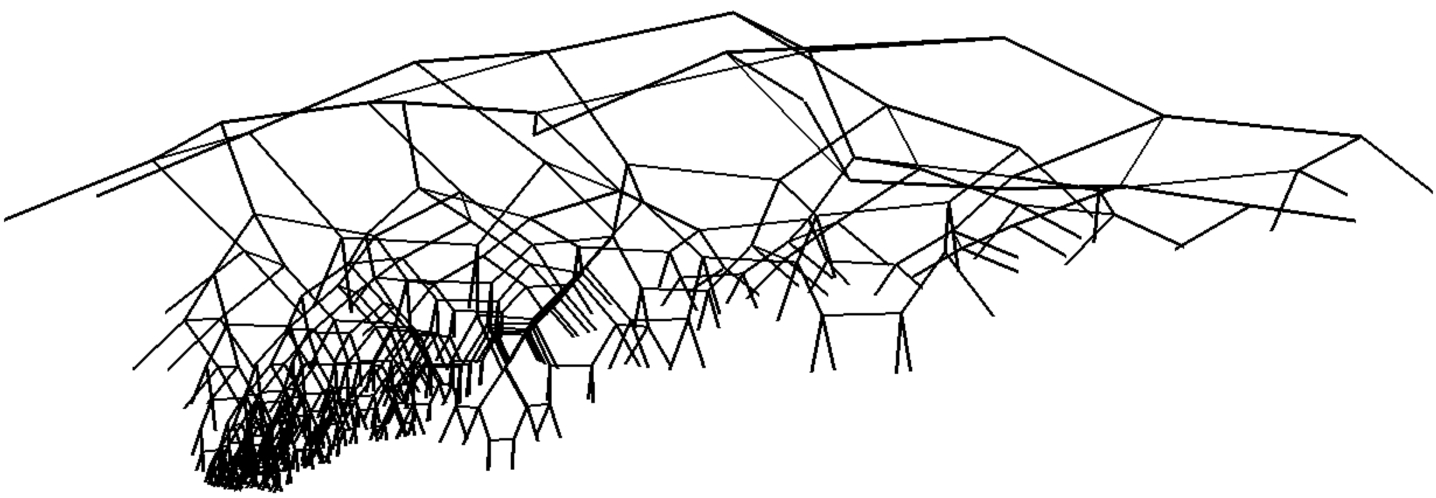
Institut für Informatik
der
Technischen Universität München

Entwicklung und Implementierung adaptiver Datenstrukturen und Algorithmen zur effizienten Visualisierung und Speicherung digitaler Höhenmodelle

Andreas Paul



Institut für Informatik
der
Technischen Universität München

Entwicklung und Implementierung adaptiver Datenstrukturen und Algorithmen zur effizienten Visualisierung und Speicherung digitaler Höhenmodelle

Andreas Paul

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.–Prof. Dr. J. Schlichter
Prüfer der Dissertation:	1. Univ.–Prof. Dr. Chr. Zenger
	2. Univ.–Prof. Dr. H.–J. Bungartz, Universität Augsburg

Die Dissertation wurde am 2.10.2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 24.11.2000 angenommen.

Inhaltsverzeichnis

1 Einleitung

Viele Anwendungen der Visualisierung machen es erforderlich, Höhenmodelle oder vergleichbare Daten effizient auf dem Bildschirm darzustellen. Unter Höhenmodellen bzw. Höhenfeldern versteht man dabei üblicherweise auf einem regelmäßigen zweidimensionalen Gitter angeordnete Höhenwerte. Die Gitterkoordinaten der Punkte können entweder explizit gegeben oder implizit, durch Informationen zum Gitter, wie dem Abstand der Gitterpunkte sowie der Anzahl der Gitterpunkte in horizontaler bzw. vertikaler Richtung, festgelegt sein. Höhenfelder beschreiben so den Verlauf einer Oberfläche über einer meist rechteckigen Grundfläche. Eine andere Sichtweise wäre, Höhenfelder als Wertetabelle einer zweidimensionalen, reellwertigen Funktion zu betrachten. Neben dieser Art von Höhenmodell ist auch noch die Darstellung von Höheninformationen durch Höhenlinien bzw. Isolinien statt durch ein Höhenfeld gebräuchlich. Letztere, in verschiedenen Bereichen eingesetzte, Darstellungsform stellt in der Regel keine besonders hohen Anforderungen an die Visualisierung. Die Daten sind als zweidimensionaler Polygonzug vorgegeben, der eine bestimmte Höhe hat und können auch einfach als Polygonzug auf dem Bildschirm gezeichnet werden. Auch die Leistung schwächerer Graphikhardware reicht heute aus, um umfangreiche Datensätze von Höhenlinien leicht visualisieren zu können. Diese Arbeit konzentriert sich auf Höhenmodelle, die eine Oberfläche mit Hilfe von Höhenwerten über einem regelmäßigen Gitter beschreiben.

Durch digitale Höhenmodelle werden in der Regel reale oder fiktive Gelände bzw. Landschaften repräsentiert. Analog behandelt werden können auch Grauwertbilder, die durch Pixel beschrieben werden. Bildformate, die diese Beschreibungsform einsetzen, sind z.B. TIFF, JPEG, PNG oder GIF. Auch hier liegt den Daten, in der Regel implizit, ein regelmäßiges, zweidimensionales, rechteckiges Gitter zugrunde. Die Gitterpunkte entsprechen den Pixeln, und was bei Höhenmodellen der Höhenwert ist, macht bei Grauwertbildern die Helligkeit des einzelnen Pixels aus. Den reinen Daten sieht man nicht notwendigerweise unmittelbar an, von welcher Art sie sind, der Unterschied besteht in der Visualisierung der Daten. Höhendaten werden meist als dreidimensionales Dreiecksnetz dargestellt, während Bilddaten durch die Farbe bzw. Helligkeit der Pixel auf dem Bildschirm dargestellt werden. Aufgrund dieser Verwandtschaft der Daten können Höhendaten leicht als Pixelbild und Bilder als dreidimensionales Dreiecksnetz visualisiert werden. Jeder Datensatz, der den Punkten eines zweidimensionalen, regelmäßigen Gitters einen Wert zuordnet, fällt in diese Kategorie und kann auf die eine oder andere Weise visualisiert werden. Die in dieser Arbeit entwickelten Algorithmen und Datenstrukturen können auf jeden Vertreter dieser Kategorie angewendet werden, was zu einem breiten Anwendungsspektrum führt.

Die effiziente Darstellung von realen oder fiktiven Landschaften spielt bei vielen Anwendungen eine wesentliche Rolle. Mit effizienter Darstellung ist dabei die Visualisierung in Echtzeit gemeint. Das heißt, daß die Zeit, die zur Erzeugung der Visualisierung auf dem Bildschirm benötigt wird, für den Benutzer zu keiner merklichen bzw. störenden Verzögerung führen darf.

Anwendungen, die Echtzeit–Visualisierung von Höhendaten erforderlich machen, sind z.B. Flugsimulatoren, Computerspiele oder die am Lehrstuhl für Flugmechanik und Flugregelung der TU München entwickelte *virtuelle Sicht* (siehe [32]). Bei letzterer handelt es sich um eine Anwendung, die vom Standpunkt der Geländedarstellung mit Flugsimulatoren und manchen Computerspielen eng verwandt ist. Die virtuelle Sicht stellt jedoch deutlich höhere Ansprüche an die Qualität und die Geschwindigkeit der Geländevisualisierung als Spiele oder Simulatoren. Sie stellt dem Piloten eines Flugzeugs eine, vom Computer erzeugte, graphische Repräsentation des umgebenden Geländes als Navigationshilfe, zur Verfügung. Je nach Position und Lage des Flugzeugs sowie gegebenenfalls der Blickrichtung des Piloten muß eine Visualisierung des sichtbaren Geländes aus einer Datenbasis erstellt werden, die das reale Gelände möglichst gut repräsentiert. Da sich im Flug die Position, Lage und Blickrichtung kontinuierlich ändern, muß die virtuelle Sicht möglichst schnell, d.h. in Echtzeit, aktualisiert werden, damit der Pilot sich nicht anhand von veralteten Informationen orientiert und das Flugzeug durch das Gelände manövriert. Die Bedeutung der Echtzeitvisualisierung des Geländes zeigt sich darin, daß in verschiedenen Testszenarien, unter anderem auch während einer Landung, dem Piloten, neben der virtuellen Sicht, keine weiteren Orientierungshilfen zur Verfügung stehen, auch nicht der Blick aus dem Cockpitfenster (vergleiche [35] und [37]).

Die Datenbasis, aufgrund der die virtuelle Sicht die graphische Repräsentation des Geländes erzeugt, besteht unter anderem aus Höhenwerten, die im Bereich von Süddeutschland einen Abstand der Gitterpunkte von 60 Metern horizontal und 90 Metern vertikal aufweisen. Eine Visualisierung des Geländes, bei der jeder Höhenwert des Datensatzes bei der Erzeugung des Dreiecksnetzes, das zur Visualisierung der Geländedaten eingesetzt wird, berücksichtigt wird, stößt auch bei relativ kleinen Gebieten, die noch groß genug sind, um in ihnen Flugversuche durchführen zu können, schnell an die Leistungskapazität moderner Graphikhardware. Aus diesem Grund ist die Reduktion der Anzahl der zur interaktiven Visualisierung eingesetzten Dreiecke unumgänglich. Vor allem dann, wenn die Geländevisualisierung auf einem Rechner laufen soll, der zum einen leicht genug sein muß, um sich nicht signifikant auf die Betriebskosten auch kleiner Flugzeuge niederzuschlagen, zum anderen in seiner Anschaffung kostengünstig sein soll. Damit scheidet der Einsatz von Graphik–Supercomputern und High–End–Rechnern aus.

Die Erzeugung des Dreiecksnetzes zur Geländevisualisierung, womit sich diese Arbeit beschäftigt, wird auch als Geländetriangulierung bezeichnet. Im Rahmen der virtuellen Sicht muß das Gelände zum einen schnell genug trianguliert werden, damit die Visualisierung in Echtzeit erfolgen kann, zum anderen muß die Darstellungsqualität gut genug sein, damit sich ein Pilot anhand der erzeugten Geländeapproximation orientieren kann. Es gibt zahlreiche Algorithmen zur Erzeugung von Geländetriangulierungen, die die Anzahl der Dreiecke stark reduziert und durch eine Fehlerschranke sicherstellen, daß die erzeugte Geländeapproximation eine gewisse Qualität nicht unterschreitet. Zu den erfolgreichsten veröffentlichten Verfahren gehören die von Lindstrom [25] und der ROAM–Algorithmus [7]. Die Anzahl der Dreiecke wird bei diesen und anderen Algorithmen unter Verwendung bestimmter Eigenschaften des Geländes, meist seiner Glattheit, sowie der Eigenheiten der graphischen Darstellung durch die Zentralprojektion und der angestrebten Anwendung reduziert. Die perspektivische Darstellung führt dazu, daß vom Betrachter weit entfernte Geländeteile auf dem Bildschirm sehr klein abgebildet werden und aufgrund der Anwendung auch nur von geringerem Interesse für diesen sind. Die relative Anzahl der erzeugten Dreiecke nimmt aus diesen Gründen mit dem Abstand des gerade triangulierten Geländeteils vom Betrachter ab. So wird ein Level–of–Detail Konzept realisiert, das im Bereich der Computergraphik die mit dem Abstand zum Betrachter zunehmend ungenauere bzw. gröbere Darstellung von Objekten bezeichnet. Bei Geländen wird die Umsetzung des Level–of–Detail Konzeptes dadurch erschwert, daß diese im Prinzip ein einzelnes Objekt darstellen, das aufgrund seiner Ausdehnung und Lage so groß ist, daß es mehrere Level–of–Detail schneidet. Die Probleme entstehen an den Grenzen zweier Level–of–Detail, an denen es zu Lücken bzw. den sogenannten vertikalen Löchern im visualisierten Gelände kommen kann. Dieser Effekt stellt ein wesentliches Defizit der ursprünglichen Implementierung der virtuellen Sicht am Lehrstuhl für Flugmechanik und Flugregelung dar (siehe [28]). Aktuelle Algorithmen zur Geländetriangulierung mit kontinuierlichem Level–of–Detail wie [25], [7], [20] oder [34] setzen verschiedene effektive Techniken ein, um Löcher bei der Triangulierung zu verhindern. Die erzeugten Dreiecksnetze gleichen sich bei den erfolgreichsten Algorithmen, da diese alle auf der gleichen Triangulierungsstrategie basieren, die auf dem Divide&Conquer–Prinzip basiert (siehe [25] und [7]). Sie wird durch eine Hierarchie von Dreiecken repräsentiert, die auch in dieser Arbeit eingesetzt wurde, so daß sich die erzeugte Triangulierung wenig bis gar nicht von der anderer Standardverfahren unterscheidet. Der Unterschied zu den bereits existierenden Verfahren ist, daß diese meist auf und für Graphik–Supercomputern entwickelt wurden und auch nur auf Rechnern dieser Leistungsklasse befriedigende Leistung bieten, während die hier entworfenen und implementierten Verfahren explizit und von vornherein für den Einsatz auf Rechnern der mittleren bis unteren Leistungsklasse konzipiert wurden. Um auch auf schwächeren Systemen ausreichende Ergebnisse erzielen zu können wurde auf die weitreichende Skalierbarkeit hierarchisch adaptiver Verfahren zurückgegriffen, die es ermöglicht, die zur Verfügung stehenden Ressourcen optimal auszunutzen, wobei bereits geringe Leistung für gute Ergebnisse ausreicht.

Die Meßlatte für die im Rahmen dieser Arbeit reimplementierte virtuelle Sicht war der Einsatz des Systems bei Testflügen, wobei sich der Pilot im Tiefflug ausschließlich anhand der virtuellen Sicht orientieren konnte. Der erfolgreiche Abschluss dieser Testflüge, unter Einsatz eines Rechners, der mehrere Leistungsklassen unterhalb des bei der ursprünglichen Implementierung der virtuellen Sicht eingesetzten Graphik–Supercomputers einzuordnen ist, zeigt die Tauglichkeit der entwickelten Geländetriangulierung unter realen Einsatzbedingungen.

Im zweiten Teil der Arbeit wird die für die virtuelle Sicht entwickelte Geländetriangulierung erweitert, so daß Gelände, deren Datenvolumen den Hauptspeicher jedes eingesetzten Rechners sprengen würde, interaktiv überflogen werden können, wobei Daten dynamisch nachgeladen und freigegeben werden. Dabei wird mit der Einführung einer Metahierarchie von Kacheln über dem Dreiecks–Binärbaum, in dem die Daten abgelegt sind, ein Konzept aufgezeigt und umgesetzt, das das dynamische Nachladen von Höhendaten und deren Triangulierung integriert, d.h. in einem Durchgang durchführt, wobei konsequent dem hierarchischen Ansatz gefolgt wurde. Systeme, die die Visualisierung beliebig großer Höhenfelder erlauben, arbeiten bisher z.B. mit diskreten, separat gespeicherten Level–of–Detail, die über eine Datenbank organisiert und nachgeladen werden (vergleiche [33]). Andere integrierte Visualisierungssysteme wie [24] nutzen zwar vergleichbare Triangulierungstechniken, gehen dabei aber nicht konsequent hierarchisch vor und sind wiederum ausschließlich auf Graphik–Supercomputern der obersten Leistungsklasse einsetzbar.

Die entwickelten Algorithmen ermöglichen es, bei Anwendungen, die bisher nur auf High–

End–Rechnern möglich waren, auf den Einsatz von Graphik–Supercomputern zu verzichten und auf kostengünstige Alternativen zurückzugreifen. Ebenso können, aufgrund der implementierten Visualisierung für Datenmengen jenseits jeder Hauptspeichergrenze, Datensätze interaktiv dargestellt werden, die bisher auch auf Graphik–Supercomputern kaum handhabbar waren. Der Hauptvorteil der im Rahmen dieser Arbeit entwickelten Verfahren ist die Möglichkeit ihres Einsatzes auf einem breiten Spektrum von Plattformen mit unterschiedlicher Graphikleistung. Im Bereich der Graphik–Supercomputer ist die Leistung jedoch effektiv geringer, da im Vergleich zu dem Verfahren von Lindstrom und dem ROAM–Algorithmus, in dieser Arbeit die Implementierung nicht speziell auf die Charakteristika einer speziellen Graphik–Hardware hin optimiert werden konnte, da dies die Portabilität des Systems kompromittiert hätte. Werden speziell in diesem Bereich an den entwickelten Algorithmen weitere Verbesserungen unternommen, so läßt sich auch hier eine vergleichbare Leistung erzielen.

Im folgenden Kapitel werden die Grundlagen und Rahmenbedingungen, die bei dem Entwurf der entwickelten Triangulierungsalgorithmen von wesentlicher Bedeutung waren, dargelegt. Dazu gehört zum einen die virtuelle Sicht, wobei sich aus deren beabsichtigtem Einsatz die wichtigsten Entwurfskriterien ergeben. Es folgt eine Übersicht über die verschiedenen Verfahren zur Geländetriangulierung. Abschließend wird kurz auf die numerischen Grundlagen eingegangen, auf denen das Adaptionskriterium der Triangulierung beruht. Im dritten Kapitel werden zunächst gewisse Designentscheidungen der Implementierung dargelegt, worauf die für die virtuelle Sicht entwickelte und implementierte Geländetriangulierung erläutert wird. Das Kapitel endet mit einer Übersicht über weitere Verbesserungsmöglichkeiten des Verfahrens und zeigt schließlich einige Ergebnisse der Geländetriangulierung. Das vierte Kapitel widmet sich der interaktiven Visualisierung großer Datenbasen. Zunächst werden die erforderlichen Datenstrukturen eingeführt und begründet. Daraufhin wird die Geländetriangulierung der virtuellen Sicht auf diesen Datenstrukturen reimplementiert und anschließend um das dynamische Nachladen und Freigeben von Daten erweitert. Hierzu wird die Dreieckshierarchie, auf der die Geländetriangulierung basiert, von einer Metahierarchie von Kacheln überlagert, mit deren Hilfe das Nachladen organisiert wird. Das Kapitel schließt mit einem Überblick über die im Rahmen der Implementierung gewonnenen Erfahrungen, unter anderem auch dem Einsatz des Verfahrens zur Visualisierung von Bilddaten, sowie weiteren Verbesserungsmöglichkeiten. Im letzten Kapitel werden die gewonnenen Erkenntnisse und Ergebnisse noch einmal zusammengefasst. Schließlich werden Möglichkeiten für weitere Entwicklungen und Verbesserungen der im Rahmen dieser Arbeit entstandenen Implementierungen aufgezeigt.

An dieser Stelle möchte ich mich bedanken bei Prof. Sachs vom Lehrstuhl für Flugmechanik und Flugregelung und seinen Mitarbeitern, deren konstruktive Begleitung wesentlich zum Entstehen dieser Arbeit beigetragen hat, bei meinen Kollegen am Lehrstuhl für Ingenieuranwendungen in der Informatik, numerische Programmierung für zahlreiche sehr hilfreiche Diskussionen und schließlich ganz besonders bei Prof. Zenger für die Überlassung des Themas und die freundliche und anspornende Betreuung, die mir auch stets Spielraum zum Gehen eigener Wege eröffnete.

2 Grundlagen

Wesentlich für dieser Arbeit ist der enge Bezug zur praktischen Anwendung der zu entwickelnden Algorithmen. Dementsprechend waren es auch vor allem die angestrebten Anwendungen, die die Grundlage für die Rahmenbedingungen und Anforderungen an diese Algorithmen bildeten.

2.1 *Virtuelle Sicht*

Die „natürliche Sicht“ ist für Flugzeugbesatzungen nach wie vor das wichtigste Hilfsmittel zur Orientierung bei bodennahen Manövern wie etwa Landeanflug, Tiefflug oder auch nur dem Rollen zum Terminal. Ist die Sicht durch die Cockpitfenster auf die Umgebung z.B. durch Nebel, Regen oder auch einfach durch Dunkelheit eingeschränkt bzw. gar nicht möglich, kommt es zu Behinderungen des Flugverkehrs. Gerade kleine Flugzeuge und Flughäfen verfügen nur selten über die zum Teil recht kostspielige Ausrüstung, die Allwetterflugbetrieb ermöglicht. So ist es nicht erstaunlich, daß bei Berichten in den Medien von Abstürzen kleinerer Privatflugzeuge häufig schlechte Außensichtbedingungen als Absturzursache angegeben werden. Große Verkehrsflugzeuge sind zwar alle mit den erforderlichen Flugführungsinstrumenten ausgestattet, die Allwetterflugbetrieb ermöglichen, das gleiche gilt für die großen Flughäfen, dennoch kommt es auch heute noch, selbst an gut ausgestatteten Flughäfen, bei einer Verschlechterung der Außensicht zu Verspätungen und Absagen von Flügen. Die Nachteile für Passagiere und Fluggesellschaften sind offensichtlich.

2.1.1 Herkömmliche Hilfsmittel zur Flugführung bei schlechter Außensicht

Das übliche Navigationshilfsmittel, das die Landung bei schlechter oder fehlender Außensicht ermöglicht, ist das sogenannte Instrument Landing System (ILS). Dieses System wurde am 23. September 1929 von Lt. James Doolittle zum ersten mal erfolgreich eingesetzt und von der Internationalen Zivilen Luftfahrt Organisation 1949 genehmigt und übernommen (siehe [22]). Abbildung 2.1 eines ILS–Instrumentes verdeutlicht, daß die derzeit eingesetzten Flugführungsinstrumente in ihrer Darstellung sehr abstrakt und wenig intuitiv sind. Aus diesem Grund ist der erforderliche Interpretationsaufwand relativ hoch. Die Anzeige, mit der der Pilot überprüfen kann, wie gut er dem Leitstrahl[1] folgt, besteht aus zwei Markierungen, von denen sich die eine

[1] ein vom Flugplatz gesendetes Signal, das die ideale Anfluglinie markiert und Landungen bei Null–Sicht ermöglicht

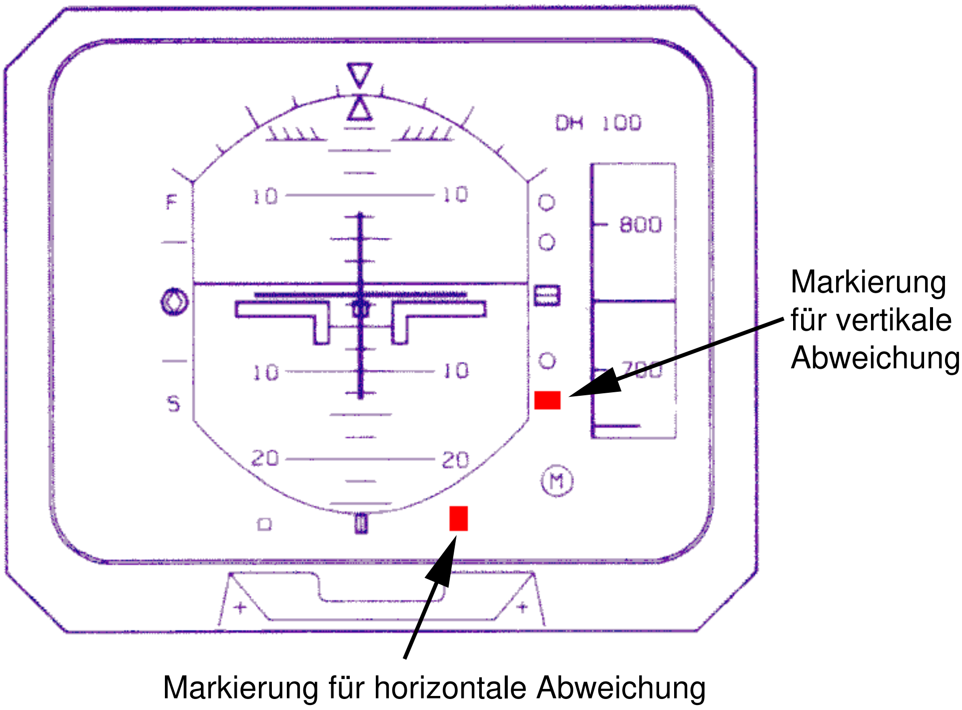
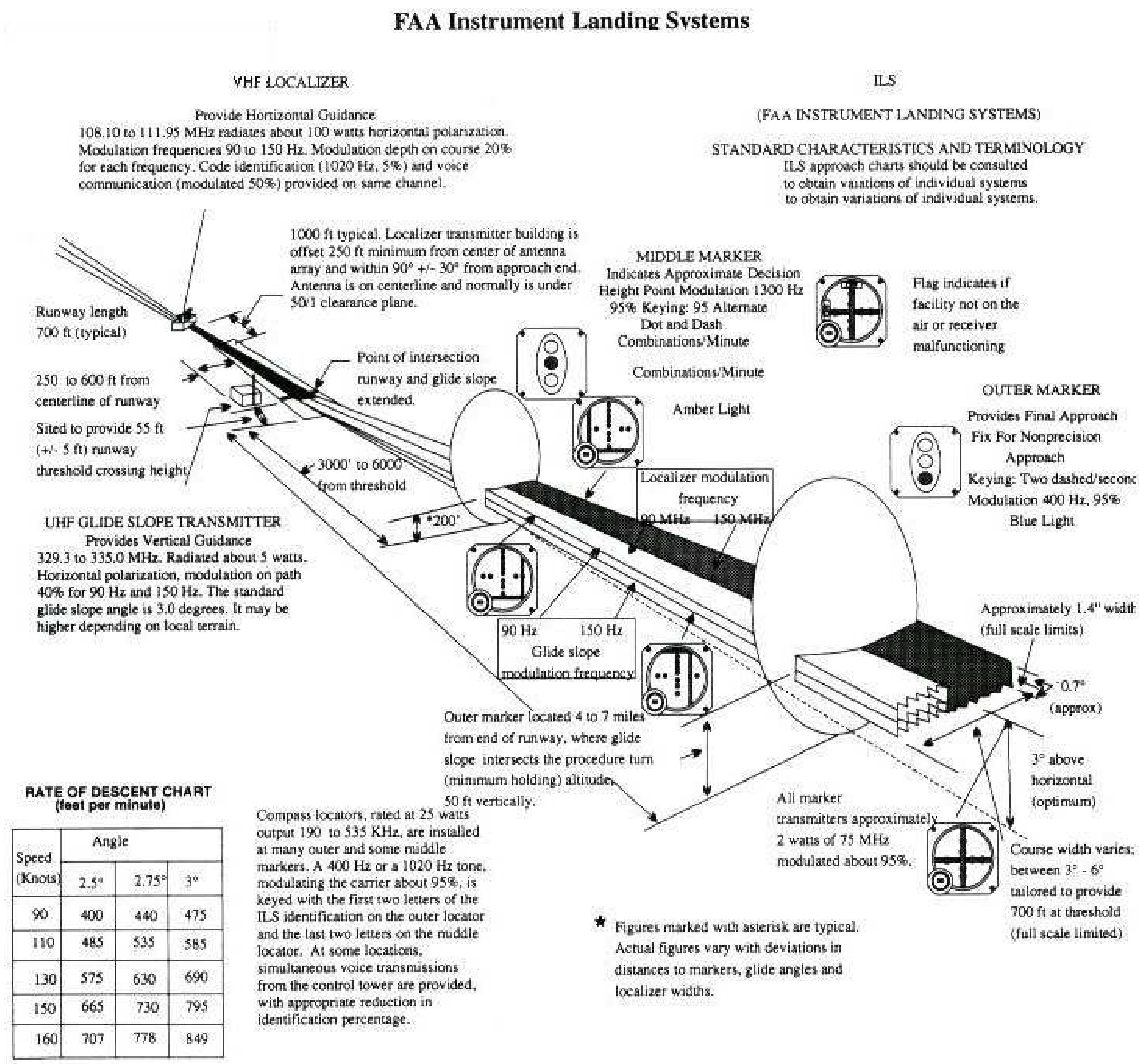


Abbildung 2.1 Anzeige für Instrumenten gestützte Landung.

horizontal, die andere vertikal verschiebt und so die Abweichung von der Ideallinie wiedergeben (siehe Abbildung 2.1). Abbildung 2.2, die Charakteristika und Terminologie des Instrument Landing Systems illustriert, belegt die Komplexität des Systems. Mit einem so gestalteten Instrument sind ausschließlich gradlinige Anflüge möglich, die Einflugschneisen von erheblicher Länge (zum Teil mehrere Kilometer) erforderlich machen. Es ist nicht möglich, Ortschaften zu umfliegen und so einen Beitrag zum Lärmschutz zu leisten. Der Grund für diese Einschränkung liegt einerseits in der Tatsache, daß sich der Leitstrahl nicht abknicken läßt[2], andererseits in der Anzeigetechnik, bei der auf den Piloten zukommende Kurven im Landeanflug erst zu erkennen wären, wenn die, durch den Leitstrahl markierte, Ideallinie bereits verlassen wurde. Der Pilot kann somit erst zu spät auf diese Situation reagieren. Ein weiterer Nachteil dieser Technologie ist der hohe technische und finanzielle Aufwand, der auf Seiten des Flugplatzes betrieben werden muß, um Flugzeugen eine leitstrahlunterstützte Landung anbieten zu können. Aus diesem Grund sind kleinere Flugplätze, die hauptsächlich von Privatflugzeugen genutzt werden, praktisch nie mit dieser Flugführungshilfe ausgerüstet.

Die Flugführungsinstrumente, in anderen Flugsituationen als der Landung, beschränken sich im wesentlichen auf Kompass, künstlichen Horizont und Höhenmesser. Auch hierbei handelt es sich um Instrumente, die seit Beginn des 20. Jahrhunderts in dieser Form eingesetzt werden. Die mechanische Einfachheit und Zuverlässigkeit dieser Anzeigen ist zwar über jeden Zweifel erhaben, der dramatische allgemeine technische Fortschritt seit ihrer Einführung hat sich jedoch bisher nicht wesentlich auf Flugführungshilfen ausgewirkt. Die technische Ausführung mag sich geändert haben, die zugrundeliegenden Prinzipien sind jedoch seit 50 Jahren unverändert,

[2]auch wenn Gemeinden wie z.B. Augsburg sich mit der gradlinigen Ausbreitung von elektromagnetischen Wellen als Erklärung für diese Tatsache nicht abspeisen lassen wollen



Speed (Knots)	Angle 2.5°	2.75°	3°
90	400	440	475
110	485	535	585
130	575	630	690
150	665	730	795
160	707	778	849

Abbildung 2.2 Charakteristika und Terminologie des Instrument Landing System.

so daß auch moderne Flugführungsanzeigen den selben Einschränkungen unterliegen, die seit Beginn des 20. Jahrhunderts bestehen.

2.1.2 Idee und Konzept der virtuellen Sicht

Die sogenannte virtuelle Sicht ist ein Weg zur Lösung der aufgeführten Nachteile und Einschränkungen der herkömmlichen Flugführungsinstrumente (siehe [32] oder [29]). Die Idee ist, dem Piloten eine vom Computer generierte graphische Darstellung der aktuellen Umgebung anzubieten, die mit innovativen Flugführungsanzeigen ergänzt werden kann (siehe [6]). Diese graphische Darstellung kann z.B. auf einen Monitor in der Instrumentenkonsole (Head–Down–Display) gelegt werden, auf die Cockpitfenster bzw. auf ein Visier vor den Augen des Piloten (Head–Up–Display) projiziert werden oder auf zwei Monitoren, die ähnlich wie ein Brille vom

Piloten getragen werden (Head–Mounted–Display), angezeigt werden. Der Unterschied zum Head–Up–Display besteht bei letzterem darin, daß die Kopfbewegungen bzw. die Blickrichtung des Piloten bei der Erzeugung der graphischen Geländedarstellung berücksichtigt werden.

