Diplomarbeit

Entwicklung und Implementierung eines effizienten, hardwarenahen Antialiasing-Algorithmus

Fachbereich Informatik der
Technischen Hochschule Darmstadt
Institut für Informationsverwaltung und Interaktive Systeme
Fachgebiet Graphisch-Interaktive Systeme

von

Patrick Baudisch

Matr.Nr: 839284

Prüfer: Prof. Dr.-Ing. J. Encarnação

Betreuer: Dipl.-Ing. H.-J. Ackermann

Darmstadt, September 1994

Inhaltsverzeichnis

Hiermit erkläre ich an Eides statt, daß ich diese Diplomarbeit ohne unzulässige fremde Hilfe und nur unter Verwendung der angegebenen Literatur angefertigt habe.

Darmstadt, den 28. September 1994

(Patrick Baudisch)

Kurzfassung

Dank spezieller Graphik-Hardware können heute Applikationen wie Virtual Reality und Sichtsimulation auch auf Standardplattformen wie PCs ausgeführt werden. Wegen der beschränkten Leistungsfähigkeit solcher Systeme werden jedoch häufig Kompromisse eingegangen, die zuungunsten der Bildqualität gehen. Aliasing-Effekte, die bei der Generierung von Rasterbildern auftreten, werden dabei toleriert, da klassische Antialiasing-Algorithmen einen sehr hohen Rechenleistungs- und Speicherbedarf haben.

Das im Rahmen dieser Diplomarbeit entwickelte ‘Vier-Zeiger-Verfahren’ ist ein Antialiasing-Verfahren, das nur einen geringen Mehraufwand gegenüber dem normalen Rendering erfordert. Aufbauend auf dem PHIGS-Renderer Da*Render* wurde es in Software implementiert. Darüberhinaus eignet es sich gut für eine Realisierung in einem VLSI-Entwurf.

Das Verfahren arbeitet in zwei Schritten, einer Generierung von Subpixelmasken beim Rendering und einer Nachbearbeitung , bei der die generierten Masken ausgewertet und die resultierenden Farben aller Bildpunkte bestimmt werden. Um verbleibende Artefakte soweit wie möglich auszuschließen, wurde das Verfahren nach der Analyse von Testbildern nochmals erweitert.

Eine Hardware-Realisierung wird skizziert. Der Hardware-Mehraufwand des Verfahrens ergibt sich einerseits durch die notwendige Vergrößerung der Bildspeichertiefe und andererseits durch die benötigten Rechenwerke. Da die Erzeugung der Maskeninformation parallel zum Rendering erfolgen kann, resultiert ein Geschwindigkeitsverlust gegenüber dem einfachen Rendering nur aus der zusätzliche Traversierung von Polygonkanten. Dieser Verlust ist nur bei sehr vielen kleinen Polygonen signifikant, einer Situation wie sie aus Leistungsgründen bei Echtzeitapplikationen generell vermieden wird. Die Nachbearbeitung des Bildes kann in einer Pipeline parallel zur Generierung des nächsten Bildes erfolgen und wirkt sich so nicht negativ auf die Systemleistung aus. Das Verfahren stellt somit einen guten Kompromiß zwischen Bildqualität, Geschwindigkeit und Hardware-Aufwand dar.

Abstract

The anti-aliasing method presented in this paper is designed for the use in a real time rendering system as needed for simulation or virtual reality. It requires just a small amount of extra processing time compared to image generation without anti-aliasing. The method is divided into two distinct processes, i.e. rendering and postprocessing. During the rendering process a modified subpixel mask containing coverage information is generated for each pixel. These masks are used by the postprocess to determine the ratio by which pixel colors are mixed from the original pixel color and the colors of the adjacent pixels.

The anti-aliasing method was implemented based on the PHIGS renderer Da*Render*. An outline of a hardware realization is presented. Since the hardware realization is able to generate subpixel masks in parallel with the pixel colors, the system performance is reduced only by the need of additionally processing edge pixels. The percentage of the edge pixels is relevant only if lots of small triangles are processed, a situation generally avoided in real time systems. If a rendering pipeline with at least one extra framebuffer is used, postprocessing can be executed in parallel to the image generation, i.e. it does not lead to extra computation time. Therefore the method presented makes up a good compromise between image quality, system performance and hardware effort.

