Technische Universit"at Dresden

Fakult"at Informatik
Institut f"ur Software- und Multimediatechnik
Professur f"ur Computergraphik und Visualisierung
Prof. Dr. Stefan Gumhold

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker

Optimierung und Übertragung von Tiefengeometrie für Remote-Visualisierung

Josef Schulz
(Geboren am 20. Oktober 1989 in Naumburg (Saale), Mat.-Nr.: 3658867)

Betreuer: Dr. rer. nat. Sebastian Grottel

Dresden, 7. Dezember 2016

Aufgabenstellung

In Big-Data-Szenarien in der Visualisierung spielt der Ansatz der Remote-Visualisierung eine zunehmende Rolle. Moderne Netzwerktechnologien bieten große Datenübertragungsraten und niedrige Latenzzeiten. Für die interaktive Visualisierung sind aber selbst kleinste Latenzzeiten problematisch. Um diese vor dem Benutzer maskieren zu können, kann eine Extrapolation der Darstellung durchgeführt. Diese Berechnungen erfordern zusätzlich zum normalen Farbbild weitere Daten, beispielsweise ein Tiefenbild und die Daten der verwendeten Kameraeinstellung. Für die Darstellungsextrapolation werden Farb- und Tiefenbild zusammen interpretiert, beispielsweise als Punktwolke oder Höhenfeldgeometrie. Im Rahmen dieser Arbeit soll untersucht werden, wie die Darstellung mittels Höhenfeldgeometrie optimiert werden kann. Ansätze sind hierfür Algorithmen aus der Netzvereinfachung. Zu erwarten sind sowohl harte Kanten als auch glatte Verläufe der Tiefenwerte, welche sich in der Netzgeometrie durch adaptive Vernetzung mit reduziertem Datenaufwand darstellen lassen.

Dem Szenario der Web-basierten Remote-Visualisierung folgend soll der Web-Browser als Klient-Komponente eingesetzt werden. Die einzusetzenden Technologien sind HTML5, Javascript, WebGL und WebSockets. Entsprechende Javascript-Bibliotheken sollen genutzt werden um die Qualität und Wartbarkeit des Quellcodes zu steigern. Für die Server-Komponente darf die Technologie vom Bearbeiter frei gewählt werden.

Zu Beginn der Arbeit wird eine Literatur-Recherche zu Web-basierter Visualisierung und Remote-Visualisierung erfolgen. Schwerpunkte sind hierbei die Bild-Extrapolation, Vernetzung und Rekonstruktion auf Basis von Tiefenbildern und die Netzoptimierung und -Vereinfachung. Im Anschluss an die Literaturrecherche wird ein Konzept für die Implementierung mit dem Betreuer abgesprochen und anschließend als prototypische Software umgesetzt. Folgendes Szenario dient als Grundlage für dieses Konzept:

Als Eingabedaten stehen mehrere Datensätze aus unterschiedlichen Szenarien der wissenschaftlichen Visualisierung zur Verfügung. Für jeden Datensatz sind mehrere Tripel aus Farbbild, Tiefenbild und Kamera-Parameter gegeben. Die Serverkomponente bereitet einen Datensatz auf und bietet ihn dem Klienten an. Diese Aufbereitung ist vor allem die Generierung einer optimierten Tiefennetzgeometrie aus den Tiefenbilddaten. Der Klient fordert Farbbilder, Kameraeinstellungen und Tiefengeometrie von Tripel-Paaren an. Konzeptuell wird ein Tripel als aktueller Zustand und das zweite Tripel als Ground-Truth einer Bildextrapolation verstanden. Diese können daher auch in dieser Reihenfolge angefordert werden. Die Tripel werden zwischen Klient und Server direkt per Sockets/WebSockets übertragen. Die Daten des ersten Tripels werden anschließend genutzt um dessen Farbbild in die Ansicht des zweiten Tripels extrapoliert. Hierbei werden vom zweiten Tripel nur die Kameraeinstellung genutzt. Diese Extrapolation wird Klient-seitig in WebGL implementiert damit alle Berechnungen auf der GPU ausgeführt werden. Anschließend wird das extrapolierte Bild mit dem originalen Ground-Truth-Farbbild aus dem zweiten Tripel verglichen um die Qualität der Extrapolation zu bewerten, z.B. durch SSIM.

Die umgesetzte Lösung wird ausführlich evaluiert. Zentraler Wert ist hierbei die Bildqualität nach der Extrapolation abhängig vom Winkelunterschied zwischen den Kameraeinstellungen und den Parametern der Vereinfachung der Tiefennetzgeometrie. Hierfür werden Tripel-Paare aus den Datensätzen und Variationen der Parameter der Algorithmen systematisch und automatisiert vermessen. Untersuchungen zum Laufzeitverhalten der Netzoptimierung im Server und der Bildextrapolation im Klienten sind optional durchzuführen.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Arbeit zum Thema:

Optimierung und Übertragung von Tiefengeometrie für Remote-Visualisierung

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 7. Dezember 2016

Josef Schulz

Kurzfassung

Zusammenfassung Text Deutsch

Abstract

abstract text english

Inhaltsverzeichnis

1 Einleitung

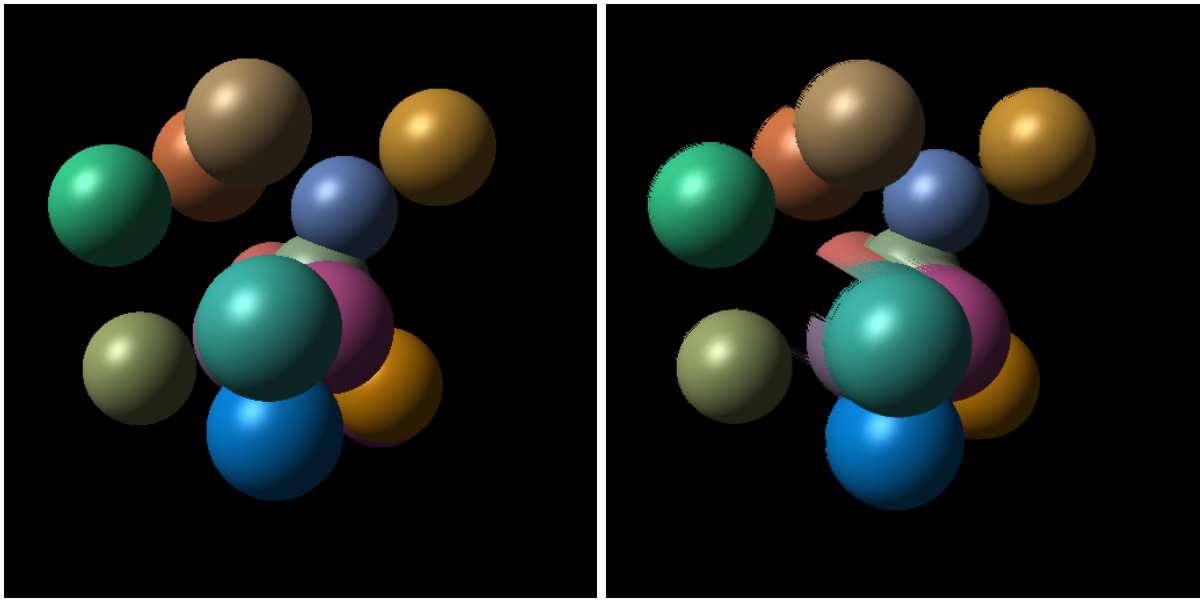
Bei der Remote-Visualisierung, wird die Bildsynthese und die eigentliche Darstellung voneinander getrennt. Der Server-Prozess erzeugt und kodiert jedes Bild zu einem kompakten Datenpaket, welches an den Client-Prozess gesendet wird. Der Client empfängt und dekodiert das Datenpaket und gibt das Bild auf einem Bildschirm aus.

Remote-Visualisierung, ist ein insbesondere für mobile Endgeräte interessantes Konzept, weil es die Visualisierung von komplexen Szenen auch auf leistungsarmen Geräten ermöglicht. Neben Computerspielen, ist die wissenschaftliche Visualisierung ein wichtiges Anwendungsgebiet, da Datensätze Größenordnungen erreichen können, die den Speicher herkömmlicher Desktops, Laptops, Smartphones etc. bei weitem übersteigen. Auch wenn ausreichend Speicher zur Verfügung steht, kann die Übertragung dieser Daten viel Zeit in Anspruch nehmen.

Mit Hilfe der Remote-Visualisierung ist es möglich, dass der Server die Bildsynthese übernimmt und nicht der komplette Datensatz übertragen werden muss. Ein weiterer Vorteil der mit leistungsstarken Serversystemen einhergeht, ist der, dass sich komplexe Visualisierungs- und Beleuchtungsmethoden verwenden lassen, die mit normalen Endgeräten nicht zu realisieren sind.

Die Latenz bezeichnet in der Netzwerktechnik die Übertragungszeit von einem zum anderen Gerät. Diese ist in modernen Netzwerken gering, für die interaktive Visualisierung allerdings immer noch zu groß, um eine für den Menschen nicht wahrnehmbare Verzögerungszeit und damit eine interaktive Visualisierung zu gewährleisten. Ein möglicher Ausweg besteht darin, dass der Client-Prozess ein bereits empfangenes Bild extrapoliert. Das bedeutet, dass das noch nicht empfangene Bild aus den vorhanden Informationen approximiert wird, wodurch die Bildwiederholfrequenz akzeptabel bleibt.

Bei der Bildsynthese des Server-Prozesses lässt sich, aus dem Tiefenpuffer ein Tiefenbild erzeugen, zusätzlich zum Farbbild. Geometrisch entspricht das Tiefenbild einer 2.5D Ansicht der Szene. Wird es zum Client gesendet, kann dieser das Tiefenbild als Punktwolke interpretieren. Das approximierte Bild entsteht, indem die Punktwolke aus einer neuen Kameraperspektive gezeichnet wird. Mit zunehmendem Winkelunterschied, zwischen der alten und der neuen Kameraperspektive, werden die Lücken zwischen den Punkten, der Punktwolke, immer größer. Durch die Erzeugung eines Dreiecksnetzes, aus dem Tiefenbild, lassen sich die Lücken schließen. Das Farbbild wird dabei als Textur über das Netz gelegt.



(a) (b)

Abbildung 1.1: Die Abbildungen zeigen die Extrapolation des ersten Bildes a, aus dem Datensatz *TestSpheres*. Der Winkelunterschied zum Bild b beträgt 8 Grad.

Die Zentrale Aufgabe dieser Arbeit, ist die Optimierung und die Übertragung der Tiefeninformationen, vom Server zum Client. Zu diesem Zweck erzeugt der Server, aus dem Tiefenbild ein adaptives Dreiecksnetz. Dieses wird an den Client übertragen und extrapoliert. Mit Ground-Truth Datensätzen werden, in Abhängigkeit des Kamerawinkels, verschiedene Ansätze hinsichtlich ihrer Güte evaluiert. Um die Algorithmen zu testen, wurde eine Server- und eine Client-Komponente entwickelt. Beide Komponenten tauschen Informationen über das auf TCP basierende Websocket-Protokoll aus. Der Client ist ein Browser-basierter Web-Client, damit die implementierten Algorithmen mit den Einschränkungen durch JavaScript und WebGl evaluiert werden können.

2 Verwandte Arbeiten

Einen Überblick über Architekturen und Methoden der interaktiven Remote-Visualisierung geben Shu Shi *et al.* [**?**]. Zum zentralen Problem ihrer Arbeit wird die Latenz und die effiziente Übertragung der Daten vom Server zum Client. Lösungen hängen vom Anwendungsfall ab. Sie definieren *THIN*-Systeme als eine Klasse von Remote-Visualisierungssystemen, die auf die Übertragung von 2D Informationen beschränkt sind und die Bildwiederholfrequenz weitaus geringer sein kann, als es in interaktiven 3D-Anwendungen erforderlich ist. Das Ziel von *THIN*-Systemen besteht, im darin Desktop-Anwendungen fernsteuern zu können. Beispiele für existierende Architekturen sind *SLIM* [**?**] und *THiNC* [**?**]. Sie sind für die Übertragung von 2D-Daten optimiert und profitieren davon, dass die meisten Änderungen nur Teilbereiche der eigentlichen Oberfläche betreffen. Ein anderes Extrem der Remote-Visualisierung ist die Aufteilung der Bildsynthese auf verschiedene Host-Systeme, wie bei dem Cluster-basierten System WireGL [**?**]. Jin Zhefan stellt in diesem Kontext Kompressionsmethoden für Zeicheninstruktionen, Vektoren, Normalen und Texturinformationen vor [**?**]. Peter Eisert und Philipp Fechteler haben ein Remote-Visualisierungssystem für Computerspiele entwickelt [**?**]. Ihr System überträgt bei kleinen Auflösungen, für Endgeräte ohne GPU die Bilder kodiert mit H.264 oder wahlweise MPEG-4. Bei größeren Auflösungen werden Zeicheninstruktionen gesendet, und die Bilder werden vom Client synthetisiert. Auch wenn ihr System ausschließlich für den Einsatz in lokalen Netzwerken konzipiert wurde, ist die Latenz zu hoch, um vom Benutzer nicht registriert zu werden. Mit Hilfe einer Bildextrapolation durch den Client kann die Latenz reduziert werden. Zu diesen Zweck schätzen Shu *et al.* in ihrer Arbeit [**?**] zusätzliche Referenz-Kamerapositionen und erzeugen zusätzlich zum Farbbild der originalen Kameraposition weitere Tiefenbilder. Durch Warping werden die Farbinformationen unter Zuhilfenahme der Tiefenbilder in die neue Kameraperspektive überführt. Dabei lassen sich durch Verdeckung bedingte Lücken, durch Pixel aus den Referenztiefenbilder schließen. Das Farbbild wird mit JPEG komprimiert und die Tiefenbilder mit ZLIB. Ein ähnlicher Ansatz wurde im VR-Bereich von Smit *et al.* erprobt [**?**]. Auch in diesem System werden Lücken mit Hilfe von Tiefenbildern aus Referenz-Kamerapositionen geschlossen. Palomo *et al.* nutzen ebenfalls einen Warping-Algorithmus, um mehrere Tiefenbilder und ein Farbbild miteinander zu vereinen. Zu diesem Zweck konstruieren sie mit Hilfe des Alpha-Farbkanals der Bilder einen speziellen z-Buffer [**?**], um ihren Algorithmus komplett auf die GPU auslagern zu können. Die Wahl des Warping-Algorithmus ist entscheidend für die Performance der Client-Komponente. Neben der im VR-System zum Einsatz kommenden Variante [**?**] bieten Mark *et al.* eine Übersicht über verschiedene Warping-Verfahren [**?**]. Die Erzeugung von mehreren Referenztiefenbildern ist eine rechenintensive Operation, da von diesen nur wenige Pixel tatsächlich benötigt werden, um die verdeckungsbedingten Lücken zu füllen. Popescu und Aliaga haben zu diesen Zweck ein spezielles Kameramodell entworfen, dass verdeckte Bildbereiche mit Hilfe gekrümmter Sichtstrahlen mitberechnen kann [**?**]. Die bisher vorgestellten Ansätze werden in dieser Arbeit nicht aufgegriffen; sie sollen verdeutlichen, welche alternativen Möglichkeiten existieren.

Die effiziente Übertragung und Visualisierung von Tiefenbildern ist auch für die Arbeit mit Tiefensensorsystemen von großem Interesse. Banno *et al.* erzeugen aus Tiefenbildern, die durch Tiefensensoren erhoben wurden, Dreicksnetze mit Hilfe der Delaunay-Triangulierung [**?**]. Ausgangspunkt für die Vernetzung ist eine Menge von Punkten, die Kanten und Krümmungen im Tiefenbild repräsentieren. Zur Erzeugung dieser Punkte kommt ein Quadtree zum Einsatz. Zusätzlich haben sie ein Verfahren entwickelt, um das erzeugte Netz in einem weiteren Schritt,

hinsichtlich der optischen Qualität, zu verbessern. Einen alternativen Ansatz für die Erzeugung der zur Vernetzung benötigten Punktmenge wird von Lee *et al.* vorgestellt [**?**]. Sie nutzen ein Fehlerdiffusionsverfahren, um in der Nähe von Kanten Punkte zu platzieren. In beiden Arbeiten wird die Extrapolation der Bilder durch die Rückprojektion der Dreiecksnetze durchgeführt. Diese beiden Ansätze bilden den Kern dieser Arbeit.

Um die Latenz bei der Übertragung zu verbessern, können mehrere Qualitätsstufen von einem Dreiecksnetz erzeugt werden, wie in der Arbeit von Chai *et al.* [**?**], die auf dem Verfahren von Lindstrom *et al.* basiert [**?**]. Mwalongo *et al.* haben einen hybriden Ansatz entwickelt, der zusätzlich zu einzelnen Vertices des Netzes Parameter für Kugeln übermittelt, die anschließend vom Client rekonstruiert werden [**?**]. Evans *et al.* haben ein Dateiformat für Punktwolken entwickelt, um diese progressiv an einen auf WebGL basierten Client zu senden [**?**].

Wessels *et al.* stellen eine Konzeption für den Programmaufbau eines interaktiven Remote-Visualisierungssystem basierend auf dem WebSocket-Protokoll vor [**?**]. In ihrem System besteht der Server-Prozess aus zwei Hauptkomponenten, der Visualisierungs-Engine und dem Deamon. Während die Visualisierungs-Engine für die Bildsynthese zuständig ist, übernimmt der Deamon die Kommunikation mit dem Client-Prozess. Der Client schickt dabei seine Eingabeinformationen von Maus und Tastatur direkt an den Server. Dieser wertet die Daten aus und erzeugt daraufhin ein mit JPEG komprimiertes Bild, das mit Base64 kodiert wird und schließlich an den Client-Prozess geschickt wird. Dieser kann das Bild nativ mit Hilfe eines HTML5 Canvas dekodieren und darstellen. Ihr System wird zur Grundlage für die in dieser Arbeit verwendete Softwarearchitektur.

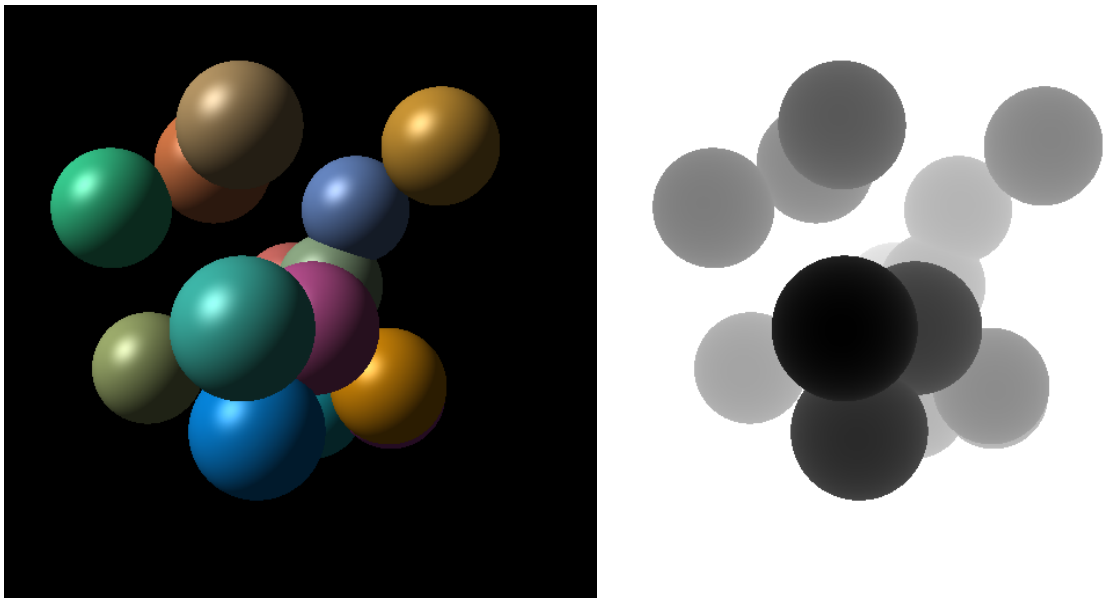
Auch wenn die Kompression und kompakte Kodierung der Dreiecksnetze nicht Teil dieser Arbeit sind, sollen die folgenden Arbeiten einen Überblick über mögliche Techniken geben. Gabriel Taubin und Jarek Rossignac haben ein Algorithmus zur Erzeugung und effizienten Kodierung von Dreiecksstreifen aus Dreiecksnetzen entwickelt [**?**]. Dazu konstruiert ihr Algorithmus Spannbäume über dem Netz, die zur Erzeugung möglichst großer Dreiecksstreifen genutzt werden. Die Kompression kann wahlweise verlustfrei oder verlustbehaftet durchgeführt werden. Typische Kompressionsraten werden mit 1:50 angegeben. Eine weitere Arbeit, die sich mit der Kompression von Dreiecksnetzen und einer kompakten Repräsentation von diesen beschäftigt, wurde Stefan Gumhold und Wolfgang Straßer geschrieben [**?**]. Kompression und Dekompression sind echtzeitfähig. Weitere geometrische Kompressionsverfahren für Vertices, Indices und Normalen, sind in der Arbeit von Michael Deering zu finden [**?**]. Sowie in der Arbeit [**?**] von den Autoren Touma und Gotsman.

3 Grundlagen

In diesem Kapitel werden die Datensätze im Detail vorgestellt und es wird gezeigt, wie die Bildextrapolation anhand der zur Verfügung stehenden Informationen durchgeführt wird. Anschließend werden die zwei Metriken PSNR und SSIM zum Vergleich von Bildern vorgestellt, die für die Evaluation verwendet werden.

