUNG DES MESHES

6. Alle Nachbarn des neu entstandenen Vertex und er selbst werden bezüglich optimalem Dezmierungspartner und entstehendem Error aktualisiert, da sich diese durch die vollzogene Dezmierung geändert haben können. Daraufhin werden sie neu in die Liste aller Vertices und ihrer Dezmierungsfehler eingesortiert.
7. Die Schritte ab Punkt 5 werden wiederholt, bis die Schleife abgebrochen wird oder eine gewünschte Zahl an Reduzierungen durchgeführt wurde.

5.3. ERWEITERUNGEN

5.3 Erweiterungen

Um einen noch geringeren Fehler beim Dezimieren von Kanten zu erhalten, wird die in Abschnitt 5.1 vorgestellte Metrik auf diverse Weisen erweitert. Hierdurch soll der Error verfeinert und an manchen Stellen das Vereinen von Vertex-Paaren sogar komplett verboten werden.

5.3.1 Tiefenabhängiger Fehler

Ähnlich wie im Kapitel 4.2.3 wird auch hier die Tiefe in den Fehler mit eingerechnet. Der Grund bleibt ebenfalls der gleiche: Um dem Rauschen der Tiefendaten in der Ferne entgegenzuwirken, wird dort auch ein erhöhter Fehler zugelassen. Genau genommen wird der bisher berechnete Fehler $error$ durch den quadrierten Abstand $depth^2$ des betrachteten Vertex zum Tiefensor geteilt:

$$error_{new} = \frac{error}{depth^2} \quad (5.3.1)$$

5.3.2 Verhindern der Invertierung von Dreiecken

Das simple Dezimieren von Kanten, wie es bisher gezeigt wurde, kann dazu führen, dass sich die Orientierung von beteiligten Faces invertiert und dass sich Faces überschneiden. Ein Beispiel für dieses Problem im zweidimensionalen Raum wird in Abbildung 5.2 aufgeführt: Sobald Vertex B auf Vertex C kollabiert wird, überschneiden sich die übrigen Dreiecke. Zudem kehrt sich die Orientierung des roten Dreiecks um.

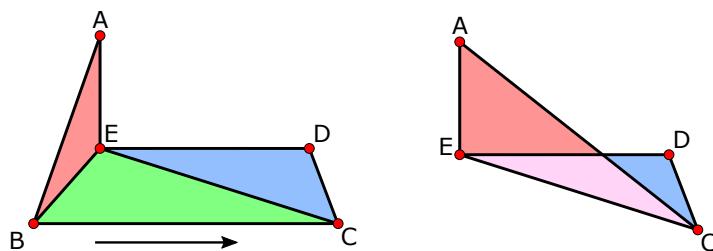


Abbildung 5.2: Eine Möglichkeit, wie ein Dreieck bei der Dezimierung einer Kante invertiert werden kann.

Im dreidimensionalen Raum ist dieses Problem zwar weniger vorhanden als im zweidimensionalen, allerdings kann es auch dort auftreten: Wenn ein Teil eines zu bearbeitenden Meshes eine glatte Fläche abbildet, entstehen dort leicht invertierte Dreiecke.

Zwar ist eine derartige Invertierung von Faces topologisch kein Fehler, allerdings sorgt sie für Probleme sowohl bei der Darstellung als auch bei der Weiterverarbeitung. Daher ist es von Vorteil, die Kontraktionen aller Kanten zu verhindern, welche eine Invertierung eines oder mehrerer anliegender Faces verursachen würden.

Um dies zu erreichen, werden beim Prüfen jeder Kante auf den Fehler ihrer Dezimierung zusätzlich die Normalen aller anliegenden Faces vor und nach der potentiellen Dezimierung berechnet. Liegen diese Normalen eines einzelnen Faces weiter als 60° auseinander, so wird das Kollabieren der Kante verboten.

5.3.3 Einschränkung der Kontraktionspartner von Rand-Vertices

Um die Menge der vom Mesh dargestellten Geometrie möglichst über den Verlauf der Dezimierung hinweg zu erhalten, ist es besonders wichtig, den Rand des Meshes zu betrachten. Das liegt daran, dass das Entfernen eines einzelnen Vertex am Rand häufig zu einem Verlust an Geometrie und somit an Information über die Szene sorgt. Um dieses Problem einzuschränken, dürfen Vertices am Rand eines Meshes einzig mit anderen Rand-Vertices kollabiert werden. Somit können sie nicht ins Innere des Meshes wandern, wodurch Geometrie verloren ginge. Werden Rand-Vertices nur noch aufeinander geschoben, ist ein Verlust von Geometrie bereits wesentlich unwahrscheinlicher.

Abbildung 5.3 stellt dies dar: Würde Vertex B auf Vertex A geschoben werden, ginge ein Großteil der Fläche des Meshes verloren. Innere Knoten wie Vertex A dürfen weiterhin auf äußere Knoten geschoben werden. Ebenfalls erlaubt ist das Kollabieren von Vertex B auf Vertex C, da beide am Rand des Meshes liegen. Dadurch wird der Verlust an dargestellter Geometrie verringert.

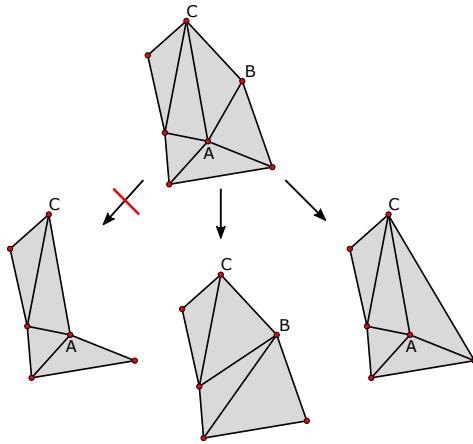


Abbildung 5.3: Verlust von Geometrie durch die Kontraktion von Rand-Vertices.

5.3.4 Dezimierung von nur parallelen Rand-Kanten

Die Erweiterung aus Abschnitt 5.3.3 sorgt bereits für einen wesentlich geringeren Grad an entfernter Geometrie während der Dezimierung. Jedoch kann es weiterhin vorkommen, dass beispielsweise entlang einer Ecke des Meshes kollabiert wird, wodurch diese Ecke nicht mehr komplett dargestellt wird. Dieses Verhalten ist in Abbildung 5.5 zu sehen.

Um diese Art von Genauigkeitsverlust zu verhindern, wird die Dezimierung derart erweitert, dass Rand-Vertices nur noch auf Nachbarn kollabiert werden dürfen, wenn alle betroffenen Kanten nahezu parallel zueinander sind. Genauer bedeutet das, dass diese Vertices nur noch entlang gerader Kanten dezimiert werden dürfen. Kontraktionen um Ecken werden somit verboten und ein Verlust oder ein Erfinden von Geometrie wird weiter eingeschränkt.

In Abbildung 5.4 ist zu sehen, wie sich diese Erweiterung auswirkt: Wird Vertex B auf Vertex C kollabiert, sind zwei Kanten betroffen, die beinahe senkrecht aufeinander stehen, daher wird die Kontraktion verboten. Wird Vertex A auf Vertex B kollabiert, sind jedoch nur nahezu parallele Kanten betroffen und die Aktion wird erlaubt. Zwar geht auch dadurch weiterhin Geometrie verloren, jedoch

5.3. ERWEITERUNGEN

wird der Verlust eingeschränkt.

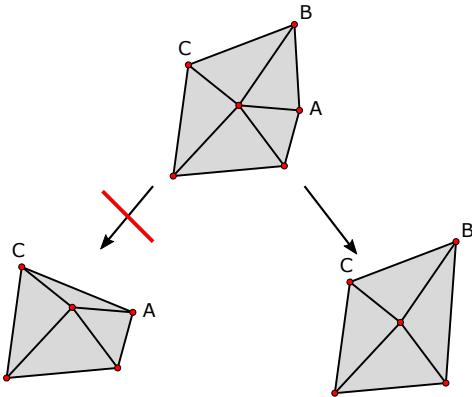


Abbildung 5.4: Verlust von Geometrie durch eine Kontraktion von Rand-Vertices um eine Ecke.