Kapitel 1

Einführung

1.1 Motivation

Aufgrund der explosiven Leistungssteigerung von Prozessoren und VLSI-Bausteinen gewinnen Echtzeitapplikationen von 3D-Graphik auch im PC-Bereich stark an Bedeutung. Typische Applikationen aus diesem Bereich sind Sichtsimulatoren und Virtual-Reality-Systeme. Ein starker Druck auf den Low-Cost-Graphikmarkt geht z.Zt. von multimedialen Entertainment-Programmen aus, die 3D-Graphik, Video und Audio vereinen. Um eine Echtzeitfähigkeit zu erreichen, wird für die Generierung der Graphik spezielle Rendering-Hardware, meistens proprietäre Polygonrenderer eingesetzt. Wegen der limitierten Leistungsfähigkeit solcher Systeme werden jedoch häufig Kompromisse zwischen Bildqualität und Echtzeitanforderungen eingegangen, die zuungunsten der Bildqualität gehen. Objekte werden hierbei sehr grob modelliert und die Bildauflösung wird niedrig gehalten, um die Zahl der zu generierenden und darzustellenden Bildpunkte zu reduzieren. Aliasing-Effekte, die bei der Generierung von Rasterbildern auftreten, sind aber bei geringer Bildschirmauflösung und animierten Bildsequenzen besonders störend. Trotzdem werden sie toleriert, da klassische Antialiasing-Algorithmen einen sehr hohen Rechenleistungs- und Speicherbedarf haben, der von den beschränkten Systemresourcen nicht gedeckt werden kann. Supersampling, ein gängiges Antialiasing-Verfahren, benötigt das n-fache der Renderingzeit und zusätzlich den bis zu n-fachen Speicherplatz im Bildspeicher im Vergleich zum Renderen ohne Antialiasing. Der Faktor n ist dabei der Oversampling-Faktor, der die virtuelle Vergrößerung der Auflösung angibt. Sinnvolle n liegen im Bereich 4 bis 256 wobei häufig 16 verwendet wird. Ein System, das gerade 16 Bilder pro Sekunde ohne Antialiasing erreichen kann, ist beim Einsatz von Supersampling mit n = 16 dann nur noch in der Lage etwa 1 Bild pro Sekunde zu generieren. Es hat seine Echtzeitfähigkeit somit verloren.

Ziel der Arbeit war es also, ein Antialiasing-Verfahren zu entwickeln, das bei möglichst geringem Hardware-Mehraufwand gegenüber dem normalen Rendering die Renderingleistung so wenig wie möglich beeinträchtigt. Eine weitere wesentlich Forderung war die Eignung zur Realisierung in Hardware. Die Idee ist dabei, das Antialiasing als Zusatzfunktionalität in den Entwurf eines Rendering-Chips zu integrieren.

1.2 Aliasing

1.2.1 Was ist Aliasing?

Im Gegensatz zu Vektor-Displays können Raster-Graphiksysteme graphische Primitive wie Linien und Polygone nur durch Approximation mit Hilfe von Bildpunkten (Pixeln) auf dem Raster darstellen. Dadurch entstehen die bekannten Treppeneffekte bei Linien und an den Kanten von Polygonen. Diese durch die Diskretisierung entstehenden Artefakte werden in der Signalverarbeitungs-Theorie ‘Aliases’ genannt. Sie treten auf, wenn eine kontinuierliche Funktion durch diskrete Abtastwerte (Samples) approximiert wird.

Die Abbildungen 1.1 und 1.2 zeigen Beispiele von geometrischen Primitiven und den Ergebnissen der Rasterung mit dem Bresenham-Algorithmus [Bre65] bzw. einem Pointsampling-Verfahren.

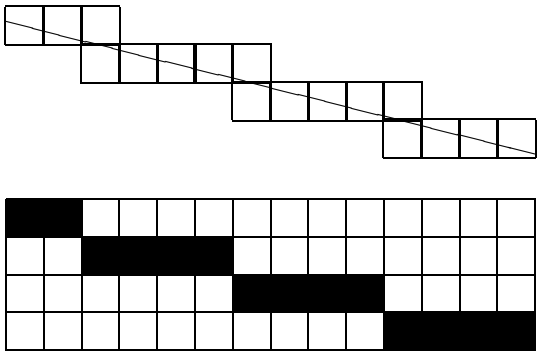


Abbildung 1.1: Linie und Ergebnis der Rasterung mit Hilfe des Bresenham-Algorithmus. Die Linie erscheint treppenartig.

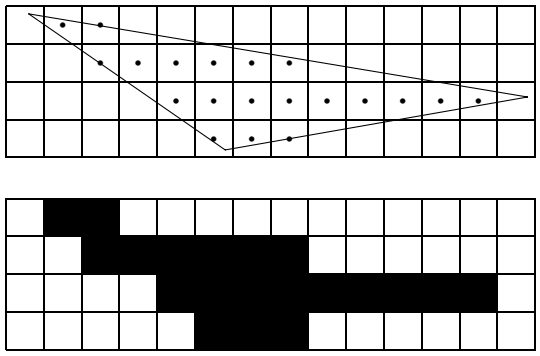


Abbildung 1.2: Dreieck und Ergebnis der Rasterung mit Hilfe eines Pointsampling-Algorithmus. Da beim Pointsampling die Farbe jedes Pixels durch ein einziges Sample in der Pixelmitte bestimmt wird, erscheint das Dreieck ‘zackig’, die ursprünglichen Konturen sind kaum noch zu erkennen.

Abbildung 1.3 zeigt ein weites typisches Aliasing-Phänomen: Schmale Rechtecke im Originalbild erscheinen im gerasterten Bild als Rechtecke einer anderen Breite.