3.1 Datensätze

Zur Analyse der verwendeten Methoden stehen zwei Datensätze zur Verfügung. Jeder Datensatz besteht aus einer Sequenz von Tripeln. Dabei setzt sich jedes Tripel aus einem Farbbild I, einem Tiefenbild D sowie einem Satz von Kameraparametern zusammen. Die Auflösung der Farb- und Tiefenbilder beträgt für alle Datensätze $w \times h = 512 \times 512$. Die Tiefenbilder wurden mit 16 Bit und die Farbbilder mit 24 Bit quantisiert, dabei entfallen jeweils 8 Bit auf die einzelnen Farbkanäle. In der Abbildung 3.1 werden Farb- und Tiefenbild des ersten Tripels aus dem Datensatz *TestSpheres* dargestellt.



(a) (b)

Abbildung 3.1: Farb- und Tiefenbild aus dem Datensatz *TestSpheres*

Die Kameraparameter setzen sich aus einem Vektor für die Kameraposition, einem *lookAt*-Vektor, der Punkt, auf den die Kamera schaut und dem *up*-Vektor zusammen. Für die Projektion werden die Positionen der Clippingebenen durch die Werte z_{near} und z_{far} beschrieben. Des Weiteren wird der Öffnungswinkel der Kamera benötigt.

Von beiden Szenen stehen 493 Bilder zur Verfügung. Beide Datensätze lassen sich in Abhängigkeit der Winkelschrittweite in 3 Sequenzen unterteilen. Der Winkel gibt dabei den Unterschied zwischen der Kameraposition aus dem ersten und dem gerade betrachteten Frame an. Eine

Auflistung der Sequenzen zeigt die Tabelle 3.1.

#	Anzahl Bilder	min Winkel	max Winkel	Winkel Schritt
1	241	0	5	0.25
2	60	6	10	1
3	192	15	90	5

Tabelle 3.1: Die Tabelle zeigt eine Übersicht über die Winkel der aufgenommenen Sequenzen.

Die Datensätze *CoolRandom* und *TestSpheres* wurden aus Partikeldatensätzen erzeugt. *TestSpheres* ist ein synthetisch erzeugter Datensatz, der aus wenigen Partikeln generiert wurde. Bei dem Datensatz *CoolRandom* handelt es sich dagegen um einen komplexen Datensatz, der mit Molekül-Datensätzen vergleichbar ist, welche in existierenden Anwendungen visualisiert werden.

3.2 Extrapolation

Eine einfache Möglichkeit, um die Extrapolation durchzuführen, besteht darin, das Tiefenbild als Punktwolke zu interpretieren und diese aus einer neuen Kameraperspektive zu zeichnen. Dazu wird aus jedem Pixel $d \in D$ ein Vertex erzeugt. Mit Hilfe der Rückprojektion lassen sich die Vertices in die gewünschten Positionen transformieren.

Um das Prinzip der Rücktransformation zu erläutern, soll zunächst betrachtet werden, wie ein Vertex v aus dem Modellkoordinatensystem in normalisierte Gerätekoordinaten transformiert wird. Die Abbildung 3.2 verdeutlicht diesen Vorgang.

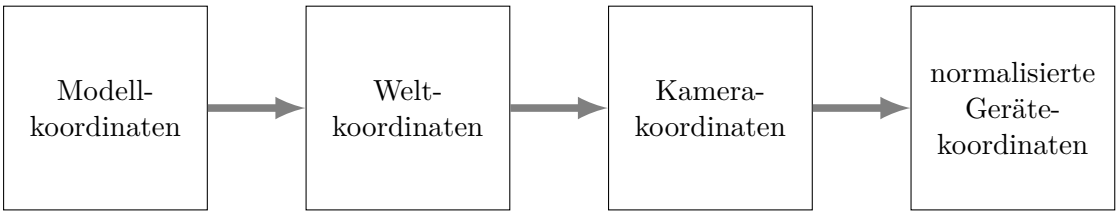


Abbildung 3.2: Die Abbildung zeigt die Teilschritte der Transformation eines Vertices aus den Modellkoordinaten in Bildschirmkoordinaten.

Der Vertex $v = (x, y, z, 1)^T$ liegt zunächst in homogenen Modellkoordinaten vor. Mit Hilfe der Modellmatrix $M = RT$, die aus einer Rotations- und einer Translationsmatrix besteht, lässt sich der Vektor in das Weltkoordinatensystem überführen. Die Transformation in das Kamerakoordinatensystem wird durch die Multiplikation der Kameramatrix V mit dem Vertex v berechnet. Schließlich kann die Projektion aus dem Kamerakoordinatensystem in normalisierte Gerätekoordinaten mit Hilfe der Projektionsmatrix P berechnet werden. In der Gleichung 3.1 wird die Transformation von dem Vertex v aus den Modellkoordinaten in die normalisierten Gerätekoordinaten v' zusammengefasst:

$$v' = M \cdot V \cdot P \cdot v. \tag{3.1}$$

Normalisierte Gerätekoordinaten haben für die x-,y- und z-Komponente den Wertebereich von -1 bis 1. Diese lassen sich in Bildschirmkoordinaten umrechnen, indem die drei Komponenten in das Intervall $[0, 1]$ übersetzt werden. Anschließend werden die x- und die y-Komponenten auf den Bildbereich gestreckt, indem sie mit $w - 1$ und $h - 1$ der gewünschten Auflösung $w \times h$ multipliziert werden.

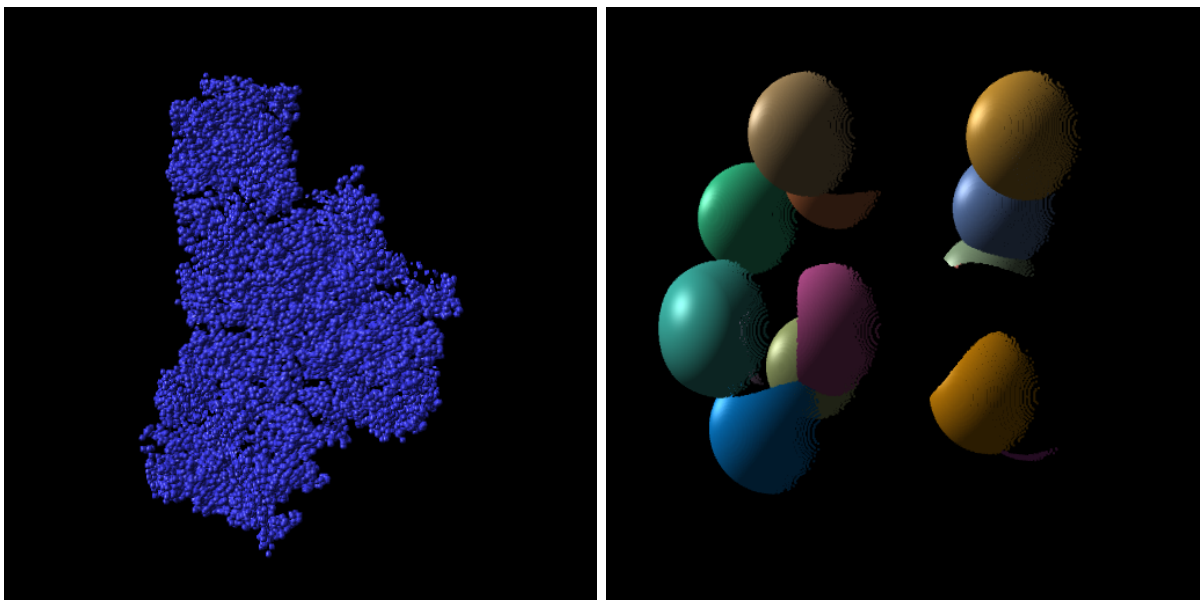
Im Folgenden wird die Menge der Tiefenbilder D_i und den dazugehörigen Transformationsmatrizen T_i, mit $i \in \{0, 1, ..., N\}$ betrachtet. Jede Transformatiosmatrix T_i setzt sich dabei aus einer Modellmatrix M_i, einer Kameramatrix V_i und einer Projektionsmatrix P_i zusammen:

$$T_i = M_i \cdot V_i \cdot P_i. \tag{3.2}$$

Um die Punktwolke des Tiefenbildes D_i mit der Transformationsmatrix T_j auf das Tiefenbild D_j abzubilden, muss zunächst für jeden Pixel $d_i \in D_i$ ein Vertex v_i erzeugt werden, mit $i, j \in \{0, 1, ..., N\}$. Zunächst liegt v_i in Bildschirmkoordinaten des Pixels d_i vor und muss in normalisierte Gerätekoordinaten übersetzt werden. Zu diesem Zweck werden zuerst die x- und y-Kompontenten durch $w-1$ beziehungsweise $h-1$ geteilt. Anschließend müssen alle drei Komponenten des Vektors v_i aus dem Werteintervall $[0, 1]$ in das Intervall $[-1, 1]$ transformiert werden. Jetzt liegt v_i in normalisierten Gerätekoordinaten vor. Durch die Multiplikation der inversen Transformationsmatrix T_i^{-1} mit v_i, kann der Vertex in seine Modellkoordinaten überführt und im Anschluss mit Hilfe von T_j aus einer neuen Kameraperspektive gezeichnet werden:

$$v_i' = T_j \cdot T_i^{-1} v_i. \tag{3.3}$$

Die Interpretation der Tiefenbilder als Punktwolke ist keine optimale Lösung, da bei der Bildsynthese Lücken im Bild entstehen. Eine leistungsfähigere Lösung ist es, aus dem Tiefenbild ein Dreiecksnetz zu erzeugen und dieses für die Bildsynthese zu verwenden. Mit Hilfe der Delaunay-Triangulierung ist es möglich, aus Tiefenbildern adaptive Dreiecksnetze zu erzeugen, welche die Lücken füllen.



(a) (b)

3.3 Delaunay-Triangulierung

Die Delaunay-Triangulierung ist ein Verfahren, um ein Dreiecknetz aus einer Menge von Punkten $p \in \mathbb{R}^2$ zu erzeugen. Dabei wird für jedes Dreieck ein Umkreis erzeugt, innerhalb dessen keine Punkte eines anderen Dreiecks enthalten sein dürfen. Jedes Dreieck des zu erzeugenden

Netzes muss diese Bedingung erfüllen. Das Resultat dieser Forderung ist die maximierte Innenwinkelsumme aller Dreiecke. Für eine gegebene Punktmenge ist die Lösung nicht eindeutig, es kann verschiedene Netzkonfigurationen geben, welche die Forderung erfüllen.

Es existieren verschiedene Algorithmen die Delaunay-Triangulierung durchzuführen. Die schnellsten erreichen eine Laufzeit von $O(n \log n)$ und sind damit für den Einsatz in Echtzeitanwendungen tauglich. Beispiele hierfür sind der Sweep-Algorithmus und die inkrementelle Konstruktion.

3.4 Bildvergleich

Im Folgenden werden zwei Metriken zum Vergleich von Bildern vorgestellt. Sie werden genutzt, um die Ähnlichkeit zwischen den extrapolierten und den Ground-Truth Bildern zu bestimmen. Beide Metriken werden für jeden Farbkanal separat berechnet.

3.4.1 PSNR

Die Abkürzung PSNR, im Englischen *Peak signal-to-noise ratio*, gibt das Verhältnis zwischen dem Maximalwert des Signals, auch als Nutzleistung bezeichnet, und dem Einfluss des Rauschens, also der Störleistung, an. Der PSNR wird auf eine logarithmische Skala abgebildet, weil die Nutzleistung in der Regel wesentlich größer als die Störleistung ist.

Um die Stärke des Fehlers zu bestimmen, wird der *mean squared error*, kurz MSE verwendet. Dieser lässt sich bestimmen, indem innerhalb eines Fensters der Größe $m \times n$ die quadratischen Abstände zwischen dem Originalbild I und dem rekonstruierten Bild K aufsummiert werden:

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2. \tag{3.4}$$

Der PSNR ist der Quotient, gebildet aus dem maximal möglichen Farbwert MAX_I, des untersuchten Farbkanals und dem MSE:

$$PSNR = 10 \times \log_{10} \left(\frac{MAX_I^2}{MSE} \right) \tag{3.5}$$

$$= 20 \times \log_{10} \left(\frac{MAX_I}{\sqrt{MSE}} \right) \tag{3.6}$$

$$= 20 \times \log_{10}(MAX_I) - 10 \times \log_{10}(MSE) \tag{3.7}$$

Bei einer Farbtiefe von 8 Bit pro Kanal, stehen Werte von 30 - 40 dB für ein geringes Störsignal und gleichzeitig für eine höhere Übereinstimmung der Bilder. Der PSNR ist ein häufig genutztes Maß, um die Ähnlichkeit zweier Bilder zu bestimmen.

3.5 SSIM

Im Gegensatz zum PSNR ist die von Wang *et al.* entwickelte Metrik *structural similarity index* ein auf die menschliche Wahrnehmung angepasstes Modell[**?**]. Die Metrik basiert auf der Idee auf, dass die Struktur der abgebildeten Objekte unabhängig von Beleuchtung und Kontrast gemessen werden kann.

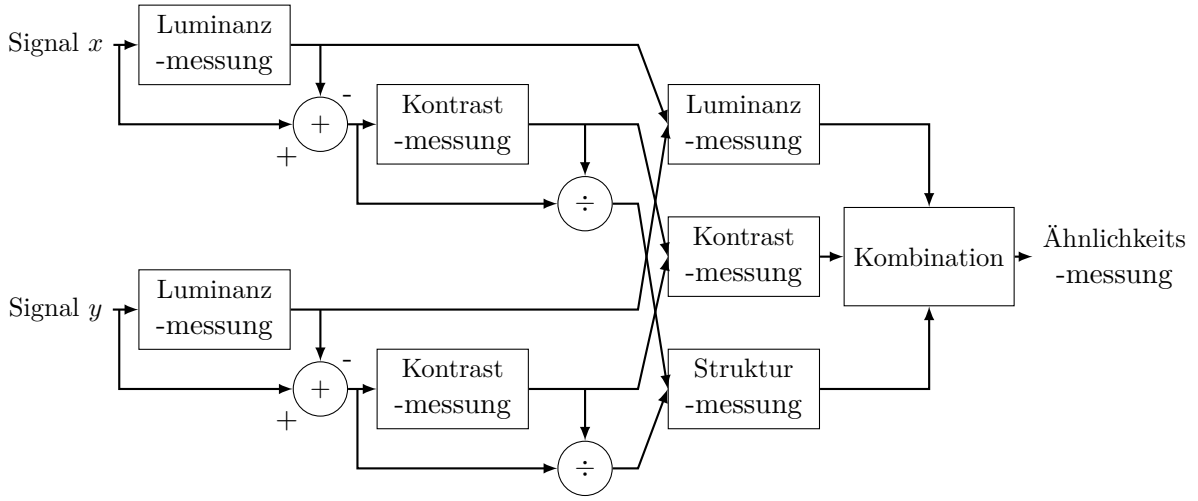


Abbildung 3.3: Das Modell zeigt das Vorgehen von dem SSIM-Algorithmus [**?**].

Die mittlere Intensität des diskreten Signals x lässt sich in einem lokalen Bereich der größe N folgendermaßen bestimmen:

$$\mu_x = \frac{1}{N} \sum_{i=1}^{N} x_i. \tag{3.8}$$

Die Vergleichsfunktion, welche die Beleuchtung zwischen zwei Signalen x und y misst, wird durch dem Term $l(x, y)$ repräsentiert. Dieser setzt die mittleren Intensitäten μ_x und μ_y wie folgt ins Verhältnis:

$$l(x, y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}. \tag{3.9}$$

Die Konstante C_1 wird benötigt, um die numerische Stabilität zu gewährleisten. Analog zur Abbildung 3.3 wird im Anschluss an die Beleuchtungsmessung die mittlere Intensität von beiden Signalen $x - \mu_x$ und $y - \mu_y$ abgezogen.

Der Kontrast wird unter Verwendung der Standardabweichung σ_x approximiert. In diskreter Form lässt sich diese mit der folgenden Formel berechnen:

$$\sigma_x = \left(\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \mu_x)^2 \right)^{\frac{1}{2}}. \tag{3.10}$$

Die Vergleichsfunktion für den Kontrast wird mit $c(x, y)$ bezeichnet und definiert sich wie folgt:

$$c(x, y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}. \tag{3.11}$$

Wie bei der Berechnung der mittleren Intensität wird eine Konstante C_2 eingeführt. An dieser Stelle wird das Signal normalisiert, indem es durch seine eigene Standardabweichung geteilt wird und die beiden Signale $(x - \mu_x)/\sigma_x$, $(y - \mu_y)/\sigma_y$ entstehen. Der Vergleich der strukturellen Eigenschaften $s(x, y)$ wird mit den normalisierten Signalen durchgeführt. Für den Vergleich der Struktur sollen folgende Eigenschaften gelten:

Erstens: Die Funktion $s(x, y)$ muss symmetrisch sein $s(x, y) = s(y, x)$.

Zweitens: Soll die Funktion auf einen Wert kleiner oder gleich 1 beschränkt werden $s(x, y) \leq 1$.

Drittens: Es soll nur ein Maximum $s(x, y) = 1$ geben, genau dann und nur dann, wenn gilt, dass $x = y$.

Die folgende Definition erfüllt alle drei Forderungen:

$$s(x, y) = \frac{\sigma_{xy} + C_3}{\sigma_x + \sigma_y + C_3}. \tag{3.12}$$

Die Kovarianz σ_{xy} berechnet sich folgendermaßen:

$$\sigma_{xy} = \frac{1}{N-1} \sum_{i=1}^{N} (x_i - \mu_x)(y_i - \mu_y). \tag{3.13}$$

Sie korreliert mit dem Kosinus des Winkels zwischen den beiden Vektoren $x - \mu_x$ und $y - \mu_y$. Die Konstanten C_1, C_2 und C_3 in den drei Vergleichsfunktionen sorgen dafür, dass eine Division durch 0 nicht möglich wird, sollten die Werte in den Nennern zu klein werden. Letztlich kann die Gesamtqualität gemessen werden, indem Beleuchtung, Kontrast und Strukturvergleich kombiniert werden:

$$SSIM(x, y) = [l(x, y)]^{\alpha} \times [c(x, y)]^{\beta} \times [s(x, y)]^{\gamma}. \tag{3.14}$$

Mit den Parametern $\alpha > 0, \beta > 0, \gamma > 0$ kann die Gewichtung zwischen den Vergleichsfunktionen variiert werden. Im Rahmen dieser Arbeit gilt $\alpha = \beta = \gamma = 1$ und es gilt für die Konstanten: $C_3 = C_2/2$, mit $C_1 = 6.5025$ und $C_2 = 58.5225$. Der $SSIM$ wird lokal, berechnet. Dabei werden die Signalwerte x_i und y_i mit einer Gaussianfunktion gewichtet. Damit lassen sich die mittlere Intensität, die Standardabweichung und die Kovarianz zu folgenden Gleichungen umformen, wobei w_i die Gewichtung an dem Punkt i bezeichnet:

$$\mu_x = \sum_{i=1}^{N} w_i x_i \tag{3.15}$$

$$\sigma_x = \left(\sum_{i=1}^{N} w_i (x_i - \mu_x)^2 \right)^{\frac{1}{2}} \tag{3.16}$$

$$\sigma_{xy} = \sum_{i=1}^{N} w_i (x_i - \mu_x)(y_i - \mu_y). \tag{3.17}$$

Damit lässt sich der $SSIM(x, y)$ durch die folgende Gleichung bestimmen:

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}. \tag{3.18}$$

Der $SSIM(x, y)$ misst die Güte jedoch nur lokal. Um eine Aussage über das gesamte Bild treffen zu können, wird der Mittelwert über allen $x \in X$ und $y \in Y$ berechnet:

$$MSSIM(X, Y) = \frac{1}{M} \sum_{j=1}^{M} SSIM(x_j, y_j). \tag{3.19}$$

4 Methodik

In dieser Arbeit werden zwei grundsätzliche Verfahren untersucht, um die Tiefeninformationen zu übertragen und als Dreiecksnetz darzustellen. Der erste Ansatz besteht darin, das Tiefenbild als solches verlustfrei zu komprimieren und zu übertragen. Der Client empfängt und dekodiert das Tiefenbild und erzeugt daraus ein voll vernetztes Dreiecksnetz. Im zweiten Ansatz wird aus dem Tiefenbild ein adaptives Dreiecksnetz mit Hilfe der Delaunay-Triangulierung vom Server konstruiert und an den Client übertragen.

Die Erzeugung von adaptiven Dreiecksnetzen lässt sich als eine Form der verlustbehafteten geometrischen Kompression bezeichnen. Die zu übertragende Informationsmenge lässt sich durch Valenz-basierte Kodierung weiter verringern.

Weil die Tiefenwerte des Tiefenbildes komplett und ohne Informationsverlust im voll vernetzen Dreiecksnetz enthalten sind, liefert dieses Verfahren Ergebnisse, die als *Ground-Truth*-Information zum Vergleich der adaptiven Vernetzung dienen.

4.1 Vollvernetzung

Bei der Vollvernetzung wird für jeden Pixel aus dem Tiefenbild D ein Vertex erzeugt. Die entstandenen Vertices werden über Kanten zu Dreiecken miteinander verbunden. Die Abbildung 4.1 zeigt drei unterschiedliche Varianten, wie sich die Vertices zu voll vernetzten Dreiecksnetzen verbinden lassen. Alle drei Varianten wurden im Rahmen dieser Arbeit implementiert, weisen bei entsprechend großen Auflösungen jedoch keine Unterschiede hinsichtlich der Qualität der Darstellung auf.