Um die virtuelle Sicht möglich zu machen, muß zunächst eine Datenbank vorliegen, die die Umgebung des Flugzeugs an seiner aktuellen Position enthält und genau genug beschreibt, um eine Darstellung des Geländes zu erzeugen, die der Orientierung dienen kann. Je ähnlicher die graphische Darstellung der tatsächlichen Umgebung ist, desto besser kann sich der Pilot in der Regel daran orientieren, d.h. je genauer und umfangreicher die Datenbank ist, desto „besser“ ist die virtuelle Sicht. Da sich Position und Lage des Flugzeugs kontinuierlich ändern und sich damit auch der Betrachterstandort und der Blickwinkel auf das umgebende Gelände ändern, müssen diese Parameter ständig erfasst werden und in die Erstellung der graphischen Darstellung einfließen. Daraus folgt unmittelbar, daß auch die computergenerierte Anzeige möglichst oft aktualisiert werden muß, um der geänderten Position und Fluglage Rechnung zu tragen. Wird die virtuelle Sicht über ein Head–Mounted–Displays angeboten, so muß zusätzlich die Blickrichtung des Piloten ermittelt werden und in die Anzeige einfließen.

Die Position des Flugzeugs kann über GPS bzw. DGPS[3] unterstützt durch INS[4] ermittelt werden. Die Flughöhe, die bei der Positionsbestimmung über DGPS typischerweise zu ungenau ist, um z.B. aufgrund dieser Werte einen Landeanflug durchzuführen, kann mit Hilfe von Radarmessungen bestimmt werden. Die Fluglage läßt sich mit Hilfe der Bordinstrumente ermitteln. Falls erforderlich, wird die Blickrichtung des Piloten mit den gleichen Verfahren bestimmt, mit denen dies auch im Bereich der Virtual Reality geschieht[5].

Das Konzept der virtuellen Sicht wird in Abbildung 2.3 dargestellt. Aus dem angestrebten Einsatz der virtuellen Sicht ergeben sich folgende Entwurfskriterien für die zu implementierende graphische Geländedarstellung:

- Die Sichtweite in der virtuellen Sicht soll mindestens 25km betragen.
- Die computergenerierte Anzeige soll 30 mal pro Sekunde, unter Berücksichtigung von Positions- und Lageänderungen, aktualisiert werden.
- Die verwendeten Algorithmen sollen die Möglichkeit zur Kompression der Geländedatenbank bieten.
- Die Bilderzeugung soll auch auf relativ kostengünstigen Rechnern möglich sein und nicht nur auf Graphik–Supercomputern.
- Die zu schreibenden Programme sollen modular und portabel gehalten werden. Letzteres, um neue Entwicklungen im Bereich kostengünstiger Graphikhardware nutzen zu können.

Neben den geringeren Anschaffungskosten spricht auch das geringere Gewicht für den Einsatz von „kleinen“ Rechnern zur Erzeugung der virtuellen Sicht anstelle von Graphik–Supercomputern. Gewicht hat beim Fliegen unmittelbaren Einfluss auf den Treibstoffverbrauch, was sich wiederum unmittelbar auf die Betriebskosten auswirkt.

[3] **D**ifferential **G**lobal **P**ositioning **S**ystem
[4] **I**nertial **N**avigation **S**ystem $\hat{=}$ trägheitsbasiertes Navigationssystem
[5] drei Sensoren am Head–Mounted–Display, deren Position auf der Basis von Laufzeitunterschieden bzw. Signalstärke bestimmt wird

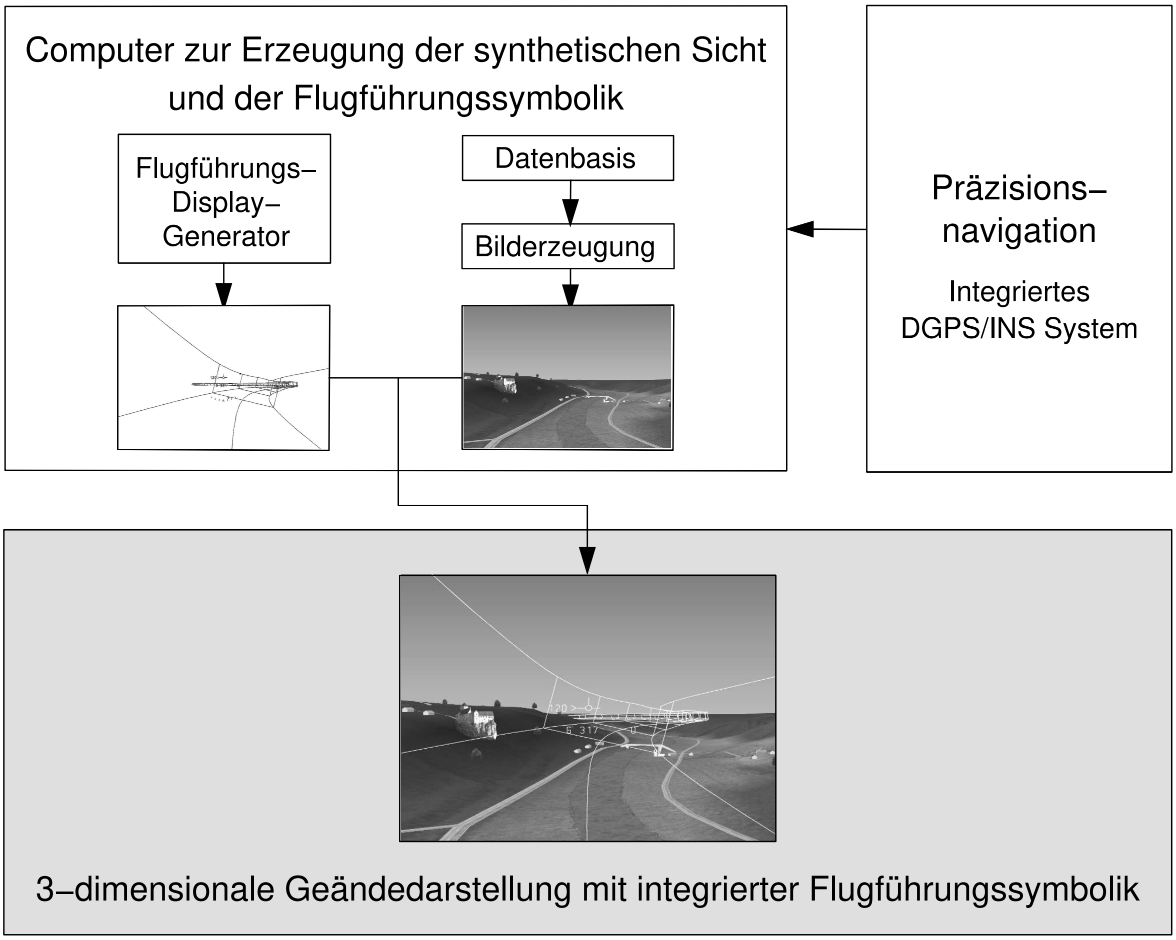


Abbildung 2.3 Konzept der virtuellen Sicht.

Eine computergenerierte dreidimensionale Darstellung der Umgebung ermöglicht das Einblenden von Anzeigen zur Flugführung, die bisher nicht denkbar waren (siehe [36] oder [6]). Prominentes Beispiel für derartige Anzeigen ist ein Flugführungstunnel, der dem Piloten den vorgegebenen Flugweg zeigt. Abbildung 2.3 zeigt einen derartigen Tunnel als Ergebnis des Flugführungs–Display–Generators. Gleichzeitig besteht selbstverständlich auch die Möglichkeit, herkömmliche Flugführungsanzeigen nachzubilden und analog zu einem Head–Up–Display in die Geländedarstellung einzublenden.

2.1.3 Erzeugung der Geländedarstellung für die virtuelle Sicht

Als Datenbasis, die die zur graphischen Darstellung des Geländes benötigten Informationen enthält und Voraussetzung für die virtuelle Sicht ist, werden die sogenannten DTED[6] und DFAD[7] Datenbanken verwendet. Die Beschreibung des Geländes ist in Höhendaten (DTED) und Strukturdaten (DFAD) unterteilt.

[6] **D**igital **T**errain **E**levation **D**ata
[7] **D**igital **F**eature **A**nalysis **D**ata

Datenbasis

Die DTED–Daten liegen als regelmäßige Gitter vor, die es in verschiedene Auflösungen (*Level*) gibt. Der Abstand der Gitterpunkte des Level Zwei ist eine Bogensekunde, bei Level Eins sind es drei und bei Level Null 30 Bogensekunden. Die Daten des Level Null sind inzwischen frei verfügbar und können unter der URL

`http://www.nima.mil/geospatial/products/DTED/dted.html`

im World Wide Web abgerufen werden. Die für diese Arbeit verwendeten Daten sind die des Level Eins und beinhalten das in Abbildung 2.4 gezeigte Gebiet. Im Bereich von Süddeutschland

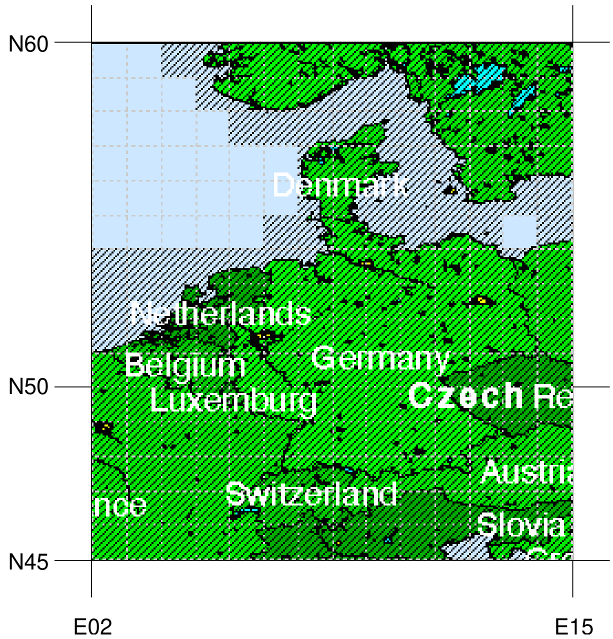


Abbildung 2.4 Von den verwendeten DTED–Daten abgedecktes Gebiet.

entsprechen drei Bogensekunden einem Abstand von etwa 60 Metern horizontal (geographische Länge) und 90 Metern vertikal (geographische Breite). Die Daten sind in Blöcke von 1×1 Grad unterteilt, was jeweils 1200×1200 Höhenwerten entspricht. Weiter nördlich, ab dem 50. Breitengrad, reduziert sich die horizontale Auflösung auf sechs Bogensekunden, was zu 600×1200 Höhenwerten pro Gradblock führt. Die Größe der Original–Daten beträgt etwa 407MB (inklusive Header–Informationen zu den einzelnen Datenblöcken), die Anzahl der Höhenwerte ist etwa 200 Millionen. Bei zwei Byte pro Höhenwert (ein `short`) ist das reine Datenvolumen 384,5MB. Um das Gebiet von ganz Deutschland abzudecken, werden ca. 22 1200×1200– und ca. 41 600×1200–Gradblöcke benötigt, das entspricht 61,2 Millionen Höhenwerten bzw. 116,7MB. Die Genauigkeit der Höhenwerte der DTED Level Eins Daten wird mit ± 30 Metern angegeben. Dabei beträgt der Fehler in flachen Gebieten in der Regel nur wenige Meter, während in Gebirgen die 30 Meter erreicht werden können.

Der zweite Teil der Daten, die in der DFAD–Datenbank enthaltenen Strukturdaten, geben Geländemerkmale wieder. Darunter sind bestimmte Eigenschaften des Geländes zu verstehen, wie z.B. Bewuchs, Bebauung oder Beschaffenheit. Die Struktumdaten spalten sich in nulldimensionale, eindimensionale und zweidimensionale Strukturen auf. Gebäude wie Brücken, Türme, Hochspannungsmasten oder Kirchen werden durch nulldimensionale Strukturen in Form ihrer Position in geographischer Länge und Breite repräsentiert. Zusätzlich sind Breite, Tiefe, Höhe und Orientierung des Objektes gespeichert. Die Art des Gebäudes wird durch einen Objekt–Typ festgelegt. Die Anzahl der unterschiedenen Objekt–Typen ist recht umfangreich und füllt einen DIN–A 4 Ordner. Darin werden verschiedene Arten von Hochspannungsmasten, Industriegebäude mit unterschiedlich gestalteten Dächern bis hin zu Obelisken und Pyramiden spezifiziert. Eindimensionale Strukturen, wie Flüsse, Straßen und Eisenbahngleise, sind in Form einer Sequenz von Positionen (Länge/Breite) zusammen mit einer Breite abgelegt. Zweidimensionale Strukturen, wie etwa Seen, Wälder, Städte, Dörfer oder felsige Gebiete, werden durch einen geschlossenen Polygonzug repräsentiert. In beiden Fällen werden, ähnlich wie bei nulldimensionalen Strukturen, jeweils verschiedene Typen unterschieden.

Erzeugung der computergenerierten graphischen Darstellung

Abbildung 2.5 illustriert das Vorgehen, um aus den beschriebenen Ausgangsdaten eine dreidimensionale graphische Darstellung des Geländes zu erzeugen.

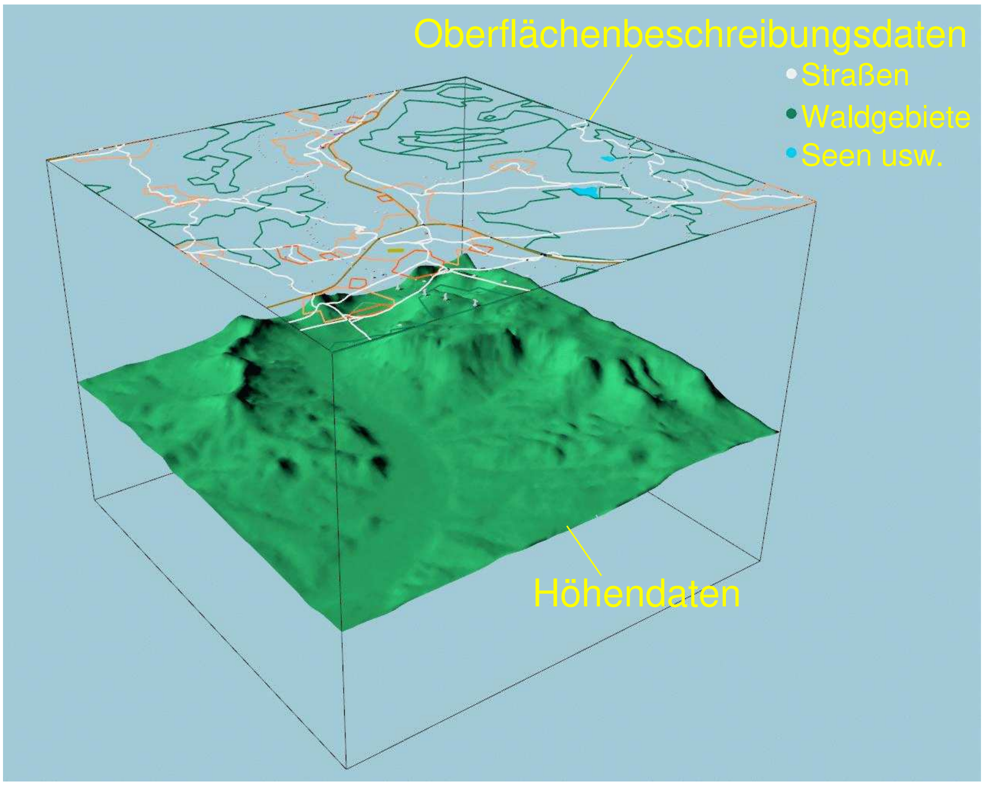


Abbildung 2.5 Digitales Geländemodell.

Aus den DTED–Daten wird in einem Triangulierungsschritt ein dreidimensionales Dreiecksnetz erzeugt. Werden hierbei alle Gitterpunkte berücksichtigt, so ergeben sich doppelt soviele zu zeichnende Dreiecke wie Gitterpunkte. Die Triangulierung sieht dabei etwa wie in Abbildung 2.6 aus. Würde dieser naive Ansatz zur Triangulierung verfolgt, so würde das für den gesamten

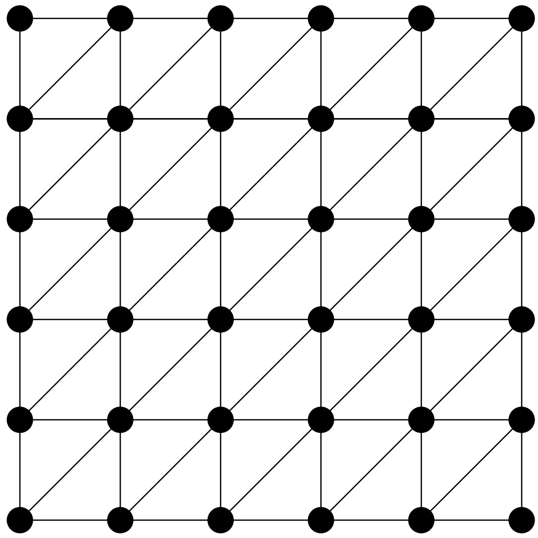


Abbildung 2.6 Triangulierung bei Berücksichtigung aller Gitterpunkte.

Deutschland–Datensatz zu 122,4 Millionen Dreiecken führen, die gezeichnet werden müssten. Selbst wenn nur ein Gradblock gezeichnet werden soll, so sind das immer noch 2,9 Millionen Dreiecke. Für eine Bildwiederholungsrate von 30 Bildern pro Sekunde müsste die Graphikhardware in der Lage sein, 87 Millionen Dreiecke pro Sekunde zu zeichnen, um einen Gradblock darstellen zu können bzw. 3,7 Milliarden Dreiecke für ganz Deutschland. High–End Graphiksysteme, die mehrere Millionen DM kosten und schon allein aufgrund ihrer Dimensionen und ihres Gewichts für einen Einsatz in Flugzeugen völlig ungeeignet sind, können einige 100 Millionen Dreiecke pro Sekunde zeichnen[8]. Aktuelle Graphik–Chips[9], die auf PC–Graphikkarten eingesetzt werden, leisten laut Hersteller bis zu 31 Millionen Dreiecke pro Sekunde. Dabei ist zu berücksichtigen, daß solche Herstellerangaben in der Regel nur unter sehr speziellen Voraussetzungen erreicht werden und daß man sich in der Praxis meist mit deutlich geringeren Werten zufrieden geben muß. Bei naiver Triangulierung ist demnach eine Darstellung von ganz Deutschland auf Jahre hinaus undenkbar. Die Darstellung eines naiv triangulierten Gradblocks dürfte in einigen Jahren möglich sein, doch dann würde wenig bis kein Spielraum für die Darstellung anderer Elemente, wie z.B. Feature–Daten oder Flugführungsanzeigen, bleiben. Ziel dieser Arbeit ist es, das Gelände bei nur geringem Qualitätsverlust mit einer um mehrere Größenordnungen reduzierten Anzahl von Dreiecken zu triangulieren und darzustellen.

Um das im Triangulierungsschritt erzeugte Dreiecksnetz einzufärben, werden die Informationen der zweidimensionalen Strukturdaten herangezogen. Den als Polygon gespeicherten Flächen mit bestimmter Eigenschaft wird eine Farbe zugeordnet, mit der sie dann gezeichnet und als Bitmap abgespeichert werden. Diese Bitmap wird dann mit Hilfe der Informationen der Höhendaten „beleuchtet“. D.h. die Helligkeit des Punktes wird in Abhängigkeit vom Winkel zwischen

[8] Der höchste gefundene Wert wurde mit 210 Millionen Polygonen pro Sekunde für eine SGI Onyx2 InfiniteReality2 Rack 16–Pipe mit Multipipe Rendering angegeben.

[9] z.B. GeForce2 Ultra der Firma nVIDIA

Oberflächennormale an diesem Punkten und dem Vektor von diesem Punkt zu einer willkürlich gesetzten Lichtquelle variiert. Abbildung 2.7 zeigt den Effekt dieses Vorgangs. Die Textur, die

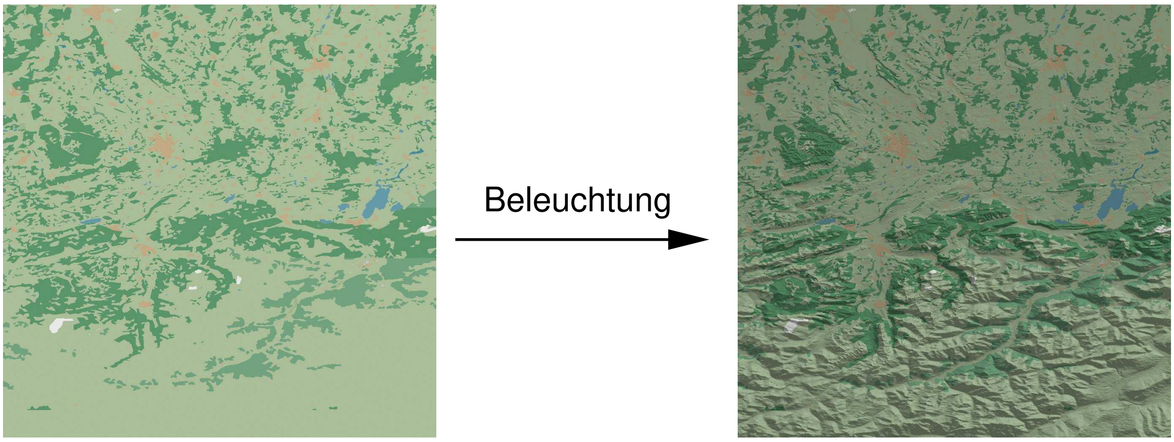


Abbildung 2.7 „Beleuchtung“ der zweidimensionalen Strukturdaten.

über das Dreiecksnetz gelegt wird, enthält somit bereits einen sehr wichtigen optischen Hinweis auf die Beschaffenheit des Geländes und unterstützt so seine räumliche Wahrnehmung. Ein weiterer Vorteil ist, daß kleinere Höhenfehler in der Geländedarstellung als weniger störend empfunden werden, da sie kaum bemerkt werden. Schließlich kann auf die Beleuchtung des Geländes während des Zeichnens verzichtet werden, da die durch eine Beleuchtung hervorgerufenen Helligkeitsunterschiede bereits Bestandteil der Textur sind. Dies beschleunigt die Erzeugung der graphischen Darstellung.

Die verbleibenden null- und eindimensionalen Strukturdaten aus den DFAD–Daten werden in einem separaten Durchgang gezeichnet. Hierzu werden die eindimensionalen Strukturdaten jeweils anhand der ihnen zugeordneten Breite in ein Polygon konvertiert und aufgrund des ihnen zugeordneten Typs eingefärbt bzw. texturiert. Für besonders wichtige nulldimensionale Strukturen (z.B. Hochspannungsmasten) werden Displaylisten definiert, die ein schnelles wiederholtes Zeichnen eines Objektes durch die Graphikhardware erlauben. Andere Objekte werden durch eine geeignete Standard–Form dargestellt. Alle Objekte, die für nulldimensionale Strukturdaten gezeichnet werden, werden jeweils aufgrund der zusätzlichen zur Position gespeicherten Informationen skaliert und gedreht.

Eine aus diesem Vorgehen resultierende Geländedarstellung ist in Abbildung 2.8 wiedergegeben.

2.2 Geländetriangulierung

Wie im vorangegangenen Abschnitt 2.1.3 dargelegt wurde, ist für die Realisierung der virtuellen Sicht eine ausgereifte Triangulierungstechnik erforderlich, die in der Lage ist, die Anzahl der Dreiecke gegenüber der naiven Triangulierung (Abbildung 2.6) drastisch zu reduzieren.

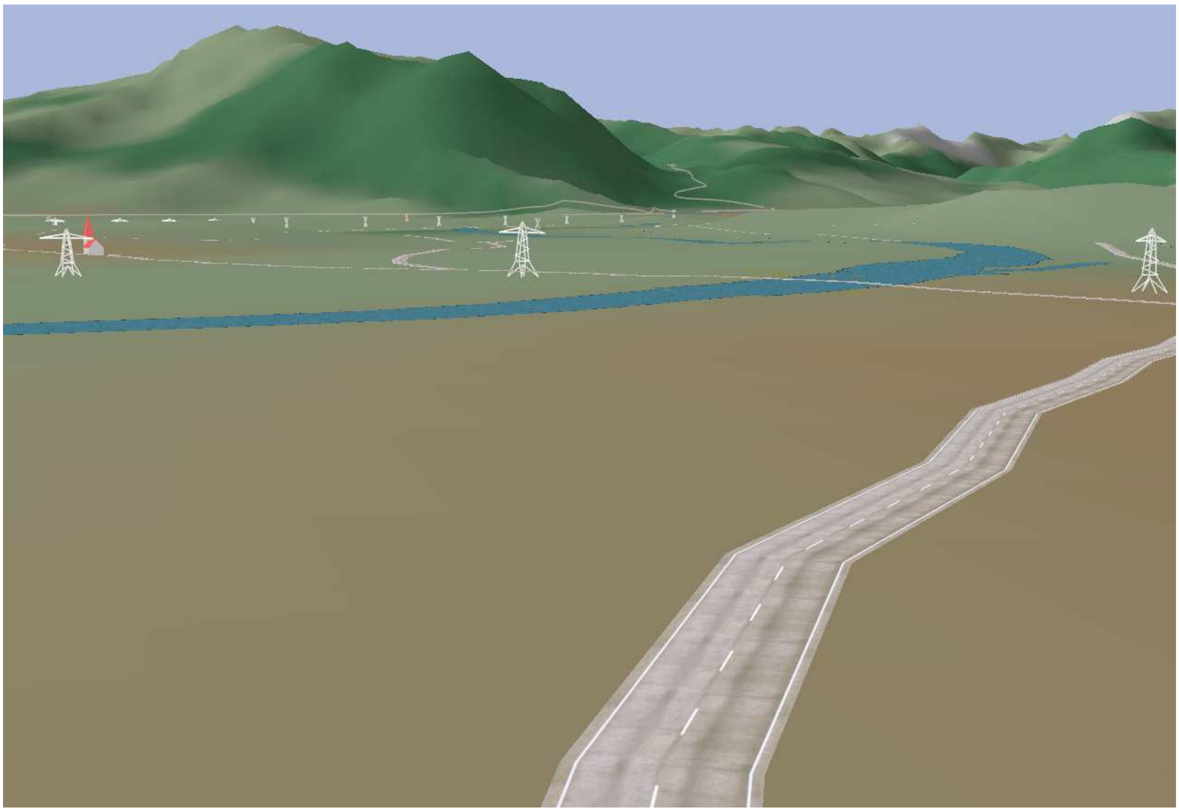


Abbildung 2.8 Geländedarstellung nach Kombination der DTED– und DFAD–Daten.

2.2.1 Triangulierungsarten

Vor allem in der zweiten Hälfte der 90er Jahre wurden zahlreiche wissenschaftliche Arbeiten veröffentlicht, die sich mit dem Problem der Echtzeitgenerierung von Dreiecksnetzen mit sich änderndem Detailgrad (*Level–of–Detail*) beschäftigen. Dies liegt vor allem an den technischen Fortschritten auf dem Feld der Computergraphik. Leistungsfähige Graphikhardware ist etwa seit Beginn der 90er in der Lage, genügend Polygone pro Sekunde auf dem Bildschirm zu zeichnen, so daß die Echtzeitdarstellung von Geländen mit interessanter Größe bei geeigneter Triangulierung möglich wird.

Alle in diesem Abschnitt aufgeführten Techniken zur Reduktion der Dreiecksanzahl nutzen eine oder beide der folgende zwei wesentliche Eigenschaften von Höhendaten:

- Natürliche Gelände bestehen zu nicht unwesentlichem Anteil aus flachen Gebieten, die bereits mit Hilfe weniger großer Dreiecke korrekt dargestellt werden können.
- Geländestrukturen, die weit vom Betrachter entfernt sind, werden aufgrund der Zentralprojektion nur auf einen kleinen Bereich auf dem Bildschirm abgebildet und können deswegen ebenfalls schon durch wenige Dreiecke dargestellt werden.

Die erste Eigenschaft wird ausgenutzt, indem den Höhenwerten eine „Wichtigkeit“ zugeordnet wird, aufgrund der sie bei einer Triangulierung berücksichtigt bzw. ignoriert werden. In der Regel handelt es sich dabei um eine Art von Fehler, der durch das Weglassen des Höhenwertes bei der Triangulierung in der Darstellung gemacht wird. Die zweite Eigenschaft führt üblicherweise dazu, daß nahe am Betrachter relativ viele Dreiecke für die Darstellung des Geländes verwendet werden, während am hinteren Rand des sichtbaren Bereiches nur relativ wenige Dreiecke gezeichnet werden. Bleibt z.B. die Größe der Dreiecke auf dem Bildschirm, d.h. ihre Fläche in Pixeln gemessen, für das ganze gezeichnete Gelände etwa gleich, so entspricht Dreiecken am hinteren Rand des sichtbaren Bereichs eine größerer Bereich der Höhendaten als bei Dreiecken nahe am Betrachter. Am hinteren Rand werden also relativ zur Geländefläche weniger Dreiecke gezeichnet.

Die verschiedenen Triangulierungstechniken können nach unterschiedlichen Kriterien kategorisiert werden. Eine weit verbreitete Unterscheidung richtet sich nach der Art der erzeugten Dreiecksnetze. Hier läßt sich zunächst grob nach regulären und irregulären Dreiecksnetzen trennen. Letztere werden im allgemeinen als *Triangular Irregular Network*, kurz *TIN* bezeichnet.

Triangular Irregular Networks

Die prominenteste Technik, die diese Art von Dreiecksnetz erzeugt, dürfte die *Delauny* Triangulierung sein (siehe [13]). Sie wird in der Regel durch ihre Orthogonalität zum Voronoi Diagramm definiert (siehe auch [11]). Eine andere Beschreibung liefert [8], die die Delauny Triangulierung als diejenige aller möglichen Triangulierungen bezeichnen, bei der die erzeugten Dreiecke die größtmögliche Winkelähnlichkeit besitzen. Solche Dreiecke sind in der Computergraphik sehr beliebt, da sie in ihrer Darstellung unproblematisch sind. Langgestreckte, schmale Dreiecke mit spitzen Winkeln führen bei der Darstellung auf dem Bildschirm oft zu unerwünschten Artefakten (dem sogenannten *Aliasing*), sowie zu Problemen mit der Texturierung und der Z–Puffer Genauigkeit.

Meist werden TINs nach einer von zwei Vorgehensweisen erzeugt. Bei der einen Variante wird mit einer möglichst groben Triangulierung begonnen, die dann fortschreitend verbessert wird, indem Kanten in die Triangulierung eingefügt werden. Dies erfolgt meist durch die Teilung von Dreiecken, die bereits Bestandteil der Triangulierung sind. Soll eine Delauny Triangulierung erzeugt werden, so kann durch die Teilung eines Dreiecks das Delauny Kriterium verletzt werden. Um dieses wieder herzustellen, sind dann unter Umständen lokale Veränderungen der Triangulierung nötig. Bei der zweiten Variante wird der umgekehrte Weg gegangen. Es wird mit einer möglichst feinen Triangulierung begonnen, die dann durch die Entfernung von Kanten schrittweise vergröbert wird. Auch hier können Probleme bezüglich dem Delauny Kriterium auftreten, denen ähnlich wie bei der ersten Variante begegnet werden kann.