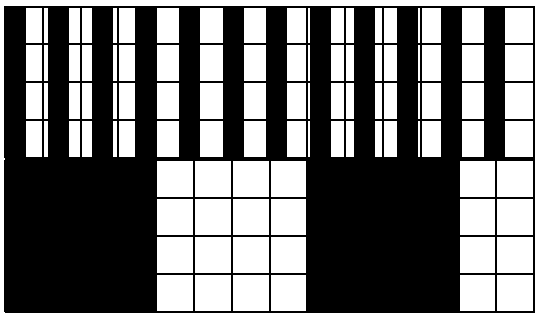


Abbildung 1.3: Ein Beispiel für Aliasing-Effekte an hochfrequenten Bildinhalten: Links das ursprüngliche Bild eines ‘Lattenzauns’, rechts das Ergebnis des Pointsamplings. Da die Frequenz des Musters ähnlich groß ist wie die Samplingfrequenz, fallen immer gleich ganze Serien von Samples auf bzw. zwischen die Latten. Im gerasterten Bild entsteht der Eindruck von wenigen, breiten Streifen.

Das in der Abbildung gezeigte Phänomen wird in [FvDFH90, Seite 627] wie folgt charakterisiert.

> “The phenomenon of high frequencies masking as low frequencies in the reconstructed signal is known as aliasing.”

Gute Beschreibungen des Aliasing-Problems finden sich unter anderem in [ES88, Seiten 335–346] und [FvDFH90, Seiten 14, 617–647].

1.2.2 Typen von Aliases in gerenderten Bildern

Nach Crow [Cro77] gibt es drei Klassen von Aliasing-Problemen:

Die äußeren Kanten eines Objektes
Hier treten die im vorhergehenden Abschnitt gezeigten Effekte auf: Kanten wirken ‘zackig’ und zeigen treppenartige Strukturen. (Siehe Abbildungen 1.1 – 1.2).

Sehr kleine Objekte
Bei Pointsampling-basierten Renderingsystemen — also auch bei Varianten des Supersamplings — verschwinden Objektteile oder sogar ganze Objekte, wenn sie keine Samplepoints überdecken. Abbildung 1.4 zeigt Beispiele für Objekte die im gerasterten Bild unsichtbar bleiben, weil sie keine Pixelmittelpunkte überdecken.

Die in den Abbildungen 1.5 und 1.6 dargestellten Effekte fallen besonders ins Auge, wenn es sich bei dem schmalen Polygon um ein isoliertes Objekte wie beispielsweise einen Fahnenmast handelt. Werden Bilder mit solchen Objekten animiert, kommt es zu sichtbaren Flackereffekten, da Objektteile abwechselnd sichtbar und unsichtbar werden.

Die hier beschriebenen Effekte treten in abgeschwächter Form im Bereich der Ecken beliebiger Dreiecke auf, da die Dreiecke dort zwangsläufig schmal sind.

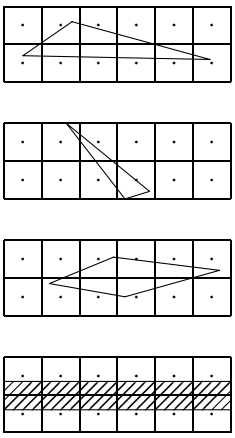


Abbildung 1.4: Objekte, die so klein oder schmal sind, daß sie keinen einzigen Samplepoint überdecken, sind im gerasterten Bild unsichtbar. Wie das schraffierte Rechteck zeigt, können diese unsichtbaren Objekte bei beliebiger Länge fast ein Pixel breit werden.

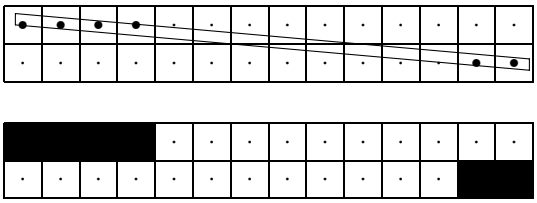


Abbildung 1.5: Das schmale, nahezu waagerecht liegende Rechteck überdeckt an manchen Stellen einige Samplepoints, an anderen Stellen nicht. Das gerasterte Bild weist deshalb Löcher auf, ein längeres Rechteck dieser Art erscheint als nicht zusammenhängend.

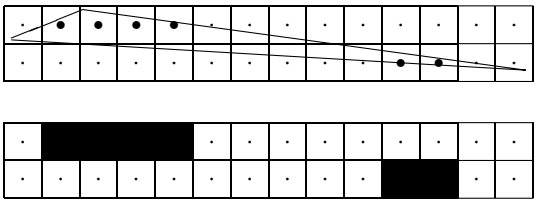


Abbildung 1.6: Bei schmalen Dreiecken tritt derselbe Effekt auf. Auf dem gerasterten Bild sind manche Teile des Dreiecks unsichtbar, da in diesem Bereich keine Samplepoints überdeckt wurden.

Flächen mit komplizierter Oberfläche

Werden Objekte mit hochfrequenten Texturen überzogen, kann es zum Entstehen neuer Muster kommen, die nicht Bestandteil des ursprünglichen Bildes waren. Abbildung 1.3 zeigte bereits ein Beispiel, das der Abbildung 1.7 habe ich [FvDFH90, Seite 828] entnommen.