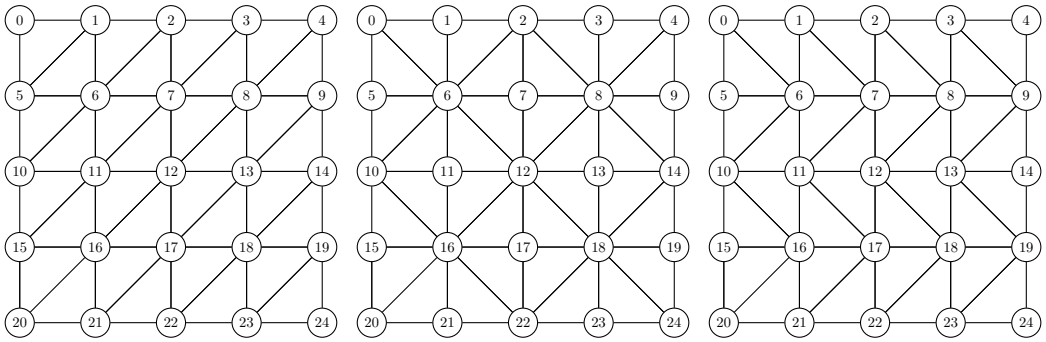


Abbildung 4.1: Darstellung von drei verschiedenen Varianten der Vollvernetzung.

Die Anzahl der Vertices entspricht der Anzahl der insgesamt gegebenen Bildpunkte. Die x- und y-Komponenten lassen sich aus der bekannten Auflösung vorausberechnen und müssen im Falle einer Auflösungsänderung neu bestimmt werden. Für jeden Vertex wird die z-Komponente mit Hilfe der Pixelwerte aus dem Tiefenbild während der Bildsynthese gesetzt. Die Anzahl der Dreiecke lässt sich dabei wie folgt berechnen: $2(w-1)(h-1)$.

Problematisch bei diesem Verfahren ist, dass die Anzahl der Dreiecke von der Auflösung abhängt

und der Rechenaufwand der clientseitigen Bildsynthese linear mit der Auflösung steigt.

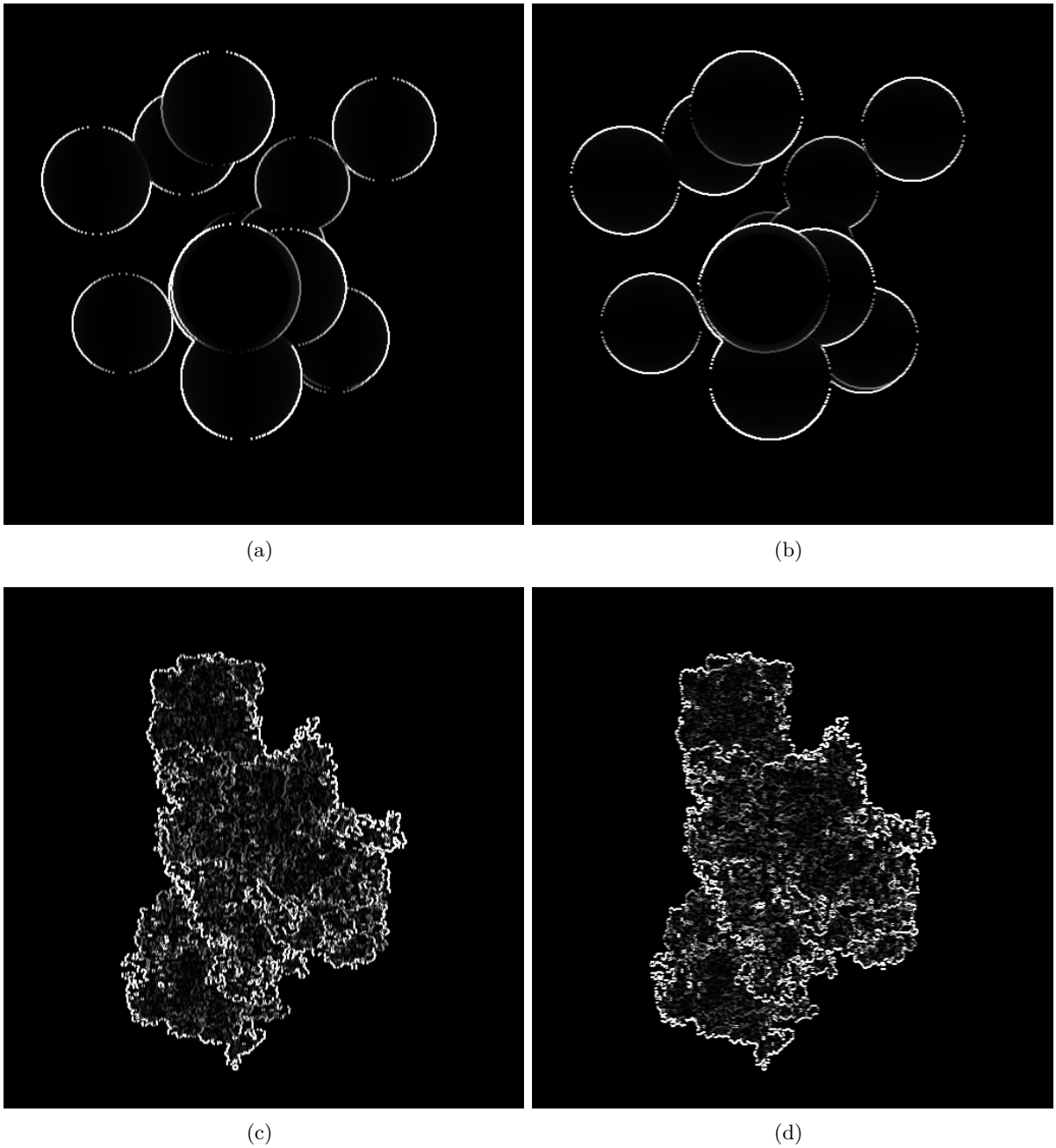
4.2 Delaunay-Triangulierung

Die zweite in dieser Arbeit untersuchte Variante zur Erzeugung eines Dreiecksnetzes aus einem Tiefenbild D, besteht in der serverseitigen Erzeugung einer adaptiven Vernetzung. Prinzipiell, lässt sich die Vorgehensweise in drei Schritte unterteilen: Im ersten Schritt wird eine Menge von Punkten $P \subset D$ definiert. Dabei muss die Menge P so gewählt werden, dass die Werteverläufe des Tiefenbildes möglichst optimal wiedergegeben werden. Der zweite Schritt besteht in der Vernetzung der Punkte $p \in P$ zu Dreiecken. Das dabei eingesetzte Verfahren ist die Delaunay-Triangulierung. Abschließend kann das erzeugte Netz in einem dritten Schritt weiter verfeinert werden.

Entscheidend für das Ergebnisnetz und für die Geschwindigkeit des gesamten Verfahrens ist die Wahl der Punktmenge P. Das Ziel bei der Verteilung der Punktemenge P ist es, eine hohe Punktdichte im Bereich von Kanten und gekrümmten Flächen zu erzielen. Im Gegensatz dazu sollten planare Regionen möglichst keine Punkte enthalten, weil diese mit wenigen Dreiecken approximiert werden können. Ausgangspunkt für die Erzeugung der Punktmenge P sind die Gradienten des Tiefenbildes ∇_x und ∇_y. Die Gradienten werden mit Hilfe des Sobel-Operators berechnet. Dazu wird die erste Ableitung berechnet und gleichzeitig wird die dazu orthogonale Richtung geglättet. Mit Hilfe einer Faltungsmatrix der Größe 3×3 lassen sich die Gradienten berechnen:

$$\nabla_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * D, \qquad \nabla_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * D.$$

Auf diese Weise entstehen für ein Tiefenbild zwei Gradientenbilder, wie in Abbildung 4.2 dargestellt. Zur Erzeugung der Punktmenge P wurden zwei Verfahren untersucht, die im Folgenden näher betrachtet werden.



(a) (b)

(c) (d)

Abbildung 4.2: Gradientenbilder beider Datensätze vom jeweils ersten Tiefenbild.

4.2.1 Quadtree

Die erste in dieser Arbeit vorgestellte Methode, zur Erzeugung der Punktmenge P, basiert auf einer Quadtree-Datenstruktur. Ein Quadtree ist eine Baumdatenstruktur, in der die Anzahl der Kindknoten, eines Knotens auf vier Kindknoten beschränkt wird. Im Rahmen dieser Arbeit repräsentiert jeder Knoten eine rechteckige Region R des Tiefenbildes D. Eine Region R wird durch seine Eckpunkte definiert. Um die Baumdatenstruktur zu erzeugen, wird ein Wurzelknoten definiert, dessen Region $R = (0, 0, w-1, h-1)$ den Ausmaßen des Tiefenbildes D entspricht. Im Anschluss wird die Region des Wurzelknotens in vier gleichgroße Teilregionen zerlegt. Diese Teilregionen sind die Kindknoten des Wurzelknotens. Alle Kindknoten werden auf diese Weise sukzessive weiter unterteilt, bis die Tiefe des erzeugten Baumes eine maximale Baumtiefe d_{max}

erreicht. Die Kindknoten der Baumtiefe d_{max} enthalten selbst keine weiteren Kindknoten und werden als Blattknoten bezeichnet, die Regionen die sie repräsentieren als Blattregionen. Die maximale Tiefe des Quadtrees ist dabei invers proportional zur Breite der Blattregionen. Mit dem Parameter d_{max} kann die Qualität und die Kompression des erzeugten Netzes sowie der Berechnungsaufwand skaliert werden. Die erzeugte Baumdatenstruktur hängt von der Auflösung der Tiefenbilder ab und muss nur neu erzeugt werden, wenn sich die Auflösung oder der Parameter d_{max} ändert.

Im nächsten Schritt wird mit Hilfe der Baumdatenstruktur die Punktmenge P erzeugt. Zu diesem Zweck wird der Baum, beginnend mit den Blattknoten, zum Wurzelknoten traversiert. Mit Hilfe der Gradienten kann die Region R eines Knoten auf planarität getestet werden. Ist eine Region R nicht planar, dann werden die Eckpunkte von R zur Menge P hinzugefügt.

Um zu überprüfen, ob eine Region R als planar bezeichnet werden kann, lässt sich mit Hilfe der Differenz zwischen dem maximalen und dem minimalen Wert der Gradienten innerhalb von R bestimmten:

$$c_x = \max_R \nabla_x - \min_R \nabla_x \tag{4.1}$$

$$c_y = \max_R \nabla_y - \min_R \nabla_y. \tag{4.2}$$

Wenn entweder c_x oder c_y größer als ein zuvor definierter Schwellwert ist, dann ist die Region R nicht planar und die Eckpunkte werden der Menge P hinzugefügt. Die Werte für c_x und c_y der inneren Konten lassen sich effizient aus den bereits berechneten maximalen und minimalen Gradienten seiner Kindknoten berechnen.

Die Wahl des Schwellwerts ist entscheidend für die Anzahl der eingefügten Punkte in P. Dieser kann adaptiv gewählt werden, um die Komplexität des Dreiecksnetzes anzupassen.

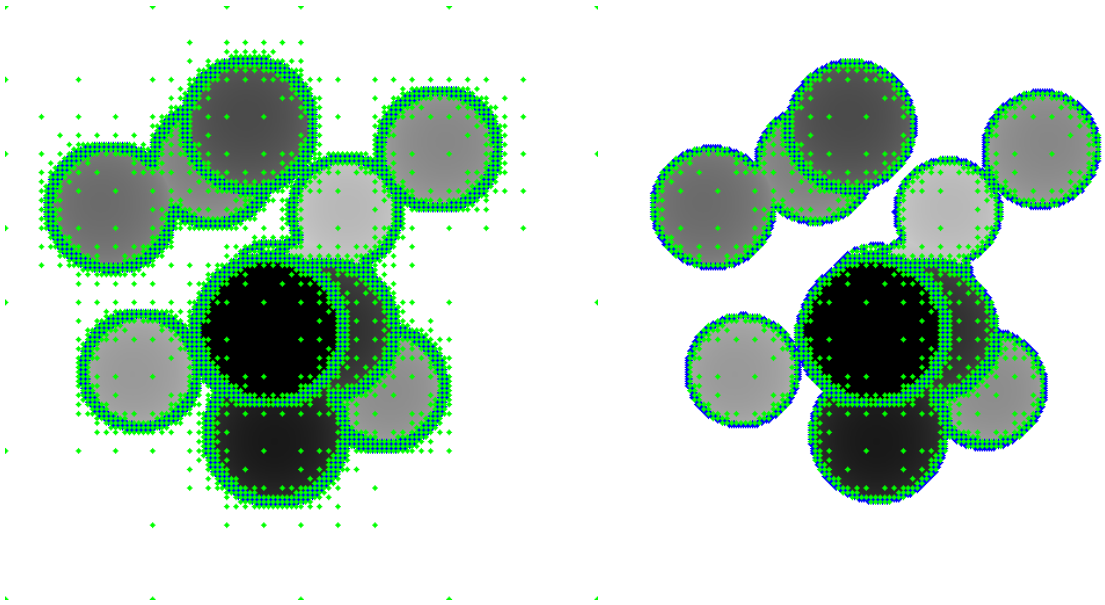
Für den eigentlichen Planaritätstest unterscheidet sich dieser Schwellwert in Abhängigkeit, ob es sich um einen inneren oder einen Blattknoten handelt. Der Schwellwert für die Blattknoten wird mit l bezeichnet und der für die inneren Knoten mit i. Blattknoten sollten vorrangig Tiefenunterspränge, Kanten detektieren und somit die Kontur der dargestellten Objekte kennzeichnen. Innere Knoten dagegen repräsentieren die Oberfläche der Objekte und werden gesetzt, um Verläufe von nicht planaren Flächen abzubilden zu können. Weil Tiefensprünge größere Gradienten zur Folge haben als gekrümmte Oberflächen, sollte der Schwellwert i für die inneren Knoten kleiner sein als der Schwellwert l für die Blattknoten, $i < l$. Die Werte von i und l liegen im Intervall $[0, 1]$.

Das Ergebnis der Traversierung des Baums ist eine Menge von Punkten P, die sich aus den Eckpunkten der getesteten Regionen zusammensetzt, in denen die Gradientdifferenzen den entsprechenden Schwellwert überschreitet.

Diese Punktmenge P kann weiter reduziert werden, indem getestet wird, ob ein Punkt $p \in P$ zum Hintergrund gehört. Das ist der Fall, wenn der Tiefenwert des Tiefenbildes an Stelle p dem maximalen Tiefenwert entspricht. Zur Erinnerung: Die Tiefenwerte liegen linear zwischen den beiden Clippingebenen, wobei ein großer Wert bedeutet, dass dieser nahe der Bildschirmebene liegt.

4.2.2 Floyd-Steinberg

Der zweite in dieser Arbeit vorgestellte Ansatz, zur Erzeugung der Punktmenge P basiert auf einem Fehler-Diffusionsverfahren. Im ersten Schritt wird aus den beiden Gradientenbildern ∇_x, ∇_y ein Merkmalsbild σ erzeugt:



(a) (b)

Abbildung 4.3: Szene *TestSpheres*, Baumtiefe 9, l=5, i=4

$$\sigma_d = \left(\frac{\| \nabla d \|}{A} \right)^{\gamma}. \tag{4.3}$$

Dabei entspricht der Nenner von 4.3 der Länge des Gradientenvektors ∇d, des Pixels $d \in D$. Dieser wird mit A, dem größtmöglichen Wert von $\| \nabla d \|$, normiert. Der Parameter $\gamma \in [0, 1]$ wird genutzt, um die Ausprägung von schwachen Kanten im Merkmalsbild σ zu erhöhen.

Um aus dem Merkmalsbild σ die Punktmenge P zu erzeugen, kommt der Floyd-Steinberg-Algorithmus zum Einsatz. Bei dem Floyd-Steinberg-Algorithmus handelt es sich um einen einfachen und effizienten Dithering-Algorithmus, der auf einem, Fehler-Diffusionsverfahren fußt. Das Merkmalsbild wird mit Hilfe des Schwellwerts δ binarisiert. Zu diesem Zweck wird das Merkmalsbild σ Pixel für Pixel zeilenweise durchlaufen. Der betrachtete Pixel wird mit einem Schwellwert δ verglichen. Wenn der Wert des Pixels größer als der Schwellwert ist, dann wird der Wert des Pixels mit δ, andernfalls mit 0 ersetzt. Zudem wird durch die Binarisierung resultierende Quantisierungsfehler des Pixels $d \in D$ entsprechend der Abbildung 4.4 auf die benachbarten Pixel verteilt.

	d	$\frac{7}{16}$
$\frac{3}{16}$	$\frac{5}{16}$	$\frac{1}{16}$

Abbildung 4.4: Darstellung der gewichteten Verteilung des Quantisierungsfehlers [**?**].

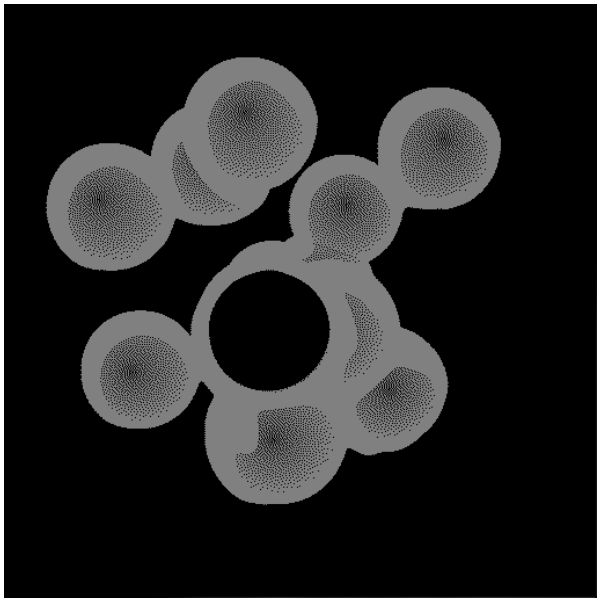
Dadurch lässt sich die Verteilung des Quantisierungsfehlers ohne einen zusätzlichen Puffer in einem einzigen Durchlauf berechnen. Das folgende Pseudocodebeispiel 4.1 verdeutlicht das Verfahren:

```
1  for each y
2          for each x
3             oldpixel         := pixel[x][y]
4             newpixel         := (pixel[x][y] > δ) ? δ : 0
5             pixel[x][y]      := newpixel
6             quant_error      := oldpixel - newpixel
7             pixel[x+1][y  ]  := pixel[x+1][y  ] + quant_error * 7 / 16
8             pixel[x-1][y+1]  := pixel[x-1][y+1] + quant_error * 3 / 16
9             pixel[x  ][y+1]  := pixel[x  ][y+1] + quant_error * 5 / 16
10            pixel[x+1][y+1]  := pixel[x+1][y+1] + quant_error * 1 / 16
```

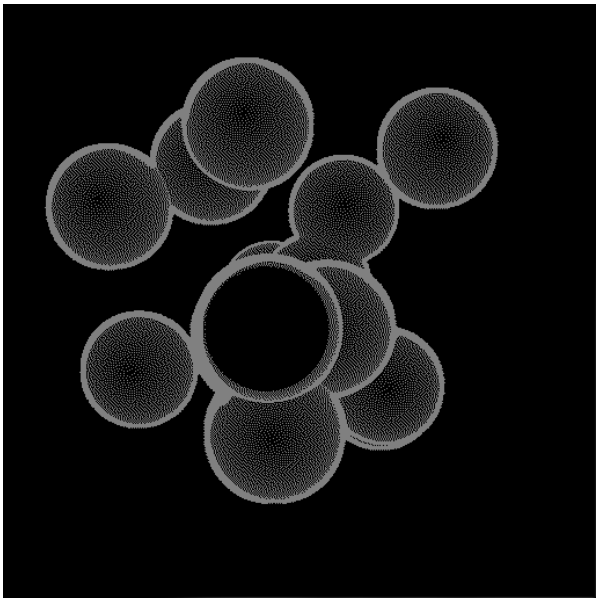
Listing 4.1: Floyd-Steinberg-Algorithmus [**?**]

Immer wenn die Bindung erfüllt wird und der Wert eines Pixels auf δ gesetzt wird, dann wird die Menge P mit der Position des gerade bearbeiteten Pixels erweitert. Die Abbildung 4.5 zeigt verschiedene bereits auf δ und 0 reduzierte Merkmalsbilder σ.

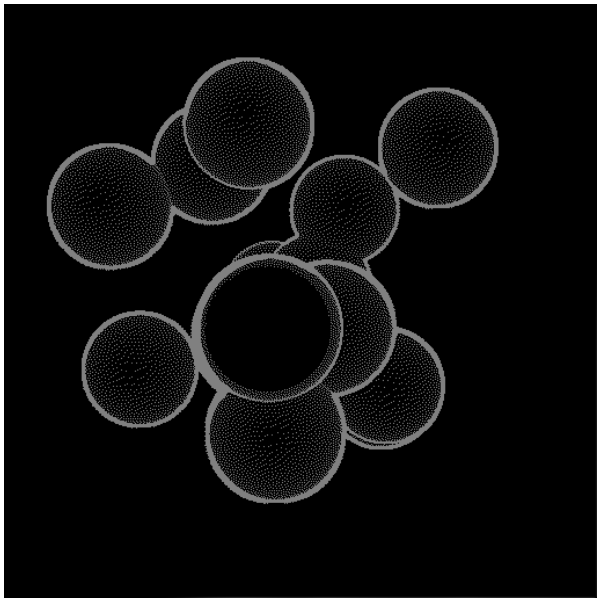
Ein Vorteil dieses Verfahrens ist die effiziente Erzeugung der Menge P. Mit Hilfe der beiden Parameter γ, δ lässt sich die Anzahl der Punktmenge P variieren. Dabei spielt insbesondere δ eine entscheidende Rolle, da durch diese Größe die Punktdichte angepasst werden kann. Wenn δ kleine Werte annimmt, dann wird die Punktdichte erhöht und im Fall großer Werten reduziert.