Die zahlreichen Verfahren zur Erzeugung von TINs unterscheiden sich im wesentlichen in den Kriterien, die zur Auswahl der einzufügenden bzw. der zu eliminierenden Kanten führen und der Art und Weise, wie lokal verletzte Kriterien wieder hergestellt werden.

Ein wichtiger Vorteil der irregulären Triangulierung ist, daß es keinerlei Voraussetzung für die zu triangulierenden Gitter gibt. D.h. insbesondere, daß die Lage der Gitterpunkte beliebig ist und diese nicht auf einem regelmäßigen Raster liegen müssen.

Reguläre Triangulierungen

Während bei TINs in der Regel beliebige Punkte, die keiner geometrischen Einschränkung unterliegen, in die Triangulierung eingefügt bzw. aus ihr entfernt werden können, so liegt einer regulären Triangulierung eine bestimmte Reihenfolge der Punkte, sowie meist auch ein regelmäßiges Gitter, zugrunde. Die Reihenfolge wird in der Regel durch eine geeignete Hierarchie festgelegt, die auf den Höhenwerten definiert wird.

Die einfachste Hierarchie läuft auf das sogenannte *Subsampling* hinaus. Dabei wird bei der Triangulierung der naive Ansatz verfolgt, jedoch wird nur jede $n$te Zeile und jede $n$te Spalte des zugrundeliegenden Höhendatensatzes verwendet. Der Wert n kann dabei abhängig von der Entfernung zum Betrachter variiert werden, um verschiedene, diskrete Level–of–Detail zu realisieren. Ein Beispiel für drei Level–of–Detail, die durch Subsampling generiert werden zeigt Abbildung 2.9. Genau dieses Verfahren zur Reduktion der Dreiecksanzahl wurde bei der ursprünglichen Implementierung der virtuellen Sicht (siehe [28]) am Lehrstuhl für Flugmechanik und Flugregelung von Prof. Sachs eingesetzt. Im dem dem Beobachter am nächsten gelegenen

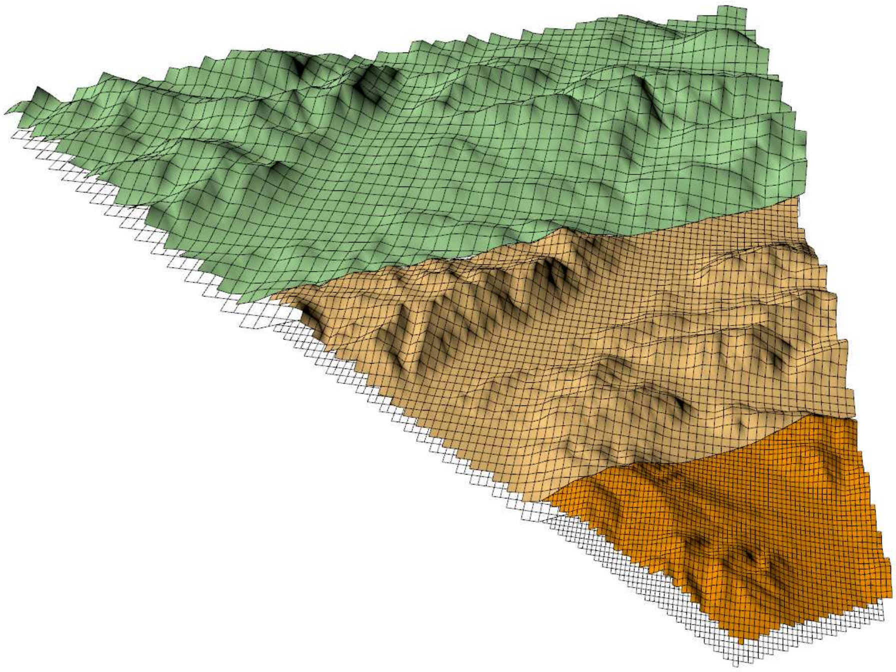


Abbildung 2.9 Drei Level–of–Detail durch Subsampling.

Bereich (rechts unten) wird jeder Gitterpunkt bei der Triangulierung genutzt, im anschließenden Bereich nur noch jeder zweite und im am weitesten entfernten Bereich schließlich nur noch jeder vierte Gitterpunkt berücksichtigt. Die Flachheit des Geländes wird nicht zur weiteren Reduktion der Dreiecksanzahl genutzt. Ein wesentliches Problem dieser Implementierung ist bereits in der Abbildung erkennbar, an den Grenzen zwischen zwei Level–of–Detail kann es zu Lücken im Gelände kommen. Diese Art von Löchern tritt in der Praxis wesentlich deutlicher zu Tage, als in Abbildung 2.9 dargestellt. Sie sind auf die Differenz zwischen einem Höhenwert und dessen Interpolant im nächst gröberen Level zurückzuführen. Da diese Löcher keine Ausdehnung in der horizontalen Ebene besitzen und nicht sichtbar sind, wenn sie senkrecht von oben betrachtet werden, nennt man sie auch *vertikale Löcher*. Abbildung 2.10 verdeutlicht das Zustandekommen von vertikalen Löchern, die ein häufig auftretendes Problem bei Geländetriangulationen mit verschiedenen Level–of–Detail sind.

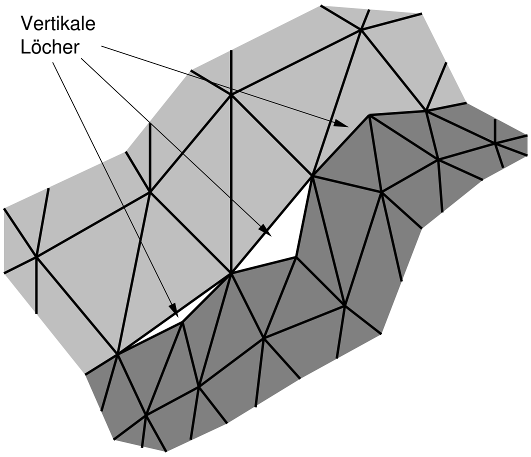


Abbildung 2.10 Vertikale Löcher an der Grenze zweier Level–of–Detail.

Die im Bereich der Geländetriangulierung am häufigsten genutzte Hierarchie ergibt sich direkt aus einem Dreiecks–Binärbaum, der sich wiederum unmittelbar aus dem naheliegensten Ansatz zur rekursiven Triangulierung eines quadratischen Gebiets nach dem *Divide&Conquer*–Prinzip ergibt. Abbildung 2.11 zeigt wie ein quadratisches Gebiet, das durch ein 5×5–Gitter gegeben ist, Level für Level rekursiv trianguliert wird. Dabei wird das Quadrat zunächst entlang sei-

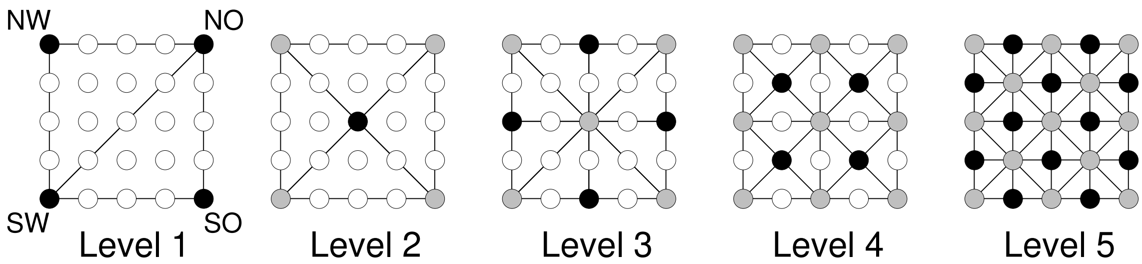


Abbildung 2.11 Rekursive Triangulierung eines 5×5 Gitters.

ner SW–NO–Diagonale geteilt. Es könnte auch die andere Diagonale zur Teilung herangezogen werden, dies stellt den einzigen Freiheitsgrad dieser Triangulierung dar. Die entstandenen rechtwinkligen Dreiecke SW–NO–NW und NO–SW–SO werden weiter unterteilt, indem sie entlang der Verbindung zwischen dem Mittelpunkt ihrer Hypotenuse und der Ecke NW bzw. SO geteilt werden. Die vier nun vorliegenden rechtwinkligen Dreiecke, werden nach dem gleichen Schema rekursiv weiter unterteilt, bis der unterste Level erreicht ist und alle Punkte des Gitters bei der Triangulierung berücksichtigt werden. Als Baum dargestellt ergibt sich die in Abbildung

2.12 gezeigte Struktur eines Dreiecks–Binärbaums. Diese Hierarchie wird z.B. von Lindstrom [25], Duchaineau [7] und Röttger [34] genutzt.

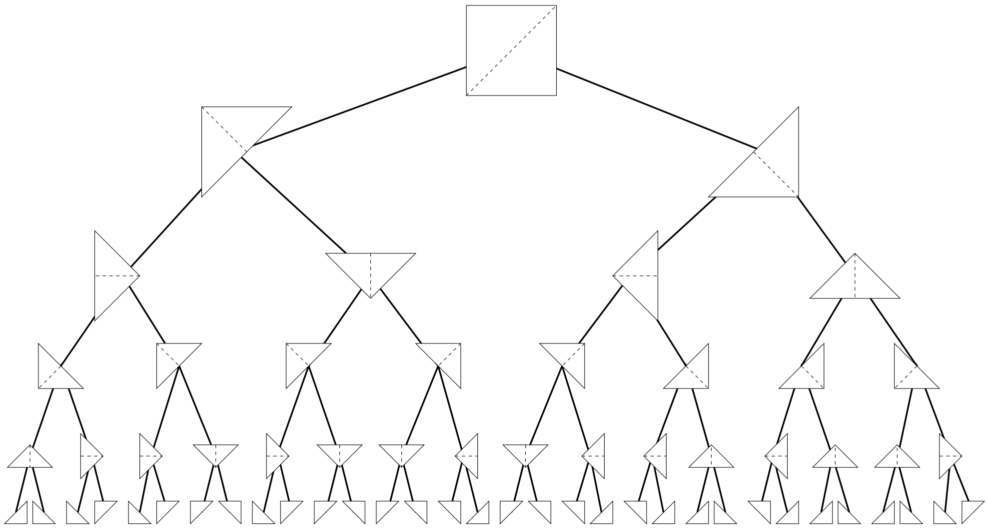


Abbildung 2.12 Struktur des Dreiecks–Binärbaums eines 5×5 Gitters.

Wird bei der Triangulierung nicht überall gleichmäßig verfeinert bzw. die Rekursion auf verschiedenen Leveln abgebrochen, so ergibt sich ein Baum, dessen Äste nicht alle gleich „tief“ sind. Verwendet man ein geeignetes Entscheidungskriterium, ob ein Teilbaum weiter verfeinert werden soll oder nicht, so führt dies zu einer adaptiven hierarchischen Triangulation. Praktisch alle Algorithmen, die auf der Hierarchie des Dreiecks–Binärbaums basieren, sind von diesem Typ. Sie unterscheiden sich, wie bereits die Verfahren zur Erzeugung irregulärer Triangulierungen, in dem Kriterium, das lokal über die Verfeinerung entscheidet, in der Art und Weise, wie vertikale Löcher vermieden werden sowie in der Verwaltung der für den Algorithmus benötigten Daten.

Der ROAM–Algorithmus (siehe [7]) legt z.B. besonderen Wert darauf, daß eine vorgegebene Anzahl von Dreiecken bei der Triangulierung nicht überschritten wird. Hierzu wird eine Split– und ein Merge–Queue verwendet, in die zu verfeinernde bzw. vergröberbare Knoten gemäß ihrer Wichtigkeit für die Geländetriangulation eingefügt werden. Diese Warteschlangen werden abgearbeitet, bis die gewünschte Dreiecksanzahl erreicht ist. Eine weitere Besonderheit ist, daß die Triangulierung inkrementell erfolgt. Üblicherweise wird das Gelände für jedes zu zeichnende Bild komplett neu trianguliert. ROAM stützt sich auf die Annahme, daß der Unterschied in der Triangulierung zwischen zwei aufeinanderfolgenden Bildern nur gering ist. Dieses Vorgehen birgt allerdings die Gefahr einer negativen Feedback–Schleife in sich (siehe [4]). Bei geringen Änderungen des Betrachterstandpunktes benötigt der Algorithmus nur wenig Zeit, um das Gelände zu retriangulieren. Sind die Änderungen größer, so dauert auch die Retriangulierung länger. Bei einer Anwendung, in der sich der Betrachter mit einer bestimmten Geschwindigkeit fortbewegt, kann es zu folgender Situation kommen: Liegen zwei Bilder kurz hintereinander, so hat der Betrachter seine Position in der Zwischenzeit nur wenig verändert, die Retriangulierung benötigt nur wenig Zeit. Beschleunigt der Betrachter, so wächst die Positionsänderung zwischen zwei aufeinanderfolgenden Bildern. Damit steigt auch die für die Retriangulierung benötigte Zeit, was wiederum zu größeren Intervallen zwischen zwei Bildern führt, was zu größerer Positionsänderung führt und so weiter. Dieses Problem tritt vor allem auf, wenn man eine zu hohe Dreiecksanzahl fordert, was gerade auf „schwächerer“ Graphikhardware auch in der Praxis vorkommt.

Lindstrom [25] verwaltet sein Gelände in Blöcken. Diese werden mit unterschiedliche Level–of–Detail Trianguliert, was mittels einem Bottom–Up Durchlauf durch die Hierarchie erfolgt. An den Grenzen von Blöcken muß speziell darauf geachtet werden vertikale Löcher zu vermeiden. Der Bottom–Up Durchlauf durch die Höhendaten stellt den entscheidenden Nachteil, der von Lindstrom vorgeschlagenen Triangulierung dar (siehe auch [4] und [7]). In der Regel ist nur ein Bruchteil der Daten des Höhenfeldes sichtbar. Der Bottom–Up Durchlauf macht es erforderlich alle Daten zu durchlaufen, obwohl diese nicht sichtbar sind bzw. schließlich für die Triangulierung nicht benötigt werden. Der daraus resultierende Aufwand ist letztendlich dafür verantwortlich, daß der Algorithmus von Lindstrom zu langsam arbeitet.

Röttger [34] verwaltet sein Gelände ebenfalls in Blöcken, die bei ihm als Quadtree organisiert sind. Er setzt eine Matrix von Skalierungsfaktoren ein, um einerseits die Triangulierung zu repräsentieren und um andererseits *Geomorphing* zu realisieren. Letzteres bezeichnet die lineare Interpolation zwischen zwei aufeinanderfolgenden Triangulierungszuständen, um abrupte Änderungen in der Geländedarstellung zu vermeiden.

Gross [20] verwendet eine auf Quadtrees basierende Hierarchie. Zur Bestimmung der einzelnen Quadtreeelemente, die für die Triangulierung verwendet werden, wird mit Wavelets gearbeitet. Die Triangulierung innerhalb der einzelnen Elemente entspricht zunächst der naiven Triangulierung (vergleiche Abbildung 2.6). Zur Vermeidung von vertikalen Löchern arbeitet Gross mit einem Katalog von Randkonfigurationen seiner Quadtreeelemente, aufgrund dessen er einen Unterschied von zwei Leveln zwischen benachbarten Elementen kompensieren kann. Aufgrund der naiven Triangulierung innerhalb von Quadtreeelementen zeichnet sich die erzeugte Triangulierung durch große zusammenhängende Gebiete von gleichgroßen Dreiecken aus, wodurch die Flachheit des Geländes nicht in vollem Umfang ausgenutzt wurde, um die Dreiecksanzahl zu reduzieren. Der Algorithmus ist nicht für den Realtime–Einsatz ausgelegt [20] und ist demnach für den Einsatz in dieser Arbeit nicht geeignet.

2.2.2 Hierarchische numerische Interpolation

Die Triangulierung der Höhendaten zur Darstellung des Geländes, das sie repräsentieren, entspricht numerisch gesehen dem Problem der Interpolation. Der Interpolant, also die triangulierte Oberfläche des Geländes, kann als Summe von Basisfunktionen dargestellt werden. Höhendaten liegen in der Regel in der durch die Stützpunktbasis festgelegten Form vor, d.h. an jedem Gitterpunkt wird der Funktionswert (in diesem Fall die Höhe) gespeichert. Abbildung 2.13 zeigt eine zweidimensionale, kontinuierliche Funktion, ihren stückweise linearen Interpolant und die zugehörige Stützpunktbasis. Im Falle von Höhendaten ist die kontinuierliche Funktion unbekannt, lediglich die Funktionswerte an den Stützstellen sind bekannt. Im Allgemeinen wird der Verlauf des Höhenfeldes zwischen den Gitterpunkten als linear angenommen und dementsprechend interpoliert. Interpolationen höherer Ordnung spielen in der Praxis für die hier angestrebte Anwendung keine Rolle.

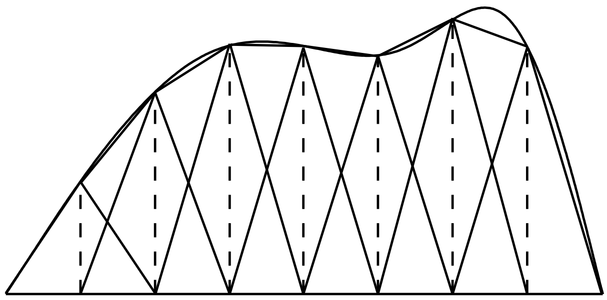


Abbildung 2.13 Funktion mit stückweise linearer Stützpunktbasis interpoliert.

Abbildung 2.14 zeigt die gleiche Funktion wie Abbildung 2.13, nur daß der Interpolant diesmal mit Hilfe einer stückweise linearen hierarchischen Basis dargestellt wird. Die Form des Interpolanten ändert sich dadurch nicht, lediglich die für die Darstellung mit der hierarchischen Basis abzuspeichernden Werte unterscheiden sich von der Variante in der eine Stützpunktbasis verwendet wird. Eine wichtige Eigenschaft der zu speichernden Werte bei der hierarchischen

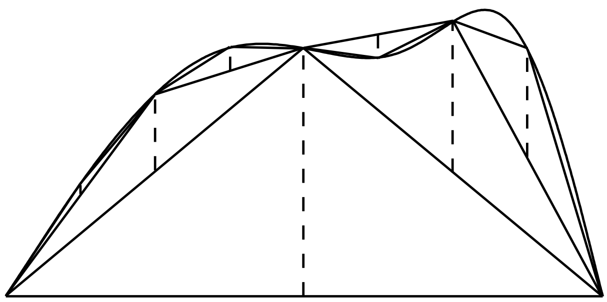


Abbildung 2.14 Funktion mit stückweise linearer hierarchischer Basis interpoliert.

Basis ist, daß diese betragsmäßig immer kleiner werden, wie an der Länge der vertikalen gestrichelten Linien in Abbildung 2.14 erkennbar ist. Die Werte bei der Stützpunktbasis liegen alle in der gleichen Größenordnung. Diese Eigenschaft kann z.B. genutzt werden, um eine Kompression der Daten durchzuführen. Zum einen können genügend kleine Werte einfach weggelassen werden, also nicht gespeichert werden, ohne das ein signifikanter Fehler gemacht wird. Andererseits müssen für die Speicherung der Werte aus tieferen Leveln der Hierarchie weniger Bits investiert werden, da die Werte bei genügender Glattheit konstant abnehmen. Ausnahmen stellen Unstetigkeitsstellen dar, die bei Höhendaten zwar vorkommen, aber in ihrer Anzahl gegenüber den glatten Bereichen nicht ins Gewicht fallen.

Eine weitere sehr wichtige Eigenschaft der zu speichernden Werte ist, daß sie unmittelbar den Fehler wiedergeben, der bei der Interpolation gemacht wird, wenn der jeweilige Wert nicht berücksichtigt wird. In diesem Fall wird der Wert an der entsprechenden Stelle über lineare Interpolation aus den hierarchischen Vorfahren approximiert. Die zu speichernden Werte werden auch als *hierarchischer Überschuss* bezeichnet. Dieser Begriff steht allgemein für die Differenz zwischen einem Funktionswert und dem aus seinen hierarchischen Vorfahren interpolierten Wert an der selben Stelle.

3 Geländetriangulierung für die virtuelle Sicht

3.1 Designentscheidungen

Ausschlaggebend für die im Laufe der Implementierung der virtuellen Sicht getroffenen Designentscheidungen sind die im Abschnitt 2.1.2 auf Seite 8 festgelegten Entwurfskriterien. Der letzte Punkt dieser Liste, die Forderung nach Modularität und Portabilität, legte die Rahmenbedingungen der Implementierung fest. Diese bestehen aus dem Betriebssystem, der Programmiersprache und den Bibliotheken, die bei der Entwicklung eingesetzt werden.

3.1.1 Rahmenbedingungen der Implementierung

Die Entwicklung erfolgte unter dem Betriebssystem IRIX der Firma Silicon Graphics [2]. Dabei wurde darauf geachtet, keine proprietären Komponenten einzusetzen, so daß der Transfer zu einem anderen Vertreter der UNIX–Betriebssystem–Familie leicht möglich ist. Im Laufe der Entwicklung wurde die virtuelle Sicht ohne großen zusätzlichen Aufwand unter den Betriebssystemen HP–UX der Firma Hewlett–Packard [1], Solaris der Firma Sun [18] sowie den frei verfügbaren Open–Source Betriebssystem für verschiedenste Plattformen Linux [15] und FreeBSD[1] [23] übersetzt und getestet. Der Forderung nach Modularität wurde durch den Einsatz der Programmiersprache C++ [26] Rechnung getragen. Diese Sprache, für die auf allen relevanten Plattformen Übersetzer existieren, unterstützt durch die Sprachkonzepte der objektorientierten Programmierung die nötige getrennte Entwicklung von Teilkomponenten und gewährleistet automatisch einen hohen Grad von Modularität. Die Eingesetzte Graphik–API[2], die Bibliothek OpenGL [38], ist der Industriestandard für Graphikapplikationen. Sie existiert ebenfalls auf allen relevanten Plattformen, zusätzlich gibt es eine Open–Source Implementierung MesaGL, die auf jedem UNIX–System mit X–Oberfläche übersetzt werden kann. OSF/Motif [3] wurde zur Implementierung der benötigten Benutzeroberflächenkomponenten herangezogen. Auch diese Bibliothek ist inzwischen, dem Modell von Open–Source Software folgend, der Öffentlichkeit zur Verfügung gestellt worden. Abgesehen davon existiert schon seit längerem eine Open–Source Alternativ–Implementierung der Motif–API unter dem Namen LessTif. Bibliotheken wie Performer [33] oder Inventor [19], die Graphikprogrammierung auf einem höheren Level ermöglichen und auf OpenGL aufsetzen, wurden nicht verwendet. Zunächst, weil es sich um proprietäre Produkte handelte, die nur auf der Plattform des Herstellers Silicon Graphics zur Verfügung standen. Inzwischen sind sie zwar auch auf anderen Plattformen verfügbar, es fehlt ihnen aber die Kontrollmöglichkeit über einzelne Graphik–Primitiven, die

[1] und damit im Prinzip auch OpenBSD und NetBSD

[2] **A**pplication **P**rogrammer **I**nterface

für die Realisierung einer eigenen Geländetriangulierungsstrategie unerlässlich ist.

Durch den Einsatz von Komponenten, bis hin zum Betriebssystem, die auf allen relevanten Plattformen als Open–Source zur Verfügung stehen, wurde die Forderung nach Portabilität optimal erfüllt. Insbesondere durch den Einsatz von Linux bzw. FreeBSD und Bibliotheken, die auf diese Betriebssysteme portiert wurden, ist der Einsatz der virtuellen Sicht auf der Basis kostengünstiger PC–Hardware ermöglicht worden. Damit ihre Leistung, d.h. insbesondere die Bildwiederholraten, auf solchen Plattformen auch ausreichend ist, muß zum einen Hardware–Unterstützung für schnelle OpenGL basierte 3D–Graphik gegeben sein, zum anderen muß die Geländetriangulation die Anzahl der darzustellenden Dreiecke erheblich und vor allem schnell reduzieren. Der Hardware–Support von High–End–Graphikkarten unter Linux und FreeBSD weitet sich ständig aus. Die Hersteller der leistungsfähigsten Karten stellen selbst entsprechende Treiber zur Verfügung. Es bleibt also eine gute und schnelle Geländetriangulierung zu implementieren.

3.1.2 Triangulierungsstrategie

Das Kriterium der Möglichkeit zur Datenkompression war ausschlaggebend für die Wahl einer regulären Triangulierung, die auf hierarchischer Interpolation basiert. Hierarchische Interpolation bietet inhärent die Möglichkeit zur Datenkompression, die z.B. an Hand der *dünnen Gitter* (siehe [40] und [5]) eingesetzt zur Kompression von Bildern ([21]), Audio–Daten ([12]) oder Bildfolgen ([31]) untersucht wurde. Irreguläre Triangulierungen wurden aufgrund der in Abschnitt 2.2.1 erwähnten Nachteile bei der graphischen Darstellung von langen, dünnen Dreiecken nicht in Erwägung gezogen.

Im Rahmen einer Diplomarbeit [14] wurde eine auf dem Dreiecks–Binärbaum (vergleiche Abbildung 2.12 auf Seite 18) basierende Triangulierungsstrategie auf ihre Tauglichkeit im Einsatz zur Generierung einer virtuelle Sicht hin untersucht. Das Ergebnis dieser Evaluierung war positiv, was die Qualität der erzeugten Geländedarstellung angeht, so daß dieser Ansatz auch bei dieser Implementierung der virtuellen Sicht zum Einsatz kommt. Da der im Rahmen dieser Untersuchung implementierte Algorithmus lediglich die Machbarkeit nachweisen sollte und dabei wenig auf Effizienz geachtet wurde, war eine Reimplementierung aufgrund der geringen Bildwiederholrate unumgänglich. Die erzeugte Triangulierung, die auf jedem Level aus rechtwinkligen, gleichschenkligen Dreiecken besteht, ist für eine beschleunigte Darstellung durch Graphik–Hardware deutlich besser geeignet, als z.B. eine, die mit Hilfe der Basisfunktionen der dünnen Gitter konstruiert wurde. Deren Träger überlappen sich innerhalb eines Levels, was dazu führt, daß die Zahl der Facetten aus denen die interpolierte Oberfläche zusammengesetzt wird deutlich höher ist. Zudem verursachen die zahlreichen langen und schmalen Träger der Basisfunktionen der dünnen Gittern bei der Triangulierung zu ebensolchen Dreiecken, womit die Gefahr von Problemen bei deren Darstellung gegeben ist. Die Dreiecks–Binärbaum Hierarchie ist in Hinsicht beider Punkte optimal, da die Träger der Basisfunktionen auf jedem Level quadratisch sind und sich nicht schneiden.

Das von Lindstrom [25] vorgestellte System basiert zwar auf der gleichen Hierarchie, dies wird aber von Lindstrom nicht explizit verwendet. Sein Vorgehen, Blöcke mit verschiedenem Level–of–Detail zu verwalten und deren Triangulierung in einem Bottom–Up Durchlauf zu vergröbern, ist daher umständlich und kostspielig, da die Vorteile der hierarchischen Struktur nicht genutzt werden. Dazu gehört auch die Klarheit und Einfachheit der algorithmischen Beschreibung, die allen Divide&Conquer–Techniken zu eigen ist. Ein Bottom–Up Durchlauf durch die Daten, der auch noch in [14] vorhanden ist stellt bei großen Geländen einen erheblichen Mehraufwand dar, der bei dieser Implementierung mit Hilfe von Hilfsdatenstrukturen vermieden wird. Lindstrom zeigt die Echtzeittauglichkeit seines Systems, indem er ein Gelände als Drahtgittermodell auf einem Zweiprozessor–Rechner vom Typ SGI Onyx RealityEngine^2 (2×150 MHz) darstellt. Das hier zu entwickelnde System muß mit volltexturierten Polygonen bei zusätzlicher Belastung der Graphik durch weitere zu zeichnende Objekte (Gebäude, Straßen, Head–Up–Display etc.) echtzeittauglich sein. Die angestrebte Hardwareplattform ist dabei eine Einprozessor–Workstation SGI Indigo^2 (195 MHz) mit MXI–Graphikkarte, eine deutlich schwächere Graphikworkstation. Vor allem durch die Vermeidung eines Bottom–Up Durchlaufes durch die Daten wird die Leistung der hier vorgestellten Implementierung gegenüber Lindstrom so gesteigert, daß die virtuelle Sicht auf der signifikant schwächeren Plattform befriedigende Ergebnisse liefert.

Der ROAM–Algorithmus [7] scheidet aufgrund seiner ihm innewohnenden Gefahr der in Abschnitt 2.2.1 erläuterten negativen Feedback–Schleife aus. Da der hier zu entwickelnde Algorithmus gerade auch auf kostengünstigen Rechnern zufriedenstellende Ergebnisse liefern muß, deren Leistung weit unter den in [25] und [7] eingesetzten Graphiksupercomputern liegt, ist die Gefahr, daß sich das Problem der negativen Feedback–Schleife in der Praxis zeigt, zu hoch. Aus diesem Grund wurde auf das, für ROAM wesentliche, inkrementelle Vorgehen bei der Triangulierung von Bild zu Bild verzichtet.

3.2 Datenstrukturen

Bei der Implementierung der Geländetriangulierung für die virtuelle Sicht wird die Hierarchie zugrundegelegt, die sich aus dem Dreiecks–Binärbaum ergibt. Damit diese zum Einsatz kommen kann und die als reguläres Gitter vorgegebenen Höhendaten hierarchisch interpoliert werden können, müssen die Höhendaten zunächst in rechteckige Gebiete mit $(2^n{+}1)\times(2^n{+}1)$ Gitterpunkten unterteilt werden. Für den praktischen Einsatz der virtuellen Sicht werden Datensätze mit 1025×1025 Gitterpunkten verwendet. Das entspricht im Bereich von Süddeutschland einem Gebiet von etwa $60\text{km}\times90\text{km}$. Diese Größe ist ausreichend, um Flugversuche durchzuführen. Für den Einsatz der virtuellen Sicht über die Forschung hinaus ist es erforderlich, größere Gebiete zu überfliegen. Dieser Problematik wird in Kapitel 4 behandelt.

3.2.1 Hierarchisierung

Der nächste Schritt besteht darin, diese $(2^n{+}1)\times(2^n{+}1)$ Gitter (implizit) in hierarchische Level zu unterteilen. Abbildung 3.1 zeigt die Level eines 5×5 Gitters. Hierarchische Vorgänger sind grau markiert. Es ist dabei nicht nötig, die Daten tatsächlich in ihre Level zu trennen. Der hierarchische Zugriff kann allein über den Algorithmus realisiert werden, der auf den in einem Array

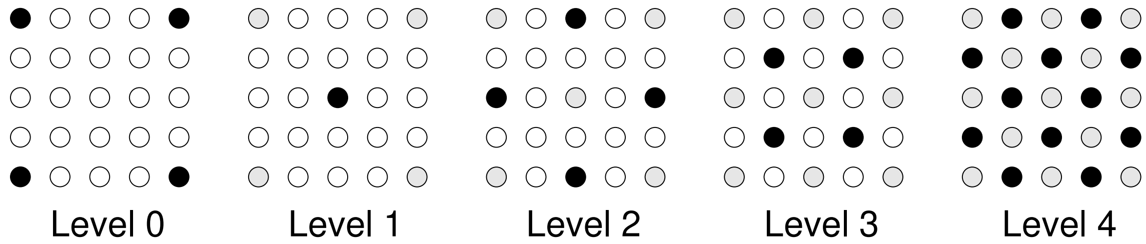


Abbildung 3.1 Hierarchische Level eines 5×5 Gitters.

gespeicherten Höhendaten arbeitet. Dies vermeidet die Probleme, die sich oft beim Arbeiten mit komplizierten und (programmier-) fehleranfälligen Datenstrukturen wie Listen oder Bäumen ergeben. Ein weiterer Vorteil ist, daß aufgrund der kompakt im Speicher liegenden Höhendaten sogenannte *Cache–Prefetch*–Mechanismen, wie sie von modernen Prozessoren eingesetzt werden, die Chance haben, die Zugriffsgeschwindigkeit auf die Daten zu verbessern. Bei Bäumen und Listen ist von vornherein nicht oder nur schwer abschätzbar, wo und in welcher Reihenfolge die Daten schließlich im Speicher abgelegt werden.