Abbildung 1.7: Aliasing durch hochfrequente Texturen. Durch die perspektivische Projektion werden die Felder des Schachbrettmusters beliebig klein. Das linke Bild zeigt das Ergebnis eines Pointsampling-Verfahrens. Das rechte Bild wurde mit Hilfe des Mip-Map-Verfahrens erstellt, das speziell für die Vermeidung der Aliasing-Effekte an hochfrequenten Texturen entwickelt wurde [Wil83], [FvDFH90, Seite 826 ff].

1.2.3 Vermeidung von Aliases

Alle Ansätze zur Vermeidung von Aliases, dem sog. Antialiasing, basieren darauf, daß ein Pixel nicht als einzelnes Sample, sondern vielmehr als etwas Flächenhaftes begriffen wird. Die Farbe des Pixels wird dann nicht mehr durch das einzelne Sample, sondern durch die gemittelte Farbe der gesamten Pixelfläche bestimmt. Die verschiedenen Verfahren unterscheiden sich in erster Linie in der Berechnung bzw. Approximation des dafür benötigten Integrals der Farbe über die Pixelfläche. Beim Antialiasing von gerenderten Polygonen gibt es folgende zwei prinzipiell unterschiedliche Ansätze:

Areasampling

Beim Areasampling wird die sichtbare Überdeckung der Pixel durch die einzelnen Polygone auf analytische Weise berechnet. Aus Sicht der Signalverarbeitung leitet sich das Areasampling wie folgt her: Nach dem Abtasttheorem von Shannon [Jer77], [OS75] läßt sich ein Signal, in diesem Fall das Bild, korrekt reproduzieren, wenn eine Abtastrate verwendet wird, die mindestens doppelt so groß wie die größte Frequenz im Bild ist. Um Artefakte zu vermeiden wird das Bild deshalb vor der Abtastung Tiefpass-gefiltert. Bei dieser Filterung werden alle Frequenzen entfernt, die größer als die Hälfte der Abtastrate sind. Das gefilterte Bild erfüllt dann die Forderung von Shannon und kann durch die Abtastung korrekt wiedergegeben werden. In [FvDFH90, Seiten 623–642] findet sich eine Beschreibung, wie die bei der Filterung benötigte Konvolution von Signal und Filterfunktion mit Hilfe von Fouriertransformationen berechnet werden kann. In [Cro77] wird die Bedeutung der Filterung näher erklärt.

Supersampling

Beim Supersampling wird die sichtbare Überdeckung der Pixel durch die einzelnen Polygone durch eine große Anzahl von Samples approximiert. Aus Sicht der Signalverarbei-

tung leitet sich das Supersampling aus der Erhöhung der Abtastfrequenz her. Je höher die Abtastfrequenz ist, desto mehr Bilddetails können rekonstruiert werden. Enthält das Bild jedoch Unstetigkeiten, wie etwa die Kante eines Dreiecks, so ist keine endliche Abtastfunktion ausreichend. Oder, mit den Worten von Edwin Catmull [Cat78] ausgedrückt:

> "Point sampling of an unfiltered object is never correct at any resolution."

Beide Ansätze werden in Kapitel 2.1 zusammen mit anderen, daraus hervorgegangenen Verfahren, kurz vorgestellt.

1.3 Zielsetzung

Das im Rahmen dieser Diplomarbeit zu entwickelnde Antialiasing-Verfahren soll als Bestandteil eines Echtzeit-Renderingsystems Verwendung finden. Ein Verfahren muß folgende Bedingungen erfüllen, um als geeignet bezeichnet werden zu können:

Geschwindigkeit
: Um den Echtzeitanforderungen zu genügen, muß in jedem Takt ein Pixel fertiggestellt werden. Dazu ist es erforderlich, daß die Architektur des Verfahrens einer Hardware-Realisierung entgegenkommt. Um von Konzepten wie Pipelining und Caching profitieren zu können, darf die zugrundeliegende Speicherstruktur nicht zu aufwendig sein.

Bildqualität
: Die Anwendung des Verfahrens sollte zu einer möglichst großen Qualitätsverbesserung der erzeugten Bilder führen.

Kostengünstige Realisierbarkeit
: Das Antialiasing-Verfahren sollte zusammen mit dem Dreiecks-Renderer auf einem Chip realisierbar sein. Darüberhinaus sollte das Verfahren mit möglichst wenig zusätzlichem Speicher auskommen.

Kapitel 2

Übersicht über einige ausgewählte Antialiasing-Verfahren

2.1 Areasampling

Beim Areasampling, das auf Edwin Catmull [Cat78] zurückgeht, wird die Aufteilung der sichtbaren Anteile eines Pixel auf die einzelnen Polygone auf analytische Weise bestimmt.

Stark vereinfacht beschrieben geht das Verfahren wie folgt vor: Nachdem alle Polygone nach ihrem höchsten Y-Wert sortiert wurden, wird das gesamte Bild nach einzelnen Scanlines durchgegangen. Alle Polygone, die die gerade bearbeitete Scanline berühren, werden in pixelgroße Abschnitte zerteilt und in die 'Buckets' ihrer jeweiligen x-Koordinate einsortiert. Anschließend werden nacheinander die Listen aller x-Koordinaten durchgegangen, wobei mit Hilfe des Hidden-Surface-Algorithmus von Sutherland [SH73][1] die jeweils sichtbaren Fragmente bestimmt werden. Die Farben dieser Fragmente werden gewichtet aufsummiert und ergeben die endgültige Pixelfarbe.