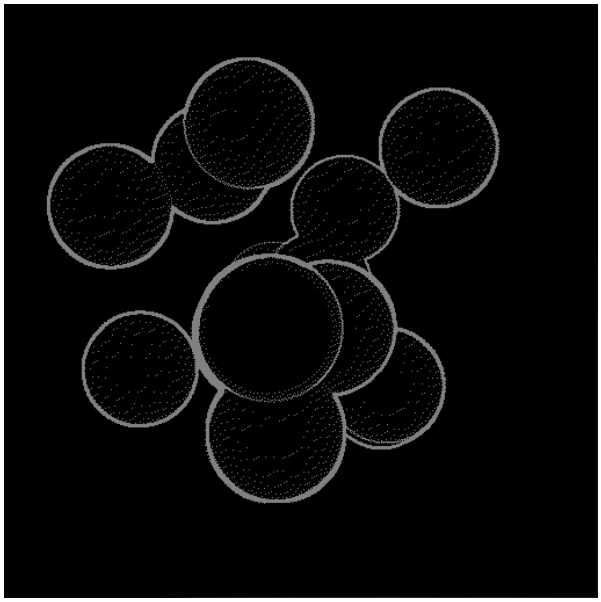
(a) $\gamma = 0.2$



(b) $\gamma = 0.4$



(c) $\gamma = 0.6$



(d) $\gamma = 1.0$

Abbildung 4.5: Die Bilder a bis d zeigen das Merkmalsbild in Abhängigkeit des Parameters γ bei einem festgesetzten Schwellwert von 0.5.

4.2.3 Erzeugung des Netzes

Die aus den beiden Methoden erzeugten Punkte $p \in P$ können jetzt mit Hilfe der Delaunay-Triangulierung zu einem Dreiecksnetz miteinander verbunden werden. In dieser Arbeit wird zu diesem Zweck die Delaunay-Triangulierung aus dem opencv-Framework verwendet.

4.2.4 Netzoptimierung

Das auf diese Weise entstandene Netze können, in Abhängigkeit der Eingabedaten, zwei arten von Artefakten enthalten. Zum einen können Dreiecke falsche Tiefenregionen approximieren, weil nur die Eckpunkte der jeweiligen Regionen R zur Konstruktion der Dreiecke genutzt werden. Zum anderen können Dreiecke über Tiefensprüngen liegen, wodurch Kanten nicht korrekt vom Netz abgebildet werden. Diese Art von Artefakte kommen zustande, wenn die Blattregionen zu große Bildregionen repräsentieren. Um die Artefakte zu reduzieren, werden nicht valide Dreiecke nochmals unterteilt oder verworfen.

Die Validität eines Dreiecks wird anhand seiner Kanten bestimmt. Eine Kante wird als nicht valide Kante bezeichnet, wenn die Tiefenwerte einiger von ihr überspannten Pixel abweicht, oder die 3D Richtung der Kante sich dem Lot der Bildebene nähert, während gleichzeitig eine bestimmte Länge in der xy-Ebene überschritten wird.

Die Eckpunkte der zu prüfenden Kante, werden im folgenden mit p_1 und p_2 bezeichnet. Der Punkt p_m bezeichnet dabei den Mittelpunkt der Kante p_1p_2.

$$\frac{d_{p_1} + d_{p_2}}{2} - d_{p_m} < \varepsilon \tag{4.4}$$

Eine Kante ist nicht valide, wenn der eigentliche Tiefenwert $D(p_m)$ In der Gleichung 4.5 approximiert der erste Term das Verhältnis, zwischen dem Betrag des Tiefenunterschieds von p_1 und p_2 zu der Länge der Kante projiziert auf die xy-Ebene.

$$\frac{\mid d_{p_1} - d_{p_2} \mid}{\parallel p_1 - p_2 \parallel (d_{p_1} + d_{p_2})} < \alpha \tag{4.5}$$

Ist der Wert des ersten Terms der Gleichung 4.5 einer Kannte größer als der Schwellwert T_{angle}, dann steht diese Kante nahezu senkrecht auf der Bildebene und sie liegt mit hoher Wahrscheinlich über einen Tiefensprung. Eine Kante auf die das zutrifft wird als nicht valide bezeichnet. Enthält ein Dreieck mindestens zwei Kanten die valide sind, wird es direkt zum endgültigen Dreiecksnetz hinzugefügt. Wenn ein Dreieck dagegen mehr als zwei nicht valide Kanten enthält, wird es weiter unterteilt.

Zwei Bildpunkte p_1 und p_2 werden als verbindbar bezeichnet, wenn alle Bildpunkte zwischen p_1 und p_2 valide Tiefenwerte besitzen und ihre Tiefe sich zum größten Teil linear ändert. Um Rechenzeit zu sparen wird empfohlen nur den Median p_m zwischen p_1 und p_2 zu testen. Eine Strecke wird als Verbindbar bezeichnet, wenn sie die folgende Gleichung erfüllt:

$$\mid (d(p_2) - d(p_m)) - (d(p_m) - d(p_1)) \mid < \beta \tag{4.6}$$

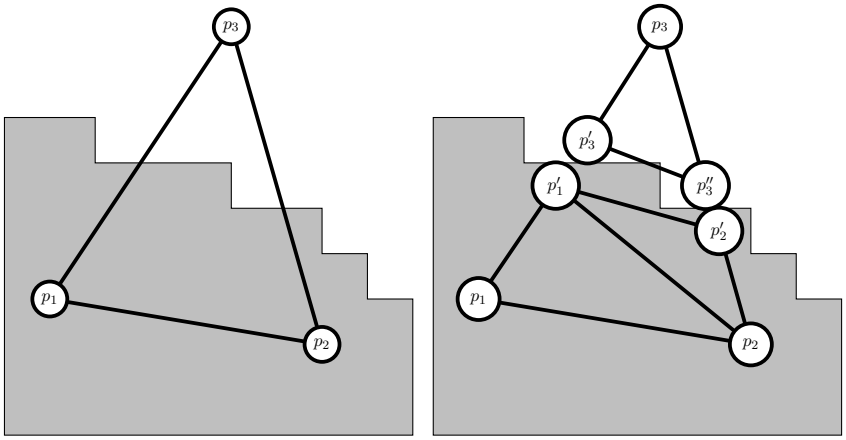
.

Um ein Dreieck zu unterteilen muss eine Fallunterscheidung durchgeführt werden. Hat das Dreieck nur eine valide Kante p_1p_2, dann müssen die anderen beiden Kanten wie in der Abbildung 4.6 geteilt werden und es entstehen aus dem ursprünglichen Dreieck drei neue Dreiecke.

Dazu werden vier neue Vertices iterativ entlang der alten Strecken eingefügt:

Der Vertex p'_1 ist der von Punkt auf der Strecke p_3p_1, der am weitesten von p_1 entfernt liegt und mit dem Punkten p_1 und p'_2 verbindbar ist.

Der Vertex p'_2 ist der von Punkt auf der Strecke p_3p_2, der am weitesten von p_2 entfernt liegt und mit dem Punkten p_2 und p'_1 verbindbar ist.



(a) (b)

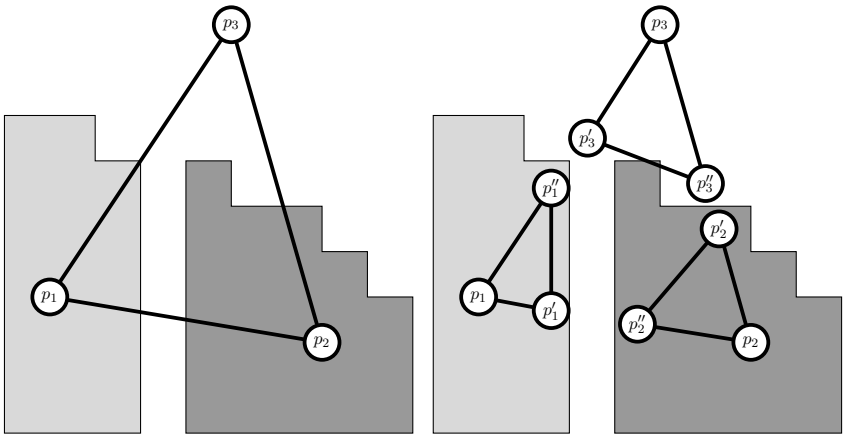
Abbildung 4.6: Die Bilder a und b zeigen die Unterteilung des Dreiecks $p_1p_2p_3$, wenn die zwei Kanten p_1p_3, p_2p_3 nicht valide sind.

Der Vertex p'_3 ist der von Punkt auf der Strecke p_3p_1, der am weitesten von p_3 entfernt liegt und mit dem Punkt p_3 verbindbar ist.

Der Vertex p''_3 ist der von Punkt auf der Strecke p_3p_2, der am weitesten von p_3 entfernt liegt und mit dem Punkt p_3 verbindbar ist.

Anschließend wird das Dreieck $p_3p'_3p''_3$ zum endgültigen Dreiecksnetz hinzugefügt. Um die anderen beiden Dreiecke einzufügen muss eine weitere Fallunterscheidung durchgeführt werden. Wenn die Strecke $p_1p'_2$ kleiner ist als $p_2p'_1$, dann werden die Dreiecke $p_1p_2p'_2$, $p_1p'_2p'_1$ zu dem finalen Netz hinzugefügt, andernfalls die beiden Dreiecke $p_1p_2p'_1$ und $p_2p'_2p'_1$.

In dem Fall, das alle drei Kanten nicht valide sind, werden sechs neue Vertices eingefügt, die Abbildung 4.7 verdeutlicht diesen Fall. Die Berechnung aller Vertices geschieht analog zu dem Vertex p'_3 aus dem vorhergehenden Betrachtung mit einer validen Kante.



(a) (b)

Abbildung 4.7: Die Bilder a und b zeigen die Unterteilung des Dreiecks $p_1p_2p_3$, wenn keine Kante valide ist.

Zu beachten ist, dass jedes neu erzeugte Dreieck auf Kollinearität zu überprüfen und gegebenenfalls zu verwerfen.

Die Pixelpositionen, in den Gleichungen 4.5 und 4.6 werden in Texturkoordinaten angegeben und die Farbwerte der Pixel auf das Intervall $[0, 1]$ transformiert.

Die Abbildung 4.8 zeigt ein nachbearbeitetes Dreiecksnetz. Rot markierte Kanten sind nicht valide und werden nicht zum endgültigen Netz hinzugefügt. Alle grünen Kanten, gehören zu Dreiecken die aus einem Dreieck, ohne valide Kante erzeugt wurden. Analog dazu wurden Dreiecke, hervorgehoben durch blaue Kanten aus Dreiecken erzeugt mit nur einer validen Kante. Kollineare Dreiecke fallen zu Schwarzen Kanten oder Punkten zusammen.

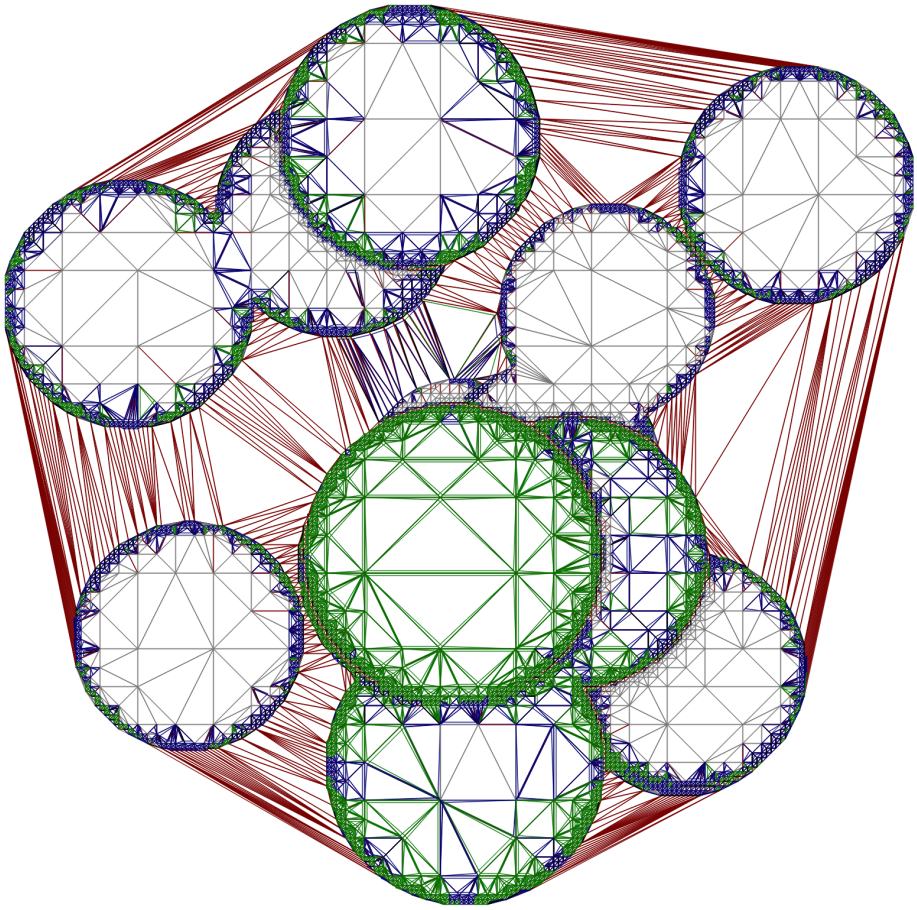


Abbildung 4.8: Nachbearbeites Netz des ersten Frames aus dem *TestSpheres*-Datensatz.

5 Implementierung

Die Client-Anwendung ist eine Browser basierte Web-Anwendung, die mit dem Server über das WebSocket-Protokoll kommuniziert. Dabei handelt es sich um ein auf TCP basierendes Netzwerkprotokoll, das eine Bidirektionale Verbindung zwischen den Verbindungsteilnehmern erlaubt, ohne auf Methoden wie Long Polling zurückgreifen zu müssen. Die klientseitge Extrapolation der Bilddaten wird mit Hilfe von WebGL durchgeführt wird. Bei WebGL handelt es sich um eine Bibliothek die von modernen Browsern zur Verfügung gestellt wird, um eine Hardware beschleunigte Bildsynthese zu ermöglichen. Der Vorteil dieser Technologie ist die Unabhängigkeit der Anwendung im Bezug, zur Plattform und dem Gerät, in Kombination mit einer hohen Rechengeschwindigkeit.

Diese Kapitel wird dabei in zwei Hauptbereiche unterteilt. Im ersten, werden Details über die Kommunikation zwischen Client und Server behandelt. Der zweite Abschnitt beschäftigt sich mit Implementierungsspezifischen Details der Algorithmen.

5.1 Details zur Kommunikation

Die Server-Anwendung wurde mit der Sprache c++ entwickelt. Dabei wird auf die Implementierung der WebSocket-Serverkomponente aus dem QT-Framework zurückgegriffen. Der Server-Prozess besteht aus einem Thread. Zur Verwaltung der Anfragen wird das Event-Managementsystem von QT verwendet. Beim Start der Serveranwendung werden die Szenen geladen. Aus diesen wird eine *Playlist* erstellt, diese vereint die Informationen aller Szenen. Im Anschluss daran wird Standardparameterkonfiguration geladen und der Server wartet darauf, dass sich ein Client verbindet.

Die Kommunikation zwischen Server- und die Client-Komponente basiert auf dem JSON-RPC 2.0 Protokoll. Die Abkürzung JSON-RPC steht für *JavaScript Object Notation Remote Procedure Call*. Es handelt sich dabei, um ein Protokoll, das den Aufbau und die Nominatur von Nachrichten beschreibt. Im speziellen um Nachrichten, die dazu dienen, dass der Kommunikationsteilnehmer bestimmte Methoden mit bestimmten Parameterkonfigurationen zu starten. Es basiert auf dem Textbasierten Datenformat JSON. Nachrichten werden in Anfragen und Antworten unterteilt. Mit Hilfe einer *id* lassen sich Nachrichten bei asynchroner Kommunikation zuordnen. Es werden drei verschiedene Typen von Anfragen unterschieden, *Request*, *Notification* und *Batch Request*. Ein *Request* ist eine Anfrage, welche die Antwort eines Kommunikationsteilnehmers verlangt. Diese Form der Anfragen, kann zu Synchronisierten Nachrichtenübertragung genutzt werden. Im Gegensatz dazu, entfällt bei einer *Notification* die *id*, weil keine Antwort verlangt wird. Der *Batch Request* fast eine Menge von *Requests* und *Notifications* in eine einzige Anfrage zusammen.

In dieser Arbeit implementierte Variante von JSON-RPC 2.0, wird auf Anfragen vom Typ *Request* beschränkt, weil die gesamte Kommunikation synchronisiert stattfindet.

Eine JSON-RPC-Anfrage ist ein JSON-Objekt, das sich aus vier Teilen zusammensetzt. Der erste ist ein String mit der JSON-RPC Version, gefolgt vom Namen der aufzurufenden Methode. Im Anschluss folgt ein JSON-Objekt, das die Parameter der Aufzurufenden Methode enthält und zum Schluss kommt die *id* der Anfrage. Als Beispielanfrage soll der Aufruf der Methode

resize dienen. Möchte der Client die Auflösung ändern schickt er die Anfrage, dem Beispiel 5.1 entsprechend zum Server.

```
1 {
2         "jsonrpc"       : "2.0",
3         "method"        : "resize",
4         "params"        : [512, 512],
5         "id",           : 0
6 }
```

Listing 5.1: RPC-Request

Hat der Server die Anfrage empfangen, ruft dieser darauf hin die Methode *resize* mit den entsprechenden Parametern aus der Anfrage auf. Im Anschluss daran, wird eine Antwort zurückgesendet. Eine JSON-RPC-Antwort ist ebenfalls ein JSON-Objekt, dass aus drei Teilen besteht. Der erste Teil ist ebenfalls die JSON-RPC Version als String kodiert. Der zweite Teil enthält den Rückgabewert in Form eines JSON-Objektes und zum Schluss folgt die *id* der Anfrage, wie im Beispiel 5.2 demonstriert

```
1 {
2         "jsonrpc"       : "2.0",
3         "result"        : [512, 512],
4         "id",           : 0
5 }
```

Listing 5.2: RPC-Response

Der Server stellt eine Menge von Methoden zur Verfügung, die durch den Client initiiert, von dem Serverprozess ausführt werden.

Eine Beispielkommunikation, dargestellt im Sequenzdiagramm 5.1 veranschaulicht den Remote-Aufruf verschiedener Methoden.

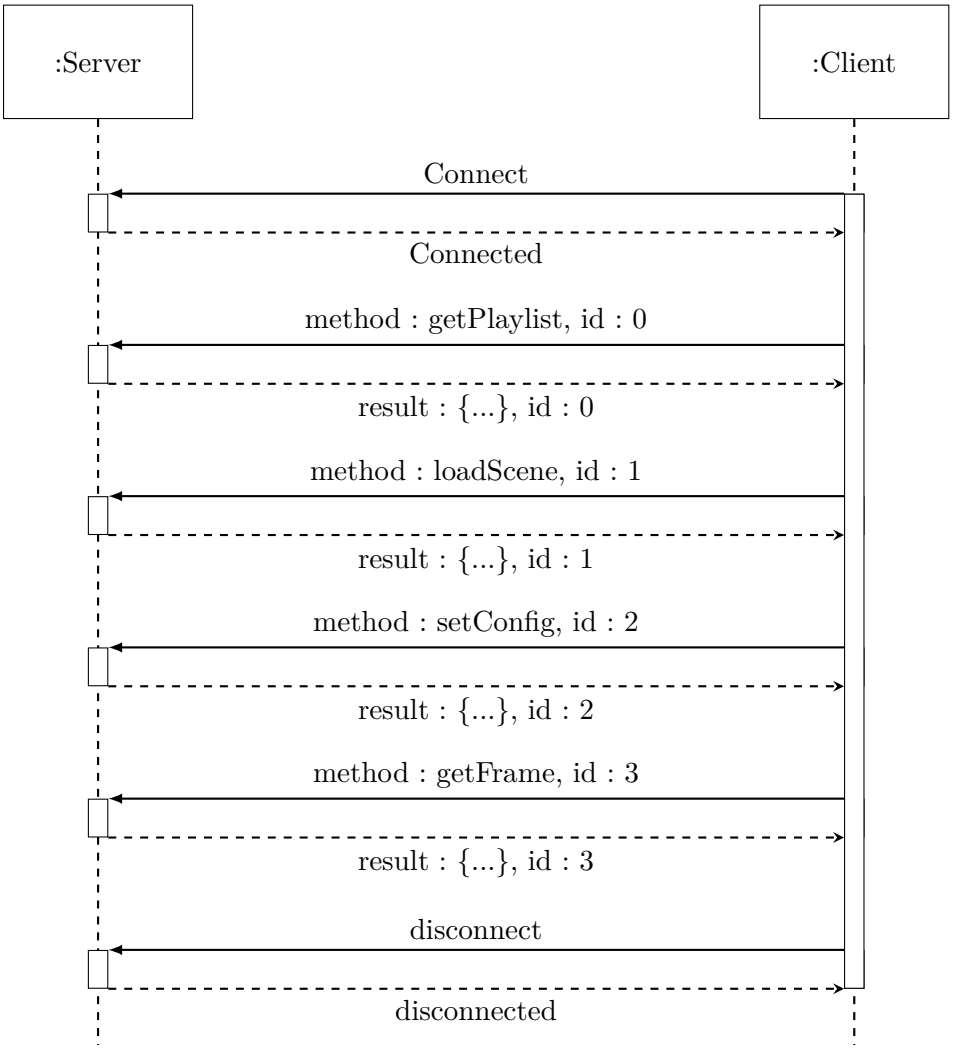


Abbildung 5.1: Das Sequenzdiagramm zeigt eine Beispielkommunikation, um ein Bild vom Server anzufordern.

Kann sich der Client erfolgreich mit dem Server verbinden, fragt er die *Playlist* ab. Diese liefert dem Client alle Informationen über die zur Verfügung stehenden Szenen. Sie enthält für jeden Frame einer Szene, dessen Kamerainformationen. Nachdem der Client alle Szenen kennt, kann er den Server mitteilen, welche dieser laden soll. Anschließend wird mit der Methode *setConfig* eine gewählte Parameterkonfiguration gesetzt. Mit Hilfe der *Playlist*, kann ein beliebiger Frame der geladenen Szene, angefordert werden. Wird der Frame empfangen, kann dieser vom Client anschließend extrapoliert werden. Neben den Vorgestellten, stehen zusätzliche Methoden, zum Starten einer Messung beziehungsweise zum Speichern eines Bildes durch den Server zur Verfügung.