Abbildung 5.5 zeigt den Einsatz auf einem größeren Mesh. Dabei wurde das linke Mesh ohne und das rechte mit der Erweiterung dieses Kapitels bearbeitet. Wie zu erkennen ist, bleiben Löcher und Ränder des Meshes durch diese Erweiterung sehr gut erhalten. Als Nachteil ist zu nennen, dass uneinheitliche Rändern zwar sehr detailliert erhalten bleiben, jedoch muss dafür in Kauf genommen werden, dass viele kleine Primitiven nötig sind und dass diese auch bei starker Reduzierung des Meshes erhalten bleiben.

Des Weiteren kann das Mesh durch dieses Verbot nicht mehr vollständig dezimiert werden, da die Konturen des Meshes unabhängig vom Grad der Dezimierung erhalten bleiben.

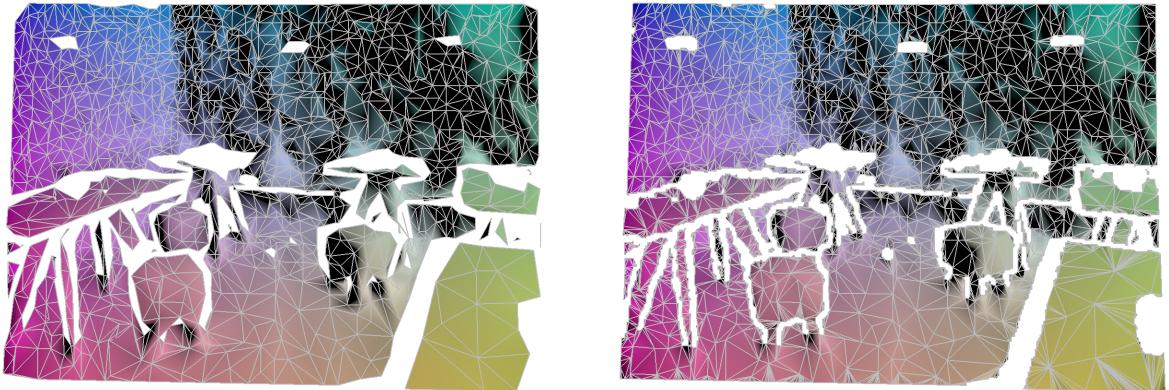


Abbildung 5.5: Auswirkung der Beschränkung des Kapitels 5.3.4 auf ein Mesh.

5.3.5 Kantenerhaltung im Mesh

Ecken am Rand des Meshes werden mit den behandelten Erweiterungen bereits erkannt und erhalten, jegliche Ecken und Wölbungen innerhalb eines Meshes werden jedoch bisher einzig der Errormetrik überlassen. So könnte beispielsweise weiterhin die Ecke eines Schrankes durch Dezimierung abgerundet werden. Nun soll speziell an solchen Stellen der Fehler der Kontraktion von Kanten erhöht werden.

Um dies zu erreichen, wird die Vorgehensweise von Hedman et. al. [5] verwendet: Vor jeder Vereinigung von Vertices werden all die anliegenden Faces betrachtet und deren Normalen gesammelt. Liegt irgendeine Kombination dieser Normalen weiter als 60° auseinander, so wird der Error der Kontraktion der Vertices mit 10 multipliziert. Das Ergebnis der Erweiterung ist in Abbildung 5.6 zu sehen: Das linke Bild zeigt ein Mesh, welches ohne diese Erweiterung dezimiert wurde. Im Vergleich zum rechten Mesh, bei dessen Dezimierung diese Erweiterung angewandt wurde, bleiben weniger Vertices erhalten und die Wölbung wird schlechter dargestellt.

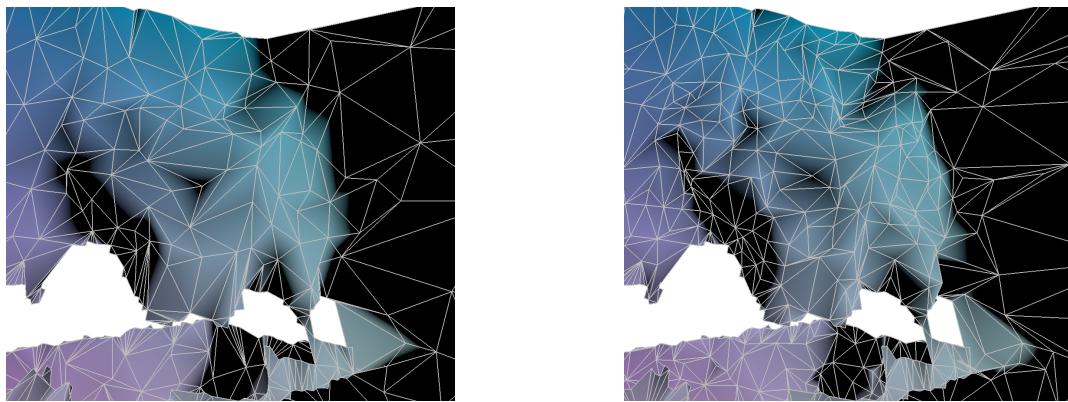


Abbildung 5.6: Auswirkung der Beschränkung des Kapitels 5.3.5 auf ein Mesh.

5.3.6 Minimaler Innenwinkel der Dreiecke

Auf glatten Flächen können sehr viele Vertices zusammengefasst werden, da der entstehende Fehler an diesen Stellen gering ist. Wird eine solche Fläche allerdings von uneinheitlichen Rändern begrenzt, dann sorgen dort vor allem die Erweiterungen aus den Abschnitten 5.3.3 und 5.3.4 für viele kleine Primitiven. Als Folge dessen können auch die Faces auf der Fläche nicht weiter zusammengefasst werden, wenn sie zum Teil den Rand berühren. Dadurch entstehen viele sehr spitze Dreiecke während der Dezimierung, welche in Meshes unerwünscht sind.

Um diesem Problem entgegenzuwirken, werden vor jeder Vereinigung von Vertex-Paaren die dadurch entstehenden Faces betrachtet. Falls eines der Faces einen Innenwinkel besitzt, der kleiner als 13° ist, wird der berechnete Fehler für das Kollabieren der entsprechenden Kante mit 10 multipliziert, um die Entstehung des spitzen Dreiecks zu bestrafen. Abbildung 5.7 zeigt eine Tischfläche, wie sie ohne und mit dieser Erweiterung nach einer Dezimierung aussieht.

5.3.7 Optimale Vertex-Position nach der Kontraktion einer Kante

Jegliche in dieser Arbeit genannten Dezimierungen gehen davon aus, dass ein Vertex einer Kante auf den anderen geschoben wird. Garland et. al. [4] schlagen eine alternative Herangehensweise vor, bei der die beiden Knoten vereint und an eine neue, der Errormetrik nach optimale Position verschoben werden.

Um den optimalen Punkt \bar{v} zu berechnen, kann die kombinierte Errormatrix K_v des betrachteten Vertex-Paars verwendet werden, wie sie in dem Abschnitt 5.1 vorgestellt wurde. Mithilfe der

5.3. ERWEITERUNGEN



Abbildung 5.7: Auswirkung der Beschränkung des Kapitels 5.3.6 auf ein Mesh.

quadratischen Gleichung 5.1.3 lässt sich so eine neue Vertex-Position in linearer Zeit bestimmen, die nach der verwendeten Errormetrik optimal ist.