Die auf dem Areasampling aufbauenden Verfahren arbeiten im Gegensatz zu Supersampling-basierten Verfahren exakt — jedem noch so kleinen Polygonfragment wird Rechnung getragen. Andererseits sind die Areasampling-basierten Verfahren extrem rechenintensiv. Besonders bei komplexen Szenen mit Objekten, die aus sehr vielen kleinen Teilen zusammmengesetzt werden, steigt der Rechenaufwand extrem an, da die zu verarbeitenden Listen in den 'Buckets' länger werden.

Varianten des Areasamplings sind in [Cro77] und [AW85] beschrieben. Bei letzterem handelt es sich um eine Weiterentwicklung, deren Ziel es ist, die Überdeckungsrechnung durch die Verwendung einer endlichen Zahl vorgefertigter Fragmente zu vereinfachen. Die Mehrzahl der Fälle kann dann mit geringem Aufwand bewältigt werden, der volle Rechenaufwand fällt nur bei einer kleinen Anzahl komplizierter Fälle an.

[1] Der Algorithmus von Sutherland bildet die Basis für das bekannte Verfahren von Weiler und Atherton [WA77]

Das Areasampling in der in diesem Abschnitt besprochenen Form eignet sich aufgrund seines hohen Rechenaufwandes nicht für den Einsatz in einem Echtzeit-Renderingsystem. Darüber hinaus sind die benötigten Rechnungen und Datenstrukturen recht komplex, so daß sich das Verfahren generell für eine Hardware-Realisierung nicht besonders anbietet. Die im vorherigen Kapitel aufgestellten Ziele werden vom Areasampling also nicht erfüllt.

2.2 A-Buffer

Das A-Buffer-Verfahren von Loren Carpenter [Car84] ist die diskrete Form des Areasamplings. Gegenüber dem Verfahren von Catmull wird eine Beschleunigung erzielt, indem die Flächenanteile der Pixel durch Subpixelmasken repräsentiert werden. Diese lassen sich durch einfache boolesche Operationen verknüpfen, so daß sich die Anwendung des Hidden-Surface-Algorithmus von Sutherland erübrigt. Um eine zusätzliche Beschleunigung zu erreichen, wird das Bild nicht scanlineweise sondern polygonweise bearbeitet. Wegen dieser Vorgehensweise wird ein Framebuffer benötigt, dessen Einträge bei Bedarf zu verketteten Listen erweitert werden können. In diesen Listen werden die zu einem Pixel gehörenden Polygonfragmente gespeichert. Der Speicherbedarf des A-Buffers paßt sich somit dynamisch der Komplexität der Szene an. Erst nachdem alle Polygone gerendert wurden, werden die zu den Pixeln gehörigen Listen traversiert. Mit Hilfe von darin abgespeicherten Informationen wie z.B. Z-Wert im Zentrum etc. lassen sich dabei die sichtbaren Flächenanteile der Polygonfragmente und damit die endgültigen Pixelfarben berechnen.

Vom A-Buffer erzeugte Artefakte, die auf eine zu ungenaue Analyse der Geometrie innerhalb des Pixels zurückzuführen sind, können mit dem in [SS93] vorgestellten Ansatz vermieden werden. Durch das zusätzliche Speichern der Werte von $\frac{\Delta z}{\Delta x}$ und $\frac{\Delta z}{\Delta y}$ im Pixelinneren könnnen die Verdeckungsberechnungen zwischen den Fragmenten exakt durchgeführt werden.

Wie Carpenter in seinem Artikel betont, wurde der A-Buffer für eine Software-Implementierung entwickelt. Die Verwendung von verketteten Listen und komplexen Fallunterscheidungen wirkt sich dabei nicht so gravierend aus wie in Hardware, wo das Traversieren von verketteten Listen zu erheblichen Geschwindigkeitsnachteilen führt. Durch die entstehenden Fallunterscheidungen kann das Verfahren von Pipeline- und Cache-Konzepten nicht mehr voll profitieren. Der A-Buffer erfüllt damit das im letzten Kapitel formulierte Ziel nach guter Hardwarerealisierbarkeit nicht.

Das im Rahmen dieser Diplomarbeit entwickelte Verfahren läßt sich als Weiterentwicklung des A-Buffers verstehen, bei dem die verketteten Listen zugunsten einer verbesserten Hardwarerealisierbarkeit aufgegeben werden.

2.3 Supersampling

Die Aufteilung der sichtbaren Fläche eines Pixels auf die Polygone, die beim Areasampling exakt berechnet wurde, wird beim Supersampling durch eine große Menge von Samples pro Pixel approximiert. Übliche Werte für die Anzahl der Samples sind dabei 2×2 bis 8×8 Samples je Pixel, d.h. die Farbe eines endgültigen Pixels wird aus den Farben von 4 bis 64 Pixeln eines intermediären Bildes zusammengemischt.