Der hierarchische Zugriff auf ein linearisiertes zweidimensionales Array führt zu großen Sprüngen innerhalb des linearen Speichersegmentes. Solche Sprünge machen in der Regel die Vorteile jeder *Cache–Prefetch*–Strategie zunichte. Es ist jedoch möglich, die Daten innerhalb des Arrays so umzuorganisieren, daß ein hierarchischer Datenzugriff lediglich zu Sprüngen nach vorne führt. Auf diese Weise können *Cache–Prefetch*–Mechanismen wieder greifen.

Die Abbildungen 3.2 und 3.3 zeigen die unmittelbaren Vorfahren und Nachfahren innerhalb der Hierarchie.

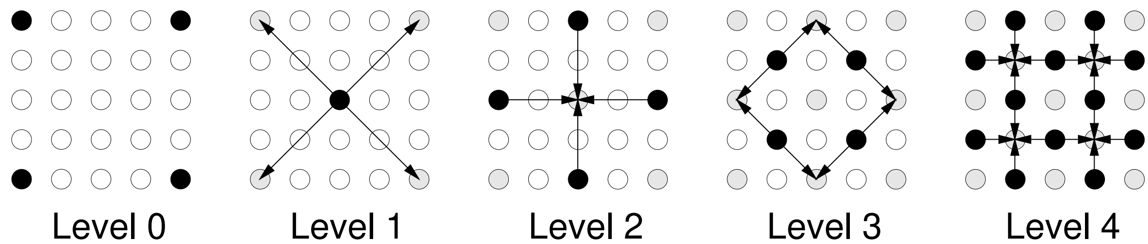


Abbildung 3.2 Hierarchische Vorfahren der Gitterpunkte.

Als Anhaltspunkt für die Wichtigkeit eines jeden Höhenwertes wird der hierarchische Überschuss genutzt, der aus der linearen Interpolation seiner hierarchischen Vorgängern gewonnen wird. Abbildung 3.4 zeigt, welche Punkte verwendet werden, um die Höhenwerte auf jedem Level zu interpolieren. Die Zuordnung folgt direkt aus der Divide&Conquer–Strategie zur Triangulierung. Jeder Punkt wird aus den Randpunkten der Hypotenuse interpoliert, die er bei der rekursiven Triangulierung teilt. Level 1 stellt einen Freiheitsgrad dar, es könnte statt der SW– und NO–Ecken auch die NW– und SO–Ecken zur Interpolation herangezogen werden, die Zuordnungen aller folgenden Level sind eindeutig. Die Berechnung der hierarchischen Überschüsse,

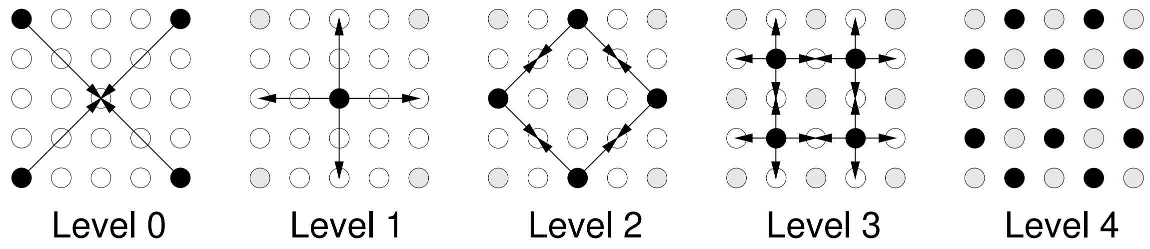


Abbildung 3.3 Hierarchische Nachfahren der Gitterpunkte.

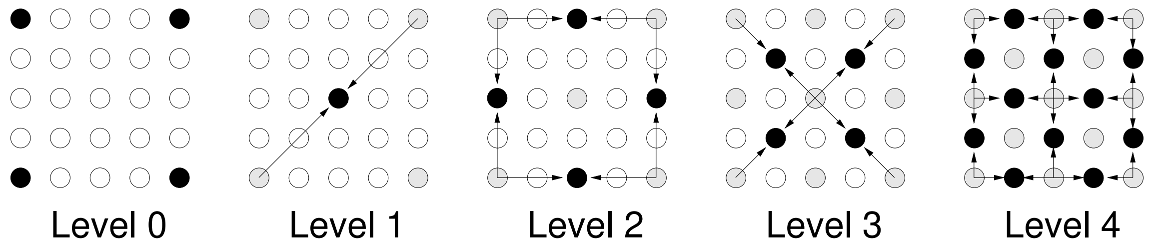


Abbildung 3.4 Vorgänger jedes Punktes, aus denen er interpoliert wird.

erfolgt einmalig in einem Bottom–Up–Durchlauf während der Initialisierungsphase. Die Differenz zwischen interpoliertem und tatsächlichem Wert wird für jeden Punkt in einem zweiten Array gespeichert.

3.2.2 Fehlerpropagation

Um Bottom–Up–Durchläufe bei der Triangulierung zu vermeiden, ist es erforderlich, daß man beim Top–Down–Durchlauf an jedem untersuchten Punkt den betragsmäßig maximalen hierarchischen Überschuss der hierarchischen Nachfahren auf allen tieferen Leveln und des aktuellen Punktes kennt. Zu diesem Zweck wird der hierarchische Überschuss bei einem Bottom–Up Durchlauf in der Hierarchie nach oben propagiert. Die Interpolation und Propagation kann gleichzeitig in einem Durchlauf erfolgen. Danach findet sich der höchste hierarchische Überschuss, der sich bei der Interpolation ergeben hat, im zentralen Punkt und in den vier Eckpunkten des Gitter wieder. Der Algorithmus zum Durchlaufen des Arrays wird im Folgenden beschrieben:
Die Makros `HEIGHT(x,y)` und `SURPLUS(x,y)` liefern den jeweilige Arrayeintrag in einer Form, die es ermöglicht, das Makro auch als L–Value (d.h. links von einem „=“ Zeichen) zu verwenden. Sollte `(x,y)` außerhalb des gültigen Bereichs liegen, so wird auf ein zusätzliches Array–Element mit entsprechendem Sonderstatus abgebildet.

```
offset=2
off=offset/2
while(offset<=2^n+1) {
  for(y=0;y<2^n+1;y+=offset) {
    for(x=off;x<2^n+1;x+=offset) {
      surplus = (HEIGHT(x-off,y)+HEIGHT(x+off,y))/2 - HEIGHT(x,y)
      if(offset==2) SURPLUS(x,y)=|surplus|
      else SURPLUS(x,y) = maximum(|surplus|,
                          SURPLUS(x-off/2,y-off/2),
                          SURPLUS(x+off/2,y-off/2),
                          SURPLUS(x-off/2,y+off/2),
                          SURPLUS(x+off/2,y+off/2))
    }
    if(y+off>=2^n) break
    for(x=0;x<2^n+1;x+=offset) {
      surplus = (HEIGHT(x,y)+HEIGHT(x,y+2*off))/2 - HEIGHT(x,y + off)
      if(offset==2) SURPLUS(x,y+off)=|surplus|
      else SURPLUS(x,y+off) = maximum(|surplus|,
                     SURPLUS(x-off/2,(y+off)-off/2),
                     SURPLUS(x+off/2,(y+off)-off/2),
                     SURPLUS(x-off/2,(y+off)+off/2),
                     SURPLUS(x+off/2,(y+off)+off/2))
    }
  }
  for(y=off;y<2^n+1;y+=offset) {
    for(x=off;x<2^n+1;x+=offset) {
      if((int)|(x-y)/offset|&(int)1)
        surplus = (HEIGHT(x-off,y+off)+HEIGHT(x+off,y-off))/2 - HEIGHT(x,y)
      else
        surplus = (HEIGHT(x-off,y-off)+HEIGHT(x+off,y+off))/2 - HEIGHT(x,y)
      SURPLUS(x,y) = maximum(|surplus|,
                     SURPLUS(x+off,y),
                     SURPLUS(x-off,y),
                     SURPLUS(x,y+off),
                     SURPLUS(x,y-off))
    }
  }
  offset = offset << 1
  off=offset/2
  }
```

Der Weg, den die hierarchischen Überschüsse gehen wenn dieser Algorithmus für ein 5×5–Gitter abgearbeitet wird (d.h. $n = 2$), ist in Abbildung 3.5 illustriert. Die Überschüsse des untersten Level (graue Punkte) wandern in Pfeilrichtung zu ihren benachbarten hierarchischen Vorgängern. Die Dicke der Pfeile spiegelt den Level wieder, die dünnsten Pfeile geben den Weg auf dem untersten, die dicksten Pfeile den Weg auf dem höchsten Level wieder. Zeigen auf

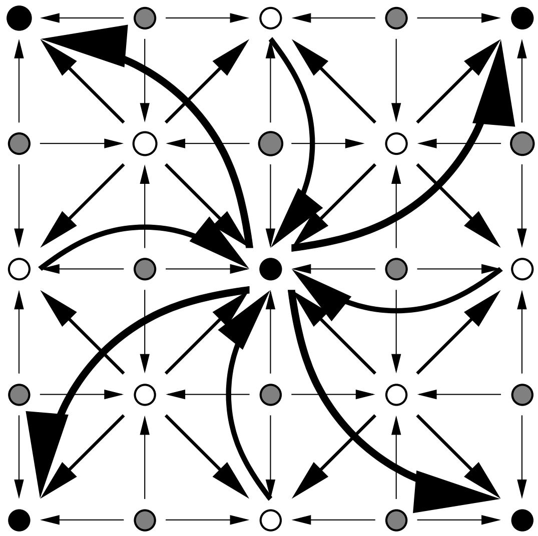


Abbildung 3.5 Weg der Überschüsse in der Hierarchie nach oben.

einen Punkt Pfeile aus verschiedenen Leveln (z.B. die Eckpunkte) so werden hier Überschüsse direkt übertragen, die auch über einen Umweg dort gelandet wären. Dieser geringfügige Mehraufwand, der im Rahmen der Initialisierung ohnehin vernachlässigbar ist, erlaubt eine stark vereinfachte algorithmischen Beschreibung, da sich die Behandlung von am Rand des Gitters auftretenden Sonderfällen erübrigt.

Die so in einem Fehler–Array gesammelten Informationen ermöglichen es im Folgenden, die Triangulierung ausschließlich mit Hilfe von Top–Down–Durchläufen durch den Datensatz zu realisieren. Zusätzlich kann durch das Nach–Oben–Propagieren der hierarchischen Überschüsse garantiert werden, daß bei der Geländetriangulierung keine Spitzen in den Höhendaten übersehen werden, da eine obere Fehlergrenze garantiert werden kann. Der im Fehler–Array abgelegte Wert gibt die maximale Höhendifferenz wieder, die jeder der hierarchischen Nachfolger des jeweils betrachteten Punktes bei seiner Berücksichtigung bei der Triangulierung beitragen könnte. Also ist der Fehler–Array Wert multipliziert mit der Anzahl der noch folgenden Level eine Obergrenze für die maximal mögliche Höhendifferenz innerhalb des Gebiets, das von dem Dreieck abgedeckt wird, das zu dem jeweiligen Fehler–Array Wert gehört. Für den Einsatz der virtuellen Sicht besonders interessant ist, daß dies mit einer geeigneten Sonderbehandlung bei der Sichtbarkeitsentscheidung auch in dem Bereich zwischen unterem Bildrand und Betrachter garantiert werden kann. Die ursprüngliche Implementierung der virtuellen Sicht ([28]) hatte u.a. für dieses Problem keine befriedigende Lösung. Abbildung 3.6 veranschaulicht die Problematik.

Bei der tatsächliche Implementierung werden noch weitere Hilfsdatenstrukturen eingesetzt. Es handelt sich um Bit–Felder zur Markierung von Punkten und Byte–Felder, um Ergebnisse von Sichtbarkeitsentscheidungen abzulegen.

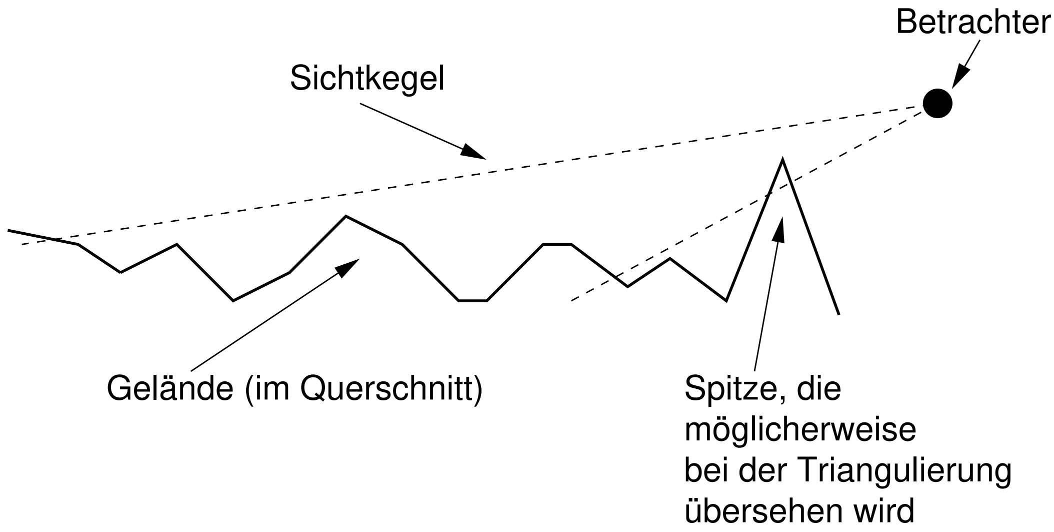


Abbildung 3.6 Spitzen in den Höhendaten, bei denen die Gefahr des Übersehens besteht.

3.3 Algorithmus

Nach dem Aufbau des Fehler–Arrays sind alle Voraussetzungen für eine Triangulierung der Höhendaten geschaffen, die in diesem Abschnitt beschrieben wird. Die Triangulierung erfolgt für jede zu erzeugende Darstellung des Geländes auf dem Bildschirm neu. Hierzu müssen die Hilfsdatenstrukturen nach erfolgter Triangulierung wieder in den Ausgangszustand zurückversetzt werden, d.h. in der Regel, daß sie mit Null vorbelegt werden müssen. Da es sich um Arrays von vergleichsweise geringer Größe handelt, ist der Zeitbedarf für diese Operation vernachlässigbar gering.

3.3.1 Triangulierung

Die Orientierung der Dreiecke ist in diesem und vor allem im folgenden Kapitel für die Erklärung der Algorithmen und die Konsistenz der abgedruckten Pseudo–Code Stücke wesentlich, da die Nachbarschaftsrelationen der Dreiecke zunächst nur durch ihre Position im Binärbaum bestimmt wird. Abbildung 3.7 zeigt die Zuordnung zwischen zugrundeliegender Geometrie und den Bezeichnern, wie sie im Folgenden verwendet werden.

Die Idee, die hinter der Triangulierung steht, ist es Top–Down, der Hierarchie folgend, durch die Daten zu laufen, bis der Wert im Fehler–Array, der dem Punkt $\frac{a+b}{2}$ entspricht, an dem das aktuell untersuchte Dreieck geteilt werden müsste, unter einer Fehlerschranke liegt. Das Dreieck, bei dem der Baumdurchlauf abbricht, wird gezeichnet.

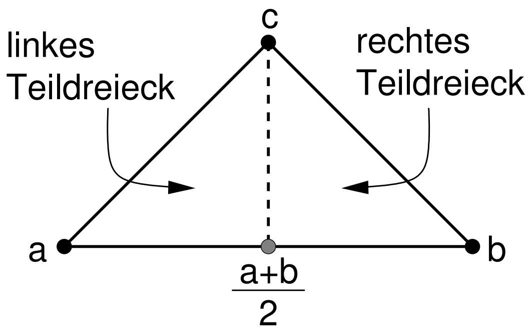


Abbildung 3.7 Zuordnung zwischen Bezeichnern und Geometrie.

Baumdurchlauf

Ein Top–Down–Durchlauf durch die Daten läßt sich mit der folgenden einfachen rekursiven Prozedur implementieren.

```
untersuche_dreieck(a,b,c)
{
  if(Wert im Fehler-Array bei (a+b)/2 < ε)
    zeichne_dreieck(a,b,c)
  else {
    untersuche_dreieck( c, a, (a+b)/2)
    untersuche_dreieck( b, c, (a+b)/2)
  }
}
```

Abbildung 3.8 zeigt zwei Folgen von Dreiecken, die während der adaptiv rekursiven Triangulierung eines 5×5–Gitters untersucht werden. Um das gesamte Gitter zu triangulieren, muß die Prozedur `untersuche_dreieck()` zweimal aufgerufen werden. Einmal für das Dreieck, das aus der SW–, NO– und NW–Ecke des Arrays gebildet wird, und einmal für das Dreieck, das durch die NO–, SW– und SO–Ecke begrenzt wird. Der Wert im Fehler–Array, das während der Initialisierung aufgebaut wurde, ist an den schwarzen Gitterpunkten höher als die vorgegebene Fehlerschranke ε, an den weißen Gitterpunkten liegt der Wert darunter. Punkte, die in einem bestimmten Level in der `if`–Abfrage untersucht werden, sind durch einen Kreis markiert. Abbildung 3.9 zeigt die sich ergebende Triangulierung des gesamten Gitters. In diesem Beispiel hängt die Fehlerschranke ε nicht vom Abstand zum Betrachter ab und ist für alle Punkte gleich.

Sichtbarkeitsentscheidung

Beim Einsatz dieses Triangulierungsschemas im Rahmen der virtuellen Sicht ist es sinnvoll, die Prozedur `untersuche_dreieck()` mit einer Terminierungsregel zu versehen, die überprüft, ob das gerade untersuchte Dreieck für den Betrachter sichtbar ist bzw. sein könnte. So

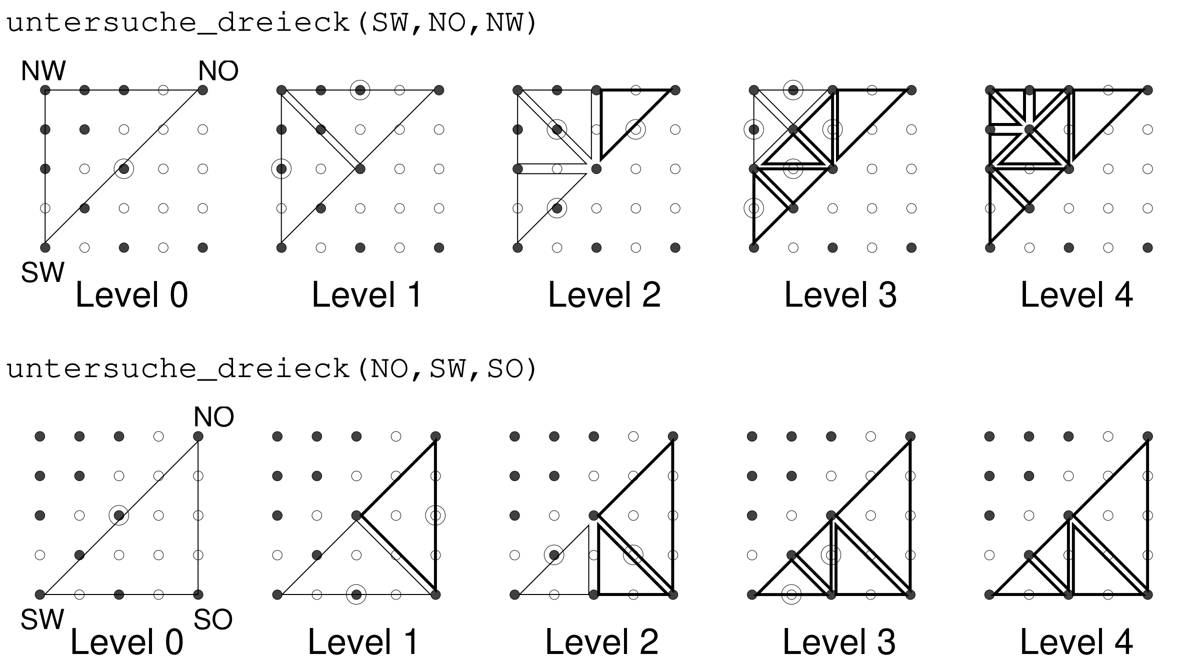


Abbildung 3.8 Folge von Dreiecken, die während einer Triangulierung untersucht werden.

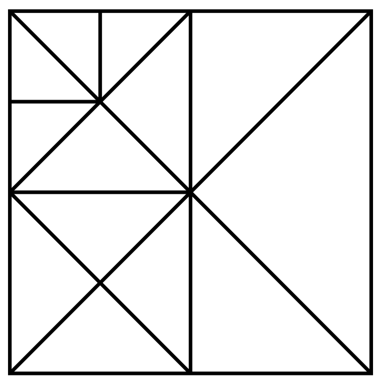


Abbildung 3.9 Die sich ergebende Triangulierung des Gitters.

kann überflüssige Arbeit vermieden werden, und die meisten der rekursiven Prozeduraufrufe können bereits zu einem sehr frühen Zeitpunkt abgebrochen werden.

Das Verfahren, das zur Ermittlung der Sichtbarkeit eines Dreiecks eingesetzt wird, ist von einer 3D–Version des bekannten Cohen–Sutherland–line–clipping–Algorithmus ([9]) abgeleitet. Der wesentliche Unterschied ist, daß es auf Dreiecke anstelle von Linien angewandt wird. Damit der Cohen–Sutherland–Algorithmus zum Einsatz kommen kann, muß für den Fall, daß ein Punkt nicht sichtbar ist, bestimmt werden, wo er außerhalb des sichtbaren Bereiches liegt (davor, dahinter, links, rechts, darüber oder darunter). Damit läßt sich für jede Ecke eines Dreiecks eine Bitmaske konstruieren, die je nachdem, in welchen Sektoren außerhalb des sichtbaren Bereiches sie liegt auf `1` bzw. `0` gesetzt wird. Durch eine logische Und–Verknüpfung dieser Bitmasken für die Ecken des zu untersuchenden Dreiecks kann dann festgestellt werden, ob sich eine Dreieck sicher außerhalb des sichtbaren Bereichs befindet. Sollte dies der Fall sein, so kann die Rekursion und damit die Triangulierung abgebrochen werden. Dieser Fall tritt z.B. dann ein, wenn sich alle drei Ecken links des sichtbaren Bereichs befinden.

Dieses Verfahren macht für jeden während einer Triangulierung untersuchten Punkt eine 4×4–Matrix–Vektor–Multiplikation erforderlich. Die Alternative, nicht auf (potentielle) Sichtbarkeit zu prüfen, würde zu einem wesentlich höheren Arbeitsaufwand bei der Untersuchung von unsichtbaren Dreiecken führen. Zudem läßt sich der Aufwand für die Matrix–Vektor–Multiplikationen durch verschiedene Maßnahmen, die in Abschnitt 3.3.2 erläutert werden, auf ein erträgliches Niveau senken.

Damit die in Abschnitt 3.2.2 angesprochene Berücksichtigung von möglichen Spitzen zwischen unterem Bildrand und Betrachter (vergleiche Abbildung 3.6) erfolgt müssen Dreiecke, die zwar ganz unterhalb des sichtbaren Bereichs liegen und somit nach Cohen–Sutherland ignoriert werden könnten einer zusätzlichen Prüfung unterzogen werden, bevor sie verworfen werden können. Dabei wird anhand der oberen Fehlerschranke, dem Produkt aus Fehler–Array Wert und der Anzahl noch folgender Level, getestet, ob in dem untersuchten Dreieck eine Spitze enthalten sein könnte, die wie in Abbildung 3.6 in der Lage wäre, den sichtbaren Bereich zu schneiden. Ist dies der Fall wird das Dreieck weiter untersucht. Dies hat zur Folge, daß wirklich nur die Dreiecke untersucht und ggf. auch gezeichnet werden, die so eine Spitze enthalten. Bei allen anderen wird aufgrund der Sichtbarkeitsprüfung die Rekursion sofort abgebrochen.

Adaptionskriterium

Bei der Triangulierung in Abbildung 3.8 wurde lediglich die Flachheit des Geländes ausgenutzt um die Anzahl der zu zeichnenden Dreiecke zu reduzieren. Damit ein kontinuierlicher Level–of–Detail ermöglicht wird, muß die Fehlerschranke ε von dem Abstand zwischen Betrachter und gerade untersuchtem Dreieck abhängig gemacht werden. Dazu wird ε durch eine Funktion `eps()` ersetzt, die geeignet konstruiert wird. Die `if`–Anweisung in `untersuche_dreieck()` ändert sich dann folgendermaßen:

```
if(Wert im Fehler-Array bei (a+b)/2 < eps(d(a,b,c)) )
```

Die Funktion `d()` liefert dabei den Abstand zwischen dem Betrachter und dem als Parameter übergebenen Dreieck.

Die Funktion `eps()`, die letztendlich den Verlauf des Level–of–Detail bei der Triangulierung bestimmt, kann völlig frei vom Benutzer gewählt werden. Einige Beispiele, wie eine brauchbare `eps()`–Funktion aussehen könnte, zeigt Abbildung 3.10. Die linke Funktion ist die, die zur Zeit in der virtuellen Sicht eingesetzt wird. Der zulässige Fehler ist hier Null innerhalb eines Abstands von 1–5km vom Betrachter. Dies entspricht einer exakten Darstellung des Geländes innerhalb dieses Radius. Danach steigt die Fehlerschranke linear auf einen Wert von 600m in einer Entfernung von 25km. Die rechte Funktion könnte eingesetzt werden, um drei separate Level–Of–Detail–Bereiche zu approximieren wie sie in der ursprünglichen Implementierung der virtuelle Sicht ([28]) verwendet wurden. Vorteil dieser `eps()`–Funktion ist es, daß sich die Triangulierung des Geländes nur in der Nähe der Sprungstellen der Funktion ändert.

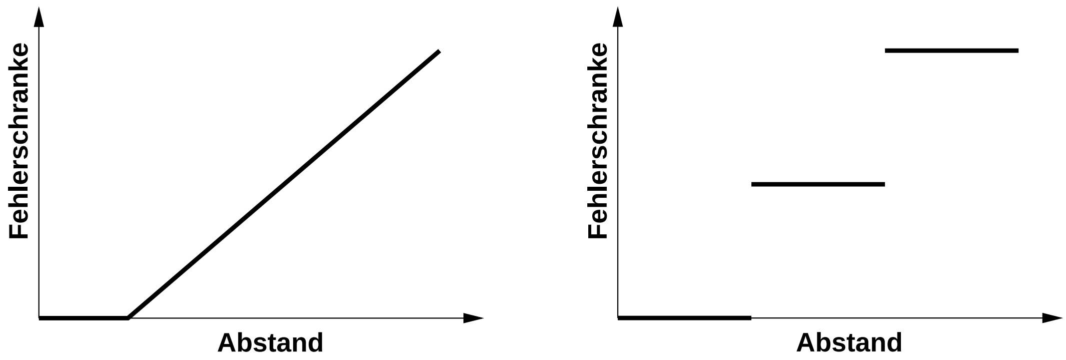


Abbildung 3.10 Beispiele für sinnvolle `eps()`–Funktionen.

Vermeiden von vertikalen Löchern

Die (beabsichtigte) Konsequenz des Wechsels von der konstanten Fehlerschranke ε zur Funktion `eps()` ist, daß unterschiedliche Fehlerschranken auf die Dreiecke angewendet werden, die während der rekursiven Aufrufe von `untersuche_dreieck()` untersucht werden. Das kann dazu führen, daß bei benachbarten Dreiecken eines tiefer rekursiv verfeinert wird als das andere. Diese Situation kann zu T–Kreuzungen in der Triangulierung führen, was sich schließlich als vertikale Löcher im Gelände bemerkbar macht. Im Bereich der Finiten Elemente ist dies ein Standardproblem und tritt unter dem Stichwort *dangling nodes* auf, siehe z.B. [27]. Diesen wird begegnet, indem Nachbardreiecke solange verfeinert werden, bis in der Triangulierung keine T–Kreuzungen mehr vorkommen. Anders als bei den Finiten Elementen existiert in der Anwendung der virtuellen Sicht ein feinstmögliches Gitter, das durch die Ausgangsdaten festgelegt wird, was dazu führt, daß eine zusätzliche Verwaltung eines *vef*–Graphen zur Bestimmung von Nachbardreiecken nicht nötig ist. Die Zuordnung zwischen Dreiecken und Nachbardreiecken ist hier durch die Lage der Dreiecke im Datensatz und die Beziehungen zwischen hierarchischen Vor- und Nachfahren gegeben. In diesem Fall ist es eine einfache Methode zur Vermeidung von T–Kreuzungen die Eckpunkte jedes untersuchten Dreiecks in einem Bitfeld zu markieren und die `if`–Anweisung in `untersuche_dreieck()` folgendermaßen zu ergänzen:

```
if(Punkt (a+b)/2 nicht im Bitfeld markiert &&
   Wert im Fehler-Array bei (a+b)/2 < eps(d(a,b,c)) )
```

Ein markierter Punkt ist also Bestandteil der Triangulierung, d.h. er dient als Ecke für mindestens eines der gezeichneten Dreieck. Die Markierung soll verhindern, daß die Rekursion abbricht, obwohl der Punkt in einem anderen Rekursionszweig bereits in die Triangulierung übernommen wurde. Damit dieser Ansatz zu dem gewünschten Ergebnis führt, muß das Dreieck, das zu einer tieferen Ebene rekursiv verfeinert wird, beim rekursiven Aufruf zuerst untersucht werden. Das führt zu dem Problem zu entscheiden, welches der Dreiecke `(c,a,(a+b)/2)` und `(b,c,(a+b)/2)` zuerst untersucht werden soll. Bis jetzt wurde noch keine einfache und vor allem sichere Methode gefunden, mit der abgeschätzt werden kann, wie tief ein Dreieck rekursiv verfeinert werden wird. Verkompliziert wird die Entscheidung dadurch, daß die virtuelle Sicht den Einsatz verzerrter Dreiecke verlangt.

Naheliegende, aber falsche Strategie

Es läßt sich leicht zeigen, daß der naive Lösungsansatz, das nähergelegene Dreieck zuerst zu verfeinern, zu vertikalen Löchern führen kann. Abbildung 3.11 illustriert diesen Fall. In diesem

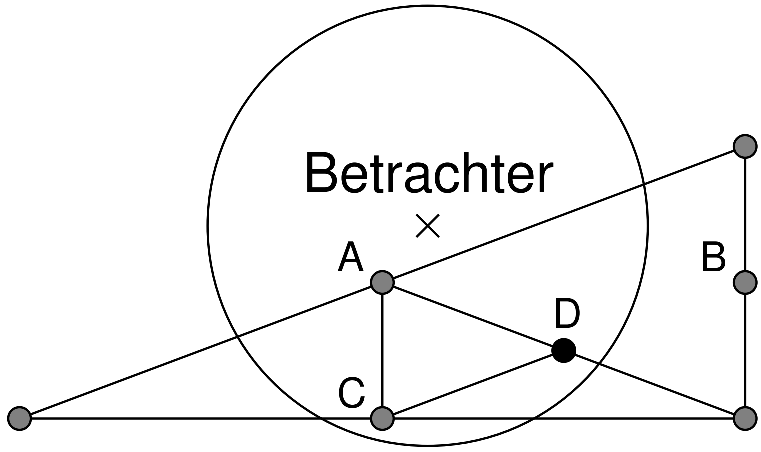


Abbildung 3.11 Beispiel einer Fehlerhaften Triangulierung.