Das Berechnen einer optimalen Position für kombinierte Vertices nach einer Dezimierung ist praktisch, um die verwendete Errormetrik gering zu halten und um ein optisch ansprechendes Ergebnis zu erzeugen. Ein großes Problem bei diesem Vorgehen ist allerdings, dass die Vertices dadurch nicht mehr an die Position der tatsächlich ursprünglich vom Tiefensensor gemessenen Werte gebunden sind und frei im Raum umpositioniert werden können. Vor allem am Rand des Meshes kann somit leicht neue Geometrie erfunden oder bekannte Geometrie abgeändert werden. Zwar könnten die Effekte eingeschränkt werden, indem beispielsweise diese Erweiterung am Rand des Meshes nicht verwendet wird, allerdings würden trotzdem innerhalb des Meshes neue und ideale Positionen nur anhand der Errormetrik bestimmt werden, was nicht für eine sichere Darstellung der Szene sorgt. Entsprechend wird die genannte Erweiterung innerhalb dieser Arbeit nicht verwendet.

KAPITEL 5. REDUZIERUNG DES MESHES

Kapitel 6

Ergebnisse

In diesem Kapitel werden die Ergebnisse der Arbeit vorgestellt. Es werden verschiedene Kombinationen der Kapitel 3, 4 und 5 miteinander verglichen. Dabei wird auf die vier Kriterien Reduktion, Ausführungszeit, Vollständigkeit und Hausdorff Distanz eingegangen.

Zur Bewertung der Methoden wird ein Strom von Bildern einer Kamera mit Tiefensensor verwendet. Dieser Strom besteht aus 3.518 Bildern einer einzelnen Szene, welche von der Kamera aufgenommen werden. Währenddessen wird die Kamera bewegt, um möglichst diverse Eingaben zu erhalten. Jegliche gezeigte Bilder innerhalb dieser Arbeit stammen aus diesem Bilder-Strom. Die genutzten Bilder haben eine Größe von 320×240 Pixeln. Abbildung 6.1 zeigt zwei solche Eingabebilder, wie sie mit den vorgestellten Verfahren bearbeitet werden.

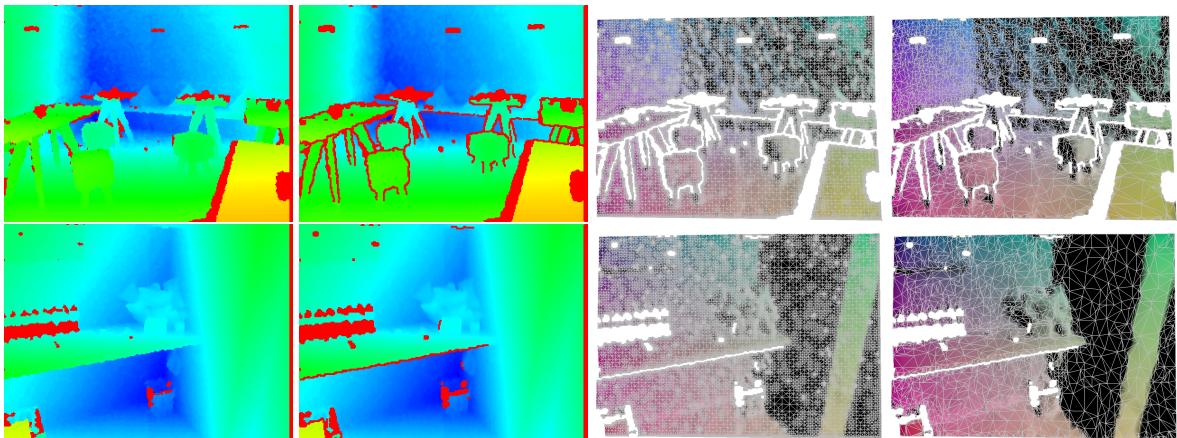


Abbildung 6.1: Beispiele für den Verlauf der Pipeline.

6.1 Verglichene Meshes

In den folgenden Schritten werden die vier Kombinationen von naiver und RQT-Triangulierung zusammen mit der vorgestellten Dezimierung und einer Vergleichsimplementierung von Open-Mesh [1] verglichen. Letztere verwendet die gleiche grundlegende Metrik, allerdings enthält sie nicht alle der vorgestellten Erweiterungen aus Abschnitt 5.3. Zusätzlich wird eine Triangulierung via RQT

KAPITEL 6. ERGEBNISSE

ohne folgende Dezimierung betrachtet. Für die ersten vier Kombinationen wird ein fester Threshold bei der Erstellung des zugehörigen RQT gewählt und die Reduktionsfaktoren der Dezimierung werden linear von 10% auf 90% in Schritten von 10% erhöht. Für den einzelnen RQT ohne folgende Verarbeitungsschritte wird der Threshold linear erhöht, bis sich die entstehende Struktur kaum mehr verändert.

Insgesamt ergeben sich also sechs verschiedene Arten von Meshes pro Eingabebild, die miteinander verglichen werden:

1. Die naive Triangulierung nach Abschnitt 4.1, welche aus dem komplett vorverarbeiteten Eingabebild erstellt wird. Dieses Mesh dient als "fehlerfreier" Vergleich für alle weiteren Meshes, da es alle Tiefendaten des Tiefensensors enthält, die auch als Eingabe für die restlichen Triangulierungen verwendet werden.
2. Das naive Mesh, welches durch OpenMesh reduziert wurde.
3. Das naive Mesh, welches wie in Kapitel 5 reduziert wurde.
4. Die Quadtree-basierte Triangulierung der Eingabedaten nach Abschnitt 4.2.2.
5. Die Quadtree-basierte Triangulierung, welche durch OpenMesh reduziert wurde.
6. Die Quadtree-basierte Triangulierung, welche wie in Kapitel 5 reduziert wurde.

6.2. UNVOLLSTÄNDIGE DEZIMIERUNGEN DER RQTs

6.2 Unvollständige Dezimierungen der RQTs

Der Fakt, dass die in dieser Arbeit vorgestellte Dezimierung bestimmte Kontraktionen komplett verbietet, sorgt dafür, dass nicht immer um die gewünschten Reduktionsfaktoren reduziert werden kann. Im Rahmen dieser Auswertung passiert dies einzig bei der Dezimierung von Daten, welche zuvor bereits mithilfe eines RQT trianguliert wurden.

Für jegliche Auswertungen, welche sich nicht um den RQT selbst drehen, wird ein RQT mit einem Threshold von 0,0015 verwendet, welcher in etwa für einen Reduzierungsfaktor von 73% sorgt. Abbildung 6.2 zeigt, wie viele der betrachteten Meshes mit diesem Threshold bei unterschiedlichen Reduzierungsfaktoren der folgenden Dezimierungen nicht ausreichend dezimiert werden können. Dies hat teilweise Auswirkungen auf die weiteren Ergebnisse bei hohen Reduzierungsfaktoren.

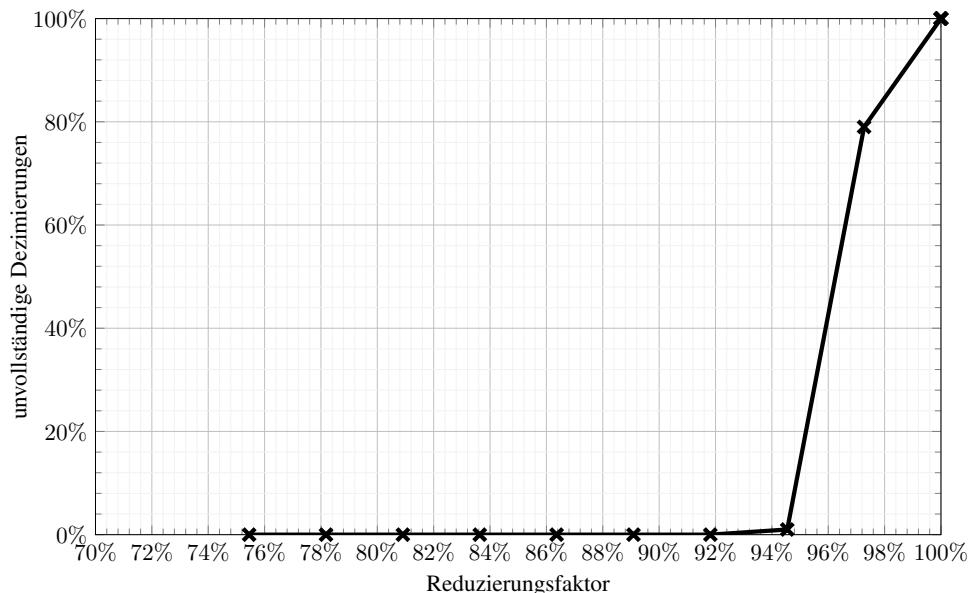


Abbildung 6.2: Der Anteil der Meshes, welcher nach einer RQT-Triangulierung nicht durch die vorgestellte Dezimierung um den gewünschten Faktor verringert werden kann.