In seiner ursprünglichen Form ist Supersampling besonders leicht zu implementieren. Zunächst wird das Bild, zum Beispiel mit Hilfe eines Z-Buffers, in einen Zwischenspeicher gerendert. Die Auflösung des Bildes im Zwischenspeicher übersteigt dabei jedoch die für das endgültige Bild gewählte Auflösung um den entsprechenden Oversampling-Faktor n sowohl horizontal als auch vertikal. In einem Nachbearbeitungs-Durchgang werden dann die Farbeinträge von je $n \times n$ Pixeln addiert und die Summe durch n^2 geteilt.

Über die im folgenden besprochenen Weiterentwicklungen des Supersamplings hinaus existiert eine Vielzahl an Variationen. Die Mittelung im Nachbearbeitungs-Durchgang, die oben mit einem sog. Boxfilter durchgeführt wurde, kann mit einem Gaußfilter oder einer anderen Art von Gewichtung durchgeführt werden (siehe z.B. [HA90][Seite 313]). Die Anordnung der Samples, die bei dem Boxfilter nach einem quadratischen Schema erfolgte, kann nach dem Gesichtspunkt der Abstandsmaximierung der einzelnen Samples optimiert werden (Abbildung 2.1).

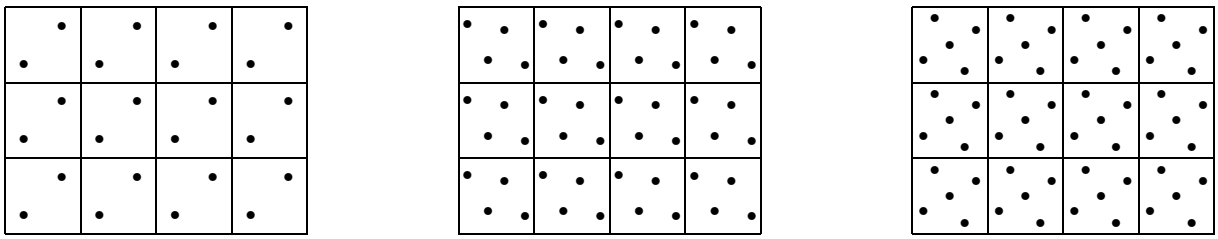


Abbildung 2.1: Plazierung der Samples nach dem Kriterium der Abstandsmaximierung. Die Generierung der Sample-Koordinaten kann für beliebige Samplingraten automatisch erfolgen (siehe z.B. [HA90, Seite 312]).

Weiterhin ist es möglich, die Samples zufällig zu plazieren, um verbleibende sichtbare Artefakte durch Rauschen zu ersetzen (stochastisches Supersampling siehe z.B. [Bar91]).

Die Hauptnachteile des Supersamplings sind:

Speicherbedarf
: Es benötigt die n^2-fache Menge an Speicher. Im nächsten Abschnitt *'Accumulation-Buffer'* werden wir sehen, wie diese Aufblähung des Speicherbedarfs vermieden werden kann.

Rechenaufwand
: Der Rechenaufwand nimmt um den Faktor n^2 zu.

Darüber hinaus hat das Supersampling, wie alle mit diskreten Samples arbeitenden Verfahren, den Nachteil, daß es nicht gewährleisten kann, daß das Ergebnisbild tatsächlich frei von Artefakten ist. Für jede Samplingrate ist es möglich ein Bild zu konstruieren, bei dem die Frequenzen im Bild zu hoch sind, um noch korrekt wiedergegeben zu werden. Verwendet das Renderingsystem eine horizontale Bildauflösung vom x und dazu ein $n \times n$ Supersampling, dann wird ein Bild, das lediglich ein gleichmäßiges Muster auf $x * n$ senkrechten schwarzen und weißen Streifen zeigt entweder völlig schwarz oder völlig weiß erscheinen. Um Supersampling wirklich zielgerichtet einsetzen zu können, wäre es an sich notwendig, die Größe der maximalen Frequenz im Bild zu bestimmen. Daß dieser Ansatz bei gerenderten Bildern nicht praktikabel ist, wird offensichtlich, wenn man versucht, die Größe des schmalsten sichtbaren Zwischenraumes zwischen Polygonen im Bild zu bestimmen. Darüberhinaus weisen gerenderte Bilder, wie bereits erwähnt, im allgemeinen sprunghafte Intensitätsänderungen an den Kanten von Polygonen auf. Das führt zu einer unendlich großen Frequenz, wie man sich an der Fouriertransformierten eines Rechtecksignals verdeutlicht. Aufgrund dieser Tatsache kann es keine Oversamplingrate geben, von der man garantieren kann, das das Ergebnisbild frei von Artefakten sein wird.

Wegen der Einfachheit der Implementierung sind Varianten des Supersamplings stark verbreitet. Das Hauptproblem des Supersamplings liegt in dem hohen Rechenaufwand, wenn ‘gute’ Ergebnisse benötigt werden. Man halte sich vor Augen, daß 4×4 Oversampling bereits eine Versechzehnfachung des Aufwandes im Verhältnis zum ungefilterten Bild bedeutet (In [Cro81] befinden sich Tabellen mit Berechnungszeiten einer Szene bei unterschiedlichen Samplingraten).