5.2 Details zur Implementierung der Algorithmen

In diesem Abschnitt werden Details zur Implementierung, der vorgestellten Algorithmen genauer betrachtet. Begonnen wird mit den Farbbildern. Um eine verlustbehaftete Komprimierung ermöglichen zu können, kommt das JPEG-Format zum Einsatz. Anschließend müssen die Bildinformationen mit Base64 kodiert werden, damit diese als String in ein JSON-Objekt integriert werden können.

5.2.1 Vollvernetzung

Im Fall der Vollvernetzung, wird neben dem Farbbild auch das Tiefenbild mit Base64 kodiert und als String in ein JSON-Objekt eingebettet. Im Gegensatz zu den Farbbildern, werden die Tiefenbilder verlustfrei als PNG komprimiert.

Die Dekodierung von Farb- und Tiefenbilder kann mit Hilfe des Bildcontainers nativ durch vom Browser durchgeführt werden. Im Bezug auf das Tiefenbild ergibt sich ein Problem. Dieser Bildcontainer erlaubt für jeden Farbkanal ausschließlich eine Farbtiefe von 8 Bit. Damit ein Shaderprogramm trotzdem auf 16 Bit Informationen aus einer Textur zurückgreifen kann, muss der 16 Bit Wert auf zwei Farbkanäle aufgeteilt werden und die Rekonstruktion erfolgt im Shader. Diese Aufteilung findet auf dem Server statt, bevor das Bild mit Base64 kodiert wird. Auf den roten-Farbkanal entfallen die ersten 8 Bit und die zweiten 8 Bit, der 16 Bit Zahl werden auf den grünen Farbkanal ausgelagert. Die Abbildung 5.2 verdeutlicht die Aufteilung.

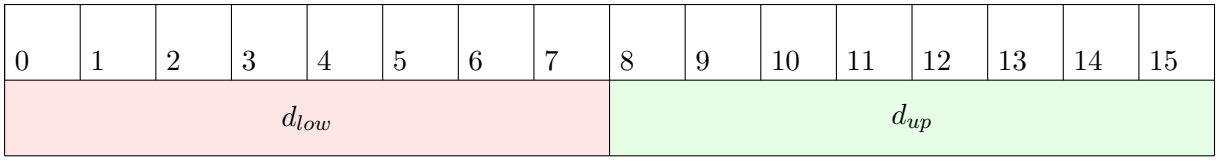


Abbildung 5.2: Hier wird die Aufteilung einer 16 Bit Zahl auf die Farbkanäle, links rot und rechts grün, visuell verdeutlicht. Das Schlüsselwort d_{up} bezeichnet die ersten 8 Bit und d_{low} die zweiten 8 Bit.

Um aus d_{up} und d_{low} die 16 Bit d Zahl zu erhalten, genügt es den Wert von d_{up} zuerst mit 255 zu multiplizieren und den Wert von d_{low} zu addieren, wie in der Gleichung 5.1 beschrieben.

$$d = d_{low} + d_{up} \cdot 256. \tag{5.1}$$

Die rekonstruierte Zahl d aus der Gleichung 5.1 liegt im Ganzzahligem Intervall $[0, 65535]$. Beim Zugriff auf die Farbwerte einer Textur im Shader, werden die jedoch normiert. Das bedeutet, im Fall eines 8 Bit Farbkanals, wird jeder Wert durch den Maximalwert 255 geteilt. Auf diese weise werden die Werte eines Pixels in das Intervall $[0, 1]$ überführt. Um eine 16 Bit Zahl in dieses Intervall zu überführen, muss sie durch den Wert 65535 geteilt werden. Aus diesem Zusammenhang ergibt sich die folgende Gleichung zur Rekonstruktion der Zahl d:

$$d = \frac{255 \cdot \frac{d_{low}}{255} + \frac{d_{up}}{255} \cdot 255 \cdot 265}{2^{16} - 1}. \tag{5.2}$$

Die Zahl 65535 lässt sich in das Produkt $65535 = 255 \cdot 257$ umschreiben und dadurch Gleichung 5.2 kann weiter vereinfacht werden:

$$d = \frac{\cancel{255} \cdot \left(\frac{d_{low}}{255} + \frac{d_{up}}{255} \cdot 265\right)}{\cancel{255} \cdot 257} \tag{5.3}$$

$$= \frac{d_{low}}{255} \cdot \frac{1}{257} + \frac{d_{up}}{255} \cdot \frac{256}{257} \tag{5.4}$$

Im Shader kann die 16 Bit zahl deshalb einfach entsprechend der Gleichung 5.4 rekonstruieren. Der Farbwert des roten Farbkanals wird mit $(1/257)$ multipliziert und der des grünen mit

(256/257). Im Anschluss daran werden die beiden Werte aufsummiert und der Wert der 16 Bit Zahl steht im Shader zur Verfügung.

Im Fall der Vollvernetzung, werden alle Dreiecke, die zum Hintergrund gehören transparent gezeichnet. Zusätzlich wird im Vertex-Shader, der Gradient des bearbeiteten Vertice berechnet. Der x- und y-Anteil des Gradienten, wird mit dem Schwellwert g verglichen. Ist einer dieser beiden Gradientkomponenten größer als der Schwellwert g, wird der Alphakanal an dieser Stelle ebenfalls auf 0 gesetzt.

Damit verdeckte Fragmente durch die transparenten sichtbar werden, kommt ein 3 und ein 5 Pass *depth peeling*-Verfahren zum Einsatz.

5.2.2 Delaunay-Triangulierung

Beim *Quadtree*, wird der Bildbereich sukzessiv in vier gleichgroße Regionen aufgeteilt, deren Eckpunkte in der Punktmenge P landen können. Die Abbildung 5.3 veranschaulicht die Aufteilung.

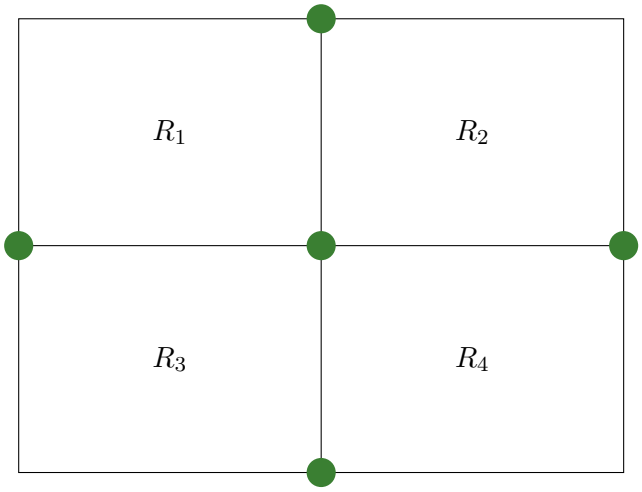


Abbildung 5.3: Unterteilung eines Rechtecks, in vier Teilflächen. An den grünen markierten Positionen, gibt es mehrere Möglichkeiten, wie die Eckpunkte der betreffenden Rechtecke gesetzt werden können.

Dabei gibt es mehrere Möglichkeiten, wie die Eckpunkte der Teilfenster gewählt werden können. Einerseits können die Flächen, exakt gleich groß gewählt werden. Mit dem Effekt, das viele kleine Dreiecke dadurch entstehen, dass die grün markierten Punkte sich aus mindestens zwei Eckpunkten zusammensetzten, die sich jeweils nur um eine Pixeleinheit voneinander unterscheiden.

Die zweite Möglichkeit besteht darin, für derartige Eckpunkte dieselbe Koordinate zu wählen. Auf diese Weise lässt sich die Anzahl der in der Punktmenge P enthaltenen Punkte, bei gleichbleibender Bildqualität senken. In dieser Arbeit wird die zweite Variante verwendet.

Bei der Erzeugung der Gradientenbilder, mit dem Sobel-operator, entsteht ein Rand mit der Breite von einem Pixel. Dieser wird auch bei der Binärisierung von dem Merkmalsbild σ ausgelassen.

Um die Netzoptimierung durchführen zu können, müssen die Punkte $p'_1, p''_1, p'_2,''_2$, sowie p'_3 und p''_3 bestimmt werden. Diese Aufgabe wird mit Hilfe einer Linienrasterisierung gelöst. Dabei wird nicht zwischen den Pixeln interpoliert.

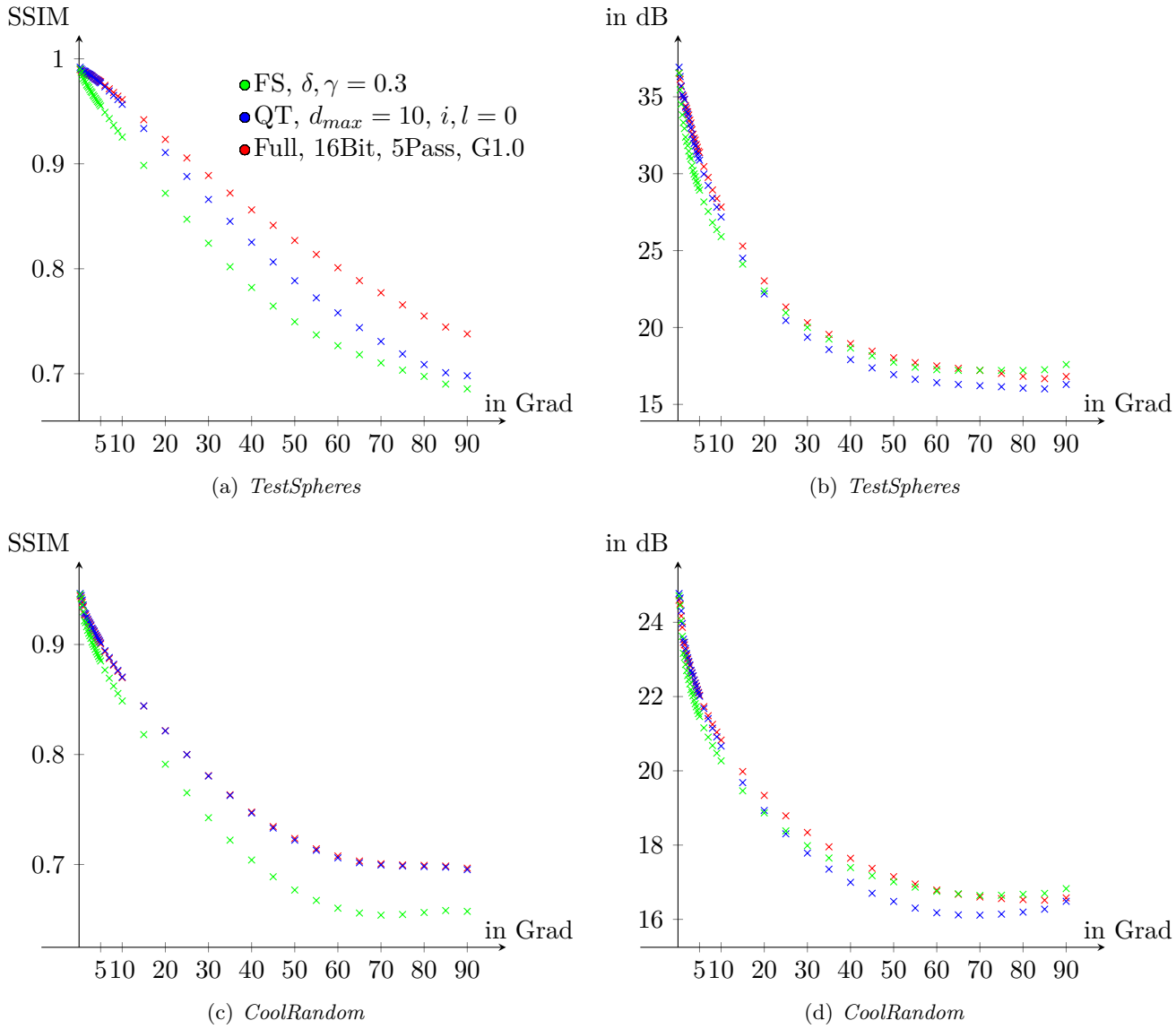
6 Ergebnisse

Zu Beginn dieses Kapitels wird beschrieben, wie die Ergebnisse erhoben werden. Anschließend werden die Parameterkonfigurationen und deren Resultate vorgestellt. Dabei auftretende Besonderheiten werden kenntlich gemacht.

Die Vollvernetzung und die beiden Varianten der Delaunay-Triangulierung werden, mit den beiden *Ground-Truth*-Datensätzen, *TestSpheres* und *CoolRandom*, evaluiert. Der erste Frame einer Szene dient als Referenz für die Extrapolation. Aus dessen Tiefenbild wird mit dem gewählten Alogithmus ein Dreiecksnetz erzeugt. Zusammen mit dem dazugehörigen Farbbild wird es zum Client gesendet. Anschließend wird Dreiecksnetz, texturiert mit dem Farbbild, aus allen Kameraperspektiven der Szene gezeichnet und die dabei entstandenen Bilder werden zurück zum Server gesendet. Die Qualität aller extrapolierten Bilder wird durch den PSNR und den SSIM bestimmt. Zusätzlich wird gemessen wie lange der Server für die Kodierung benötigt und wie lange der Client für die Extrapolation eines Bildes braucht.

Für die Vollvernetzung wurde das Tiefenbild mit 8 und 16 Bit kodiert. Es wurde ein 3 Pass und ein 5 Pass Verfahren verwendet, um die Transparenz mit *depth peeling* zu realisieren. Und schließlich wurden die Schwellwerte $g \in \{1, 0.9, 0.8\}$ für den Vergleich mit dem Gradienten getestet. Die besten Ergebnisse werden mit 16 Bit, $g = 1$ und dem 5 Pass Verfahren erzielt. Diese Konfiguration dient als Basis für den Vergleich mit den Delaunay-Verfahren.

Im Fall der Delaunay-Triangulation, wird zwischen den beiden Punktgenerierungsverfahren *Quadtree* und *Floyd-Steinberg* unterschieden. Für den *Quadtree*-Ansatz werden generell alle Hintergrundpixel verworfen. Die Bäume werden mit den maximalen Baumtiefen $d_{max} \in \{10, 9, 8\}$ erzeugt. Für jede Baumtiefe d_{max} wurden die Schwellwertkonfigurationen $l \in \{0.1, 0.2, ..., 1.0\}$ und $i \in \{0.1, ..., l\}$ gemessen. Für alle $l > 0.6$ und $i = 0.6$, wurden zusätzlich die Netzverfeinerung durchgeführt. Mit den Schwellwerten $\alpha \in \{0.1, 1, 10, 100, 1000\}$ und $\beta \in \{0.1, 0.2, ..., 1.0\}$. Die Erzeugung der Punktemenge erhoben mit dem *Floyd-Steinberg*-Fehlerdiffusionsverfahren, wird mit den Parametern $\delta, \gamma \in \{0.1, 0.2, ..., 1.0\}$ evaluiert. Die folgenden Diagramme zeigen ausgewählte Beispiele von den erhobenen Daten. Dabei wird mit den besten Ergebnissen begonnen:

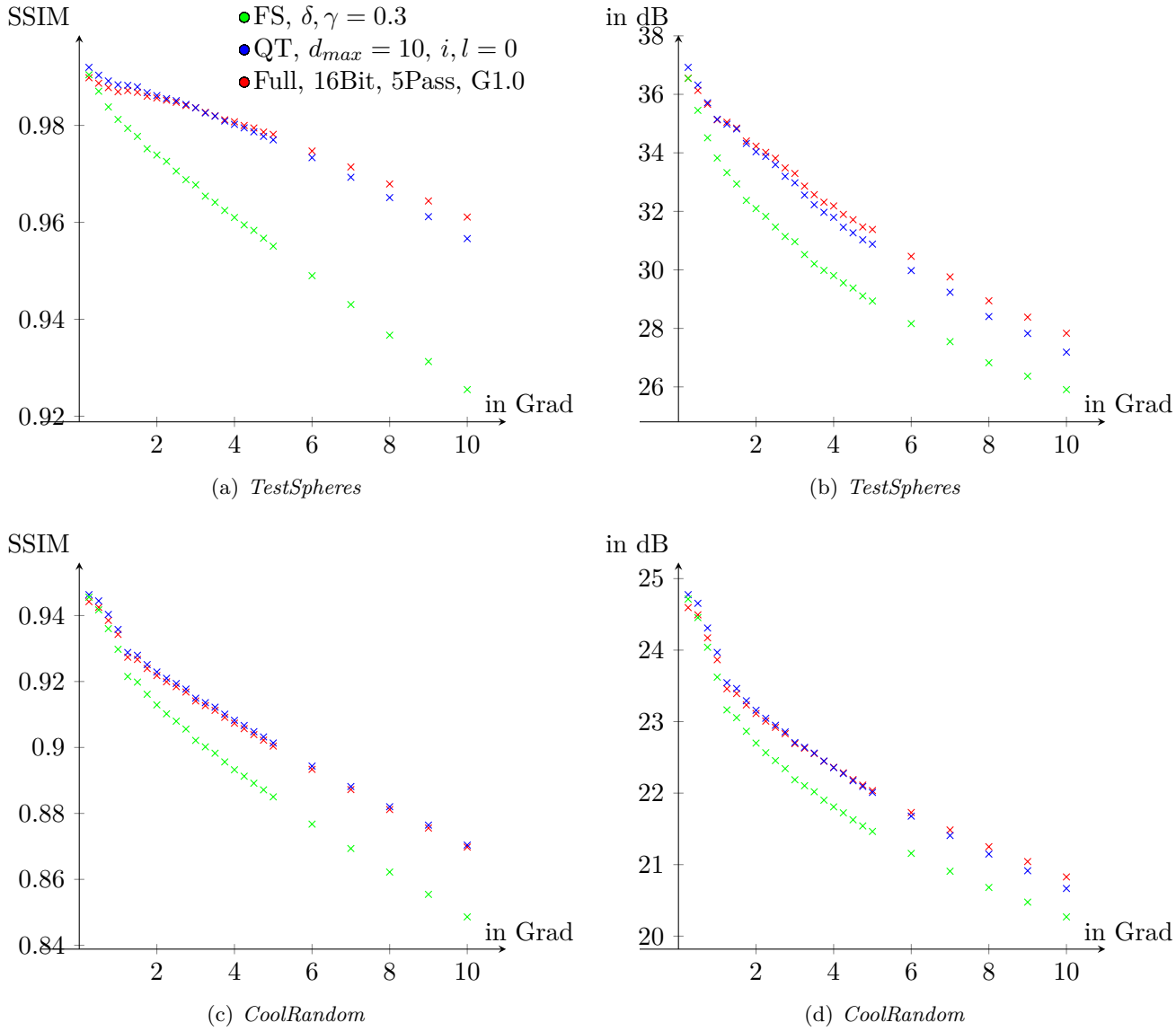


(a) *TestSpheres* (b) *TestSpheres*

(c) *CoolRandom* (d) *CoolRandom*

Abbildung 6.1: Die Diagramme a und b zeigen die Ergebnisse des Datensatzes *TestSpheres*, während c und d die Resultate des Datensatzes *CoolRandom* abtragen. Dabei zeigen a und c die Ergebnisse des SSIM, während in b und d die Ergebnisse des PSNR gezeigt werden.

Die Abbildung 6.1 zeigt die besten Resultate, der Vollvernetzung, vom *Quadtree*, sowie von der *FloydSteinberg* Variante der Delaunay-Triangulierung. Für beide Szenen gemessen mit dem SSIM liefert die Vollvernetzung insgesamt die besten Ergebnisse, dicht gefolgt vom *Quadtree*-Ansatz und die schlechtesten Ergebnisse liefert der *FloydSteinberg* Ansatz. Interessant ist, dass die Güte der Ergebnisse von dem *CoolRandom*-Datensatz, gemessen mit dem PSNR, schon bei den ersten extrapolierten Frames unter 25 *dB* liegt. Außerdem fällt beim PSNR auf, dass die Werte, ab einem Winkel von 70 Grad wieder leicht ansteigen.



(a) *TestSpheres* (b) *TestSpheres*

(c) *CoolRandom* (d) *CoolRandom*

Abbildung 6.2: Darstellung der Graphen aus der Abbildung 6.1 im Winkelintervall von 0 bis 10 Grad.

Die Abbildung 6.2 zeigt die gleichen Parameterkonfigurationen, wie die Abbildung 6.1, dieses mal jedoch beschränkt auf die Kamerawinkelunterschiede von 0 bis 10 Grad. Hier zeigt sich sowohl mit dem SSIM, als auch beim PSNR, ein signifikanter unterschied zwischen dem FloydSteinberg-Ansatz und den anderen beiden Methoden. Der *Quadtree*-Ansatz ist in den ersten Frames sogar besser als die Vollvernetzung.