6.3 Vergleich der Ausführungszeit

6.3.1 Konstante Verfahren

Als konstante Verfahren werden jene gesehen, deren Ausführungszeit von keinem Threshold oder Reduzierungsfaktor abhängen. Durch unterschiedliche Eingabedaten ergeben sich dennoch verschiedene Laufzeiten. Die Verfahren sind in Tabelle 6.1 angeführt. Dabei werden die Laufzeiten für jene Eingabedaten aufgelistet, welche innerhalb der gesamten Auswertung verwendet werden. Entsprechend werden Bilder der Größe 320×240 behandelt und der betrachtete Abhängigkeitsgraph hat eine Größe von 513×513 .

Tabelle 6.1: Die Durchschnittsdauer der konstanten Verfahren

Verfahren	Ausführungszeit (ms)
Vorverarbeitung	$11,5943 \pm 3,44808$
naive Triangulierung	$37,357 \pm 7,25665$
Erstellung des RQT Abhängigkeitsgraphen	$10,7265 \pm 0,783901$

Die gemessene Zeit für die Vorverarbeitung dauert im Durchschnitt circa 12 Millisekunden, wodurch sie sich auch für Echtzeitanwendungen gut eignet. Die Abweichungen hängen vor allem davon ab, wie viele Tiefenwerte im Eingabebild bereits vom Tiefensor nicht gemessen werden konnten und somit nicht bearbeitet werden müssen.

Die gemessene Zeit für die naive Triangulierung dient vor allem als Vergleichswert und ist ebenfalls abhängig von der Zahl der zu bearbeitenden Tiefenwerte.

Ein RQT-Abhängigkeitsgraph kann zwar manuell erstellt werden, allerdings ist die automatische Generierung vorzuziehen. Dies liegt einerseits an der geringen Laufzeit und zudem muss nur ein einzelner Graph für eine beliebige Menge an Eingabebildern derselben Größe erstellt werden. Für die Auswertung werden 4.000 Abhängigkeitsgraphen der gleichen Größe erstellt. Da das Vorgehen immer gleich ist und nicht von den Eingabedaten abhängt, ist die benötigte Zeit für die Erstellung des Abhängigkeitsgraphen stabil und dauert in etwa 11 Millisekunden. Zusätzlich ist zu erwähnen, dass die Erstellung einen Aufwand von $O(n^2)$ hat, wobei n der Breite beziehungsweise Höhe des zu erstellenden Graphen entspricht.

6.3.2 Variable Verfahren

Variable Verfahren sind alle, die von einem Threshold oder Reduzierungsfaktor abhängen. Das entspricht allen Kombinationen aus Triangulierung und Dezimierung und zusätzlich der alleinigen Triangulierung via eines RQTs. Abbildung 6.3 bildet die Laufzeiten dieser Verfahren in Kombination mit ihren nötigen konstanten Verfahren ab. Dabei wird die Zeit für das naive Triangulieren aus Tabelle 6.1 verwendet, außerdem wird die Zeit der Vorverarbeitung auf die Verfahren addiert. Die Erstellung des RQT-Abhängigkeitsgraphen wird jedoch nicht hinzugefügt, da dieser wieder verwendet oder sogar schon im Voraus erstellt werden kann.

Der erste Graph in Abbildung 6.3 zeigt dabei alle Kombinationen und die RQT-Triangulierung, der zweite Graph zeigt ein eingeschränktes Datenset, um den Bereich zwischen 0 und 600 Millisekunden übersichtlicher darzustellen.

6.3. VERGLEICH DER AUSFÜHRUNGSZEIT

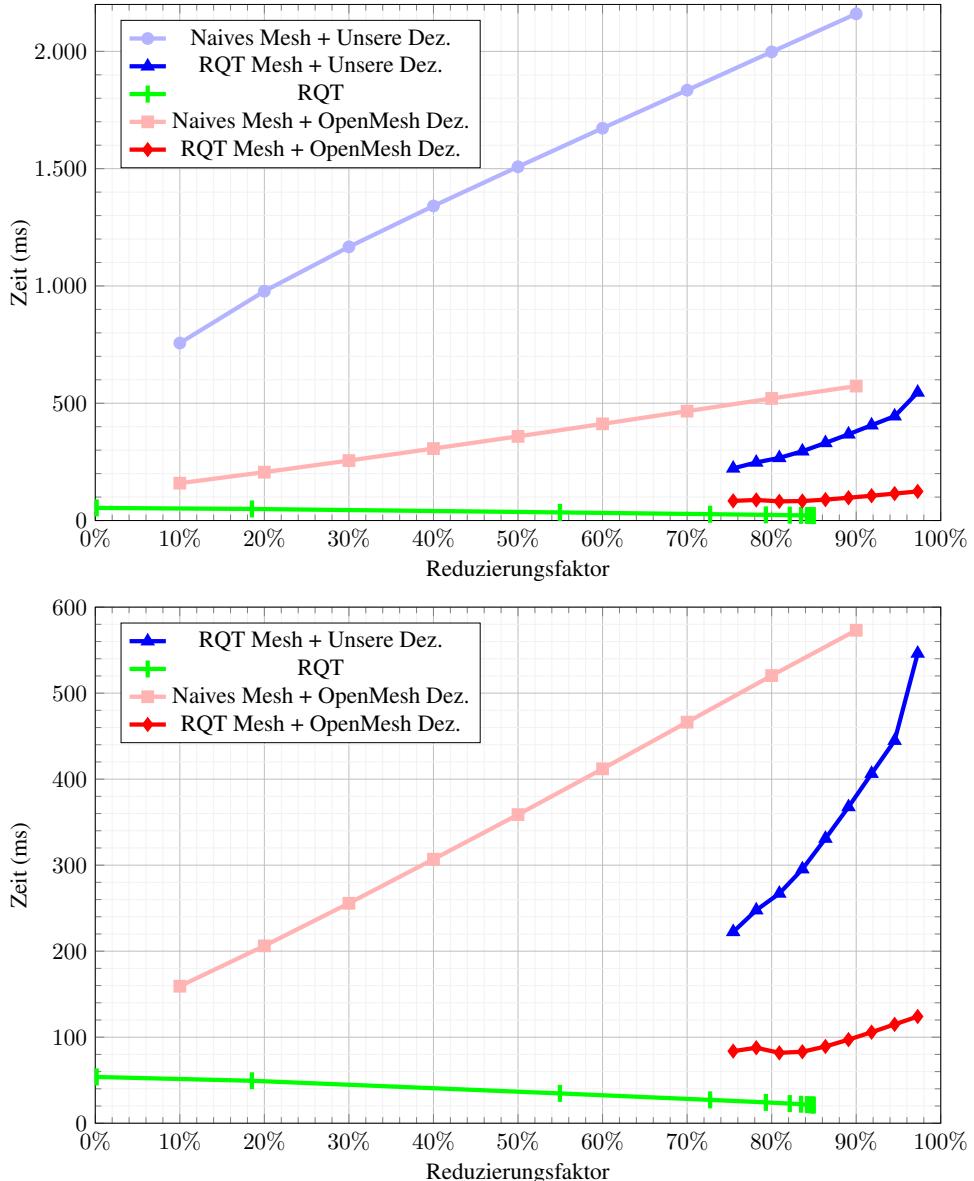


Abbildung 6.3: Die Auswertungen für die Ausführungszeit.