Supersampling an sich ist sehr gut in Hardware zu realisieren, da die vorkommenden Operationen einfach und die zugrundeliegenden Datenstrukturen simpel sind. Wegen des vervielfachten Rechenaufwandes ist es jedoch für ein Echtzeitsystem im allgemeinen zu langsam.

2.4 Accumulationbuffer

Der Accumulationbuffer, vorgestellt in [HA90], ist eine Variante des Supersamplings, bei der die Aufblähung des Speicherbedarfs vermieden wird. Verwendet wird ein Framebuffer normaler Größe, dessen Einträge lediglich eine etwas höhere Genauigkeit haben. Bei $n \times n$ Oversampling erhält jeder Eintrag $\lceil 2 \log_2 n \rceil$ zusätzliche Bits. Beim Accumulationbuffer kann das Bild nicht mehr in einem Durchgang gerendert werden. Das Rendern eines Bildes erfordert stattdessen n^2 Render-Durchgänge, zwischen denen das dem Bild zugrundeliegende Koordinatensystem jeweils im Subpixelbereich verschoben wird. Die Verschiebung wird dabei so vorgenommen, daß die Pixelmittelpunkte des einfachen Render-Durchgangs jeweils auf einem anderen Sample des entsprechenden Supersampling-Verfahrens zu liegen kommen. Die in jedem Durchgang erzeugten Farbwerte für ein Pixel werden dabei sukzessive akkumuliert, woraus sich der Name des Verfahrens herleitet. Die endgültige Farbe eines Pixels ergibt sich aus der Division des akkumulierten Farbwertes durch n^2, was sich durch eine Schiebeoperation bewerkstelligen läßt, wenn n eine Zweierpotenz ist.

Auf dem Accumulationbuffer aufbauende Verfahren sind extrem flexibel. Haeberli und Akeley

selbst führen in [HA90] vor, wie Effekte wie Depth-of-field und Motion-blur einfach realisiert werden können. Mammen [Mam89] verwendet ebenfalls einen Accumulationbuffer, um transparente Objekte zu rendern.

Durch den Ersatz des übergroßen Framebuffers durch den Accumulationbuffer wird zwar viel Speicher gespart, die Rechenzeit vermindert sich jedoch nicht. Im Gegenteil, das mehrfache Füllen der Rendering-Pipeline mit den Polygondaten führt sogar zu einem leichten Anstieg der Rechenzeit. Aufgrund der im vorhergehenden Kapitel geforderten Geschwindigkeit ist der Accumulationbuffer für unsere Anforderungen damit nicht geeignet.[2]

2.5 Approximationbuffer

Der Ansatz von Lau [Lau94] geht in eine ähnliche Richtung wie das im Rahmen dieser Diplomarbeit entwickelte Verfahren. Laus Verfahren basiert auf dem A-Buffer von Carpenter, beschränkt jedoch die Anzahl der zwischengespeicherten Fragmente auf maximal zwei. Durch diese Einschränkung kann auf die verketteten Listen des A-Buffers verzichtet werden, so daß ein Framebuffer konstanter Größe mit der Struktur eines Arrays ausreichend ist. Dadurch wird eine effiziente Hardware-Realisierung ermöglicht.

Wie Lau in seiner Arbeit betont, wurde das Verfahren für Bilder mit wenigen großen Polygonen entwickelt, wie sie etwa im Bereich der virtuellen Realität auftreten. Da der Anteil der Pixel, deren Bearbeitung mehr als zwei Einträge erfordert, dabei unter 0.8% liege, sei die Qualität der Ergebnisbilder hinreichend gut. Leider enthält der Vorabdruck noch keine Demobilder, die die tatsächlichen Grenzen des Verfahrens aufzeigen. Um einen Vergleich zwischen dem Verfahren von Lau und dem in dieser Arbeit entwickelten Verfahren zu erhalten, wäre es interessant, zu sehen, wie beispielsweise die im Anhang dieser Arbeit abgebildete Kugel mit Laus Verfahren gerendert wird.

Der Approximationbuffer benötigt ungefähr doppelt so viel Speicher wie das im Rahmen dieser Diplomarbeit entwickelte Verfahren und dürfte, in Hardware realisiert, eine ähnliche Geschwindigkeit erreichen.

2.6 Exakte Überdeckung

Bei dem von Andreas Schilling in [Sch91] vorgestellten Ansatz zur Berechnung der exakten Überdeckung handelt es sich um eine Modifikation des A-Buffers und verwandter Verfahren.

Die von Verfahren ohne exakte Überdeckung erzeugten Subpixelmasken an nahezu horizontalen (vertikalen) Kanten nutzen nur einen Bruchteil der möglichen Auflösung, da alle horizontal (vertikal) benachbarten Subpixel praktisch gleichzeitig gesetzt werden. Abbildung 2.2 zeigt

[2] Accumulationbuffer-Systeme lassen sich recht gut parallelisieren. Ein Einsatz in einem Echtzeitsystem ist damit durchaus möglich, allerdings vervielfachen sich die Kosten für die Realisierung entsprechend.

als Beispiel eine Kante mit der Steigung $\frac{1}{16}$. Die Kante ist genau so plaziert, daß immer vier Subpixel gleichzeitig aktiv werden. Es kommt daher zu sichtbaren Intensitätssprüngen an der Kante.