Beispiel hat der schwarze Punkt den höchsten hierarchischen Überschuss. Durch das Hochpropagieren dieses Überschusses beim Aufbau des Fehler–Arrays während der Initialisierung verteilt sich dieser Wert auf die grauen Punkte. Der Kreis um den mit $\times$ markierten Betrachterstandort gibt die Grenze an, ab der

```
Wert im Fehler-Array bei (a+b)/2 < eps(d(a,b,c))
```

wahr ist. Innerhalb des Kreises gilt dies nicht. Bei der rekursiven Triangulierung wird zuerst Punkt A untersucht, was zur Entscheidung der rekursiven Verfeinerung führt. Wird das nähergelegene Dreieck zuerst untersucht, führt die Auswertung der `if`–Anweisung für den Punkt B zum Abbruch der Rekursion und zum Zeichnen des Dreiecks. Anschließend wird das Dreieck, das durch den Punkt C geteilt wird, untersucht, das ebenso wie das durch D geteilte Dreieck verfeinert wird. Dadurch kommt es zu einer T–Kreuzung und damit zu einem vertikalen Loch in der Triangulierung am Punkt D.

Die weiteren untersuchten Ansätze beruhen im wesentlichen auf dem oben geschilderten, wobei die Abstandsfunktion variiert sowie verschiedene Gewichtungen der Werte im Fehler–Array untersucht wurden. Ebenso wie in Abbildung 3.11 läßt sich für jeden der untersuchten Lösungsansätze ein Gegenbeispiel, d.h. ein Triangulierung mit T–Kreuzungen konstruieren.

Korrekte zwei–pass Strategie

Um das Problem der vertikalen Löcher völlig zu vermeiden, wurde eine zwei–pass–Triangulierung implementiert. Im ersten rekursiven Top–Down–Durchgang werden die Eckpunkte von durchlaufenen Dreiecken und deren hierarchische Vorfahren in einem Bitfeld markiert, anstatt die Dreiecke sofort zu zeichnen. Das Zeichnen erfolgt dann in einem zweiten Durchgang, in dem nur noch das Bitfeld untersucht wird. Trifft die Prozedur in diesem Durchgang auf einen unmarkierten Punkt $\frac{a+b}{2}$, so wird das aktuelle Dreieck gezeichnet und dieser Zweig der Rekursion terminiert.

Durch die Markierung aller hierarchischen Vorfahren (vergleiche Abbildung 3.2 auf Seite 24) wird sichergestellt, daß die erzeugte Triangulierung zulässig, d.h. lochfrei ist. Löcher entstehen genau dann, wenn ein Punkt in einem Zweig der Rekursion markiert wird, während dies in dem anderen Rekursionszweig, der den selben Punkt erreicht bzw. erreichen würde, unterbleibt. Abbildung 3.12 zeigt ein Beispiel für solch eine Situation, die aufgrund der abstandsabhängigen Fehlerschranke auftreten kann. Die schwarzen Punkte sind markiert, die weißen Punkte nicht.

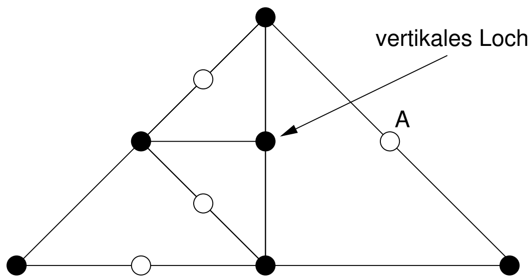


Abbildung 3.12 Vertikales Loch verursacht durch unvollständige Markierung.

Der Punkt A, der ein hierarchischer Vorfahr des Punktes ist an dem im Beispiel das vertikale Loch entsteht, wurde nicht markiert, wodurch die Rekursion beim Zeichnen des rechten Teildreiecks zu früh terminiert. Die Markierung der hierarchischen Vorfahren aller zu markierenden Dreiecksecken entspricht dem Teilen der Nachbardreiecke bei FE–Triangulierungen.

Die Markierung aller hierarchischen Vorgänger eines Punktes in einem Array ist leicht mit Hilfe einer ähnlichen Indexrechnung möglich, wie sie schon bei der Initialisierung zur Berechnung der Werte im Fehler–Array angewandt wurde. Folgende Prozedur markiert alle hierarchischen Vorfahren eines Punktes: In den Parametern `x` und `y` wird die Gitterposition des Punktes übergeben, `level` gibt den hierarchischen Level an auf dem der Punkt liegt.

```
markiere_vorfahren( x, y, level)
{
  if(level<1 || ist_markiert(x,y)) return
  markiere(x,y)
  offset = 2^n
  offset = offset>>(level/2-1+level%2)
  h_off = offset/2
  if(level%2) {
    if((int)(|x-y|/offset) & (int)1) {
      markiere_vorfahren(x-h_off,y-h_off,level-1)
      markiere_vorfahren(x+h_off,y+h_off,level-1)
      }
    else {
      markiere_vorfahren(x-h_off,y+h_off,level-1)
      markiere_vorfahren(x+h_off,y-h_off,level-1)
      }
```

```
    }
  else {
    if(x%offset == 0) {
      markiere_vorfahren(x-h_off,y,level-1)
      markiere_vorfahren(x+h_off,y,level-1)
    else {
      markiere_vorfahren(x,y+h_off,level-1)
      markiere_vorfahren(x,y-h_off,level-1)
      }
    }
}
```

Damit läßt sich die zwei–pass–Version von `untersuche_dreieck()`, die auch die Sichtbarkeitsüberprüfung enthält, wie folgt schreiben:

```
markiere_dreieck(a,b,c,level)
{
  if(dreieck_nicht_sichtbar(a,b,c)) return()
  if(Wert im Fehler-Array bei (a+b)/2 < eps(d(a,b,c)))
    markiere(a,b,c,level)
  else {
    markiere_dreieck( c, a, (a+b)/2, level+1)
    markiere_dreieck( b, c, (a+b)/2, level+1)
    }
}

untersuche_und_zeichne_dreieck(a,b,c)
{
  if(dreieck_nicht_sichtbar(a,b,c)) return()
  if(Punkt (a+b)/2 ist nicht markiert)
    zeichne_dreieck(a,b,c)
  else {
    untersuche_und_zeichne_dreieck( c, a, (a+b)/2)
    untersuche_und_zeichne_dreieck( b, c, (a+b)/2)
    }
}
```

Die Prozedur `markiere()` ruft für die drei übergebenen Punkte `markiere_vorfahren()` auf. Um nun ein rechteckiges Gebiet (NW, NO, SO, SW) vollständig zu triangulieren, sind folgende Aufrufe erforderlich:

```
markiere_dreieck(SW,NO,NW,0)
untersuche_und_zeichne_dreieck(SW,NO,NW)
markiere_dreieck(NO,SW,SO,0)
untersuche_und_zeichne_dreieck(NO,SW,SO)
```

3.3.2 Weitere Techniken zur Effizienzsteigerung

Die Implementierung der virtuellen Sicht basiert auf den im letzten Abschnitt vorgestellten Prozeduren. Um jedoch die geforderten Leistungsmerkmale erfüllen zu können, sind weitere Verbesserungen erforderlich, die die Laufzeit der Triangulierung in der Praxis deutlich reduzieren und so entweder höhere Bildwiederholraten oder eine bessere Geländeapproximation ermöglichen. Darüberhinaus werden Möglichkeiten aufgezeigt, die zwar nicht realisiert wurden, die aber für andere Anwendungen als die virtuelle Sicht deutliche Vorteile bringen können.

Berücksichtigung des Nick–Winkels

Eine zusätzliche Reduktion der Anzahl der Dreiecke kann erreicht werden, wenn der Nick–Winkel zwischen Betrachter und untersuchtem Dreieck berücksichtigt wird. Im Extremfall blickt der Betrachter senkrecht nach unten auf das Gelände. In diesem Fall sind Höhenunterschiede des Geländes nur bedingt wahrnehmbar, zumal das Dreiecksnetz mit der beleuchteten Geländestruktur (vergleiche Abbildung 2.7 auf Seite 13) texturiert ist. Der zulässige Fehler kann also mit steigendem Nick–Winkel anwachsen, im Falle von 90 Grad darf der Fehler beliebig groß sein. Dies erreicht man, indem man die Funktion `eps()` mit dem Kosinus des Nick–Winkels multipliziert. Da diese Verbesserung in der Praxis jedoch nur zu einer sehr geringfügigen Reduktion der Dreiecke führt[3] und sie zudem durch eine aufwendige Winkel– und Kosinus–Berechnung pro untersuchtem Punkt erkauft werden muß, wurde für die virtuelle Sicht darauf verzichtet.

Statt dem Nick–Winkel zwischen Betrachter und untersuchtem Dreieck könnte als grobe Näherung auch nur der Nick–Winkel des Betrachters herangezogen werden. Dies führt aufgrund der Zentralprojektion zu Fehlern bzw. Verzerrungen, die am Bildrand am stärksten sind. Auch hier ist der Nutzen in der Praxis so gering, das in der aktuellen Implementierung der virtuellen Sicht der Nick–Winkel nicht berücksichtigt wird.

Vermeidung überflüssiger Arbeit bei der Sichtbarkeitsentscheidung

Um die Sichtbarkeitsentscheidung für Dreiecke zu beschleunigen, werden die errechneten Bitmasken für jeden Punkt in einem Flag–Feld zwischengespeichert. So wird die Berechnung für jeden Punkt höchstens einmal durchgeführt und auch mehrere Top–Down–Durchläufe im Rahmen einer Triangulierung ziehen keine Mehrfachberechnungen nach sich. Sobald festgestellt wurde, daß alle Ecken eines Dreiecks im sichtbaren Bereich liegen kann auf weitere Sichtbarkeitsprüfungen verzichtet werden. Da ganz außerhalb des sichtbaren Bereichs liegende Dreiecke ohnehin nicht weiter verfeinert werden, beschränkt sich die Sichtbarkeitsentscheidung auf den Rand des sichtbaren Bereiches. Eine weitere Beschleunigung würde inkrementelles Vorgehen mit sich bringen, das jedoch für die virtuelle Sicht nicht mehr implementiert wurde.

[3] Die Blickrichtung senkrecht nach unten kommt fast nie vor, da für Piloten das Gelände das vor ihnen ist von deutlich größerem Interesse ist als das, das unter ihnen liegt.

Angabe des zulässigen Fehlers in Pixeln statt Metern

Neben dem Festlegen des Fehlers in Höhenmetern kann mit Hilfe einiger OpenGL–Funktionen auch leicht eine Transformation in den Bildraum durchgeführt werden, so daß man den zulässigen Fehlern in Pixeln angeben kann. Hierzu muß lediglich die „Höhe“ eines Pixels an der Near– und an der Far–Clipping–Plane gemessen werden[4]. Diese Messung muß nur einmal nach der Initialisierung des Graphik–Fensters, in dem die virtuelle Sicht angezeigt wird, durchgeführt werden und jedesmal, wenn sich die Projektions–Matrix oder die Fenstergröße ändert. Dies kommt in der Regel nur sehr selten vor.

Die Angabe in Pixeln statt Metern läßt eine bessere Kontrolle des Fehlers in Bezug auf die Bildqualität zu. Aus diesem Grund wird dieses Fehlermaß auch von Lindstrom [25] und anderen verwendet.

Level–of–Detail für DFAD–Daten

Die neben dem Gelände zusätzlichen zu zeichnenden Strukturmerkmale (z.B. Straßen, Flüsse, Hochspannungsmasten, Brücken) können ebenfalls einem Level–of–Detail Ansatz unterworfen werden. Für die virtuelle Sicht wurde wie folgt vorgegangen: Strukturmerkmale werden erst ab einer festgelegten Entfernung, die kleiner als die Sichtweite ist, gezeichnet und durchlaufen mit geringer werdendem Abstand zum Betrachter drei Level–of–Detail–Intervalle. Im dem Betrachter am nächsten gelegenen Intervall werden ein- und zweidimensionale Strukturmerkmale vollständig gezeichnet, d.h. entsprechende Objekte für ein- und texturierte Polygone für zweidimensionale Merkmale. Im nächsten Intervall werden für zweidimensionale Merkmale nur noch Linienzüge gezeichnet, deren Farbe der Mittelwert der Textur ist. Eindimensionale Merkmale werden durch eine Standardform dargestellt, die auch für unbekannte Objekte im ersten Level–of–Detail verwendet wird. Diese Standardform wird in der Praxis aus vier Linien zusammengesetzt, die einen das Objekt umschreibenden Quader entlang seiner Diagonalen durchlaufen, sie ist also extrem schnell zu zeichnen. Im letzten Intervall wird beim Zeichnen der Linienzüge für zweidimensionale Merkmale nur noch jeder zweite Punkt verwendet. Eindimensionale Merkmale werden nur noch durch einen senkrechten Strich dargestellt.

Suchbaum für zweidimensionale Strukturdaten

Nicht unwesentlich viel Rechenzeit wird für die Darstellung der zweidimensionalen Strukturdaten investiert. Um die Situation zu entspannen, wurden diese Strukturdaten automatisch in einem Suchbaum organisiert, so daß nicht nach jeder Triangulierung die komplette, zum Teil sehr lange Liste nach zu zeichnenden zweidimensionalen Merkmalen durchsucht werden muß. Die Aufgabenstellung, den Suchaufwand zu minimieren ist eng verwandt mit dem Problem, beim Raytracen die Anzahl der Schnitte zwischen Objekten und Lichtstrahl auf der Suche nach

[4] z.B. mit Hilfe der Funktionen `gluProject()` und `gluUnProject()`

dem Objekt, das von dem aktuell verfolgten Lichtstrahl getroffen wird, zu minimieren. Hierzu findet sich in [10] bzw. ursprünglich in [16] ein Algorithmus, der aus einer Liste vorgegebener Boundingboxes automatisch einen gut balancierten Suchbaum generiert. Der Einsatz dieses Suchbaums hat den Aufwand, der beim Zeichnen der zweidimensionalen Strukturdaten betrieben wird zwar deutlich reduziert, aufgrund ihrer oft sehr hohen Anzahl (abhängig vom jeweiligen Geländedatensatz) erfordern sie jedoch immer noch einen signifikanten Anteil der benötigten Rechenzeit.

Parallelisierung

Wie alle auf dem Divide&Conquer Prinzip aufbauenden Algorithmen läßt sich auch bei der virtuellen Sicht eine einfach Parallelisierung erreichen, indem man für die rekursiven Aufrufe während der Triangularisierung jeweils einen eigenen Thread startet. Mehrere Threads sind jedoch nur auf einem Mehrprozessorrechner wirklich sinnvoll. Da die angestrebte Zielplattform in der Regel nur eine CPU zur Verfügung stellt, wurde auf diese Form der Parallelisierung verzichtet.

Das Programm der virtuelle Sicht wird am Lehrstuhl für Flugmechanik und Flugregelung auch für die Erzeugung der Außensicht des am Lehrstuhl installierten Flugsimulators eingesetzt. In diesem Szenario, in dem das Programm auf einem SGI Onyx Graphik–Supercomputer mit InfiniteReality Graphik und zwei CPUs läuft, macht eine Parallelisierung Sinn. Es hat sich jedoch gezeigt, daß jeweils ein eigener Thread für rekursive Aufrufe bei der Triangulierung in der Praxis bei weitem zu feingranular ist. Mehr Sinn macht es, die Geländetriangulierung an sich und die Suche nach sichtbaren Strukturdaten parallel durchzuführen.

Maßnahmen, um *poppen* des Geländes zu vermeiden

Ein Problem, das sich bei den meisten Geländetriangulierungsverfahren mit kontinuierlichem Level–of–Detail zeigt, ist das sogenannte *Poppen* des Geländes. Dies wird durch den abrupten Wechsel des Detailgrades in einem Bereich beim Übergang von einem zum nächsten Bild verursacht. Wird z.B. ein Dreieck gegenüber dem vorangegangenen Bild um einen Level tiefer verfeinert, so kann durch den zusätzlich berücksichtigten Höhenwert der Eindruck entstehen, daß das Gelände um die Größe des hierarchischen Überschusses nach oben oder unten springt. Der Höhenunterschied liegt zwar innerhalb der Fehlertoleranz, aber die Geländedarstellung wird durch diesen Effekt als unruhig empfunden.

Es gibt verschiedene Möglichkeiten, diesem Eindruck entgegen zu wirken. Die aufwendigste ist es, zwischen den beiden Triangulationszuständen überzublenden. Dazu ist es erforderlich, das Gelände im kritischen Bereich zweimal zu zeichnen, wobei über mehrere Einzelbilder hinweg mittels Alpha–Blending vom alten in den neuen Zustand übergeblendet wird. Entscheidender Nachteil dieses Vorgehens ist, daß Geländestücke mehrfach gezeichnet werden müssen.

Eine weitere Methode ist das sogenannte Geomorphing. Dabei wird zwischen den beiden verschiedenen Triangulationszuständen interpoliert und über mehrere Einzelbilder die Geometrie

Schritt für Schritt von der alten in die neue Triangulation überführt. Röttger [34] verwendet in seinem Algorithmus statt einem Bitfeld zur Markierung von Punkten ein Bytefeld. Der Wert eines Bytes spiegelt den Morphfaktor des zugehörigen Höhenwertes wieder. Die damit darstellbaren 255 Zwischenschritte (die 0 entspricht dem unmarkierten Zustand) sind ausreichend, um einen scheinbar kontinuierlichen Übergang des Geländes von einem Triangulationszustand in den anderen zu realisieren.

Eine einfache Methode, die den Eindruck des „unruhigen" Geländes bereits deutlich reduziert ist es, bei der Triangulierung zu fordern, daß Höhenwerte, die ein lokales Maximum im Höhendatensatz darstellen, für die Triangulation herangezogen werden müssen. Dies kann durch Vorabmarkierung der lokalen Maxima und ihrer hierarchischen Vorfahren während der Initialisierung sicher gestellt werden. Wenn lokale Maxima bei der Triangulierung berücksichtigt werden, so bewirkt dies, daß Spitzen bei der Triangulierung nicht mehr verschwinden und demnach auch nicht mehr plötzlich auftauchen (d.h. nach oben poppen) können. Vor allem die Geländesilhouette ändert sich durch diese Maßnahme deutlich weniger heftig, was den optischen Eindruck bereits erheblich verbessert. Das Kriterium, das einen Punkt zum lokalen Maximum macht, kann auch schwächer gewählt werden, so daß auch weniger als acht niedrigere Nachbarpunkte zur Einstufung als lokales Maximum führen. Je weniger niedrigere Punkte erforderlich sind, desto besser wird die Geländedarstellung. Allerdings steigt im gleichen Maß auch die Anzahl der benötigten Dreiecke, so daß auf schwacher Graphikhardware Werte unter fünf zu unbefriedigenden Ergebnissen führen.

Ähnlich wie bei der Sichtbarkeitsentscheidung besteht bei der Triangulierung die Möglichkeit, inkrementell zu arbeiten. Bei der Triangulierung kann immer nur ein Level hinzugenommen oder entfernt werden, so daß sich tiefgreifende Änderungen bei der Triangulierung nur von Bild zu Bild fortpflanzen können. Die Änderung von einem Triangulationszustand zu einem anderen wird so auf mehrere Einzelbilder verteilt, was den Übergang in die Länge zieht und weniger abrupt erscheinen läßt.

Bis auf die Markierung der lokalen Maxima wurde keine dieser Lösungsmöglichkeiten für die virtuelle Sicht implementiert, da das Problem nicht als schwerwiegend eingestuft wurde. Diese Einschätzung folgt unmittelbar aus Gesprächen mit Testpiloten die ein Flugzeug mit Hilfe der virtuellen Sicht geflogen und auch gelandet haben (siehe [35] und [37]). Das vorhandene poppen wurde von den Piloten nicht als störend empfunden.

3.4 Ergebnisse

Der erfolgreiche Abschluss von Flugversuchen (siehe [30] und [37]) mit der virtuellen Sicht im Juli 1997 in Freiburg verdeutlicht am eindrucksvollsten, daß die gesetzten Ziele erreicht wurden. Fast alle Entwurfskriterien aus Abschnitt 2.1.2 konnten erfüllt werden. Lediglich bei der geforderten Bildwiederholrate von 30 Hz mussten zu Gunsten einer genaueren Darstellung des Geländes Abstriche gemacht werden. Die Bildfrequenz bleibt jedoch oberhalb von 20 Bildern pro Sekunde, was für den Einsatz im Rahmen der virtuellen Sicht ausreichend ist.

Abbildung. 3.13 zeigt eine Folge von Bildern, wie sie sich ergeben, wenn sich der Beobachter einem Berg nähert. Das erste und das zweite Bild wurden von derselben Position aus erzeugt, wobei das erste Bild mit einer deutlich geringeren Fehlerschranke berechnet wurde, um zu zeigen, wie das Gelände bei exakter Wiedergabe aussehen würde. Der Rest der Sequenz zeigt, daß die Triangulierung umso detaillierter wird, je näher der Betrachter an den Berg kommt. Eine Nachbildung des Geländes mit texturierten Polygonen ist in den Abbildung 3.14 und 3.15

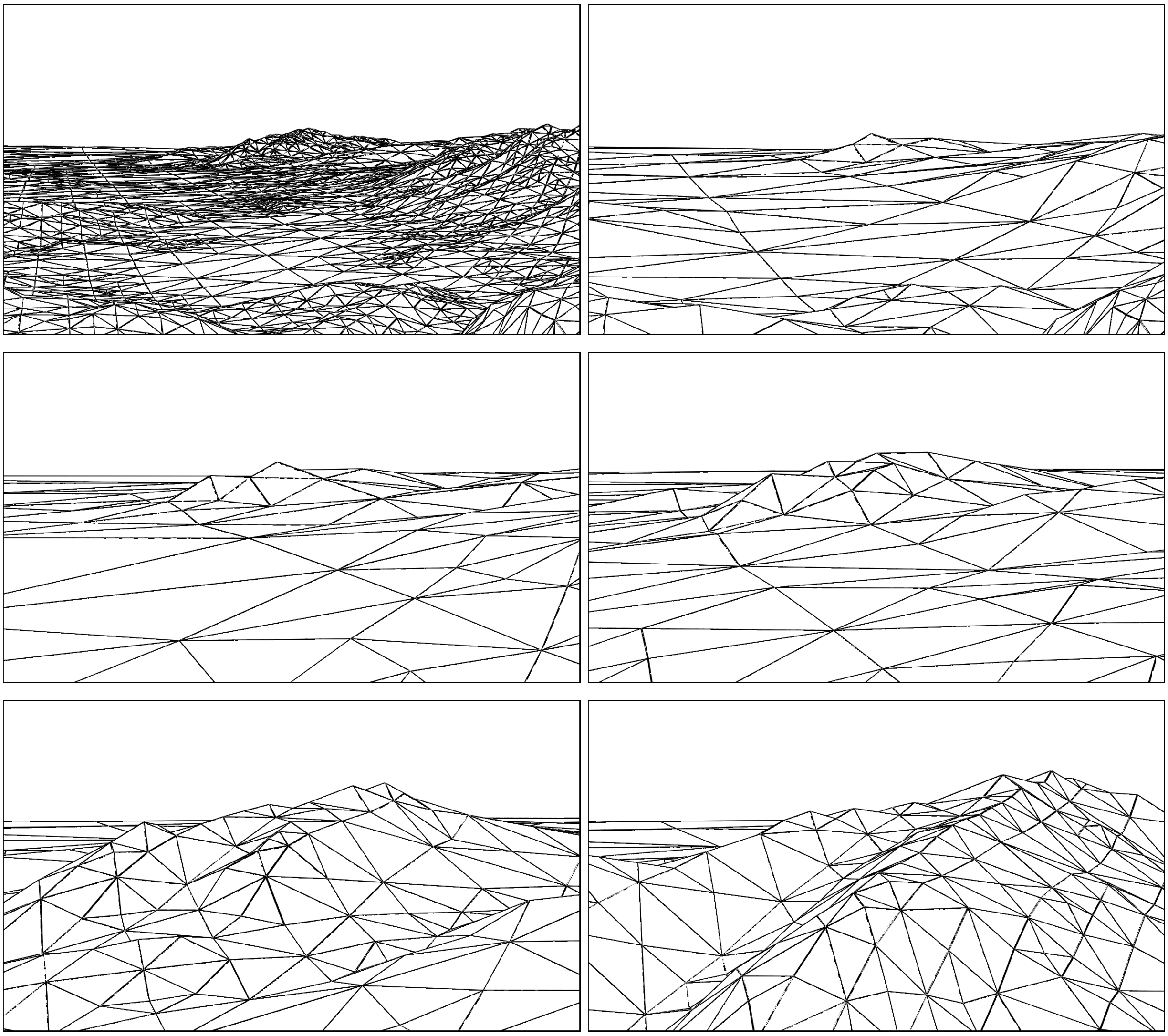


Abbildung 3.13 Annäherung an einen Berg.

gezeigt. Hierbei wurden einmal ca. 6500 Dreiecke verwendet (Abbildung 3.14) und einmal lediglich ca. 600 Dreiecke (Abbildung 3.15).

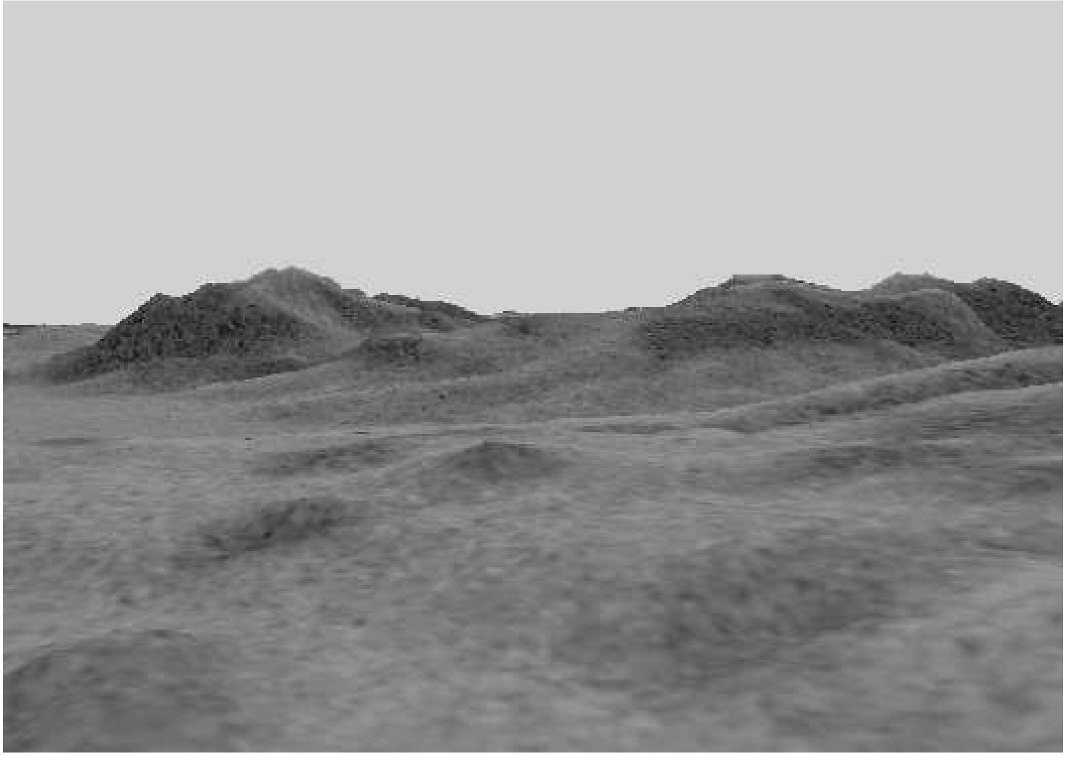


Abbildung 3.14 Geländenachbildung auf der Basis von ca. 6500 Dreiecken.

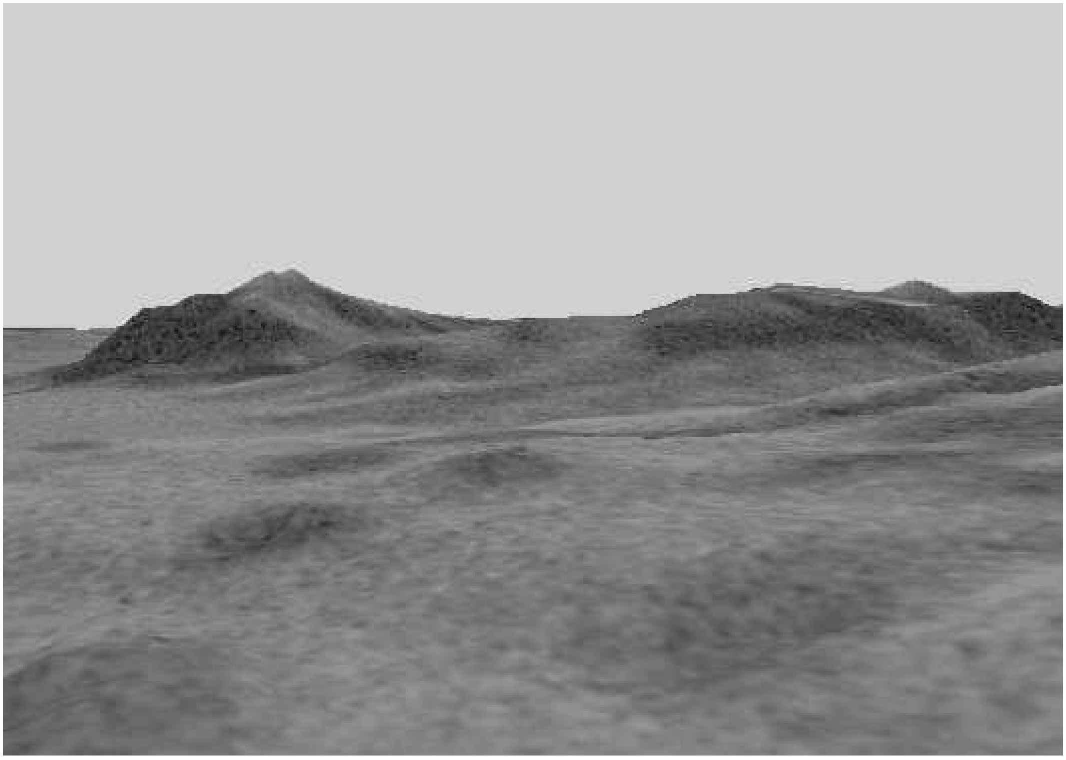


Abbildung 3.15 Geländenachbildung auf der Basis von ca. 600 Dreiecken.