Die Messungen der Dauer von RQT-basierten Triangulierungen zeigt, dass diese bei einer niedrigen Reduktion ähnlich schnell wie das naive Vorgehen in Kombination mit der Vorverarbeitung sind. Mit einem steigenden Reduktionsfaktor wird der RQT jedoch schneller. An sich ist dieses Vorgehen aufwändiger als das naive, da für jedes Pixel des Bildes Vergleiche gemacht werden müssen. Werden allerdings genug Vertices dadurch eingespart, wird der RQT vor allem dadurch schneller, dass weniger Vertices bei der Erstellung des Meshes verarbeitet werden müssen.

An der Ausführungszeit der vorgestellten Dezimierung ist ein Nachteil im Vergleich zu OpenMesh zu erkennen: Werden sie auf das gleiche Mesh angewandt, ist die vorgestellte Dezimierung wesentlich langsamer als die Vergleichsimplementierung von OpenMesh. Der Grund ist, dass die vorgestellte

KAPITEL 6. ERGEBNISSE

Dezimierung wesentlich mehr Vergleiche und Berechnungen für jede einzelne potentielle Kontraktion durchführt.

Mithilfe einer Triangulierung durch einen RQT lässt sich dieses Defizit einschränken: Wird die vorgestellte Dezimierung auf ein RQT-Mesh angewandt, ist die Laufzeit geringer als die von OpenMesh auf einem naiven Mesh. Allerdings wird die Dezimierung von OpenMesh gleichermaßen beschleunigt, wenn sie auf einem RQT-Mesh verwendet wird.

6.4. VOLLSTÄNDIGKEIT

6.4 Vollständigkeit

6.4.1 Definition der Metrik

Eine weitere Metrik wird eingeführt, um die Vollständigkeit eines Meshes im Vergleich zum naiven Mesh zu messen. Dies bedeutet, dass die vom naiven Mesh dargestellte Geometrie mit der vom Vergleichsmesh abgeglichen wird. Wenn beispielsweise Löcher im naiven Mesh während der Dezimierung entfernt werden, entsteht an dieser Stelle neue Geometrie, die nicht in den tatsächlichen Eingabedaten enthalten ist. Gleichermaßen kann ein Teil des Randes vom naiven Mesh während einer Dezimierung entfernt werden. Dadurch geht ursprünglich dargestellte Geometrie aus dem naiven Mesh verloren.

Um dies zu erreichen, werden Strahlen durch jeden Pixel des Eingabebildes auf die zu vergleichenden Meshes geschossen. Pro Strahl wird vermerkt, ob dieser das naive Mesh, das Vergleichsmesh oder beide getroffen hat. Treffen insgesamt mehr Strahlen das naive Mesh, wurde Geometrie verloren, umgekehrt wurde Geometrie erfunden. Das optimale Ergebnis wäre also, dass jegliche Strahlen, die das erste Mesh treffen, ebenfalls das zweite treffen.

Diese Metrik eignet sich, um erfundene Geometrie aus der Sicht des aufnehmenden Tiefensensors zu finden, da von dessen Position aus die Strahlen geschossen werden. Wird jedoch Geometrie erstellt, die aus seiner Sicht vor oder hinter bereits existenter Geometrie liegt, wird diese nicht erkannt und bewertet.

Als Ergebnis erhält man pro Mesh die Zahl der Strahlen, welche jeweils das naive (h_{naiv}), das Vergleichs- (h_{comp}) und beide Meshes (h_{both}) treffen. Im Folgenden werden diese Werte zu h_n und h_c zusammengefasst. h_n steht für den Prozentsatz an ursprünglicher Geometrie, die erhalten geblieben ist. h_c zeigt die Menge der erfundenen Geometrie an. 1 ist dabei der optimale Wert für beide Terme. Sie berechnen sich folgendermaßen:

$$h_n = \frac{h_{both}}{h_{naiv}}, h_c = \frac{h_{comp}}{h_{both}}$$

6.4.2 Vollständigkeit der Verfahren

Die Abbildungen 6.4 und 6.5 zeigen die durchschnittliche Vollständigkeit bezüglich h_n und h_c , mit der alle bearbeiteten Meshes ihre jeweiligen naiven Meshes darstellen.

Für die reine Triangulierung eines Meshes via RQT ergibt sich, dass alle Einstellungen des Thresholds das gleiche Ergebnis für sowohl h_n als auch h_c erzeugen. Aus $h_c = 1$ lässt sich schließen, dass keine Geometrie aus Sicht der Kamera hinzu erfunden wird. An h_n ist zu erkennen, dass jedoch Geometrie im Vergleich zur naiven Triangulierung verworfen wird. Die Stellen, an denen Geometrie verloren geht, liegen an Rändern, welche durch viele kleine Primitiven dargestellt werden müssen. An solchen Rändern kann es relevant sein, wie die Dreiecke gelegt werden. Der RQT ist jedoch an einen bestimmten Aufbau gebunden und kann dadurch manchmal nur weniger Pixel des Bildes abdecken als die naive Triangulierung. Die beiden Werte sind konstant, da jegliche verwendeten Unterteilungsregeln am Rand des Bildes und an verworfenen Pixeln im Bild unabhängig von dem gewählten Threshold immer das betroffene Quad teilen. Somit ergibt sich an allen Randstellen, welche nicht perfekt am Rand eines Quads anliegen, immer die gleiche Triangulierung und dadurch

KAPITEL 6. ERGEBNISSE

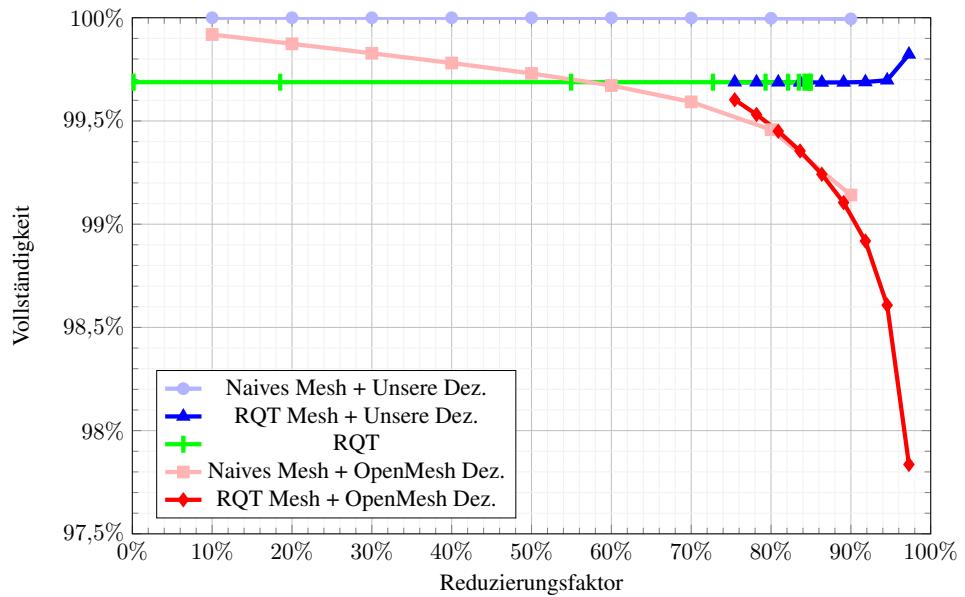


Abbildung 6.4: Die Auswertungen für h_n .