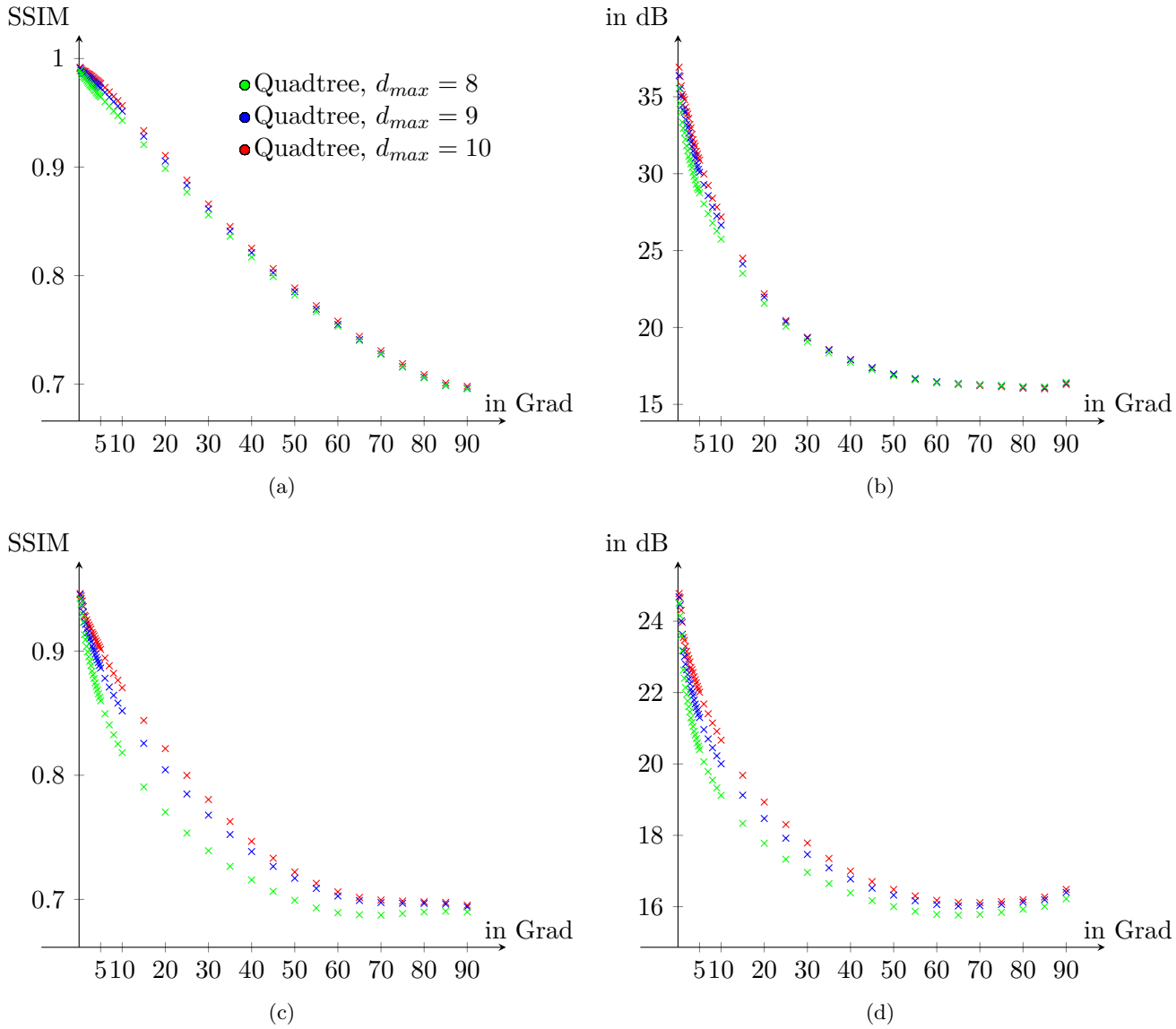
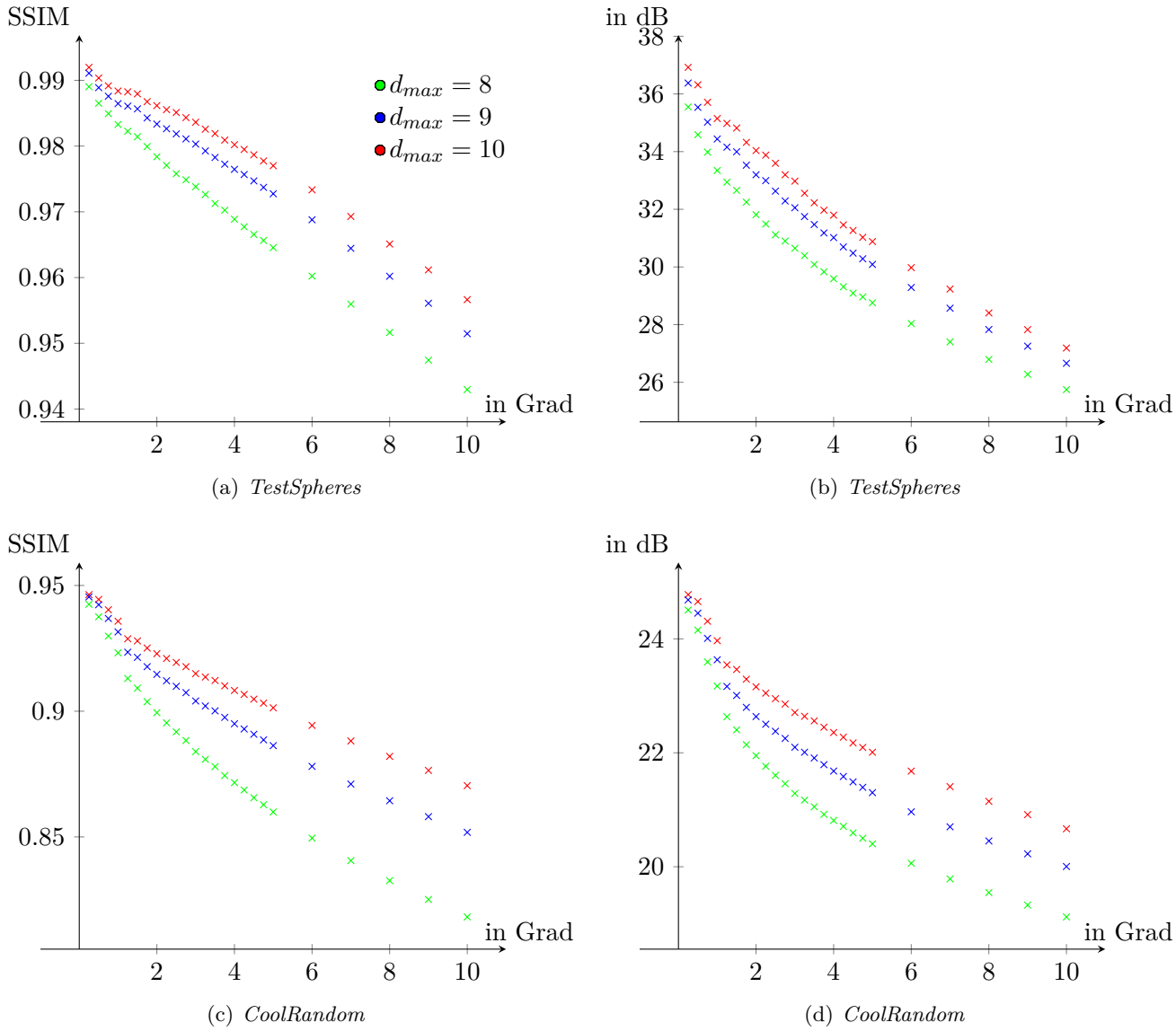


Abbildung 6.3: Die Diagramme a und b zeigen die Ergebnisse des Datensatzes *TestSpheres*, während c und d die Resultate des Datensatzes *CoolRandom* abtragen. Dabei zeigen a und c die Ergebnisse des SSIM, während in b und d die Ergebnisse des PSNR gezeigt werden.

Die Abbildung 6.3 zeigt die Ergebnisse *Quadtrees*-Ansatzes, mit unterschiedlichen maximalen Baumtiefen. Dabei zeigt sich das der unterschied in der Szene *TestSpheres* nur gering ist. Anders fallen die Ergebnisse für die *CoolRandom* Szene aus. Hier zeigt sich ein deutlicher Unterschied zwischen drei Varianten, insbesondere im Intervall von 5 bis 50 Grad. Die Gütewerte für die Baumtiefe $d_{max} = 8$, liegen dabei deutlich unterhalb der anderen. Im Fall der *CoolRandom* Szene zeigt sich bei der Messung mit dem PSNR eine erneute Verbesserung bei Winkelunterschieden, die größer als 70 Grad sind.



(a) *TestSpheres* (b) *TestSpheres*

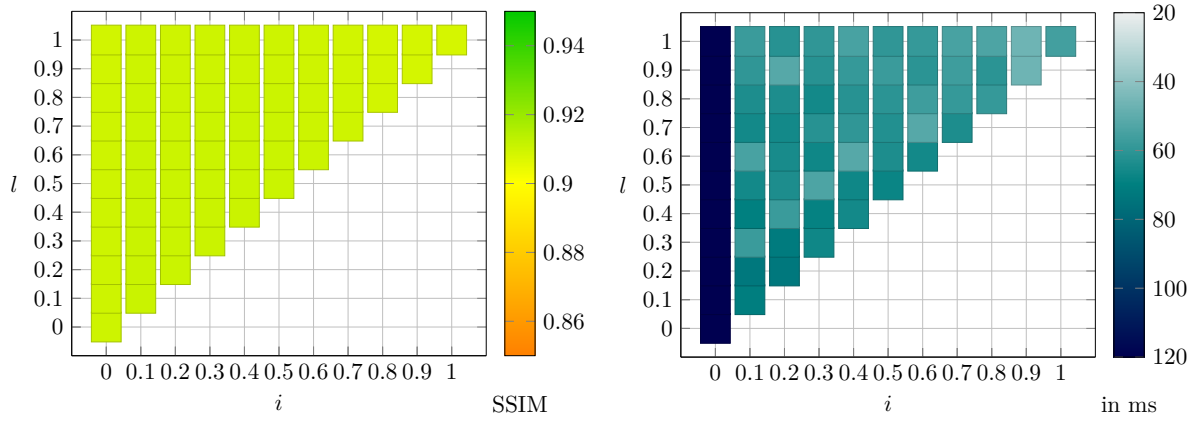
(c) *CoolRandom* (d) *CoolRandom*

Abbildung 6.4: Die Diagramme a und b zeigen die Ergebnisse des Datensatzes *TestSpheres*, während c und d die Resultate des Datensatzes *CoolRandom* abtragen. Dabei zeigen a und c die Ergebnisse des SSIM, während in b und d die Ergebnisse des PSNR gezeigt werden.

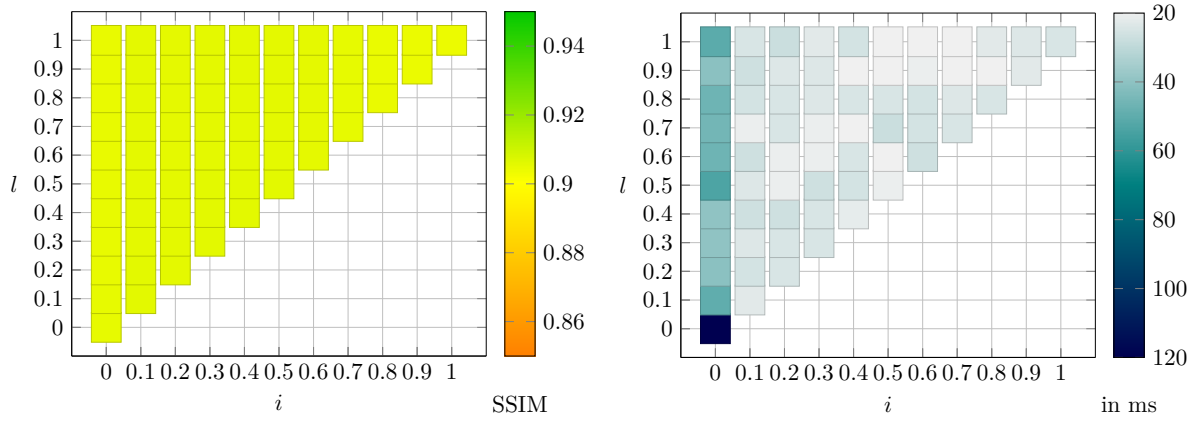
Die Diagramme aus der Abbildung 6.4, zeigen die gleichen Graphen, wie die Abbildung 6.3, allerdings im Intervall von 0 bis 10 Grad. Der Unterschied der Güte zwischen den Parameterkonfigurationen, wird hierbei noch deutlicher.

Um die Schwellwerte l und i zu untersuchen, wird der SSIM von einem konstanten Kamerawinkelunterschied betrachtet. Zu diesem Zweck wird der 20 Grad Winkel gewählt, weil wie in der Abbildung 6.3 zu erkennen, eine höhere Streuung der Werte zu erwarten ist. Dadurch werden die Qualitätsunterschiede zwischen den Parameterkonfigurationen besonders deutlich.

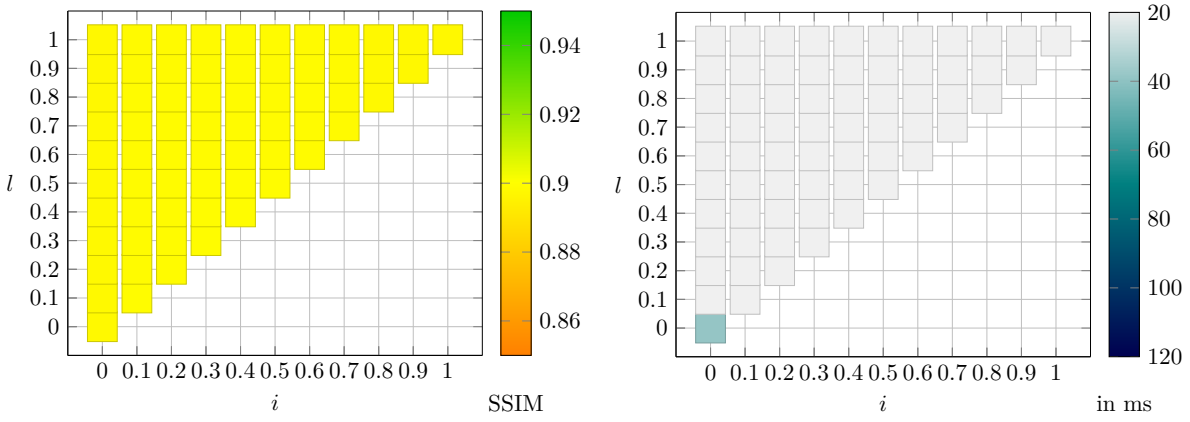
In der Abbildung 6.5 sind die Güte und die Netzkonstruktionszeit des *TestSpheres*-Datensatzes abgebildet. Es ist zu erkennen, dass die Güte mit der maximalen Baumtiefe d_{max} abnimmt. Der Güteunterschied zwischen den einzelnen Konfigurationen von l und i, ist bei konstanter Baumtiefe allerdings gering. Anders sieht es mit der Konstruktionszeit aus, hier zeigt sich deutliche



(a) *TestSpheres*, $d_{max} = 10$ (b) *TestSpheres*, $d_{max} = 10$



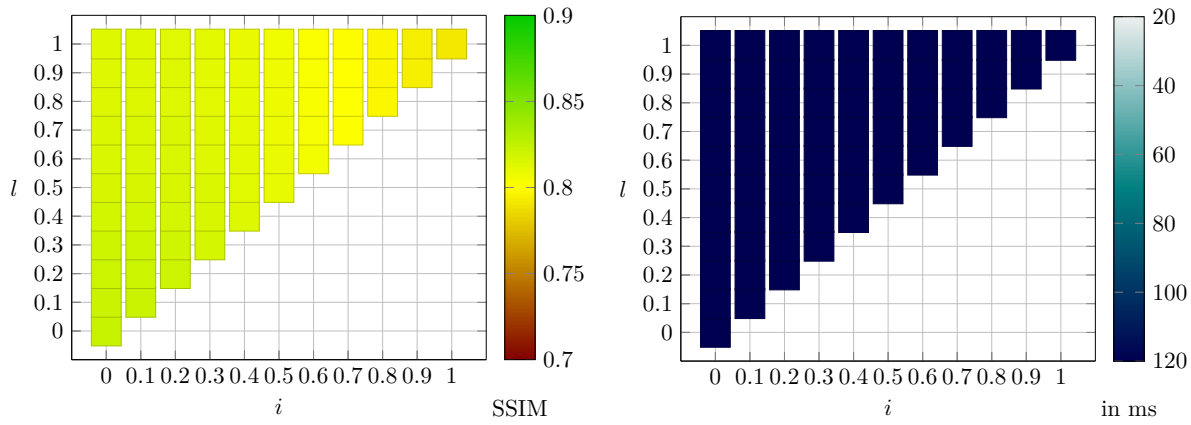
(c) *TestSpheres*, $d_{max} = 9$ (d) *TestSpheres*, $d_{max} = 9$



(e) *TestSpheres*, $d_{max} = 8$ (f) *TestSpheres*, $d_{max} = 8$

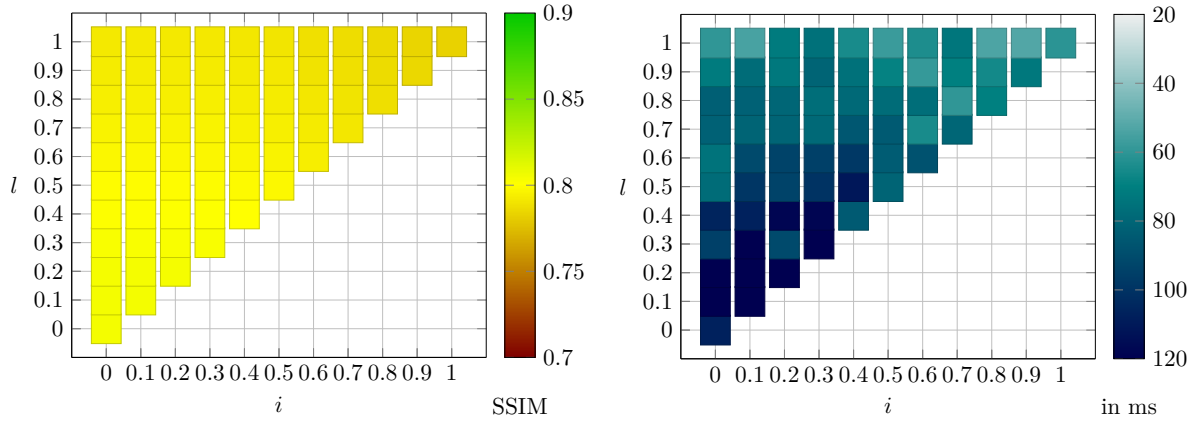
Abbildung 6.5: Ergebnisse des *Quadtree*-Ansatz mit *TestSpheres*Datensatz, bei einem konstanten Kamerawinkelunterschied von 20 Grad. Dabei liegt die maximale Baumtiefe in a,b bei 10, in c,d bei 9 und in e,f bei 8. Die Diagramme auf der linken Seite zeigen die Güte mit Hilfe des SSIM Wertes. Auf der rechten Seite wird die Konstruktionszeit des Netzes abgetragen, die der Server benötigt.

Unterschiede. Bei $d_{max} = 10$ und für $i = 0$ ist die Konstruktionszeit für alle l größer als 120 ms. Für die maximale Baumtiefe $d_{max} = 9$ überschreitet die Konfiguration $l, i = 0$ die Marke von 120 ms.



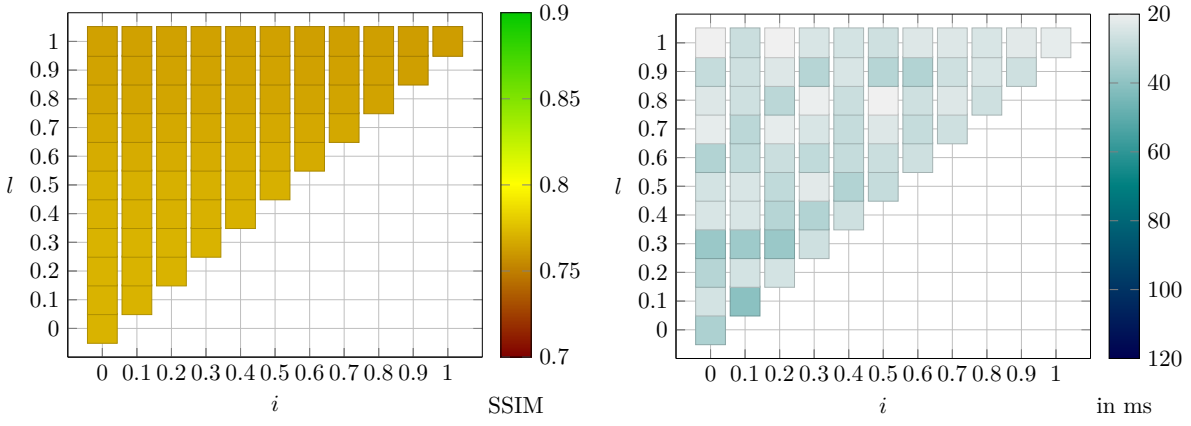
(a) *CoolRandom*, $d_{max} = 10$

(b) *CoolRandom*, $d_{max} = 10$



(c) *CoolRandom*, $d_{max} = 9$

(d) *CoolRandom*, $d_{max} = 9$



(e) *CoolRandom*, $d_{max} = 8$

(f) *CoolRandom*, $d_{max} = 8$

Abbildung 6.6: Ergebnisse des *Quadtree*-Ansatz mit *CoolRandom*, bei einem konstanten Kamerawinkelunterschied von 20 Grad. Dabei liegt die maximale Baumtiefe in a,b bei 10, in c,d bei 9 und in e,f bei 8. Die Diagramme auf der linken Seite zeigen die Güte mit Hilfe des SSIM Wertes. Auf der rechten Seite wird die Gesamtkonstruktionszeit des Netzes abgetragen.

Vergleichbar zur Abbildung 6.5, werden Güte und Konstruktionszeit, in der Abbildung 6.6 für den *CoolRandom*-Datensatz dargestellt. Die Güte variiert wesentlich stärker, als im *TestSpheres*-Datensatz. Zu erkennen ist eine Abnahme der Güte mit steigenden l und i, wenn die maximale Baumtiefe $d_{max} = 10$ beträgt. Für die anderen beiden maximalen Baumtiefen 8 und 9, ist die Güte für alle i und l relativ konstant. Auch bei dieser Szene zeigt sich das bei $d_{max} = 10$ die Konstruktionszeiten am größten sind. Unabhängig von l und i überschreiten alle die Marke von 120 ms.