4 Interaktive Visualisierung großer Datenbasen

Eine erhebliche Einschränkung der im Kapitel 3 implementierten virtuellen Sicht ist die Einschränkung auf Gelände von 1025×1025 Höhenwerten bzw. im Bereich von Süddeutschland auf ca. 60km×90km große Geländestücke. Es ist offensichtlich, daß für alles andere als den Forschungseinsatz größere Gelände überflogen werden müssen. Datensätze, die Höhendaten von ganz Deutschland oder auch von ganz Europa enthalten, stellen für den praktischen Einsatz interessante Größenordnungen dar. Angesichts der in Abschnitt 2.1.3 dargelegten Datenmengen ist der Zeitpunkt, ab dem die Höhendaten von ganz Deutschland oder gar ganz Europas in den Hauptspeicher eines kostengünstigen Computers passen, noch nicht absehbar. Dabei ist zu berücksichtigen, daß zu jedem Höhenwert zusätzliche Verwaltungsdaten erforderlich sind, deren Datenvolumen in der Regel ein Vielfaches des Volumens der eigentlichen Höhenwerte ausmacht. Sobald die Daten nicht mehr in den Hauptspeicher des Rechners, der zur Visualisierung eingesetzt wird, passen und über den Paging–Mechanismus des Betriebssystems auf den Hintergrundspeicher wie z.B. Festplatten ausgelagert werden, sind Bildwiederholraten, die zur interaktiven Visualisierung ausreichen undenkbar. Von interaktiver Visualisierung spricht man, wenn die Daten schnell genug auf den Bildschirm gezeichnet werden können, so daß die interaktive Manipulation der Daten bzw. ihrer Darstellung möglich wird. Im einfachsten Fall ist dies z.B. die Rotation, Translation oder Skalierung der Daten, die vom Benutzer mit der Maus durchgeführt werden kann. Damit dies als möglich erachtet werden kann, ohne daß dem Benutzer unzumutbare Verzögerungen zwischen seiner Eingabe und deren Auswirkung auf die Bildschirmdarstellung zugemutet werden, muß die Darstellung der Daten mindestens einmal pro Sekunde erneuert werden können.

Auch wenn die jetzt verwendeten Höhendaten in den Hauptspeicher passen würden, ist es leicht vorstellbar, daß z.B. durch eine höhere Dichte der Messpunkte, wie sie bei den DTED Level Zwei Daten bereits jetzt der Fall ist, jede beliebige Hauptspeichergröße gesprengt werden kann. Der Punkt, an dem eine Erhöhung der Messpunktdichte bei Höhendaten keine wesentliche Verbesserung der damit möglichen Geländeapproximation bringt ist noch lange nicht erreicht.

Die Visualisierung von Datenvolumen, die die Grenzen jedes verfügbare Hauptspeichers überschreiten ist neben Höhendaten auch für andere Datentypen interessant. Fernerkundungssatelliten liefern z.B. jeden Tag weit mehr Daten als ausgewertet werden könnten. Mit steigender Zahl von Satelliten und ständiger Verbesserung und Erhöhung der Auflösung der Messtechnik wird sich das täglich erfasste Datenvolumen in Zukunft noch vervielfachen. So hat auch z.B. die NASA bisher nur einen Bruchteil der auf ihren Missionen gesammelten Daten auswerten können. Die interaktive Visualisierung stellt bereits heute die einzig sinnvolle Methode zur Analyse solcher Daten dar. Ein weiteres Gebiet, in dem erfahrungsgemäß sehr große Datenmengen anfallen, deren Visualisierung in der Regel einen signifikanten Erkenntnisgewinn bedeutet, ist die numerische Strömungssimulation. Eine Verfeinerung der Gitter auf denen die Strömung simuliert wird oder die Verkürzung der Zeitschritte bei der Simulation des zeitlichen Verlaufs einer Strömung führt leicht zu einem Platzbedarf für die Ergebnisse von etlichen Gigabyte.

Ähnlich wie bei der virtuellen Sicht lassen sich zwar immer Teile der Daten interaktiv visualisieren, ein Überblick über den gesamten Datensatz erfordert in der Regel jedoch, daß der Datensatz vollständig in den Speicher passt. Um größere Datensätze zu visualisieren, wird entweder auf nicht interaktive Methoden zurückgegriffen oder es werden mit Hilfe von Subsampling eine oder mehrere Versionen des Datensatzes mit geringerem Level–of–Detail erzeugt. Somit sind makroskopische Zusammenhänge nur unter großem Aufwand zu verdeutlichen bzw. überhaupt zu erkennen. Der hierzu erforderliche Überblick über den ganzen Datensatz würde im Rahmen der virtuellen Sicht z.B. den Blick von Hamburg bis in die Alpen ermöglichen. Sichtweiten dieser Größenordnung sind zwar für die Funktion der virtuellen Sicht als Navigationshilfe unnötig, es existieren jedoch andere Anwendungen, wie z.B. der aus dem Wetterbericht bekannte sogenannte „Wetterflug" oder Geoinformationssysteme (GIS), die davon profitieren würden. Dies gilt besonders dann, wenn der Benutzer die Möglichkeit hat, sich beliebig im Datensatz zu bewegen und Teile, die von Interesse erscheinen, genauer, bis hin zur vollen Auflösung der Daten, zu untersuchen, ohne daß zu einem Datensatz mit feinerer Auflösung gewechselt werden muß.

4.1 Implementierung

Im Folgenden soll das in Kapitel 3 implementierte Verfahren zur Geländetriangulierung so erweitert werden, daß Unterbäume des Dreiecks–Binärbaums dynamisch nachgeladen bzw. der von diesen belegte Speicher wieder freigegeben werden kann. Dazu soll der kontinuierliche Level–of–Detail Übergang ausgenutzt werden, indem weit entfernte Details, deren Darstellung auf dem Bildschirm keinen Beitrag zur Verbesserung der Bildqualität leisten würden, gar nicht erst im Hauptspeicher gehalten werden. Der frei festlegbare Level–of–Detail ermöglicht es die Darstellungsqualität an die verfügbaren Ressourcen und die Ansprüche des Benutzers an die Antwortzeiten des Systems anzupassen, so daß ein einheitliches Verfahren auf Graphikhardware von unterschiedlichen Leistungsklassen zum Einsatz kommen kann.

4.1.1 Datenstruktur

Damit das in Kapitel 3 implementierte Verfahren das dynamische Nachladen von Geländeteilen und das Freigeben von nicht benötigten Teilen zuläßt, muß der Algorithmus auf der Basis einer anderen Datenstruktur neu implementiert werden. Bisher wurde die Baumstruktur und Hierarchie durch den festcodierten Zugriff auf die Daten im Triangulationsalgorithmus implizit genutzt. Die zugrundeliegende Datenstruktur, ein zweidimensionales Array, das linearisiert im Speicher liegt, wurde als ein Speichersegment alloziert und kann auch nur als ein Segment freigegeben werden. Eine Freigabe von Teilen des Arrays ist nicht möglich.

Baumstruktur im Speicher

Um die Freigabe von Teilbäumen zu ermöglichen, wird der Dreiecks–Binärbaum nun auch als Datenstruktur im Speicher aufgebaut, wobei jeder Knoten separat alloziert und über Zeiger mit seinen Vor- und Nachfahren verknüpft wird. So kann jeder Knoten individuell wieder freigegeben werden, wenn er nicht mehr benötigt wird. Diesen Vorteil erkauft man sich über den zusätzlichen Speicherplatzbedarf, der für die Zeiger anfällt. Deren Information steckte beim Einsatz von Arrays in Form von Indexrechnung im Algorithmus und musste nicht explizit im Speicher gehalten werden.

Die Erzeugung der Baumdatenstruktur erfolgt mit einer einfachen rekursiven Funktion die analog dem Durchlauf durch die Hierarchie bei der Triangulierung im letzten Kapitel konstruiert ist.

```
knoten erzeuge_baum( a, b, c, v )
{
  if( tiefste Stufe erreicht ) return( leeren Baum )
  else {
    k = erzeuge neuen Knoten
    k→vorgänger = v
    k→links = erzeuge_baum( c, a, (a+b)/2, k )
    k→rechts= erzeuge_baum( b, c, (a+b)/2, k )
    return( k )
    }
}
```

Bei der mit dieser Funktion aufgebauten Datenstruktur liegt jedes Dreieck des Dreiecks–Binärbaums als separat allozierter Knoten im Speicher vor. Neben den Koordinaten der Ecken wird auch für jeden Knoten der hierarchische Überschuss an dem Höhenwert, der die Hypotenuse des Dreiecks halbiert, ermittelt und gespeichert. Ebenso werden die Zeiger auf die unmittelbaren Vorgänger und Nachfolger gesetzt. Beim rekursiven Aufruf der Funktion `erzeuge_baum()` wird bei der Übergabe des Wertes `(a+b)/2` für die Ecke `c` der neu zu erzeugenden Dreiecke bzw. Knoten anstelle der errechneten interpolierten Höhe der tatsächliche Höhenwert an dieser Stelle übergeben, so daß er in `c.höhe` der nachfolgenden Knoten gespeichert wird[1]. Die Differenz zwischen tatsächlichem und interpoliertem Höhenwert liefert den hierarchischen Überschuss, der in dem Knoten gespeichert wird. Dieser entspricht dem Höhenfehler, der gemacht wird, wenn das Dreieck des Knotens, in dem der hierarchische Überschuss gespeichert ist, gezeichnet wird und nicht die beiden Nachfolger. Demzufolge sind die Überschüsse des untersten Baumlevels, deren Positionen nicht mehr auf einen realen noch nicht im Baum erfassten Höhenwert abgebildet werden können, Null bzw. müssen gar nicht erst gespeichert werden. Das Speichern der Ecken der Dreiecke in jedem Knoten ist extrem redundant, da jeder Eckpunkt mehrfach im Baum an verschiedenen Stellen gespeichert wird. Koordinaten, die am Rand des Datensatzes liegen, werden pro Level bis zu viermal, Koordinaten im Inneren bis zu achtmal über verschiedene Knoten verteilt gespeichert. Abbildung 4.1 verdeutlicht die Situation durch eine Detailansicht, bei der die Dreiecke eines Levels etwas auseinander gezogen wurden. Links

[1] bei der tatsächlichen Implementierung entspricht `c.höhe` der Y–Koordinate von `c`

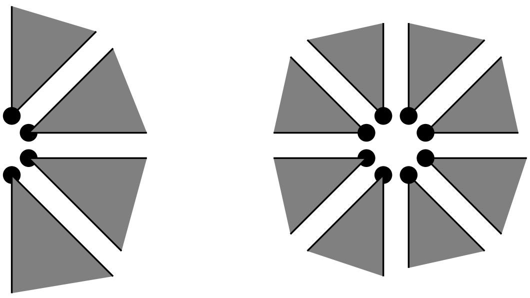


Abbildung 4.1 Redundant gespeicherte Dreiecksecken.

ist die Situation für einen Randpunkt dargestellt, rechts die eines Punktes im Inneren des Gebiets. Die schwarzen Punkte repräsentieren in beiden Fällen jeweils einen Höhenwert. Für die Entwicklung des Verfahrens in diesem Kapitel war es von Vorteil, in jedem Knoten die Koordinaten sofort zur Verfügung zu haben. In Abschnitt 4.2.2 wird eine einfache Methode aufgezeigt, die das Mehrfachspeichern von Koordinaten vermeidet.

Zwillingsdreiecke

Eine Information, die durch das separierte Anlegen der Dreiecksinformationen im Speicher gegenüber der Implementierung auf Arrays verloren geht, ist die Zuordnung zwischen einem Höhenwert und seinen hierarchischen Vorgängern. Die Zuordnung zwischen einem Dreieck und seinen Vorgängern ist eindeutig und vollständig durch die Baumstruktur erfasst, die Beziehung zwischen einem Wert und seinen hierarchischen Vorgängern sind die in Abbildung 3.2 eingezeichneten Verbindungen. Bei den Vorgängern eines Höhenwertes handelt es sich um die Werte, die beim hierarchischen Durchlauf durch den Datensatz auf dem Weg zu allen Dreiecken, deren Ecke `c` auf dem jeweiligen Höhenwert liegt, berührt werden. Die durch die Baumstruktur verlorene Information wird durch die im Folgenden eingeführte *Zwillingsrelation* repräsentiert.

Abgesehen von Höhenwerten am Rand liefert jeder Wert den hierarchischen Überschuss für genau zwei Dreiecke. Die beiden Dreiecke, denen der gleiche hierarchische Überschuss zugeordnet ist, werden als Zwillinge bezeichnet. Abbildung 4.2 zeigt ein Beispiel für zwei Dreiecke, die in Zwillingsrelation zueinander stehen. Die durch fette Umrandung gekennzeichneten Zwillinge

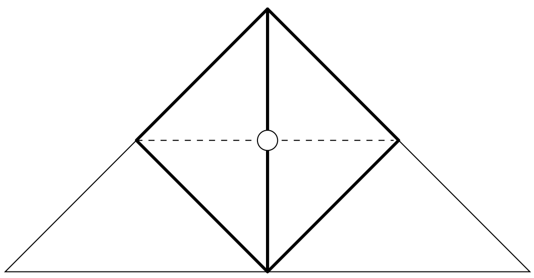


Abbildung 4.2 Beispiel für Zwillingsrelation.

im Inneren des größeren Dreiecks teilen sich den durch einen weißen Kreis markierten Mittelpunkt ihrer Hypotenuse. Die Zwillingsrelation spielt im Weiteren eine wichtige Rolle, insbesondere bei der Markierung hierarchischer Vorgänger, da für eine gültige, d.h. T–Kreuzzugs-, also lochfreie Triangulierung bei der Teilung bzw. Verfeinerung eines Dreiecks auch der Zwilling dieses Dreiecks geteilt werden muß. Würde dies versäumt werden, so würde die Triangulierung vertikale Löcher enthalten, wie man in Abbildung 4.3 leicht sieht. Hier wurde der linke der

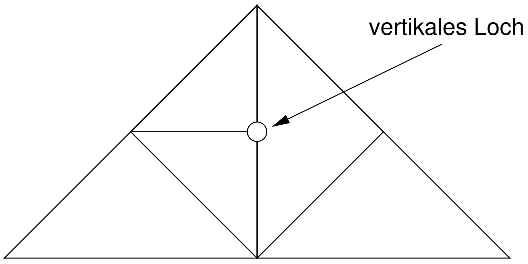


Abbildung 4.3 Beispiel für ein durch versäumte Zwillingsteilung entstandenes vertikales Loch.

beiden Zwillinge aus Abbildung 4.2 geteilt, während dies beim rechten unterlassen wurde. Am gemeinsamen Mittelpunkte der Hypotenuse entsteht so ein vertikales Loch.

Abbildung 4.4 zeigt den Dreiecks–Binärbaum mit als Querverbindungen innerhalb eines Levels erkennbaren Zwillingsrelationen. Hierzu ist zu bemerken, daß die Zwillingsrelationen, die im

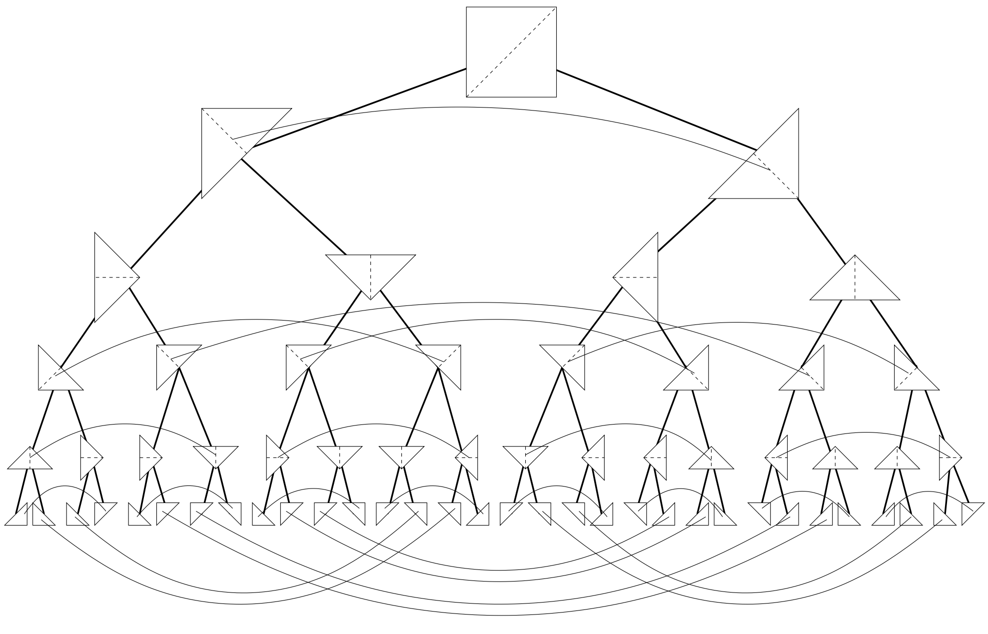


Abbildung 4.4 Dreiecks–Binärbaum mit Zwillingsrelationen.

untersten Level eingezeichnet sind, Dreiecke verknüpfen auf deren gemeinsamer Hypotenuse kein realer Höhenwert mehr liegt. Für die Markierung von Vorgängern spielen diese Verbindungen deshalb keine Rolle und könnten auch weggelassen werden.

Die Nachbarschaftsbeziehung, die die Zwillingsrelation repräsentiert, ist lediglich implizit durch die Baumstruktur festgelegt. Um die Zuordnung zwischen einem Dreieck und seinem Zwilling zu vereinfachen, wird jeder Knoten im Baum um einen Zeiger auf seinen Zwilling ergänzt. Diese recht kompliziert erscheinenden Querverbindungen aus Abbildung 4.4 lassen sich mit einer einfachen rekursiven Prozedur erzeugen, sofern die Orientierung der Dreiecke beim Baumaufbau konsequent erhalten wurde. Anders ausgedrückt, jedes Dreieck im Baum ist konform mit der in Abbildung 3.7 auf Seite 29 eingeführten Zuordnung zwischen Bezeichnern und Geometrie und es besteht eine Entsprechung von linkem bzw. rechtem Teilbaum mit dem linken bzw. rechten Teildreieck. Der von `erzeuge_baum()` generierte Baum erfüllt diese Bedingung.

```
verknüpfe_zwillinge( l, r )
{
  if( l und r sind gültige Knoten ) {
    verknüpfe( l→links, r→rechts )
    verknüpfe_zwillinge(l→links→rechts,
                        r→rechts→links)
    verknüpfe_zwillinge(r→rechts→rechts,
                        l→links→links)
    }
}
```

Die Prozedur `verknüpfe()` tut dabei nichts anderes, als zwischen den beiden übergebenen Knoten eine Verbindung mit Hilfe zweier Zeiger die auf den jeweils anderen Knoten zeigen herzustellen.

```
verknüpfe( l, r )
{
  l→zwilling = r
  r→zwilling = l
}
```

Werden die Knoten, die die linke bzw. die rechte Hälfte des Dreiecks aus Abbildung 4.5 repräsentieren, als Parameter an `verknüpfe_zwilling()` übergeben, so werden zunächst die zum grauen Knoten gehörenden Zwillingsdreiecke verknüpft, in der nächsten Stufe der Rekursion dann die zu den weißen Knoten gehörenden Dreiecke. Die Zwillingsbeziehung läßt sich

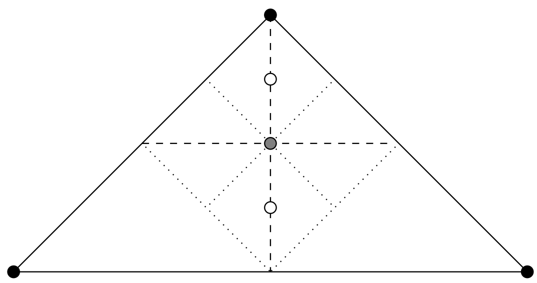


Abbildung 4.5 Strategie zur Verknüpfung von Zwillingen.

also entlang der Naht, an der die Dreieck bei der Triangulierung geteilt werden, herstellen. Soll dies gleich beim Aufbau der Baumstruktur erfolgen, so läßt sich `erzeuge_baum()` leicht entsprechend erweitern.

Der von `verknüpfe_zwillinge()` zusammen mit `erzeuge_baum()` realisierte Baumdurchlauf entspricht keiner der bekannten Standardstrategien zum Durchlaufen von Bäumen, sondern ergibt sich aus der in der Baumstruktur implizit enthaltenen geometrischen Information.

Fehlerpropagation

Die Informationen, die bei der virtuellen Sicht im Fehler–Array gesammelt und gespeichert wurden, müssen nun ebenfalls in den Knoten gespeichert werden. Das Hochpropagieren der Fehler kann analog zur Zwillingsverknüpfung bereits bei der Erzeugung der Baumstruktur erfolgen. Die erweiterte Fassung von `erzeuge_baum()` sieht dann wie folgt aus:

```
knoten erzeuge_baum( a, b, c, v )
{
  if( tiefste Stufe erreicht ) return( leeren Baum )
  else {
    k = erzeuge neuen Knoten
    k→vorgänger = v
    k→links  = erzeuge_baum( c, a, (a+b)/2, k )
    k→rechts = erzeuge_baum( b, c, (a+b)/2, k )
    k→überschuss = max(|Höhe an (a+b)/2 - Höhe von (a+b)/2|,
                       k→links→überschuss,
                       k→rechts→überschuss)
    verknüpfe_zwilling( k→links, k→rechts )
    return( k )
    }
}
```

Abbildung 4.6 zeigt einen Baum der Tiefe sechs. Bei der dreidimensionalen, perspektivischen Darstellung wurde die X– und Y–Koordinate des Schwerpunktes der Dreiecke übernommen, die Höhe der einzelnen Knoten, die jeweils am Anfang und Ende jeder Linie liegen, entspricht dem Level, auf dem sie sich in der Hierarchie befinden. Die Verbindung zum linken Nachfolger ist orange, die zum rechten Nachfolger grün gefärbt. Horizontale graue Linien zeigen Zwillingsbeziehungen zwischen Knoten. Um den perspektivischen Eindruck zu verstärken, wurde auf der X–Y–Ebene ein Gitter mit Linienabstand eins gezeichnet. Abbildung 4.7 zeigt denselben Baum senkrecht von oben gesehen, wobei das Gitter auf der X–Y–Ebene weggelassen wurde.

Triangulierung

Die Triangulierung erfolgt nach dem gleichen Schema wie bei der Implementierung auf Arrays für die virtuelle Sicht.

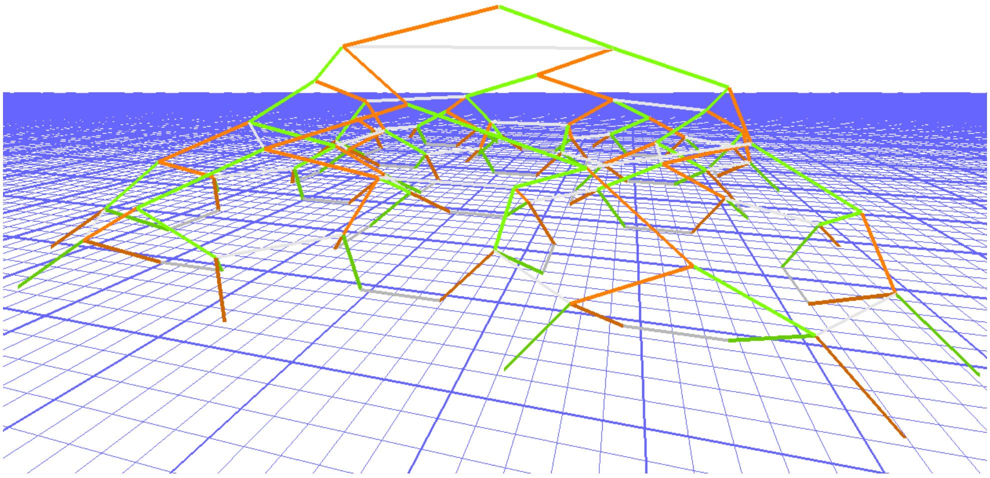


Abbildung 4.6 Dreidimensionale Baumstruktur.

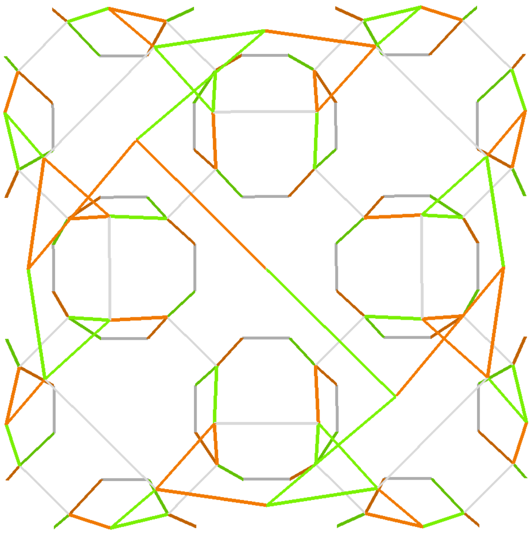


Abbildung 4.7 Dreidimensionale Baumstruktur (von oben).

```
markiere_baum( k )
{
  if( knoten_nicht_sichtbar(k) ) return
  if( k→überschuss < eps(d(k)) )
    markiere_vorgänger( k→vorgänger )
  else {
    markiere_baum( k→ links )
    markiere_baum( k→rechts )
    }
}
zeichne_markierten_baum( k )
{
  if( knoten_nicht_sichtbar(k) ) return
```

```
  if( k nicht markiert )
    zeichne_knoten( k )
  else {
    zeichne_markierten_baum( k→ links )
    zeichne_markierten_baum( k→rechts )
    }
}
```

Die Prozedur `markiere_vorgänger()` markiert den übergebenen Knoten, dessen Vorfahren sowie alle Zwillinge und deren Vorfahren, die auf dem Weg zur Wurzel durchlaufen werden.

```
markiere_vorgänger( k )
{
  if( k ist markiert ) return
  markiere k
  if( k→zwilling existiert )
    markiere_vorgänger( k→zwilling )
  markiere_vorgänger( k→vorgänger )
}
```

Die Sonderbehandlung von Zeigern, die ins Leere zeigen wurde hier der Übersichtlichkeit halber bei allen Programmstücken weggelassen. An diesem Punkt ist die Triangulierung der virtuellen Sicht reimplementiert. Im Folgenden wird eine Strategie zum dynamischen Nachladen und Freigeben von Teilbäumen entwickelt und umgesetzt.

4.1.2 Nachladen

Der in Bezug auf Speicherplatzersparnis optimale Ansatz bei der Triangulierung ist sicher der, bei dem nur genau die Daten im Speicher gehalten werden, die auch wirklich für die Darstellung benötigt werden. Abbildung 4.8 zeigt die bei einer Triangulierung durchlaufenen Knoten als dreidimensionalen Baum. Der Betrachter befindet sich dabei auf der rechten Seite und blickt

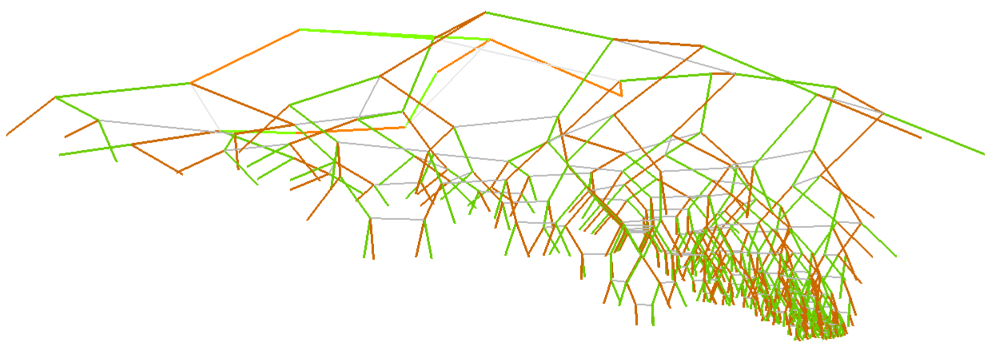


Abbildung 4.8 Bei einer Triangulierung durchlaufene Knoten als dreidimensionaler Baum.

nach links. Gemäß der Level–of–Detail Strategie wird der Baum in der Nähe des Betrachters bis zum tiefsten Level durchlaufen, während auf der linken, weit entfernten Seite nur sehr wenige Level durchlaufen werden. Um das Gelände zu zeichnen, sind nur die im Bild dargestellten Knoten erforderlich. Es genügt also, wenn nur diese Knoten im Speicher gehalten werden.

Da eine Retriangulierung des Geländes, die mehrmals pro Sekunde erfolgen soll, bei sich bewegendem Betrachter jedesmal neue Knoten durchläuft und alte, jetzt unbenutzte Knoten nicht mehr erreicht, würde dies zu einem ständiges Nachladen und Freigeben von einzelnen Knoten führen. Bewegt sich der Betrachter schnell bzw. ändert er die Blickrichtung, so wechselt der Status mehrerer hundert Knoten von „unbenutzt" auf „benutzt", die demnach einzeln nachgeladen werden müssten. Das Nachladen von Daten ist in der Regel mit IO–Operation auf einen Hintergrundspeicher (z.B. Festplatte) verbunden und diese sind typischerweise um Größenordnungen langsamer als der Zugriff auf den Hauptspeicher. Bei Verfolgung der optimalen Strategie würden die bei jedem Triangulierungsdurchlauf erforderlichen IO–Operation so viel Zeit kosten, daß man nicht mehr von interaktiver Visualisierung sprechen könnte.

Kacheln

Da feingranulares Vorgehen zuviel Zeit kostet, ist es erforderlich, IO–Operationen zusammenzufassen. Dies erreicht man am einfachsten, indem nicht einzelne Knoten, sondern immer ganze Teilbäume auf einmal geladen bzw. freigegeben werden. Es liegt also nahe, die Höhendaten in Kacheln aufzuteilen und nur die Kacheln im Speicher zu halten, die Knoten enthalten, die bei der Triangulierung benötigt werden. Abbildung 4.9 zeigt einen Baum, bei dem die Teilbäume der untersten drei Level zu Kacheln zusammengefasst wurden. Die Größe der Kacheln ent-

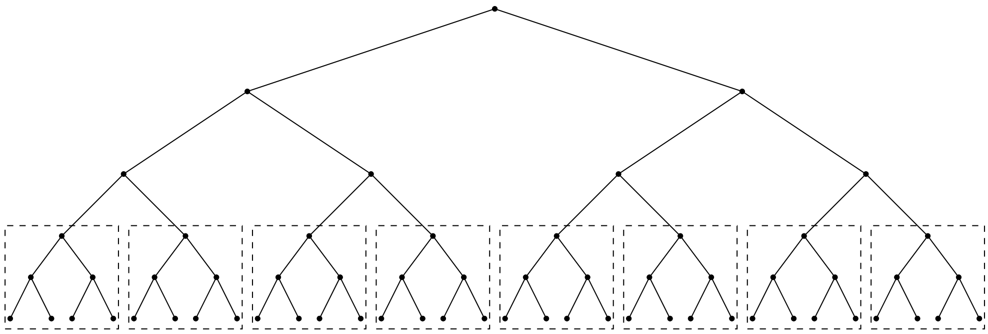


Abbildung 4.9 Teilbäume werden zu Kacheln zusammengefasst.

scheidet über die Häufigkeit der Nachladeoperationen und über deren Dauer. In die Wahl der am besten geeigneten Kachelgröße fließen Faktoren wie IO–Leistung, Blockgröße des Systems und das charakteristische Bewegungsverhalten des Betrachters ein. Aus den Erfahrungen auf der Entwicklungsplattform[2] hat sich das Zusammenfassen von sechs bis acht Level tiefen Teilbäumen als guter Kompromiss erwiesen. Kleinere Kacheln erfordern ein zu häufiges Nachladen, bei größeren Kacheln dauert der einzelne Nachladevorgang wieder so lange, daß es zu merklichen Verzögerungen bei der Aktualisierung der Bildschirmdarstellung kommt.