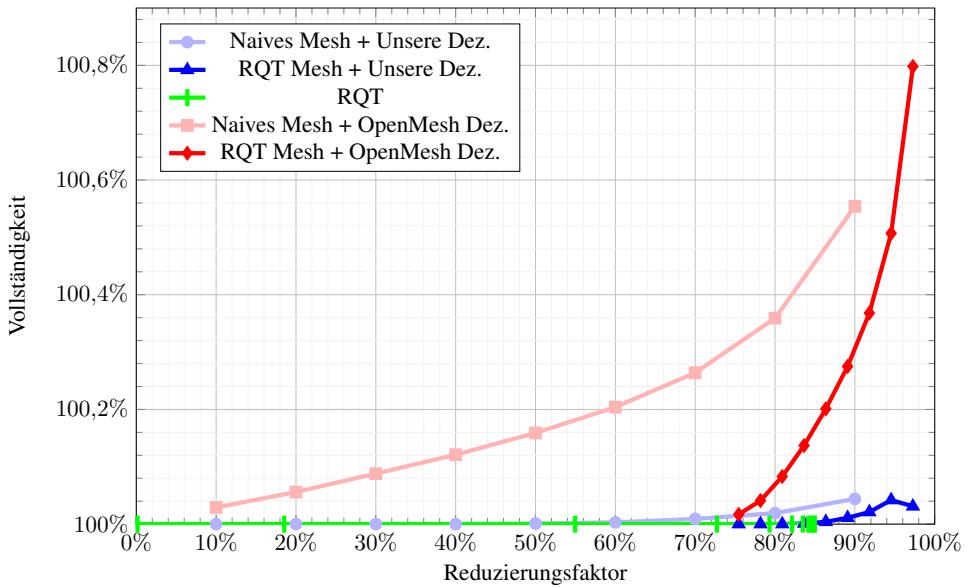


Abbildung 6.5: Die Auswertungen für h_c .

6.5. HAUSDORFF DISTANZ

eine gleichbleibende Vollständigkeit.

Werden jene Meshes betrachtet, welche durch ein Dezimieren von naiv triangulierten Meshes entstehen, kann für h_n und h_c beobachtet werden, dass die vorgestellte Dezimierung näher an dem optimalen Wert von 100% bleibt und selbst bei einer Reduktion der Vertices um 90% eine wesentlich geringere Abweichung aufweist als OpenMesh. Dies ist auf die beiden Verbote für Dezimierungen von Vertex-Paaren am Rand von Meshes zurückzuführen. Auch OpenMesh nutzt ein Verbot wie aus Abschnitt 5.3.3, wodurch Vertices am Rand des Meshes nur auf andere Vertices am Rand kollabiert werden dürfen. Das weitere Verbot aus Abschnitt 5.3.4, welches nur die Kontraktionen am Rand des Meshes erlaubt, welche ausschließlich zueinander parallele Rand-Kanten beeinflussen, wird von OpenMesh jedoch nicht verwendet. Dies sorgt dafür, dass OpenMesh Ecken des Meshes abrundet und dadurch die Vollständigkeit verschlechtert.

Der Grund dafür, dass sich die Vollständigkeit der vorgestellten Dezimierung dennoch bei hohen Reduktionsfaktoren verschlechtert, liegt darin, dass das genannte Verbot für nicht-parallele Kanten einen gewissen Threshold nutzt, um entsprechende Kanten als parallel einzustufen. Daher können weiterhin Kanten dezimiert werden, die nicht vollständig, sondern nur beinahe parallel sind.

Ein Blick auf die RQT-Meshes, welche dezimiert werden, zeigt, dass die vorgestellte Dezimierung für h_n und h_c näher an dem optimalen Wert von 100% bleibt und selbst bei stärkerer Reduktion kaum abweicht. Die Vollständigkeit des von OpenMesh reduzierten Meshes degradiert schneller als auf den naiven Meshes. Dies liegt daran, dass nach der verwendeten Triangulierung via RQT nur noch etwa 25% der ursprünglichen Vertices vorhanden sind. Durch die Regeln des RQT decken diese das Mesh noch gut ab, OpenMesh kollabiert jedoch für die Abdeckung relevante Vertices schneller, da die gesamte Auswahl an Vertices geringer ist. Insgesamt nähern sich die Graphen für h_n und h_c deshalb an die der OpenMesh-Dezimierung auf dem naiven Mesh an.

Um die spontane Verbesserung der vorgestellten Dezimierung bei einem Reduzierungsfaktor von etwa 97% zu begründen, wird auf Abbildung 6.2 verwiesen: Bei der Dezimierung von Meshes, welche bereits durch einen RQT trianguliert wurden, können durch die vorgestellte Dezimierung nicht immer die gewünschte Zahl an Vertices entfernt werden. Somit ergibt sich ab diesem Reduzierungsfaktor ein wesentlich kleinerer Datensatz, welcher entsprechend stärker von Einzelfällen beeinflusst wird.

6.5 Hausdorff Distanz

6.5.1 Definition der Metrik

Um die Abweichungen der Meshes untereinander zu messen, wird eine der Hausdorff Distanz ähnliche Metrik verwendet. Die Hausdorff Distanz zweier Meshes ist das Maximum aller kleinstmöglichen Abstände, die zwischen einem frei gewählten Punkt auf einem beliebigen der beiden Meshes und dem anderen Mesh liegen können. Ein Beispiel für diese Abstände wird in Abbildung 6.6 dargestellt.

Sind also die zwei Meshes M und N gegeben, kann deren Hausdorff Distanz $H(M, N)$ folgendermaßen berechnet werden:

$$H(M, N) = \max \left\{ \sup_{x \in M} \inf_{y \in N} d(x, y); \sup_{y \in N} \inf_{x \in M} d(x, y) \right\}$$

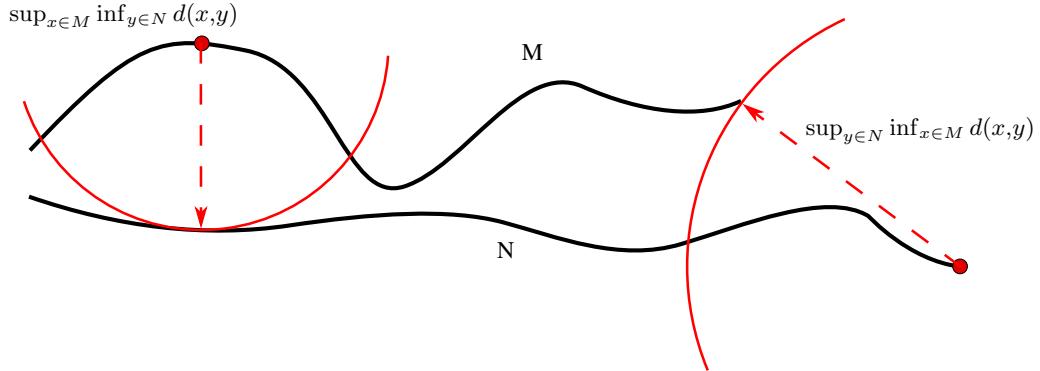


Abbildung 6.6: Die Hausdorff Distanz zwischen zwei Meshes.

$d(x, y)$ steht hierbei für die Distanz zwischen den Punkten x und y . In der verwendeten Abwandlung der Hausdorff Distanz werden als Approximation nur die Abstände zwischen Vertices und Faces verwendet.