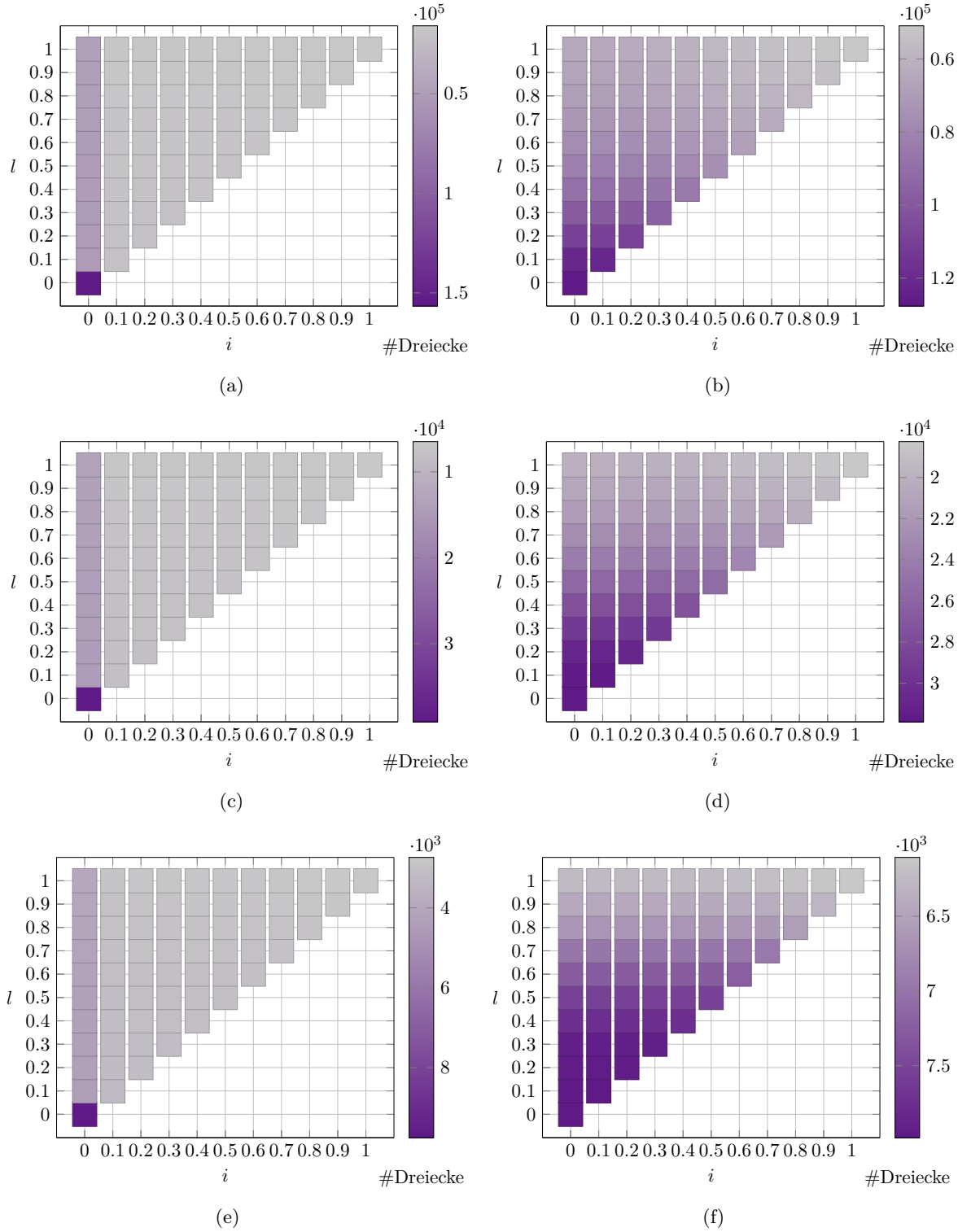


Abbildung 6.7: Die Diagramme zeigen für den *Quadtree*-Ansatz, bei einem konstanten Kamerawinkelunterschied von 20 Grad die Anzahl der Dreiecke in Abhängigkeit der Parameter l, i und d_{max}. Dabei liegt die maximale Baumtiefe in a,b bei 10, in c,d bei 9 und in e,f bei 8. Die Diagramme auf der linken Seite zeigen die Anzahl der Dreiecke von der *TestSpheres* Szene. Auf der rechten Seite wird die Anzahl der Dreiecke von *CoolRandom* dargestellt.

In der Abbildung 6.7 ist zu sehen, dass die Anzahl der Dreiecke mit der kleiner werdender Baumtiefe sinkt. Die Dreiecksanzahl von Netzkonfigurationen des *TestSpheres*-Datensatzes ist dabei für $i = 0$ größer als für alle anderen. Am größten ist diese, wenn auch $l = 0$ ist. Ein Ähnliches Bild ergibt sich für den *CoolRandom*-Datensatz. Hier ist der Unterschied zwischen den verschieden l und i gleichmäßiger. Wenn sowohl l, als auch i kleiner werden, steigt die Anzahl der Dreiecke.

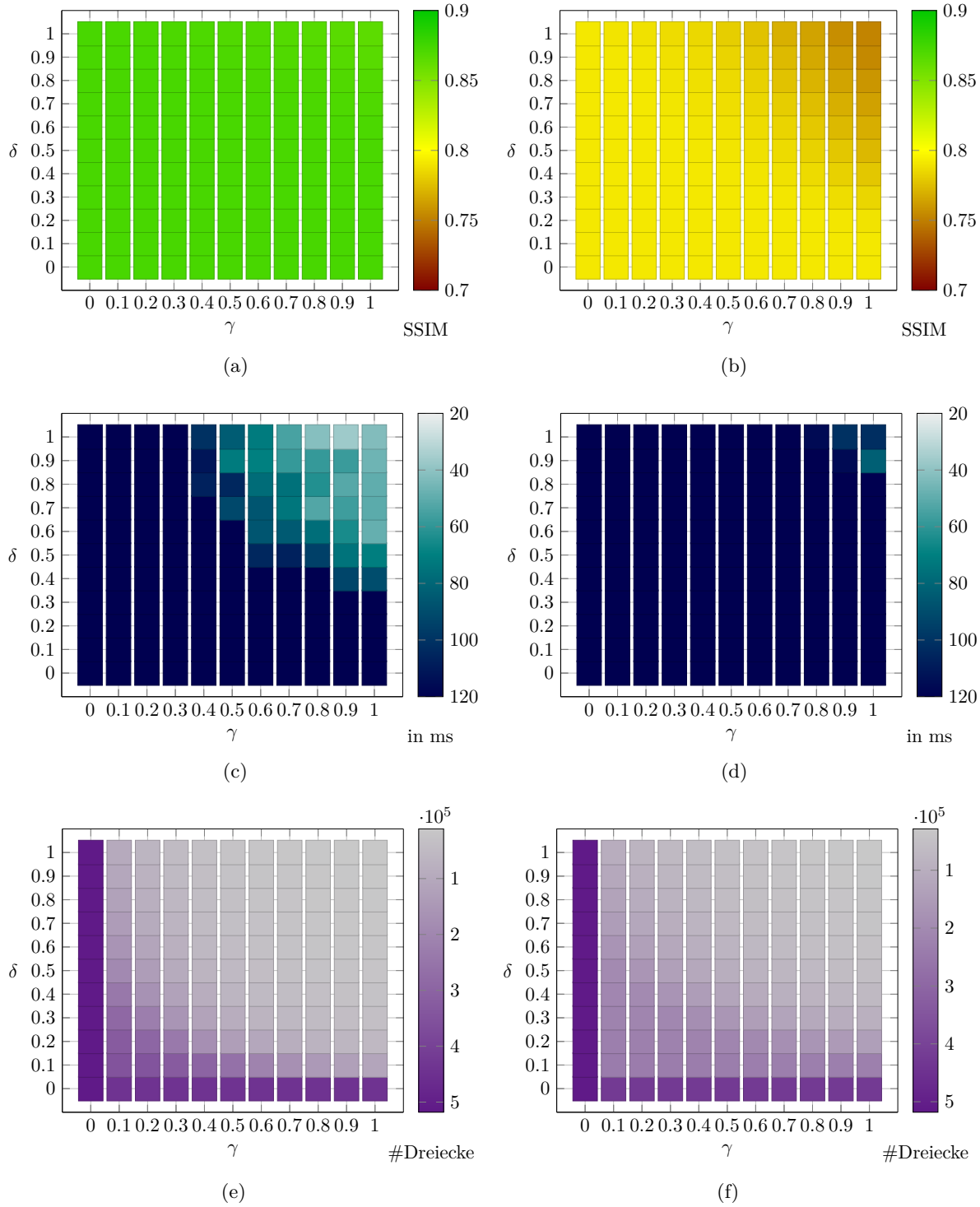
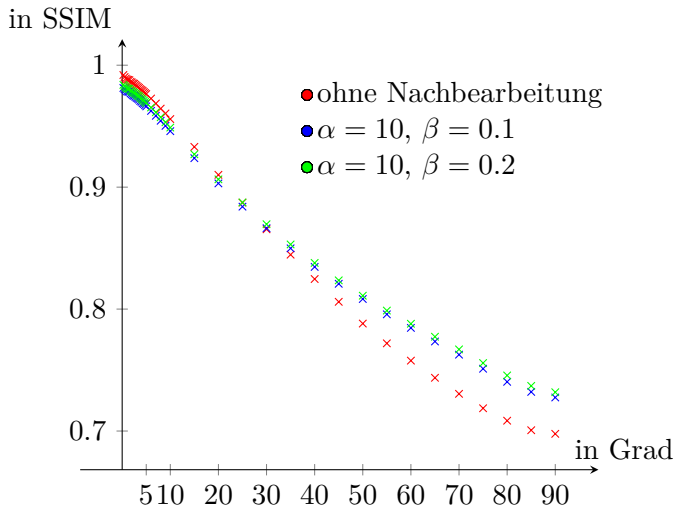


Abbildung 6.8: Die Diagramme a, b stellen die Ergebnisse von dem *CoolRandom* Datensatz und c, d die des *TestSpheres* Datensatzes erhoben mit FloydSteinberg Ansatz dar. Die Diagramme auf der linken Seite zeigen die Güte mit Hilfe des SSIM und die Diagramme auf der rechte Seite bilden die Gesamtkompressionszeit des Netzes ab.

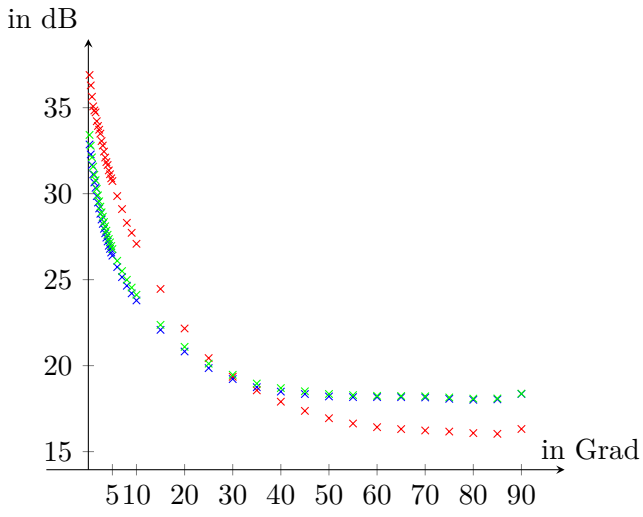
Analog zu den Schwellwerten der *Quadtree*-Methode, werden die Parameter t und g, vom *Floyd-Steinberg*-Verfahren abgebildet 6.8. Dabei sind die Ergebnisse 6.8, für die Szene *TestSpheres* hinsichtlich der Güte, für alle Konfigurationen sehr gut. Bei der Bereichnungszeit zeigt sich,

dass erst ab den Konfigurationen $\delta > 0.6$ und $\gamma > 0.5$, die Konstruktionszeiten für eine interaktive Visualisierung klein genug sind. Die Anzahl der Dreiecke ist bei kleinen Werten für γ oder für δ am Höchsten. Anders sieht es bei dem Datensatz *CoolRandom* aus. Die Gütewerte liegen alle unterhalb von 0.8, dabei zeigt sich eine zunehmende Verschlechterung für Konfigurationen mit $\delta > 0.6$ und $\gamma > 0.5$. Die Konstruktionszeiten des Netzes sind für alle Parameterkonfigurationen zu groß. Mit der Anzahl der Dreiecke ist dabei ähnlich verteilt wie für die Szene *TestSpheres*.

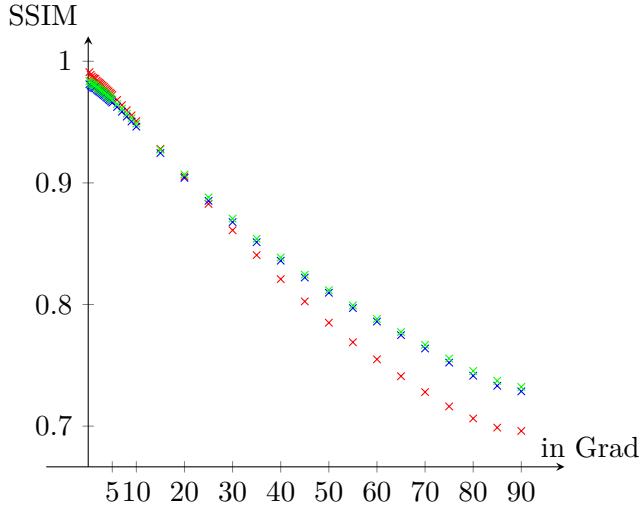
Der Netzoptimierungsschritt wird ausschließlich mit dem *Quadtree*-Ansatz untersucht, da dessen Ergebnisse weitaus besser sind, als die der FloydSteinberg-Methode. Um die Parameter α und β auszuwerten, werden l und i konstant auf einen Wert fixiert. Zu diesem Zweck wurde $l = 0.7$ und $i = 0.6$ gewählt.



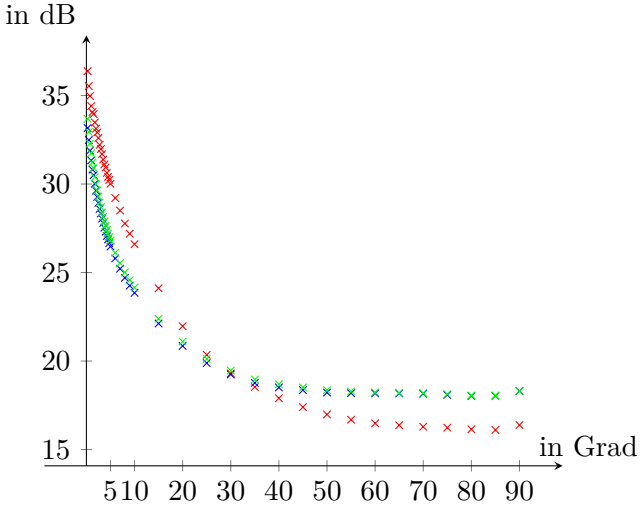
(a) *TestSpheres*, $d_{max} = 10$



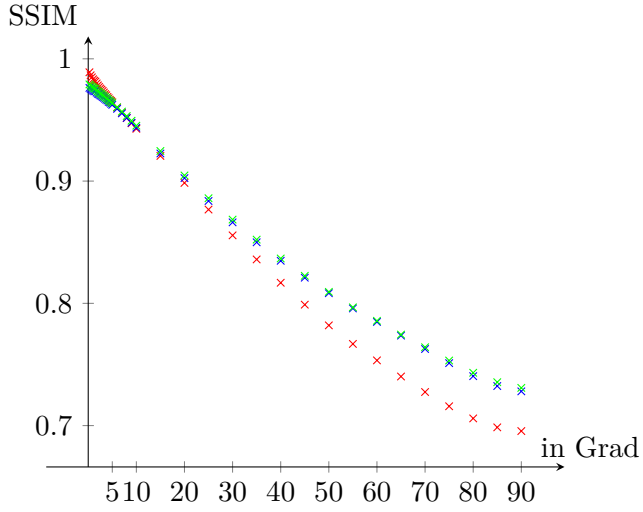
(b) *TestSpheres*, $d_{max} = 10$



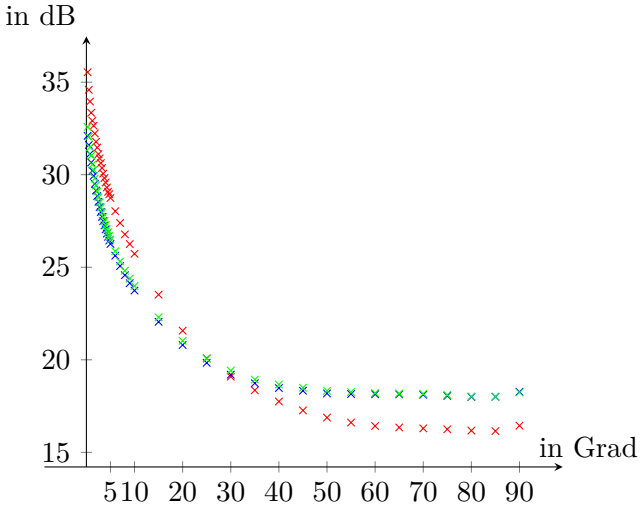
(c) *TestSpheres*, $d_{max} = 9$



(d) *TestSpheres*, $d_{max} = 9$

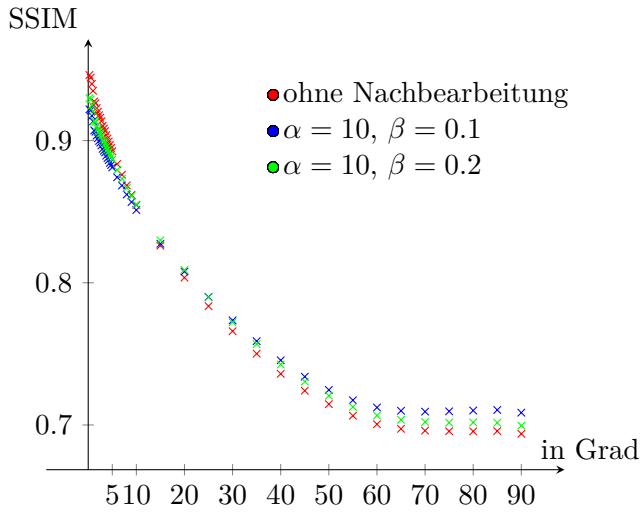


(e) *TestSpheres*, $d_{max} = 8$

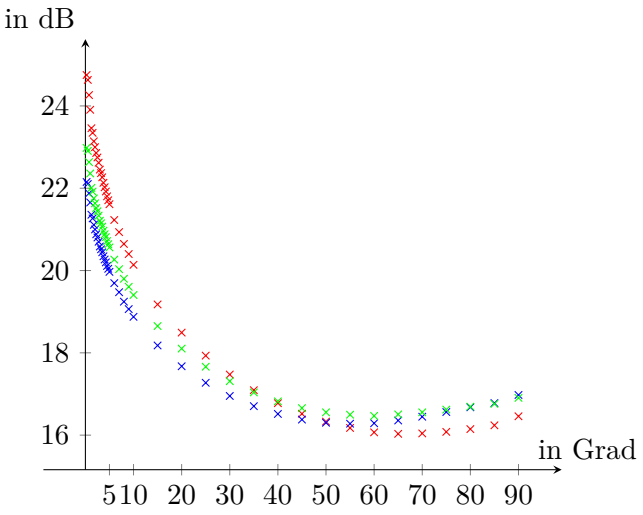


(f) *TestSpheres*, $d_{max} = 8$

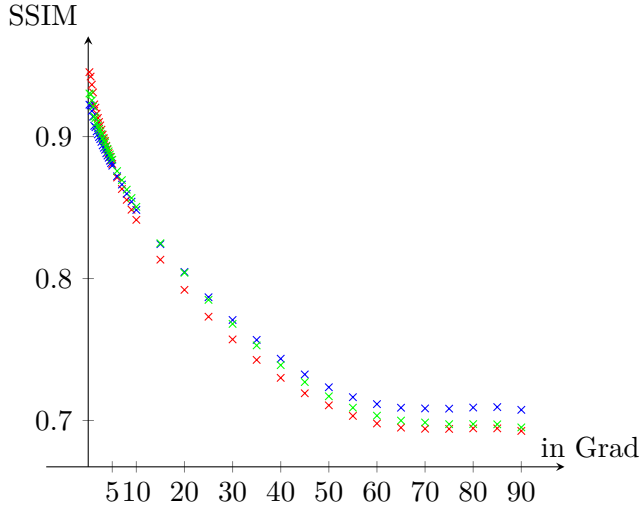
Abbildung 6.9: Es wird der Datensatz *TestSpheres* betrachtet. Die Diagramme zeigen die Güteentwicklung, in Abhängigkeit des Parameters j. Dabei liegt die maximale Baumtiefe in a,b bei 10, in c,d bei 9 und in e,f bei 8.