[2] Silicon Graphics O2, 384MB, 180MHz R10000

Kachelhierarchien

Die Aufteilung in Kacheln muß nicht auf die untersten Level des Baumes beschränkt bleiben. Der Teil des Baumes aus Abbildung 4.9 oberhalb der Kacheln, die oberen drei Level, können ebenfalls als Kachel angesehen werden. Abbildung 4.10 zeigt eine Fortsetzung des Prinzips der Unterteilung in Kacheln auf mehreren Ebenen. Verkleinert man die Kacheln soweit, daß jede

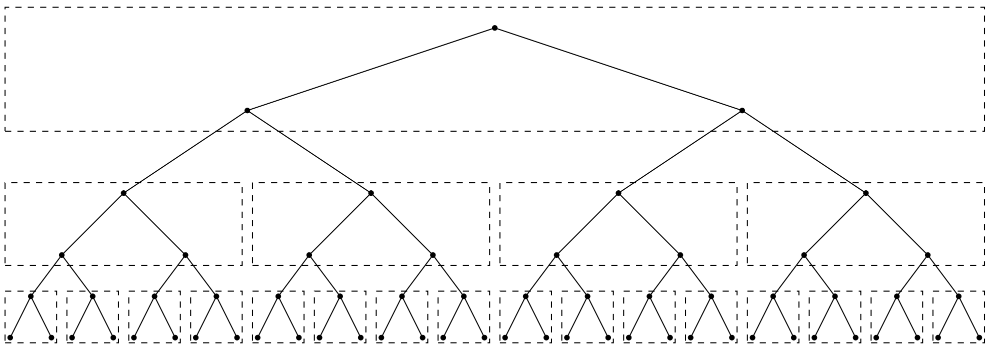


Abbildung 4.10 Mehrere Level von Kacheln.

Kachel nur noch einen Knoten enthält so würde dies der in Bezug auf Speicherplatzersparnis optimalen Strategie entsprechen. In Abbildung 4.11 sind für jeden der drei Kachellevel aus Abbildung 4.10 die Bereiche der Höhendaten zu sehen, die den einzelnen Kacheln in den drei Leveln zugeordnet sind. Die Darstellung ganz links entspricht dem obersten Kachellevel. Dabei

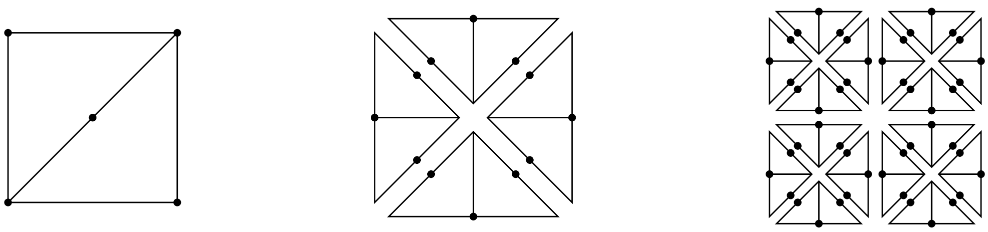


Abbildung 4.11 Zu den Kacheln der einzelnen Level gehörende Gebiete.

wird auch deutlich, daß der Wurzelknoten immer eine Ausnahme darstellt, da er das gesamte, in der Regel rechteckige Gebiet repräsentiert. Alle anderen Kacheln sind dreieckig.

Anders als in Abbildung 4.10 können die Kacheln verschiedener Level auch unterschiedlich groß bzw. tief sein. Damit läßt sich eine auf den Speicherplatzbedarf der Daten, die angestrebte Zielplattform, deren IO– und Graphik–Leistung und die typischen Anforderungen der vorgesehenen Visualisierungsart maßgeschneiderte Hierarchie von Kacheln entwerfen. Ziel der in diesem Kapitel entworfenen Algorithmen soll es sein, die Einschränkung der virtuellen Sicht auf die Darstellung von Geländen mit 1025×1025 Höhenwerten aufzuheben. Angestrebt wird, ein Gelände von der Größenordnung ganz Deutschlands zu überfliegen, wobei die Sichtweite im Optimalfall beliebig sein soll. Der zu diesem Zweck erzeugte Test–Datensatz enthält 16814×15544 Höhenwerte. Da die DTED–Daten ab dem 50. Breitengrad nur noch die halbe horizontale Auflösung haben, wurde jede Spalte dieser Datensätze verdoppelt, damit sie sich nahtlos an die Daten südlich des 50. Breitengrades anfügen lassen. Dank der Baumstruktur wäre es leicht gewesen, anstatt die Daten künstlich zu verbreitern, diese lediglich durch Teilbäume entsprechend geringerer Tiefe darzustellen. Statt dessen wurde der Weg, die Menge der zu verwaltenden Daten zu vergrößern, gewählt, um einen aussagekräftigeren Testfall zu erhalten. Für die hier beschriebene Implementierung wurde ein Level von Kacheln analog zu Abbildung 4.9 als ausreichend erachtet. Die den Kacheln entsprechenden Teilbäume werden auch als Kachelbäume, der Teil des Baumes oberhalb der Kacheln wird als Überbaum bezeichnet.

Kacheltafel

Zur Verwaltung des Zustandes der einzelnen Kachelbäume wird eine Kacheltafel eingeführt, die für jede Kachel Informationen darüber enthält, ob sie geladen bzw. bei der Triangulierung benötigt wurde sowie wo auf dem Hintergrundspeicher die zu der Kachel gehörigen Daten zu finden sind und ein Verweis darauf, wo im Überbaum die Kachel einzufügen ist. Ebenfalls enthalten sind die Koordinaten der Ecken der Kachel. Sollten mehrere Kachellevel existieren, so muß für jeden dieser Level eine separate Kacheltafel verwaltet werden. Damit während der Triangulierung beim Fehlen eines Unterbaums die entsprechende Kachel nachgeladen werden kann, müssen von dem Knoten oberhalb des Kachelbaums Zeiger auf die Einträge der nachfolgenden Kacheln in der Kacheltafel weisen. Falls der Zwilling eines Knotens, der markiert werden soll, in einer nicht geladenen Nachbarkachel liegt, so muß auch diese nachgeladen werden. Um dies zu ermöglichen, sind Verweise von den Knoten der Kachelbäume, deren Zwilling in einer benachbarten Kachel liegt, in die Kacheltafel erforderlich. Die Prozeduren zum Nachladen und Freigeben einzelner Kacheln müssen die Zeiger zwischen Kacheltafel, Überbaum und Kachelbäumen konsistent halten. Abbildung 4.12 zeigt die benötigten Zeiger zwischen Kacheltafel, dem ersten Kachelbaum sowie dem Überbaum–Knoten, zu dem der erste Kachelbaum gehört. Der Baum und insbesondere auch die Zwillingsbeziehungen sind die gleichen wie in Abbildung 4.4.

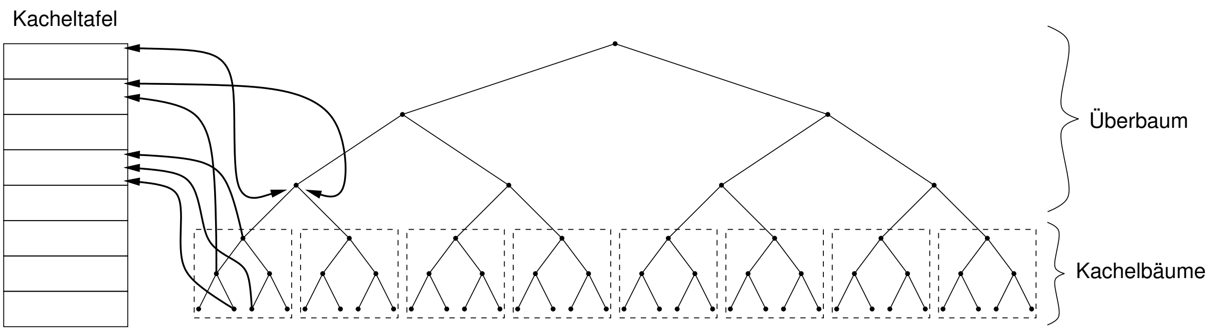


Abbildung 4.12 Zeiger zwischen Baum und Kacheltafel.

Triangulierung und Nachladen

Nach der Implementierung der Prozeduren zum Nachladen und Freigeben von Kacheln kann die Kacheltafel wie folgt in die bestehenden Triangulierungsprozeduren integriert werden.

```
markiere_baum( k )
{
  if( knoten_nicht_sichtbar(k) ) return
  if( k→überschuss < eps(d(k)) )
    markiere_vorgänger( k→vorgänger )
  else {
    if( k liegt auf dem letzten Überbaum Level )
      markiere_kacheln( k )
    markiere_baum( k→ links )
    markiere_baum( k→rechts )
    }
}
```

Die Prozedur `markiere_kachel()` markiert zum einen die Kachel als bei der Triangulierung benutzt, zum anderen muß sie sicher stellen, daß die erforderlichen Daten geladen werden und ggf. die Zeiger in dem Überbaumknoten auf die neugeladenen Kachelbäume gesetzt werden. Um diese Aufgabe erfüllen zu können, muß sie entscheiden können, welche der beiden nachfolgenden Kacheln geladen werden muß und bei welcher evtl. darauf verzichtet werden kann[3]. Dies wird durch die zusätzlich in der Kacheltafel gespeicherten `überschuss`–Werte der Wurzelknoten der jeweiligen Kachel ermöglicht.

Die einzige Prozedur, die noch erweitert werden muß, ist `markiere_vorgänger()`.

```
markiere_vorgänger( k )
{
  if( k ist markiert ) return
  markiere k
  if( k→nachbar_kachel existiert )
    markiere_kachel( k→nachbar_kachel )
  if( k→zwilling existiert )
    markiere_vorgänger( k→zwilling )
  markiere_vorgänger( k→vorgänger )
}
```

Die Prozedur `markiere_kachel()` markiert und lädt ggf. die übergebene Kachel. Bei diesem Vorgang muß auch der Zeiger `k→zwilling` gesetzt werden, damit der anschließende Aufruf `markiere_vorgänger(k→zwilling)` nicht ins Leere läuft.

Das Gegenstück zu `markiere_kachel()`, das den Speicher, den unbenutzte Kachelbäume belegen, freigibt, besteht in einer Prozedur, die nach der Triangulierung die Kacheltafel durchläuft und alle Einträge, die zwar als geladen aber nicht als benutzt markiert sind, freigibt.

[3]Hinweis: Die Sonderbehandlung von leeren Zeigern, die an `markiere_baum()` übergeben werden, wurde der Übersichtlichkeit halber weggelassen.

Speichern der Kacheldaten auf dem Hintergrundspeicher

Nachdem bei der Triangulierung jetzt Kacheln nachgeladen und anschließend falls möglich wieder freigegeben werden können, fehlen noch die Prozeduren und Funktionen, um die Daten auf den Hintergrundspeicher, der in den meisten Fälle eine Festplatte mit ausreichender Kapazität sein dürfte, zu schreiben und von dort wieder zu laden. Da der Baum ursprünglich aus einem Array von Höhenwerten erzeugt wurde, genügt es, auch genau diese Daten abzuspeichern. Alle anderen Werte wie z.B. der `überschuss` lassen sich aus den Höhenwerten errechnen. Die Prozedur `schreibe_baum()` muß den Baum so durchlaufen und dabei die Höhenwerte auf den Hintergrundspeicher schreiben, daß die Funktion `lade_baum()` die Daten in der Reihenfolge vorfindet und laden kann, in der sie beim Baumaufbau benötigt werden. Das Laden und damit automatisch auch das Schreiben der Daten hängt also eng mit dem Baumaufbau zusammen.

Wie in Abschnitt 4.1.1 dargelegt, ist jeder Höhenwert mehrfach im Baum repräsentiert. Da die Daten nur einmal auf den Hintergrundspeicher geschrieben werden sollen, scheidet der Ansatz den Baum in der gleichen Weise wie z.B. beim Zeichnen zu durchlaufen und dabei einfach den Höhenwert der Ecke `c` jedes Knotens zu speichern, aus. Ein Baumdurchlauf bei dem sicher gestellt werden kann, daß jeder Höhenwert nur einmal geschrieben wird, ist der von `verknüpfe_zwillinge()` implementierte Durchlauf. Die Prozedur `schreibe_baum` sieht dann etwa wie folgt aus.

```
schreibe_baum( k )
{
  if( k ist der leere Baum ) return
  schreibe_baum( k→links )
  schreibe_baum( k→rechts )
  schreibe_zwillinge( k→links, k→rechts )
}
```

Die Prozedur `schreibe_zwillinge()` ist analog zu `verknüpfe_zwillinge()` konstruiert, nur daß statt `verknüpfe()` eine Prozedur `schreibe()` aufgerufen wird, die den übergebenen Wert auf den Hintergrundspeicher schreibt.

```
schreibe_zwillinge( l, r )
{
  if( l und r sind gültige Knoten ) {
    schreibe( l→links→links→c.höhe )
    schreibe_zwillinge(l→links→rechts,
                       r→rechts→links)
    schreibe_zwillinge(r→rechts→rechts,
                       l→links→links)
  }
}
```

Statt `l→links→links→c.höhe` könnte auch `r→rechts→rechts→c.höhe` oder andere äquivalente Werte geschrieben werden. Abbildung 4.13 illustriert die Reihenfolge, in der die Knoten abgearbeitet werden, wenn das abgebildete Dreieck an `schreibe_baum()`

übergeben wird. Weiße Knoten gehören zum Rand und werden bei dem Baumdurchlauf von `schreibe_baum()` und `schreibe_zwillinge()` nicht erreicht, die Ziffern in den Punkten im Inneren des Dreiecks geben die Reihenfolge wieder, in der die zugehörigen Knoten durchlaufen werden. Es bleibt eine Sonderbehandlung für Randknoten bereitzustellen, um den

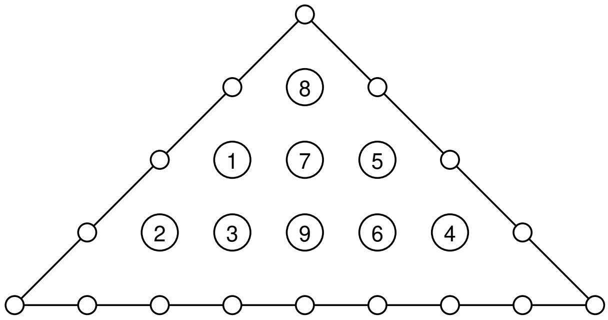


Abbildung 4.13 Reihenfolge in der Knoten geschrieben werden.

verbleibenden Rest des Gebiets auf den Hintergrundspeicher zu schreiben. Die Prozedur, die den Baum einer Kachel abspeichert sieht dann wie folgt aus.

```
schreibe_kachel( t )
{
  schreibe_baum( t→baum )
  schreibe_rand( t→baum )
}
```

Die Voraussetzung für die umgekehrte Funktion `lade_kachel()` ist `lade_baum()`, eine Funktion die eng mit `erzeuge_baum()` verwandt ist. Im wesentlichen muß für `lade_baum()` die Prozedur `verknüpfe_zwilling()` um die Möglichkeit, Daten nachzuladen erweitert werden. Das bei `erzeuge_baum()` integrierte Hochpropagieren der Fehler muß jetzt anschließend in einem separaten Baumdurchlauf erfolgen, da erst dann die erforderlichen Daten vorliegen. Demnach sieht `lade_baum()` etwa folgendermaßen aus.

```
knoten lade_baum( a, b, c, v )
{
  if( tiefste Stufe erreicht ) return( leeren Baum )
  else {
    k = erzeuge neuen Knoten
    k→vorgänger = v
    k→links  = lade_baum( c, a, (a+b)/2, k )
    k→rechts = lade_baum( b, c, (a+b)/2, k )
    lade_und_verknüpfe_zwilling( k→links, k→rechts )
    return( k )
    }
}
```

Somit läßt sich die Prozedur `lade_kachel()` wie folgt schreiben.

```
lade_kachel( t )
{
  t→baum = lade_baum( t→a, t→b, t→c )
  lade_rand( t→baum )
  propagiere_fehler( t→baum )
}
```

Wobei `propagiere_fehler()` wie folgt konstruiert ist.

```
propagiere_fehler( k )
{
  if( k ist leerer Baum ) return
  else {
    propagiere_fehler( k→links )
    propagiere_fehler( k→rechts )
    k→überschuss = max(|k→links→c.höhe - Höhe von (k→a+k→b)/2|,
                       k→links→überschuss,
                       k→rechts→überschuss)
}
```

Dabei wird `k→links→c.höhe` von der Prozedur `lade_und_verknüpfe_zwillinge()` nachgeladen, die analog zu `schreibe_zwillinge()` konstruiert ist und neben dem zu `schreibe()` dualen Aufruf von `lade()` auch einen `verknüpfe()`–Aufruf enthält.

Zugriff auf die Daten auf dem Hintergrundspeicher

Beim Zugriff auf die Daten auf dem Hintergrundspeicher wurden drei verschiedene Klassen von IO–Funktionen, die das Betriebssystem zur Verfügung stellt, getestet. Zunächst wurden die Daten auf dem Hintergrundspeicher mit Hilfe von `mmap()` auf ein Speichersegment abgebildet. Es handelt sich dabei um den gleichen Mechanismus, mit dem das Betriebssystem Teile von Prozessen aus dem Hauptspeicher auf den Hintergrundspeicher auslagert. Dieser Mechanismus ist insofern sehr elegant, als daß nach einem entsprechenden `mmap()`–Aufruf alle Zugriffe transparent erfolgen, d.h. das Betriebssystem kümmert sich um jeden weiteren Transfer von Daten zwischen Haupt- und Hintergrundspeicher, während der Benutzer einfach auf ein Array zugreift. Da nicht ganze Knoten sondern lediglich deren Höhenwerte auf den Hintergrundspeicher gesichert werden, kann der eigentliche Vorteil von `mmap()` nicht greifen. Dabei handelt es sich um eine programmgesteuerte Verwaltung von virtuellem Speicher, die aufgrund der Kenntnis über den Programmablauf effizienter wäre als die virtuelle Speicherverwaltung des Betriebssystems. Zudem kommt die Einschränkung von `mmap()` wonach nur Speicherstücke, die ein ganzzahliges Vielfaches der systemabhängigen Seitengröße[4] ausmachen, verwaltet werden können. Dieser Wert beträgt typischerweise zwischen 4 und 16kB. Da die Größe einer Kachel auf dem Hintergrundspeicher in den seltensten Fällen einem ganzzahligen Vielfachen der Seitengröße entspricht, ist ein gewisser Verschnitt und damit Verschwendung von Platz auf

[4] zu ermitteln mit `getpagesize(3)`

dem Hintergrundspeicher unvermeidlich. Der wesentliche Nachteil von `mmap()` zeigte sich jedoch bei der Verarbeitung großer Datensätze. Dabei war das Betriebssystem mit der Verwaltung von zuvielen `mmap`–Segmenten, wie sie z.B. bei dem Testdatensatz von ganz Deutschland auftreten, wenn man pro Kachel ein `mmap`–Segment anlegt, überlastet. Aus diesen Gründen wurde auf den Einsatz von `mmap()` verzichtet.

Die zweite Klasse von IO–Funktionen, die eingesetzt wurde, sind die klassischen `read(2)` und `write(2)` Funktionen. Da jedoch immer einzelne Höhenwerte separat geschrieben bzw. gelesen werden steigt die Zahl der `read()`/`write()`–Aufrufe so stark, daß der IO–Durchsatz einbricht. Schließlich wurde die gepufferte Variante `fread()` und `fwrite()` eingesetzt, die Schreib- und Leseoperationen in geeigneter Weise zusammenfassen.

Dateiformat für das dynamische Nachladen

Das Dateiformat, wie es letztendlich auf dem Hintergrundspeicher abgelegt ist, wird in Abbildung 4.14 illustriert. Der Header enthält alle Daten, die erforderlich sind, um die Größe aller

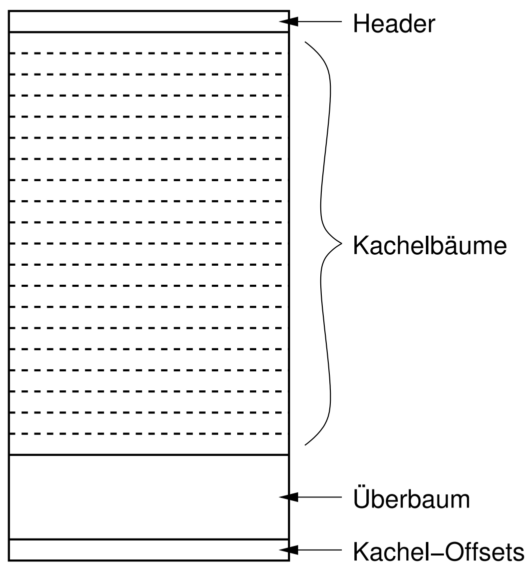


Abbildung 4.14 Dateiaufbau zum dynamischen Nachladen.

Verwaltungsstrukturen bestimmen zu können. Dazu gehören unter anderem Gelände- und Kachelgröße sowie geographische Länge und Breite, in denen sich die Höhendaten befinden, um die Geometrie des zugrundeliegenden Rasters bestimmen zu können. Im Anschluss folgen die Daten der einzelnen Kacheln und die Daten des Überbaums. Am Ende der Datei steht eine Tabelle mit Kacheloffsets, die die Position jeweils einer bestimmten Kachel innerhalb der Datei wiedergeben. Diese Reihenfolge ergibt sich aus dem Verfahren zum Schreiben der Daten. Dabei wird der Höhendatensatz soweit möglich kachelweise eingeladen, diese werden dann in einen Baum im Speicher überführt und diese wiederum auf Platte geschrieben. Danach wird der Speicher für die Kachel sofort wieder freigegeben. Bei diesem Vorgehen wird sukzessiv der Überbaum aufgebaut, der erst komplett ist, wenn alle Kacheln verarbeitet wurden. Erst dann kann er auf den Hintergrundspeicher geschrieben werden. Diese Strategie ist erforderlich, da der ganze Höhendatensatz nicht in den Speicher passt, wobei allerdings vorausgesetzt wird, daß mindestens der ganze Überbaum und eine Kachel Platz haben. Ist dies nicht der Fall, so ist eine interaktive Visualisierung des Datensatzes mit den gewählten Größen für Über- und Kachelbäume ohnehin nicht möglich.

Beim Laden der Daten wird der Abschnitt der Datei, die die Kacheldaten enthält und dessen Größe sich aus den Header–Informationen errechnen läßt, zunächst übersprungen und erst der Überbaum geladen. Im Anschluss wird die Kacheltafel um die Information der Position der Kacheln innerhalb der Datei ergänzt. Die einzelnen Kacheln, die für die Triangulierung erforderlich sind, werden beim Markierungsdurchlauf nachgeladen, indem ihre Position in der Datei mit `fseek()` angesprungen wird und dann der Baum dieser Kachel erzeugt wird, wobei die Höhenwerte mittels `fread()` eingelesen werden. Kacheln, die bei einer Triangulierung nicht benötigt wurden, können freigegeben werden.

4.2 Ergebnisse

Die Reimplementierung des Triangulierungsverfahrens der virtuellen Sicht auf Bäumen ermöglicht und zeigt die Machbarkeit des dynamischen Nachladens bei Einsatz dieses Verfahrens. Gleichzeitig zur Darstellung von Höhendaten wurden Möglichkeiten zur Darstellung anderer Strukturen anhand von Bilddaten untersucht.

4.2.1 Bilddaten

Bilddaten sind insofern mit Höhendaten verwandt, als daß sie in einem festen Raster vorliegen und jedem Rasterpunkt ein Wert zugeordnet ist. Der Einfachheit halber werden nur Graustufenbilder untersucht, eine Erweiterung auf Farbbilder ist durch separate Behandlung der Farbkanäle leicht möglich. Der bei Bilddaten gespeicherte Wert entspricht dem Grauwert bzw. der Helligkeit des jeweiligen Bildpunktes. Um die Triangulierung der virtuellen Sicht auf Bilddaten anzuwenden, werden anstelle der Höhenwerte die Grauwerte als Quelle für das Adaptionskriterium herangezogen. Der wesentliche Unterschied, der sich bei den Programmen ergibt, ist der Teil, der das Zeichnen der Dreiecke übernimmt. Hier müssen die Dreiecke an ihren Ecken entsprechend den an diesen Knoten gespeicherten Grauwerten eingefärbt werden. Da der Farbverlauf im Inneren der Dreiecke zwischen den Farben an den Ecken linear interpoliert wird, entsteht der Eindruck eines kontinuierlichen Farbverlaufs über das ganze Dreiecksnetz. Die dritte Dimension, der Höhe, fällt weg, das Bild wird in der X–Y–Ebene liegend dargestellt. Abbildung 4.15 zeigt ein Bild, das nach diesem Verfahren erzeugt wurde. Auf der rechten Seite ist das zugehörige Gitter zu sehen.

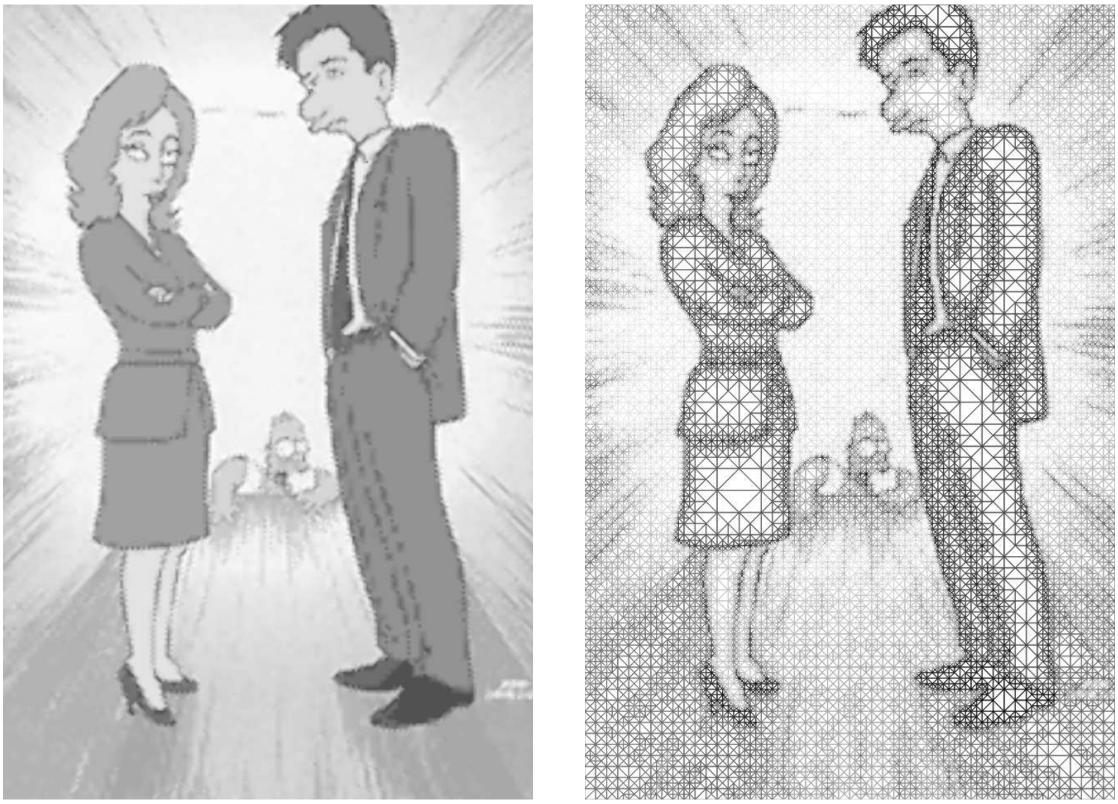


Abbildung 4.15 Triangulierte Bilddaten mit Gitter.

Fehlermaß

Während das Fehlermaß bei der Visualisierung von Geländedaten der Höhenfehler ist, muß für Bilddaten ein anderes Fehlermaß eingesetzt werden. Da sich der Fehler beim Weglassen bzw. Vergröbern von Dreiecken bei Bildern in Helligkeitsunterschieden zeigt, die sich über die Fläche des Dreiecks verteilen, wurde der Helligkeitsüberschuss im Mittelpunkt der Hypotenuse mit der Fläche des Dreiecks in Pixeln gemessen skaliert und das Ergebnis als Fehlermaß genutzt. Dies hat zur Folge, daß der Fehler geringer wird, je weiter man sich von dem Bild entfernt, da dieses aufgrund der perspektivischen Darstellung immer kleiner wird. Aufgrund des kleineren Fehlers, der gemacht wird, sinkt gleichzeitig die Zahl der Dreieck bei der Triangulierung. Die Anzahl der Dreiecke nimmt mit zunehmendem Abstand ab, während der Fehler der gemacht wird und damit die Bildqualität gleich bleibt.

Legale Löcher

Eine Besonderheit bei der Triangulierung von Bildern ist, daß Löcher, die bei Höhendaten als extrem störend empfunden werden bei Bildern unter bestimmten Umständen zulässig sein können, da sie nicht wahrgenommen werden. Abbildung 4.16 zeigt ein Beispiel für „legale Löcher“. Diese treten dann auf, wenn am Rand einer Kachel der Überschuss unterhalb der Feh-

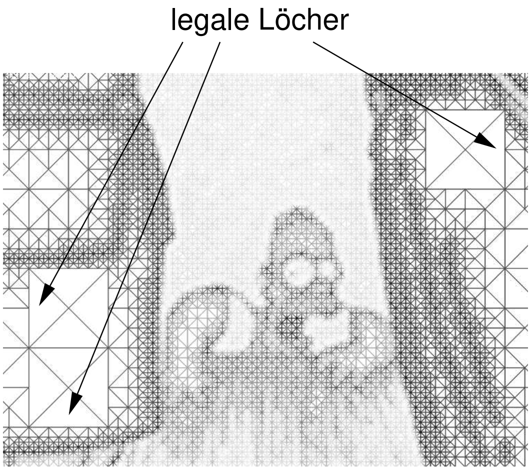


Abbildung 4.16 Beispiel legaler Löcher bei der Triangulierung von Bildern.

lerschranke liegt und die Nachbarkachel nur geladen werden müsste, um Zwillingsknoten zu markieren. Tritt diese Situation ein, so kann bei Bilddaten auf das Laden der Nachbarkachel verzichtet werden. Bei Kacheln die so wie im Beispiel einfarbige Gebiete abdecken, ist das in der Regel der Fall. Bei Höhendaten könnte es in so einer Situation zu einem vertikalen Loch kommen, das, auch wenn seine Höhe unterhalb der Fehlerschranke liegt, als störend empfunden wird. Bei Bilddaten führt ein vertikales Loch höchstens zu Helligkeitsunterschieden zwischen benachbarten, nahtlos aneinander grenzenden Dreiecken, die deutlich weniger bzw. gar nicht als störend empfunden werden.

Bewertung

Für den praktischen Einsatz ist eine Darstellung von Bildern durch grauwertgesteuerte Triangulierung wenig sinnvoll. Hardwareunterstütztes Texturemapping ist inzwischen bei besseren Graphikkarten üblich. Die damit erreichte Darstellungsqualität kann von einer Triangulierung nur erreicht werden, wenn der erlaubte Fehler nahe bei bzw. auf Null gesetzt wird, was zu einer sehr feinen Triangulierung führt und aufgrund der hohen Zahl dann zu zeichnender Dreiecke zu einer langsamen Darstellung. Da auch weit entfernte texturierte Polygone dank dem sogenannte Mip–Mapping [39] ohne Artefakte oder Aliasingeffekte dargestellt werden können und auch Verfahren zur Darstellung von Texturen existieren, die größer sind als der Texturspeicher, bleibt die Darstellung von Bilddaten durch Dreiecksnetze in der Praxis bedeutungslos.