Da sowohl die vorgestellte Dezimierung als auch die verwendete Version von OpenMesh immer einen Vertex komplett auf seinen Partner dezimieren, teilen sich alle behandelten Meshes ihre übrig gebliebenen Vertices weiterhin mit ihrem jeweiligen naiv triangulierten Mesh. Die minimale Distanz von jedem Vertex des reduzierten Meshes zum naiven Mesh ist deshalb 0 und es werden nur die Distanzen von Vertices des naiven Meshes zum reduzierten Mesh gemessen. Um eine bessere Einschätzung des gesamten Unterschieds der Meshes zu erhalten, wird zudem nicht das Maximum, sondern der Durchschnitt von allen minimalen Abständen genommen. Ein weiterer Zusatz in der Abwandlung der Hausdorff Distanz besteht darin, dass jede gemessene Distanz durch die Entfernung $depth_{x_v}$ des jeweils betrachteten Vertex zur Tiefenkamera geteilt wird, die das zum Mesh zugehörige Bild aufgenommen hat. Dies setzt den entstandenen Fehler mit der Distanz zur Kamera in Relation. Der Grund dafür ist das Rauschen in den Tiefendaten, wegen welchem in der Distanz eine stärkere Abweichung von den aufgenommenen Daten zugelassen wird. Des Weiteren werden $d(x, y)$ und $depth_{x_v}$ quadriert und von der gesamten Gleichung das quadratische Mittel gemessen, da einzelne größere Abstände stärker gewichtet sind als mehrere kleine. Somit ergibt sich die folgende Abänderung $H'(M, N')$ der Hausdorff Distanz:

$$H'(M, N') = \sqrt{\frac{\sum_{x_v \in M} \inf_{y_f \in N'} \frac{d(x_v, y_f)^2}{depth_{x_v}^2}}{n_{N'}}}$$

Dabei ist N' das naive Mesh, welches immer zum Vergleich verwendet wird. $n_{N'}$ ist die Zahl der Vertices im naiven Mesh, x_v sind Vertices des Vergleichsmeshes M und y_f sind Faces des naiven Meshes.

Für die genannten Berechnungen muss die Distanz zwischen jedem Vertex des naiven mit jedem Dreieck des Vergleichsmeshes ausgerechnet werden. Dies wird optimiert, indem zu jedem Vertex nur noch in einer kleinen Screen-Space Nachbarschaft nach Faces des anderen Meshes gesucht wird. Anstelle von allen Faces des Vergleichsmeshes werden nur die dort gefundenen Faces verwendet und deren Distanzen zum aktuellen Vertex verglichen.

6.5. HAUSDORFF DISTANZ

Letztendlich erhält man pro Mesh ein quadratisches Mittel der kürzesten Distanzen der naiven Vertices zum Vergleichsmesh, welche in Relation zur Tiefe gesetzt sind. Die in dieser Auswertung gezeigte Ausgabe entspricht wiederum dem Durchschnitt dieser Werte über alle 3.518 Eingabebilder.

6.5.2 Entstehende Hausdorff Distanz durch die Verfahren

Die Ergebnisse der Messungen sind in Abbildung 6.7 aufgeführt.

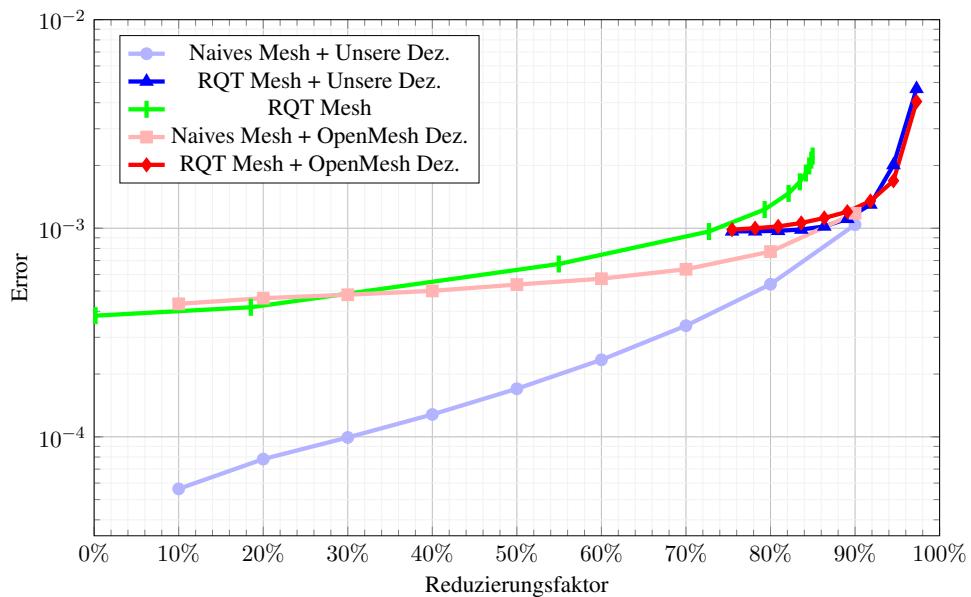


Abbildung 6.7: Die Auswertungen für die Hausdorff Distanz.

Die Hausdorff Distanz der RQT-Triangulierung erreicht selbst bei einer minimalen Dezimierung eine Abweichung zum naiv triangulierten Mesh. In diesem Fall wird das RQT-Mesh gleich dem naiven Mesh komplett in Quads der Größe 2×2 Pixel unterteilt. Um diese Quads zu triangulieren, gibt es im naiven Mesh zwei Möglichkeiten, von denen jene gewählt wird, welche die kürzere Diagonale erzeugt. Der RQT ist allerdings an einen bestimmten Aufbau und die daraus folgende Triangulierung gebunden. Ist ein einzelnes der vier Pixel in einem Quad verworfen, kann es zudem vorkommen, dass der RQT das gesamte Quad verworfen muss, wohingegen die naive Triangulierung die restlichen drei Vertices triangulieren kann.

Ein Blick auf die Meshes, welche durch ein Dezimieren von naiv triangulierten Meshes entstehen, zeigt, dass die vorgestellte Dezimierung einen geringeren Error als die Variante von OpenMesh erzielt. Der Grund dafür sind die zusätzlichen Erweiterungen der vorgestellten Dezimierung, welche OpenMesh nicht beachtet.

Die Meshes, welche durch RQTs trianguliert und danach dezimiert werden, zeigen einen zueinander sehr ähnlichen Verlauf im Fehler. Der Grund für den zu Beginn flachen Verlauf ist, dass beide Dezimierungen zuerst vor allem die Strukturen reduzieren, welche durch den RQT im Mesh erstellt wurden und überflüssig sind. Dies sind vor allem die hohen Vertex-Vorkommen um jene Stellen, an welchen das vorverarbeitete Tiefenbild fehlende Messwerte enthält. Mit einem erhöhten Reduk-

KAPITEL 6. ERGEBNISSE

tionsfaktor beginnt der Fehler immer schneller anzusteigen. Der Grund hierfür und für den weiterhin gleichen Verlauf der Graphen ist, dass der Reduktionsfaktor in diesem Bereich bei etwa 95% liegt. Dieser Wert ist so hoch, dass keine der Dezimierungen mehr einen gering bleibenden Fehler erzeugen kann.

Kapitel 7

Zusammenfassung und Ausblick

Innerhalb dieser Arbeit wird ein effizientes Verfahren vorgestellt, um aus unvollständigen Tiefenkarten ein Dreiecksnetz mit geringem Fehler zu erstellen.

Zuerst werden die Tiefenkarten selbst bearbeitet, um Kanten zwischen unabhängigen Objekten zu entfernen. Diese bearbeiteten Bilder werden daraufhin mithilfe von Restricted Quadtrees trianguliert, um die Zahl der überflüssigen Vertices, welche von einer naiven Triangulierung erzeugt werden würden, zu verringern. Im Anschluss wird das erstellte Mesh zudem reduziert, um weitere Primitiven einzusparen, welche durch die Erstellungsregeln des Restricted Quadtrees im Rahmen der ursprünglichen Triangulierung verwendet werden müssen. Während dieser Dezimierung wird neben der regulär verwendeten quadratischen Errormetrik auf diverse weitere Regeln geachtet, welche unter verschiedenen Umständen den Error weiter verfeinern, um das Mesh originalgetreuer zu halten. Die gesamte Pipeline wird als Open Source Software bereitgestellt und ist in das Saiga Framework [12] integriert.

Die Triangulierung via RQTs sorgt für eine starke Verringerung der Primitiven im Vergleich zu einem naiven Ansatz. Der Threshold für den entstehenden Fehler ist außerdem verschieden wählbar, wovon auch der tatsächliche Reduzierungsgrad der Vertices abhängt. Die Ausführungszeit kann zudem kürzer als die der naiven Triangulierung sein, wenn genug Vertices eingespart werden.