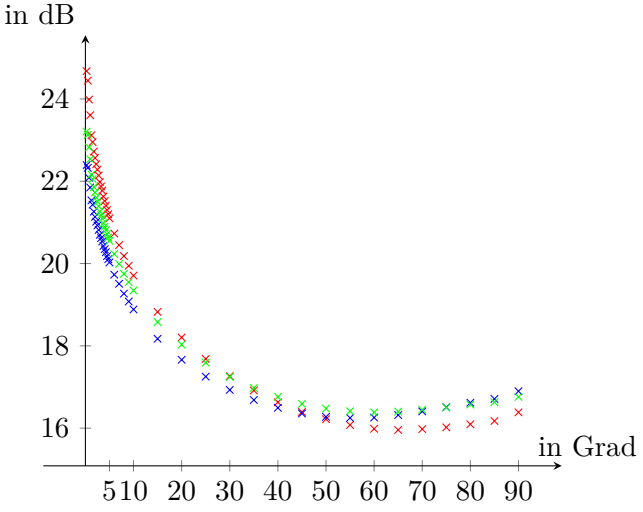
(a) *CoolRandom*, $d_{max} = 10$



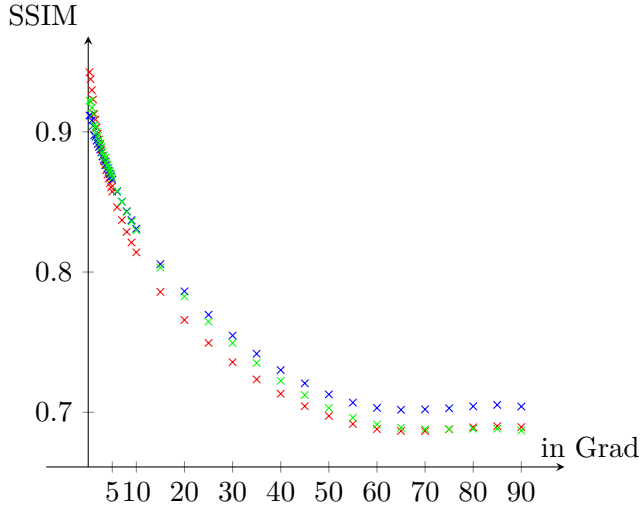
(b) *CoolRandom*, $d_{max} = 10$



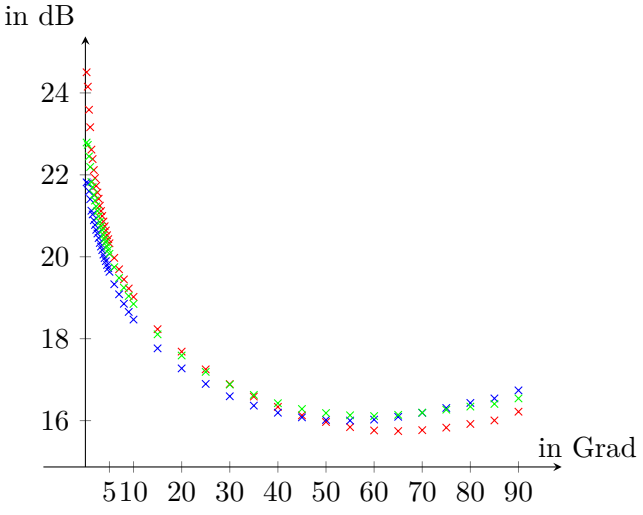
(c) *CoolRandom*, $d_{max} = 9$



(d) *CoolRandom*, $d_{max} = 9$



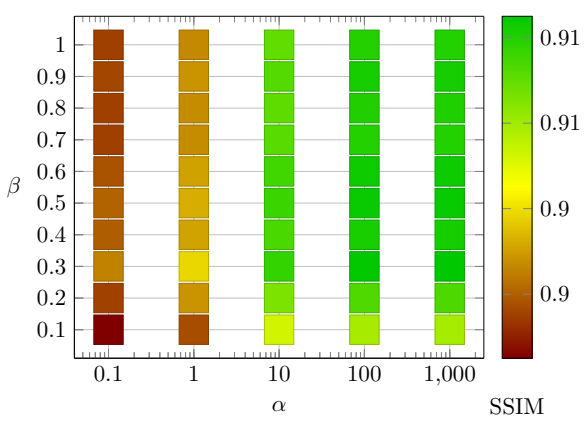
(e) *CoolRandom*, $d_{max} = 8$



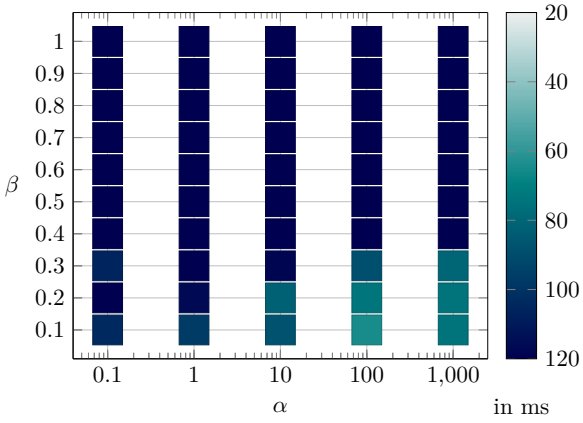
(f) *CoolRandom*, $d_{max} = 8$

Abbildung 6.10: Es wird der Datensatz *CoolRandom* betrachtet. Die Diagramme zeigen die Güteentwicklung, in Abhängigkeit des Parameters j. Dabei liegt die maximale Baumtiefe in a,b bei 10, in c,d bei 9 und in e,f bei 8.

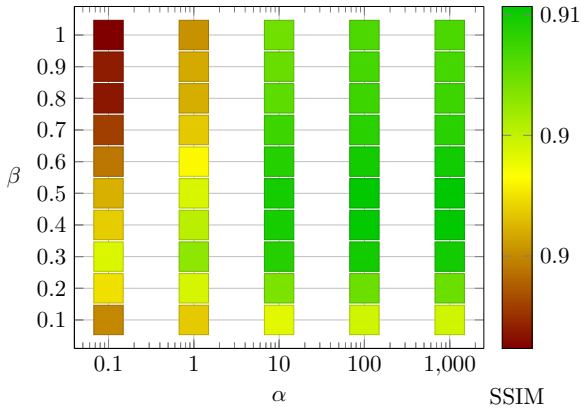
Die Diagramme aus der Abbildungen 6.9 und 6.10 stellen dar, wie sich der Netzoptimierungsschritt auf die Dreiecksnetze hinsichtlich der Güte auswirkt. Zum Vergleich dient das nicht nach bearbeitete Netz. Dabei wird $\alpha = 10$ gesetzt.



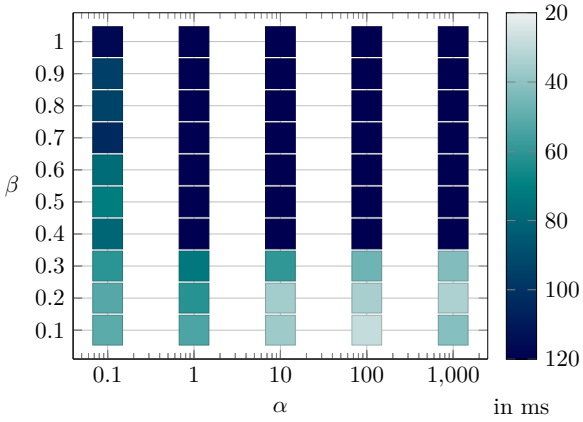
(a) *TestSpheres*, $d_{max} = 10$



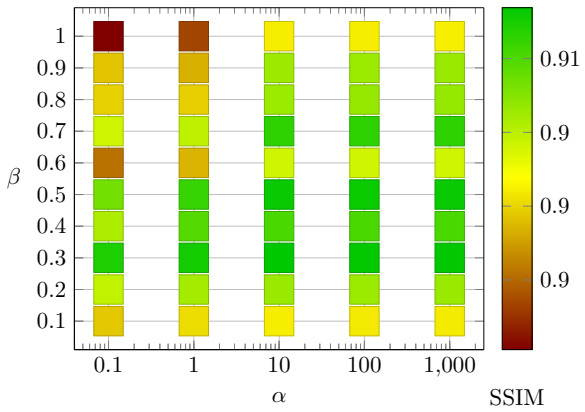
(b) *TestSpheres*, $d_{max} = 10$



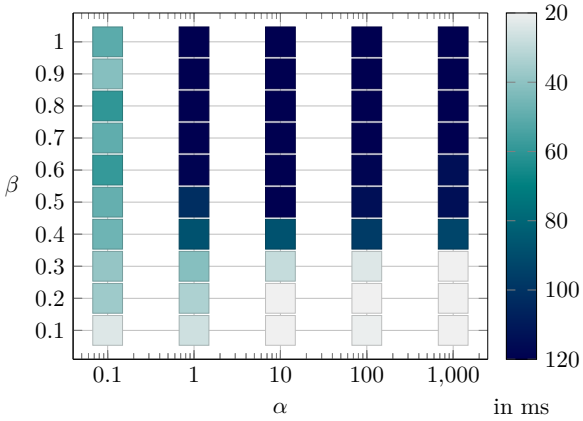
(c) *TestSpheres*, $d_{max} = 9$



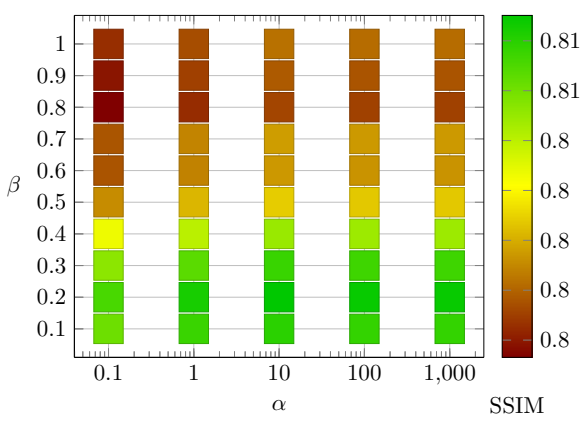
(d) *TestSpheres*, $d_{max} = 9$



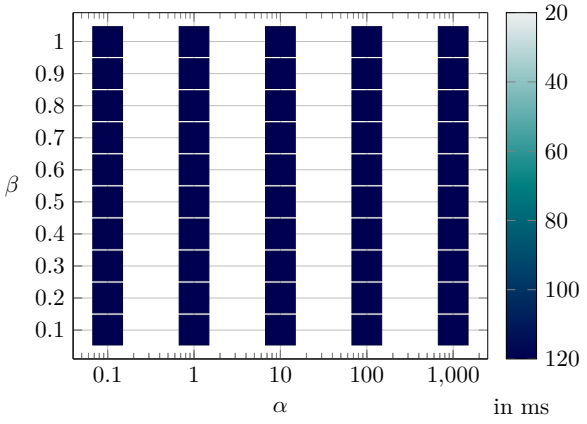
(e) *TestSpheres*, $d_{max} = 8$



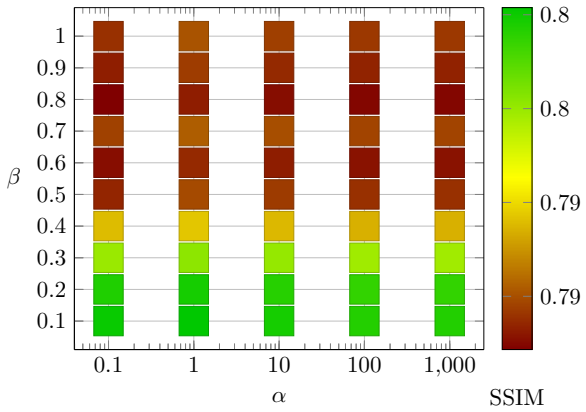
(f) *TestSpheres*, $d_{max} = 8$



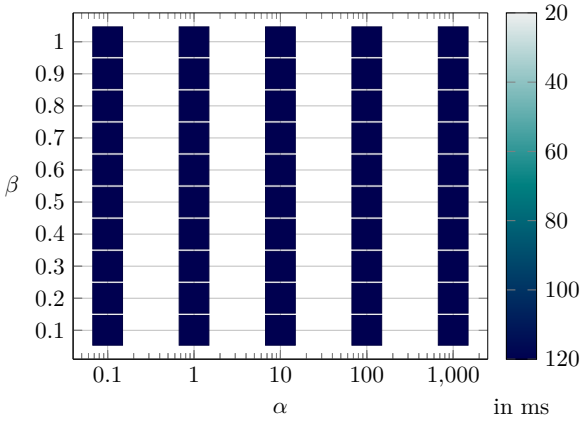
(g) *CoolRandom*, $d_{max} = 10$



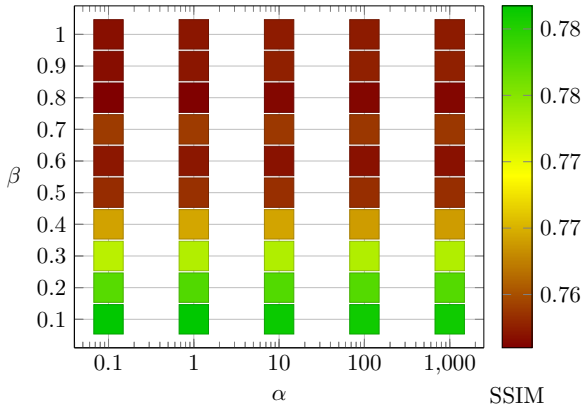
(h) *CoolRandom*, $d_{max} = 10$



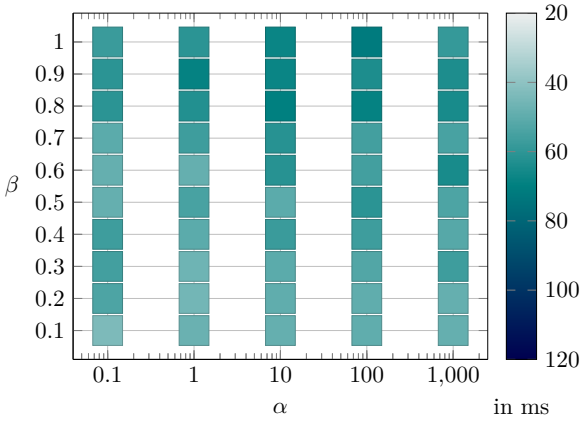
(i) *CoolRandom*, $d_{max} = 9$



(j) *CoolRandom*, $d_{max} = 9$

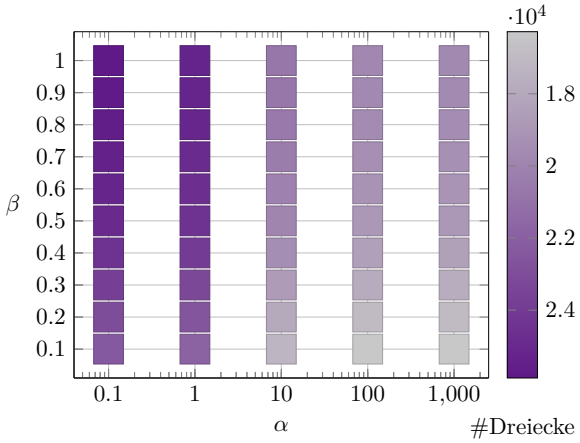


(k) *CoolRandom*, $d_{max} = 8$

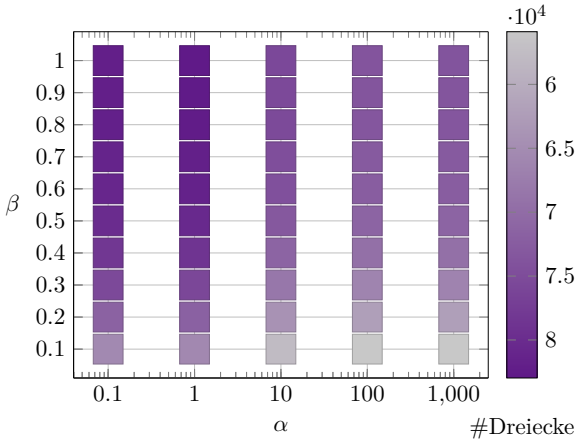


(l) *CoolRandom*, $d_{max} = 8$

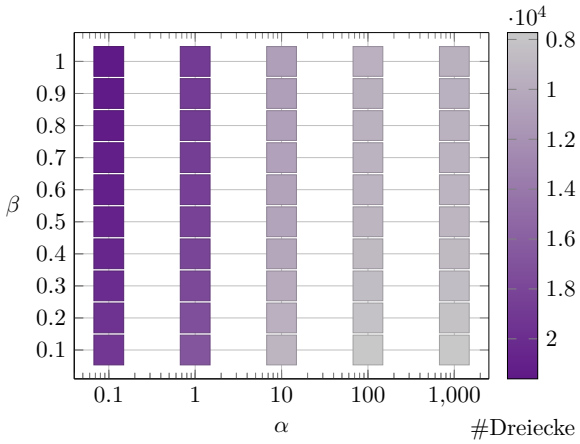
Abbildung 6.11: SSIM und Netzkonstruktionszeit, bei einem konstanten Winkelunterschied von 20 Grad.



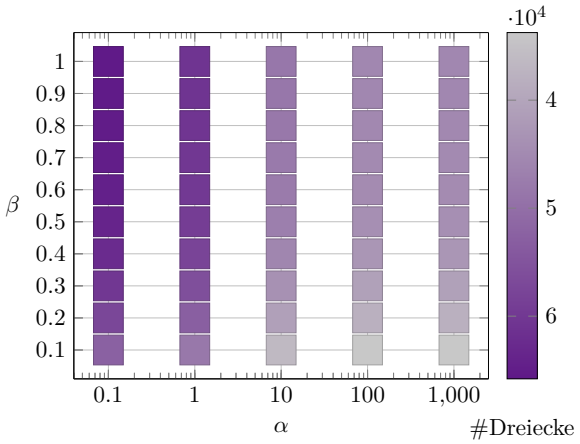
(a) *TestSpheres*, $d_{max} = 10$



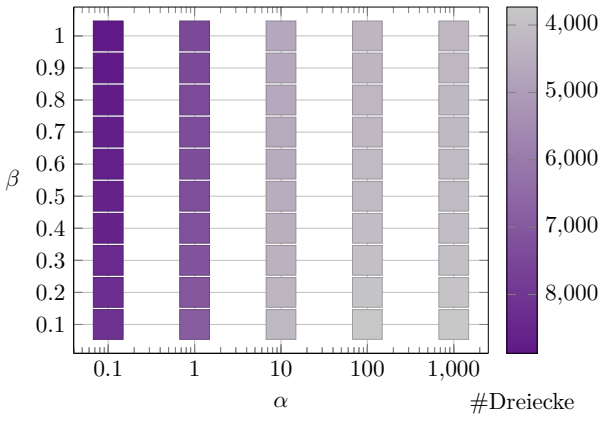
(b) *CoolRandom*, $d_{max} = 10$



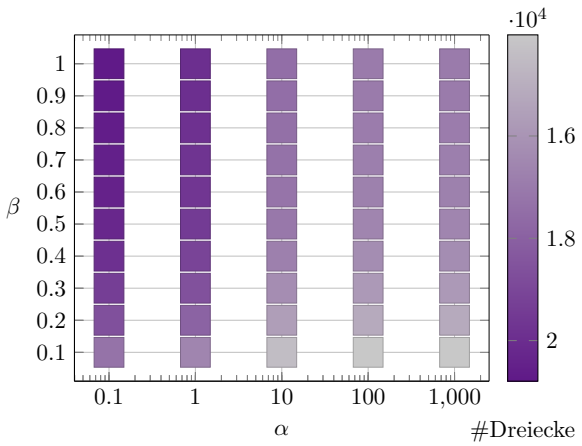
(c) *TestSpheres*, $d_{max} = 9$



(d) *CoolRandom*, $d_{max} = 9$



(e) *TestSpheres*, $d_{max} = 8$



(f) *CoolRandom*, $d_{max} = 8$

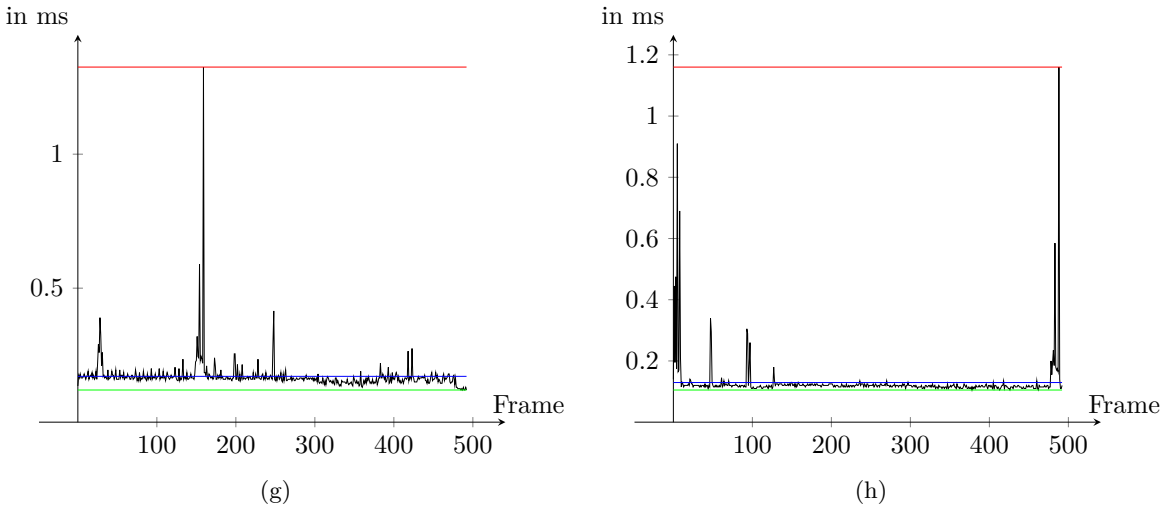


Abbildung 6.12: Die Diagramme stellen die Zeit, die der Client, zur Extrapolation der Frames benötigt grafisch dar. Dabei ist im Diagramm a die Vollvernetzung zu sehen und im Diagramm b die Delaunay-Triangulation.

7 Diskussion

Das Ziel bestand darin, ein Verfahren zu entwickeln, das den Werteverlauf von Tiefenbildern, mit Hilfe von Dreiecksnetzen approximiert und die Güte, in Abhängigkeit des Kamerawinkelunterschieds zum Originalbild, zu evaluieren. Zu diesem Zweck wurden mit einem *Quadtree* beziehungsweise der Floyd-Steinberg Methode eine Menge von Punkten bestimmt. Diese wird mit Hilfe der Delaunay-Triangulierung zu einem Dreiecksnetz verknüpft. In einen weiteren Schritt, kann die Triangulierung optimiert werden.

Die besten Resultate, der verschieden Varianten liefern die in der Abbildung 6.1 abgebildeten Parameterkonfigurationen. Wobei diese auch die meisten Dreiecke produzieren. Es zeigt sich sehr deutlich, das die Güte der Ergebnisse von der Szene und ihrem Inhalt abhängt.

8 Zusammenfassung

9 Ausblick

Abbildungsverzeichnis

Tabellenverzeichnis

Danksagung

Die Danksagung...