Die Struktur und Strategie zur Verwaltung von Kacheln, die in diesem Kapitel entwickelt wurde, kann selbstverständlich auch zur Verwaltung von Texturkacheln eingesetzt werden und somit bei der Darstellung von Texturen, die größer sind als der Texturspeicher, helfen.

4.2.2 Höhendaten

Das gesetzte Ziel der Visualisierung von Höhendatensätzen, die deutlich größer sind als der Hauptspeicher wurde erreicht. Der anfangs als Beispiel angeführte Blick von Hamburg in die Alpen ist in Abbildung 4.17 zu sehen. Links ist das untexturierte Gelände zu sehen, im Hinter-

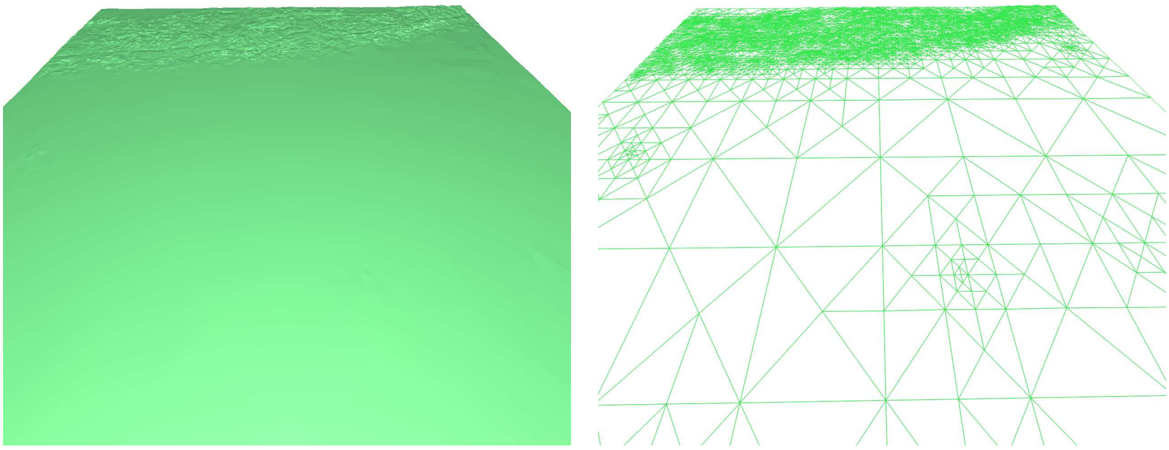


Abbildung 4.17 Blick von Hamburg in die Alpen.

grund erkennt man die Alpen, rechts ist das zugehörige Gitter abgebildet. Die Darstellung wurde auf einer Silicon Graphics O2 mit 300 MHz MIPS R12000 CPU und 384 MB Hauptspeicher erzeugt. Die aus ca. 16000 Dreiecken bestehende Triangulierung konnte mit einer Bildwiederholrate von 3–4 Hz erneuert werden. Vernachlässigt man die Bildwiederholrate, so ist die Qualität der Geländeapproximation nur durch die Größe des Hauptspeichers limitiert. Im vorliegenden Beispiel wurde bei einer Sichtweite von ca. 1000km ein Fehler von 11 Metern pro Level an der Far–Clipping–Plane zugelassen. Der Fehler fällt zwischen Far– und Near–Clipping–Plane linear bis auf einem Meter pro Level ab.

Auswirkung der Kachelgröße

Die Kachelbäume in Abbildung 4.17 sind sechs Level tief. Wird die Tiefe verringert, so verdoppelt sich mit jedem Level der Platzbedarf des Überbaums, was auf der oben angegebenen Plattform zu einem Einbruch der Bildwiederholrate auf unter ein Hz führt. Bei kleineren Kachelbäumen wird der Überbaum so groß, daß ein Durchlauf bei den gegebenen Cache– und Speicher–Größen[5] extrem ungünstig wird, wodurch sich der Leistungseinbruch erklären läßt.

Die Baumknoten wurden bei dieser Implementierung großzügig mit Informationen gefüllt, die für die eigentliche Triangulierung nicht unbedingt erforderlich sind bzw. die auch dynamisch beim Baumdurchlauf berechnet werden könnten. Ein Beispiel für letzteres sind z.B. die Koordinaten der Ecken der Dreiecke. Bei der Implementierung der virtuellen Sicht werden diese während dem rekursiven Durchlauf durch das Array dynamisch berechnet. Da hierzu bei jedem

[5] 32kB primary Cache, 1MB secondary Cache und 384MB Hauptspeicher

rekursiven Aufruf lediglich drei Additionen und drei Multiplikationen erforderlich sind fällt dieser Mehraufwand nicht ins Gewicht. Würde auf die überflüssigen und dynamisch berechenbaren Informationen in den Baumknoten verzichtet, so würde der beobachtete Leistungseinbruch erst bei geringeren Kachelbaumtiefen auftreten. Letztendlich gehören auch die Cache–Größen zu den Parametern, die Einfluss auf den Entwurf der in Abschnitt 4.1.2 angesprochenen Kachelhierarchien haben. Eine optimale Lösung muß für jedes System individuell ermittelt werden.

Nachladeverhalten

Die Häufigkeit, mit der Kacheln nachgeladen werden, hängt ab von der Beschaffenheit des Geländes, von der Geschwindigkeit, mit der sich der Betrachter bewegt bzw. die Blickrichtung ändert und von der gewählten Fehlerfunktion. Bei dem Testdatensatz war mit einer für die virtuelle Sicht typischen Geschwindigkeit und einer Kachelbaumtiefe von sechs für den Benutzer die durch das Nachladen verursachte Verzögerung in der Bildschirmdarstellung nicht festzustellen. Nur bei sehr schnellen Richtungswechseln, wie etwa bei spitzen 90 Grad Kurven in bergigem Gelände, kam es zu feststellbaren, durch das Nachladen verursachten Verzögerungen. In flachem Gelände, wozu aus der Sicht der Triangulierung bis auf die Alpen der gesamte Testdatensatz gezählt werden darf, lag die Bildwiederholrate zwischen 7–8Hz. Kacheln müssen nur sporadisch alle paar Sekunden nachgeladen werden.

4.3 Weitere Effizienzverbesserungen

Die im Rahmen dieses Kapitels entwickelte Implementierung sollte hauptsächlich die Möglichkeit der Visualisierung von großen Geländedatensätzen mit der eingesetzten Triangulierungsstrategie zeigen. Es bleibt eine Reihe von Ansatzpunkten zur Verbesserungen und Möglichkeiten zur Optimierung, von denen hier einige aufgezeigt werden sollen.

Hybrid–Implementierung

Aus den Erfahrungen, die bei der Implementierung der Triangulierung auf Baumstrukturen gewonnen wurden, läßt sich der Schluss ziehen, daß eine Baumstruktur für die Kachelbäume zu aufwendig ist und auch ihren Zweck nicht erfüllt, nämlich daß unbenutzte Teilbäume freigegeben werden können. Eine Hybrid–Implementierung, die die Vorteile der Triangulierung auf Arrays aus Kapitel 3 und die Kachelverwaltung mit Hilfe des Überbaums aus diesem Kapitel vereint, erscheint der optimale Kompromiss zu sein. Anstelle von Kachelbäumen würden Kachelarrays nachgeladen. Da dabei nicht mehr für jeden Wert ein eigener Baumknoten alloziert werden muß und die Werte mit einem `read()`–Aufruf gelesen werden können, sinkt die Zeit für das Nachladen einer Kachel drastisch. Auch ein hierarchischer Durchlauf durch eine Kachel beschleunigt sich durch die kompakt und damit cache–freundlich im Speicher liegenden

Kachel–Daten. Die Überbaum Struktur ermöglicht es, leicht festzustellen, welche Kacheln geladen werden müssen und auf welche verzichtet werden kann, gleichzeitig beinhaltet sie die Informationen über das Gelände von Bereichen, in denen die Kacheldaten nicht vorliegen.

Auch bei Bilddaten scheint der Weg der Hybrid–Implementierung der richtige Weg zu sein. Mehrere Kachelhierarchien können auf verschiedene Mip–Map–Level abgebildet werden. Im seltenen Fall, daß ein Bild mit einem sehr flachen Winkel betrachtet wird, können weit entfernte Teile bzw. Kacheln, die keine Platz mehr im Texturspeicher finden, durch gouraud–schattierte (siehe [17]) untexturierte Dreiecke approximiert werden.

Eine Kombination der Höhendaten- und Bilddarstellung ist denkbar, die es erlaubt, beide Datentypen, Höhendaten und zugehörige Texturen, in einer Struktur zu verwalten. Bei anderen integrierten Systemen für Geländedarstellung (siehe z.B. [24] oder [33]) werden diese Komponenten in der Regel getrennt verwaltet, obwohl sie in der Darstellung eng zusammenhängen.

Multithreading

Es liegt nahe, den Teil des Algorithmus, der sich um die Darstellung auf dem Bildschirm kümmert, soweit als möglich von dem Teil, der IO–Operationen durchführt und Daten vom Hintergrundspeicher nachlädt, zu entkoppeln. Dies wird in der Regel durch getrennte Threads erreicht. Eine Aufteilung in drei Threads würde bestmögliche Entkoppelung bedeuten. Der erste Thread kümmert sich ausschließlich um die Bildschirmdarstellung und läuft so schnell er kann durch die Datenstrukturen. Ein zweiter Thread kümmert sich um die Verwaltung der Daten und deren Konsistenz. Er markiert Dreiecke bzw. Baumknoten und entscheidet, welche Teile geladen werden müssen und welche freigegeben werden können. Der letzte Thread kümmert sich ausschließlich um IO, er lädt Kacheln und gibt sie wieder frei. Dabei kann er unter Umständen unbenutzte Kacheln einfach „überladen“ und spart so Allozier– und Freigebe–Zyklen.

Intelligente IO–Strategien

Die einfachste Verbesserung der aktuell implementierten IO–Strategie, bei der unbenutzte Kacheln sofort freigegeben werden, wäre, stattdessen damit einige Triangulationszyklen zu warten, da es nicht unwahrscheinlich ist, daß in diesem Zeitraum die Kachel aufgrund der sich ändernden Triangulierung wieder benötigt wird. Ähnlich verhält es sich mit Kacheln die gerade in den Bereich außerhalb des Sichtkegels gewandert sind. Auch hier ist es nicht unwahrscheinlich, daß bei einer geringfügigen Änderung der Blickrichtung, die bei der virtuellen Sicht recht häufig vorkommt, diese Kachel wieder sichtbar wird. So läßt sich einfach überflüssiges „flattern“ von Kacheln verhindern, die sonst ständig geladen und freigegeben würden.

Das Nachladen von Kacheln ist vom Prinzip her das gleiche wie das Auslagern von Teilen des einem Prozess zugeordneten Speichers auf den Hintergrundspeicher, das sogenannte *paging*, das jedes moderne Betriebssystem praktiziert. Bei der Entscheidung, welche Teile ausgelagert werden sollen wurden viele Strategien entwickelt, die auch beim Nachladen von Kacheln zum

Einsatz kommen können. Dazu gehört es eine feste Menge von Speicher für Kacheln vorzugeben und jeweils die unwichtigste durch eine nachzuladende Kachel zu ersetzen. Eine andere beim paging benutzte Strategie wäre, die Kachel zu verwerfen, die am längsten nicht benutzt wurde (*least recently used*).

Bei jeder Form von Cache–Verwaltung ist es üblich, daß Daten geladen werden, bevor diese angefordert werden, das sogenannte *Prefetching*. Dabei werden Daten, von denen es nicht unwahrscheinlich ist, daß sie in Kürze benötigt werden in den Cache geladen. Dabei kann ausgenutzt werden, daß Datentransfer der in seiner Größe einem Vielfachen der Seitengröße des Systems oder des Hintergrundspeichers entspricht, die verfügbare Bandbreite optimal ausnutzt. Da von Applikationen in den seltensten Fällen Daten in der passenden Größe bewegt werden, lassen sich die entstehenden Lücken mit Prefetching–Anfragen auffüllen und so die verfügbaren IO–Bandbreiten besser ausnutzen. Bei der virtuellen Sicht läßt sich aufgrund der Informationen, die das Programm über den Standort des Betrachters, seine Blickrichtung und seine Geschwindigkeit hat, sehr genau abschätzen, welche Kacheln in nächster Zeit benötigt werden. Somit kann die Prefetching–Strategie einen wesentlichen Beitrag zur Entlastung des IO–Threads leisten.

5 Ergebnisse

Die im Laufe dieser Arbeit erfolgte Entwicklung einer verbesserten Geländetriangulierung für die am Lehrstuhl für Flugmechanik und Flugregelung der TU München entwickelten virtuellen Sicht und deren Reimplementierung bieten signifikante Vorteile gegenüber ihrer ursprünglichen Implementierung (siehe [28]). Während die ursprüngliche Implementierung bei ihrer Umsetzung des Level–of–Detail Konzeptes einen sehr einfachen Ansatz wählt (vergleiche Abbildung 2.9 auf Seite 16) und sich so das Problem der vertikalen Löcher einhandelt, das nicht zufriedenstellend gelöst wird, wurde bei der Entwicklung der verbesserten Geländetriangulierung speziell darauf geachtet, daß dieser Effekt nicht auftritt. Durch die Umsetzung eines kontinuierlichen Level–of–Detail Ansatzes konnte die Anzahl der erforderlichen Dreiecke bei der Geländeapproximation erheblich reduziert werden, was unter anderem auch größere Sichtweiten und die Nutzung höher aufgelöster Höhendatensätze (z.B. DTED Level Zwei) erlaubt. Ebenso führt die Erfüllung der Entwurfskriterien Modularität und Portabilität zu einer wesentliche Vereinfachung der Weiterentwicklung der virtuellen Sicht. Die ursprüngliche Implementierung ist zwar in der Lage auf einem Graphik–Supercomputer[1] gute Ergebnisse zu liefern, überfordert mit ihren Anforderungen an die Graphikleistung jedoch schwächere Systeme, die in der Regel kleiner und leichter sind und deswegen für den Einsatz in Flugzeugen in Bezug auf Betriebs- und Anschaffungskosten deutliche Vorteile bieten. Im Laufe der Reimplementierung der virtuellen Sicht im Rahmen dieser Arbeit wurde das System auf verschiedene Plattformen (Solaris, HP–UX, Linux bzw. FreeBSD) portiert und so der Nachweis erbracht, daß die virtuelle Sicht sich auch auf Basis dieser Systeme einsetzen läßt. Aufgrund der sich in jüngster Zeit ausweiten Hardwareunterstützung von OpenGL auf kostengünstigen PC–basierten Systemen wie Linux und FreeBSD eröffnen sich völlig neue Perspektiven für den Einsatz der virtuellen Sicht. Die Durchführung erfolgreicher Flugversuche mit der neu entwickelten virtuellen Sicht im Juli 1997 bestätigte eindrucksvoll den erfolgreichen Abschluss dieses Teilabschnitts dieser Arbeit (siehe [37]). Dabei wurde neben einem Tiefflug durch das relativ enge Kinzigtal in der Nähe von Freiburg auch mehrere krummlinige Anflüge auf die Landebahn des Freiburger Flughafens durchgeführt. Der Effekt des poppenden Geländes, der durch den Einsatz des kontinuierlichen Level–of–Detail zustande kommt, wird von den Piloten, die letztendlich die Anwender der virtuellen Sicht sind, nicht als störend empfunden, weswegen keine weiteren Anstrengungen unternommen wurden, dem Poppen des Geländes entgegenzuwirken.

Die virtuelle Sicht wird neben ihrem eigentlich vorgesehenen Einsatzgebiet, der Navigations- und Flugführungsunterstützung, am Lehrstuhl für Flugmechanik und Flugregelung auch für die Erzeugung der Außensicht des dort installierten Flugsimulators eingesetzt. Auch hier hat sich die durch die modulare Implementierung gewonnene Flexibilität des Systems bei der Ansteuerung des Dreikanal–Displays bewährt. Der hierfür eingesetzte Graphik–Supercomputer erlaubt aufgrund seiner Mehrprozessorarchitektur die Umsetzung von in dieser Arbeit aufgezeigten

[1] Silicon Graphics Onyx mit InfiniteReality Graphik

Optimierungsmöglichkeiten wie der Parallelisierung, die auf der angestrebten Zielplattform der virtuellen Sicht für den Einsatz in Flugzeugen in der Regel noch nicht zur Verfügung steht.

Das einschränkenste Defizit der virtuellen Sicht ist die fehlende Möglichkeit, dynamisch Gelände nachzuladen, wenn das Flugzeug das Gelände verlässt, das sich gerade im Speicher befindet. So ist der Einsatz bisher auf Testflüge zu Forschungszwecken beschränkt. Im zweiten Teil dieser Arbeit wurde die für die virtuelle Sicht entwickelte Geländetriangulierung erweitert, um das dynamische Nachladen und Freigeben von Daten zu ermöglichen. So wird die interaktive Visualisierung von Höhenfeldern und vergleichbaren Daten ermöglicht, deren Umfang jenseits aller heute üblichen Hauptspeichergrenzen liegt. Gleichzeitig erlaubt die hierarchische Repräsentation des Datensatzes einen nahtlosen Übergang zwischen der Darstellung einer Gesamtübersicht über alle Daten und einer Detailansicht eines bestimmten Abschnitts. Aufgrund des Level–of–Detail Konzepts ist dieser Übergang bereits innerhalb einer graphischen Darstellung des Datensatzes möglich. Dies zeigt sich am Beispiel der Darstellung eines Blickes über ganz Deutschland hinweg von Hamburg bis in die Alpen. Dabei werden die Alpen aufgrund ihrer großen Höhenunterschiede und das näher gelegene Gelände aufgrund der geringeren Fehlerschranke in diesem Bereich feiner trianguliert als das dazwischen liegende Gelände [2]. Voraussetzung dafür war die Neuimplementierung der Geländetriangulierung auf der Basis von Bäumen, die die Möglichkeit bieten, Teilbäume dynamisch nachzuladen und freizugeben, anstatt auf den für die virtuelle Sicht eingesetzten Arrays. Zu diesem Zweck wurde für jedes Dreieck des Dreiecks–Binärbaums (vergleiche Abbildung 2.12 auf Seite 18) ein Baumknoten alloziert. In jedem der Knoten wurden zusätzliche Informationen abgelegt, die bei der Implementierung auf Arrays zum Teil dynamisch während dem rekursiven Triangulierungsdurchlauf dynamisch berechnet oder aufgrund von Indexrechnung auf dem Array ermittelt wurden. Um diese Informationen bei der Triangulierung bereitzustellen, muß beim Baumaufbau ein gewisser Aufwand getrieben werden, der bei der Array–Implementierung nicht erforderlich war. Ein vorrangiges Beispiel für die im Baum nicht per se vorhandene Information, die bei Einsatz von Arrays leicht zu ermitteln ist, ist die Nachbarschaftsbeziehung von Dreiecken. Zwei Dreiecke, die sich eine gemeinsame Hypotenuse teilen, lassen sich in Arrays leicht durch levelabhängige Index–Offsets einander zuordnen, während dies im Baum eines expliziten Verweis auf den Nachbarknoten bedarf. Um diese Nachbarschaftsbeziehungen im Baum zu ermitteln und die jeweiligen Knoten miteinander zu verlinken, wurde eine rekursive Baumdurchlaufstrategie implementiert, die sich mit keinem der üblichen Standardbaumdurchläufe vergleichen läßt.

Da das Freigeben und Nachladen von einzelnen Baumknoten in Bezug auf die IO–Performance zu aufwendig ist, wurden diese zu sogenannten Kacheln zusammengefasst. Die Zusammenfassung von Unterbäumen in Kacheln ermöglicht die Bündelung der IO–Operationen beim Nachladen. Dabei erlaubt das Kachelkonzept die Einführung mehrerer Ebenen von Kacheln, die in einer Art Metahierarchie über den Baum von Höhenwerten gelegt werden, was eine speziell an den Daten und die Anwendung angepasste Verwaltungsstruktur ermöglicht. Für die Erweiterung der virtuellen Sicht wurde ein Baum mit einem Kachellevel implementiert. Die Verwaltung der Kacheln erfolgt über eine Kacheltafel, mit deren Hilfe Kachelbäume auf dem Hintergrundspeicher lokalisiert und in den Baum eingefügt werden können. Das Nachladen von Kacheln erfolgt dynamisch während der Triangulierung und ist vollständig in die Prozeduren zur Triangulierung integriert.

[2] Dies wird in der Darstellung in Abbildung 4.17 auf Seite 63 leider nicht deutlich, da diese von einem sehr hoch gelegenen Betrachterstandpunkt aus erzeugt wurde, um im Abdruck erkennbare Strukturen zu erzeugen.

Neben der Anwendung auf Höhendaten wurde in diesem Abschnitt auch die Visualisierung von Bilddaten mit Hilfe der entwickelten Verfahren untersucht. Aufgrund der Verwandtschaft von Höhen- und Bilddaten war hierzu lediglich eine Änderung der Zeichenfunktion und des Fehlermaßes nötig. Die Erfahrungen, die bei dieser Untersuchung gewonnen wurden, zeigen, daß eine Darstellung von Bilddaten durch Dreiecksnetze Verfahren, die Texture–Mapping einsetzen, unterlegen sind. Nichtsdestotrotz kann man die Kachelhierarchie effizient zur Verwaltung und Organisation des dynamischen Nachladens von Texturen einsetzen, die insgesamt zu groß für den Texturspeicher sind und deswegen aufgeteilt werden müssen.

Bei Höhendaten hat sich die Kachelhierarchie zur Verwaltung bewährt, die konsequente Darstellung der Kachelbäume durch tatsächliche Bäume ist im Gegensatz zur Darstellung als Array jedoch zu aufwendig. Einerseits ist der Speicheraufwand aufgrund der nötigen Zusatzinformationen im Baum um ein vielfaches größer als beim Einsatz von Arrays. Zum anderen ist die Baumdarstellung hier auch in Punkto Laufzeit der Arraydarstellung unterlegen, da die Daten nicht mehr kompakt im Speicher liegen und so bei einem Durchlauf durch die Daten CPU–Caches nicht mehr sinnvoll arbeiten können. Schließlich erfordert die einzelne Allozierung der Baumknoten das individuelle Nachladen von einzelnen Werten, was in Bezug auf IO–Performance besser vermieden werden sollte.

Eine optimaler Kompromiss scheint die Integration der Array- und Baumdarstellung zu sein. Dabei sollte die Baumdarstellung für die Überbaumstruktur gewählt werden, die hauptsächlich zur Organisation des Nachladens von Kacheln genutzt wird. Gleichzeitig erlaubt sie auch die Darstellung von Gebieten, in denen die Kacheldaten noch nicht vorhanden sind bzw. aufgrund der Fehlerschranke auch nicht benötigt werden. Die Arraydarstellung ist für die Kachelbäume bzw. in diesem Fall Kachelarrays am sinnvollsten. Zum einen können die Daten schnell nachgeladen werden, da sie sowohl im Speicher als auch auf dem Hintergrundspeicher kompakt liegen. Der Durchlauf durch die Daten beim Laden und Triangulieren ist sehr schnell, da sich die Größe der Kacheln typischerweise in einer Größenordnung bewegt, die in kompakter Darstellung zusammen mit allen nötigen Hilfsinformationen, die wiederum in Arrayform vorliegen, in den Secondary–, wenn nicht sogar in den Primary–Cache moderner Prozessoren passt. So läßt sich eine sehr schnelle, flexible und effiziente Geländetriangulierung entwickeln, die auf einer speziell an die Triangulierung angepassten Datenbank arbeitet und die interaktive Visualisierung beliebig großer Höhendatensätze erlaubt.

Liste der Prozeduren und Funktionen

Abbildungsverzeichnis

Literaturverzeichnis

[1] *HP-UX Reference Guide*. Hewlett Packard Company: 1992.

[2] *IRIX Programmer's Reference Manual*. Silicon Graphics Inc., Mountain View, California: 1993.

[3] *Osf/Motif Programmer's Guide Release 1.2 (Osf/Motif Series)*. Prentice Hall: 1992.

[4] Blow, Jonathan: „Terrain Rendering at High Levels of Detail". *Game Developers' Conference, San Jose, California, USA* (2000).

[5] Bungartz, H.: *Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung*. Dissertation. Institut für Informatik, TU München. 6 1992.

[6] Dobler, K.: *Untersuchung zu räumlich integrierten Flugführungsanzeigen*. Dissertation. Fakultät für Maschinenwesen, TU München. 2000.

[7] Duchaineau, M., M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich und M.B. Mineev-Weinstein: „ROAMing terrain: Real-time Optimally Adapting Meshes". In: *Proceedings of the 8th IEEE Visualization '97 Conference*. Los Alamos Nat. Lab., NM, USA: 1997.

[8] Foiani, Leila De, und Paola Magillo: *Triangle–based surface models*. Technical report . Università di Genova, Dipartimento di Informatica e Science dell'Informazione. 1994.

[9] Foley, J.D., A. van Dam, S.K. Feiner und J.F. Hughes: *Computer Graphics – Principle and Practice*. . 2. Aufl. Reading, Massachusetts: Addison-Wesley Publishing Company: 1992: Kap. 3.12.3, 113–117.

[10] Foley, J.D., A. van Dam, S.K. Feiner und J.F. Hughes: *Computer Graphics – Principle and Practice*. . 2. Aufl. Reading, Massachusetts: Addison-Wesley Publishing Company: 1992: Kap. 15.10.2, 707–710.

[11] Fortune, S.: „Voronoi Diagrams and Delaunay triangulations". In Du, D. Z., und F. Hwang (Hrsg.): *Computing in Euclidean Geometry*. World Scientific Publ.: 1992: 193–223.

[12] Frank, A.: *Hierarchische Polynombasen zum Einsatz in der Datenkompression mit Anwendung auf Audiodaten*. Diplomarbeit. Institut für Informatik, TU München. 1995.

[13] Frederick, C. O., Y. C. Wong und F. W. Edge: „Two-dimensional Automatic Mesh Generation for Structural Analysis". *Internat. J. Numer. Methods Eng.* **2** (1970) 133–144.

[14] Gerstner, T.: *Ein adaptives hierarchisches Verfahren zur Approximation und effizienten Visualisierung von Funktionen und seine Anwendung auf digitale 3–D Höhenmodelle*. Diplomarbeit. Institut für Informatik, TU München. 1995.

[15] Goerzen, John: *Linux Programming Bible*. IDG Books Worldwide: 2000.

[16] Goldsmith, J., und J. Salmon: „Automatic Creation of Object Hierarchies for Ray Tracing“. *IEEE Computer Graphics and Applications* **7(5)** (May 1987) 14–20.

[17] Gouraud, H.: „Continuous Shading of Curved Surfaces“. *IEEE Transactions on Computers* **C-20(6)** (1971) 623–629.

[18] Graham, J.: *Solaris 2.x: Internals and Architecture*. McGraw-Hill, New York: 1995.

[19] Graphics, Silicon: *Open Inventor C++ Reference Manual*. Addison-Wesley Publishing Company: 1994.

[20] Gross, M.H., O.G. Staadt und R. Gatti: „Efficient Triangular Surface Approximations using Wavelets and Quadtree Data Structures“. *IEEE Transactions on Visualization and Computer Graphics* **2**(2) (June 1996).

[21] Hiller, K.: *Datenkompression mit dem Dünn-Gitter-Verfahren*. Diplomarbeit. Institut für Informatik, TU München. 1993.

[22] Kayton, M.: „Introduction to Aircraft Navigation“. In (Hrsg.): *Navigation: Land, Sea, Air & Space*. IEEE Press: 1989: 229–244.

[23] Lehey, Greg: *The Complete FreeBSD*. 3. Aufl. Walnut Creek: 1999.

[24] Lindstrom, P., D. Koller, W. Ribarsky, L. Hodges und N. Faust: *An integrated Global GIS and Visual Simulation System*. Tech Report GIT–GVU–97–07. Georgia Institute of Technology. März 1997.

[25] Lindstrom, Peter, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust und Gregory A. Turner: „Real–Time, Continuous Level of Detail Rendering of Height Fields“. *Computer Graphics (SIGGRAPH ’96 Proceedings)* (1996).

[26] Lippman, Stanley B., Josee Lajoie und Jose Lajoie: *C++ Primer*. Addison-Wesley Publishing Company: 1998.

[27] Mitchell, W.: „Optimal multilevel iterative methods for adaptive grids“. *SIAM J. Sci. Statist. Comput.* **13**(1) (Januar 1992) 146–167.

[28] Möller, H.: *Computergenerierte synthetische Sicht zur Verbesserung der Flugführung bei schlechter Außensicht*. Dissertation. Fakultät für Maschinenwesen, TU München. 1997.

[29] Möller, H., und G. Sachs: „Synthetic Vision for Enhancing Poor Visibility Operation“. *IEEE Aerospace and Electonics Magazine* **9** (1994) 27–33.

[30] Paul, A.: „Der FORTWIHR hebt ab – Flugversuche in Freiburg“. *FORTWIHR–Quartl* **4/1997**(16) (1997).

[31] Paul, A.: *Kompression von Bildfolgen mit hierarchischen Basen*. Diplomarbeit. Institut für Informatik, TU München. 1995.

[32] Rathmann, U.: „Künstliche Sicht“. In Carl-Cranz-Gesellschaft (Hrsg.): *CCG-Kurs LR 4.05 Moderne Unterstützungssysteme für Piloten*. Oberpfaffenhofen: 1992.

[33] Rohlf, J., und J. Helman: „Iris performer: A high performance multiprocessing toolkit for real-time 3D graphics“. In: *Proceedings of SIGGRAPH ’94, Computer Graphics*. ACM SIGGRAPH, New York: ACM Press: Juli 1994: 381–395.

[34] Röttger, S., W. Heidrich, Ph. Slusallek und H.-P. Seidel: „Real-Time Generation of Continuous Levels of Detail for Height Fields“. In: *Proc. 6th International Conference in Central Europe on Computer Graphics and Visualization ’98*. 1998: 315–322.

[35] Sachs, G., K. Dobler und P. Hermle: „Flugversuche zur Anwendung der synthetischen Sicht für den Flug in Bodennähe“. In: *Dt. Luft- und Raumfahrtkongr*. München: 14.-17.10. 1997.

[36] Sachs, G., K. Dobler, G. Schänzer und M. Dieroff: „Precision Navigation and Synthetic Vision for Poor Visibility Guidance“. In: *AGARD Flight Vehicle Integration Panel*. Lissabon: September 1996.

[37] Sachs, G., P. Hermle und W. Klöckner: „Flight Tests with Computer Generated Synthetic Vision“. In Bungartz, H.-J., F. Durst und C. Zenger (Hrsg.): *High Performance Scientific and Engineering Computing*. FORTWIHR. München: Springer–Verlag: März 1998: 177–188.

[38] Shreiner, Dave: *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*. 3. Aufl. Addison-Wesley Publishing Company: 1999.

[39] Williams, Lance: „Pyramidal Parametrics“. In: *Computer Graphics, SIGGRAPH Proceedings*. Juli 1983: 1–11.

[40] Zenger, C.: „Sparse grids“. In Hackbusch, W. (Hrsg.): *Parallel Algorithms for Partial Differential Equations: Proceedings of the 6th GAMM-Seminar, Kiel, Januar 1990, Notes on Numerical Fluid Mechanics 31*. Vieweg, Braunschweig: 1991.