Das Ergebnis nach Triangulierung und Dezimierung ist ein stark reduziertes Mesh, welches die Geometrie des bearbeiteten Bildes besser abdeckt und einen geringeren Error erzeugt als eine vergleichbare Dezimierung von OpenMesh. Diese Ergebnisse werden unter anderem durch das Verbot der Kontraktion bestimmter Vertex-Paare erreicht. Dies sorgt allerdings auch dafür, dass das resultierende Mesh nicht immer auf eine beliebige Anzahl an Vertices reduziert werden kann. Des Weiteren ist die Ausführungszeit der vorgestellten Dezimierung im Vergleich zur Implementierung von OpenMesh wesentlich erhöht. Dieser größere Zeitaufwand liegt daran, dass mehr Vergleiche vollzogen werden müssen, um den Fehler der Kontraktion eines Vertex-Paares zu bestimmen.

Die Kombination von RQT und Dezimierung sorgt insgesamt für eine gute Beschleunigung bei geringer Fehlererzeugung. Die Triangulierung via RQT verwirft eine erste Auswahl von Vertices mit simplen und somit weniger rechenaufwändigen Verfahren. Die Dezimierung läuft daraufhin wesentlich schneller als auf naiv triangulierten Meshes, da viel weniger potentielle Vertex-Kontraktionen bewertet werden müssen.

Zusammenfassend sorgt eine Kombination von RQT-basierter Triangulierung und Dezimierung via OpenMesh für das schnellste Ergebnis. Den geringsten Fehler erzeugt eine naive Triangulierung

KAPITEL 7. ZUSAMMENFASSUNG UND AUSBLICK

mit einer anschließenden Dezimierung, wie sie in dieser Arbeit vorgestellt wird. Wird ein RQT-basiertes Mesh mithilfe der vorgestellten Dezimierung reduziert, ist die resultierende Vollständigkeit besser als bei der Implementierung von OpenMesh. Allerdings kann der RQT bereits einen derartigen Fehler bezüglich der Hausdorff Distanz erzeugen, dass die vorgestellte Dezimierung und OpenMesh diesbezüglich ein ähnliches Ergebnis erzielen. In diesem Fall sorgt die vorgestellte Implementierung für eine bessere Vollständigkeit und die Version von OpenMesh für eine bessere Laufzeit.

Eine Möglichkeit zur potentiellen Verbesserung der Genauigkeit des Ergebnisses besteht in der Verwendung eines Neuronalen Netzes. Dieses könnte genutzt werden, um jeden Vertex nach der Dezimierung derart zu verschieben, dass die Hausdorff Distanz des gesamten Meshes im Bezug zur naiven Triangulierung verringert wird. Die entsprechende Ausgabe ist eine Translation. Als Eingabedaten für das Netz könnte eine Punktewolke aus der Umgebung des jeweiligen Vertex dienen, die mithilfe von Raytracing gefunden wird. Diese Punktewolke muss noch in den Ursprung geschoben und auf eine einheitliche Ebene rotiert werden, um das Problem zu verallgemeinern. Um Trainingsdaten für das Netz zu erhalten, kann für jeden betrachteten Vertex eines zu bearbeitenden Meshes der Punkt auf dem zugehörigen naiven Mesh gefunden werden, welcher die geringste Distanz zu ihm hat. Die Trainingsdaten können daraufhin die Translation vom betrachteten zum naiven Vertex als optimales Ergebnis nutzen.

Das Training des Netzes wäre zeitaufwändig, allerdings müssen zur Anwendung eines trainierten neuronalen Netzes nur noch Matrixmultiplikationen durchgeführt werden.

Literaturverzeichnis

- [1] Rheinisch-Westfälische Technische Hochschule Aachen. Openmesh. URL: <https://www.openmesh.org/>.
- [2] H. Borouchaki and P.J. Frey. Simplification of surface mesh using hausdorff envelope. *Computer Methods in Applied Mechanics and Engineering*, 194(48):4864 – 4884, 2005. Unstructured Mesh Generation. URL: <http://www.sciencedirect.com/science/article/pii/S0045782505000708>, doi:<https://doi.org/10.1016/j.cma.2004.11.016>.
- [3] Pascal Frey and Loic Marechal. Fast adaptive quadtree mesh generation. *Proceedings of the 7th International Meshing Roundtable*, 05 2000.
- [4] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. URL: <https://doi.org/10.1145/258734.258849>, doi:10.1145/258734.258849.
- [5] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (SIGGRAPH Asia Conference Proceedings)*, 37(6), November 2018. URL: <http://www-sop.inria.fr/reves/Basilic/2018/HPPFDB18>.
- [6] Muhammad Hussain, Yoshihiro Okada, and Koichi Niijima. Efficient and feature-preserving triangular mesh decimation. *Journal of WSCG*, 12:167–174, 01 2004.
- [7] Xinghua Liang, Mohamed S. Ebeida, and Yongjie Zhang. Guaranteed-quality all-quadrilateral mesh generation with feature preservation. In Brett W. Clark, editor, *Proceedings of the 18th International Meshing Roundtable*, pages 45–63, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [8] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 109–118, New York, NY, USA, 1996. ACM. URL: <http://doi.acm.org/10.1145/237170.237217>, doi:10.1145/237170.237217.

- [9] R. Mur-Artal and J. D. Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, Oct 2017. doi:10.1109/TRO.2017.2705103.
- [10] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings Visualization '98 (Cat. No.98CB36276)*, pages 19–26, Oct 1998. doi:10.1109/VISUAL.1998.745280.
- [11] Mariano Pérez, Ricardo Olanda, and Marcos Fernández. Visualization of large terrain using non-restricted quadtree triangulations. volume 3044, pages 671–681, 05 2004. doi:10.1007/978-3-540-24709-8_71.
- [12] D. Rückert and Others. Saiga. URL: <https://github.com/darglein/saiga>.
- [13] Darius Rückert, Matthias Innmann, and Marc Stamminger. FragmentFusion: A Light-weight SLAM Pipeline for Dense Reconstruction. In *ISMAR 2019*, 2019.
- [14] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. In Bianca Falcidieno and Tosiyasu L. Kunii, editors, *Modeling in Computer Graphics*, pages 455–465, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.
- [15] Ron Sivan and Hannan Samet. Algorithms for constructing quadtree surface maps. 1992.
- [16] Keisuke Tateno, Federico Tombari, Iro Laina, and Nassir Navab. CNN-SLAM: real-time dense monocular SLAM with learned depth prediction. *CoRR*, abs/1704.03489, 2017. URL: <http://arxiv.org/abs/1704.03489>, arXiv:1704.03489.
- [17] Brian Von Herzen and Alan H. Barr. Accurate triangulations of deformed, intersecting surfaces. *SIGGRAPH Comput. Graph.*, 21(4):103–110, August 1987. URL: <http://doi.acm.org/10.1145/37402.37415>, doi:10.1145/37402.37415.
- [18] Jian Wu, Yuanfeng Yang, Sheng-Rong Gong, and Zhiming Cui. A new quadtree-based terrain lod algorithm. *JSW*, 5:769–776, 07 2010. doi:10.4304/jsw.5.7.769–776.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass die Arbeit veröffentlicht wird und dass in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Graphische Datenverarbeitung, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, den 13. Dezember 2019

(Simon Mederer)

Curriculum Vitae

Persönliche Daten:

Name: Simon Mederer
E-Mail: simon.mederer@fau.de
Adresse: Zweifelsheimer Straße 41a
91074 Herzogenaurach
Deutschland
Geburtsdatum: 11. Mai 1996
Nationalität: Deutsch

Ausbildung:

2017 B.Sc. Informatik,
Friedrich-Alexander-Universität Erlangen-Nürnberg
2014 Abitur,
Gymnasium Herzogenaurach, Deutschland