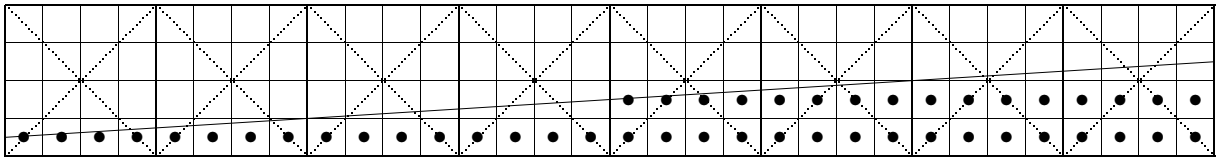


Abbildung 2.2: Sprunghafte Aktivierung von je 4 Subpixeln beim A-Buffer: Welche Subpixel gesetzt werden, wird durch die Überdeckung der Subpixel-Mittelpunkte durch die Halbebene bestimmt. Bei einer Kante mit sehr geringer Steigung führt das zu dem Effekt, daß immer gleich mehrere Subpixel auf einmal gesetzt werden. Die Farbintensität entlang der Kante wird dadurch ungleichmäßig erhöht. Im Bild tritt ein Intensitätssprung zwischen den beiden mittleren Pixeln auf.

Der Ansatz von Schilling besteht darin, zunächst die von der Halbebene überdeckte Fläche zu berechenen und erst anhand dieser *exakten* Fläche eine Subpixelmaske auszusuchen, die die entsprechende Überdeckung repräsentiert. Die Auswahl der Subpixelmaske kann effizient mit Hilfe einer Tabelle erfolgen. Abbildung 2.3 zeigt dasselbe Beispiel, diesmal wurden die Masken jedoch mit dem Algorithmus von Schilling bestimmt. Wie man sieht, nimmt die Intensität von Pixel zu Pixel linear zu.

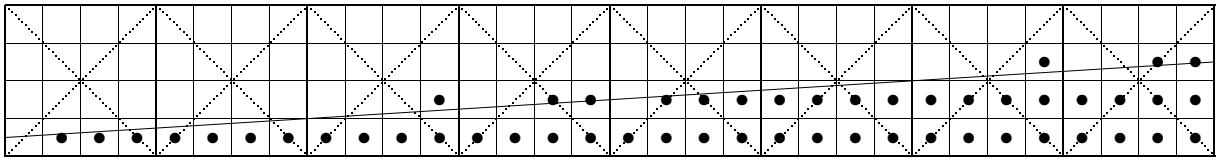


Abbildung 2.3: Gleichmäßige Aktivierung der Subpixel nach Schilling: Zunächst wird die tatsächliche Überdeckung berechnet, dann werden die Subpixelmasken anhand dieser Überdeckung bestimmt. Man sieht, daß dadurch Subpixel gesetzt wurden, deren Mittelpunkte nicht in der Halbebene liegen bzw. umgekehrt. Der Intensitätsverlauf entlang der Kante ist jetzt besonders gleichmäßig und wird nur durch die tatsächliche Auflösung der Subpixelmasken begrenzt.

Klassen zeigt in [Kla93] eine Weiterentwicklung von Schillings Ansatz, bei der durch geschickte Wahl der Subpixelmasken, die Qualität des Ergebnisbildes noch weiter verbessert werden kann, besonders wenn dieses für die Ausgabe auf einem Drucker bestimmt ist. Bei Klassens Ansatz handelt es sich um eine Verallgemeinerung des aus der Drucktechnik bekannten Half- bzw. Quarterbittings.

Der Ansatz Schillings findet auch bei dem von mir entwickelten Verfahren Anwendung.

Kapitel 3

Das Subpixel-Verfahren

3.1 Übersicht zur Funktionsweise

Das Subpixel-Verfahren arbeitet mit zwei separaten Durchgängen. Im ersten, dem Render-Durchgang wird das Bild per Pointsampling/Z-Buffering in normaler Auflösung gerendert. Dabei wird jedoch im Bereich von Dreieckskanten zusätzliche Information über die genaue Überdeckung der Pixel durch die Dreiecke gespeichert. Diese Information wird im zweiten, dem Nachbearbeitungs-Durchgang benutzt, um die endgültige Farbe der Pixel zu berechnen.

Die Grundidee, die das Subpixel-Verfahren wie auch das daraus hervorgegangenen Vier-Zeiger-Verfahren von anderen Verfahren unterscheidet, besteht in der *Ausnutzung räumlicher Koheränzen*: Statt für alle Dreiecksfragmente eines Pixels jeweils einen Farbeintrag zu erzeugen, werden die Farbeinträge der direkt angrenzenden Nachbarpixel mitverwendet. Jedes Pixel hat selbst nur genau einen Farbeintrag. Dieser beinhaltet die Farbe des Dreiecksfragments, das den Pixelmittelpunkt überdeckt (Pointsampling). Die Farben von Dreiecksfragmenten, die keinen Pixelmittelpunkt überdecken, werden, soweit möglich, anhand der Farben der direkten Nachbarn approximiert.

3.2 Aufbau der Subpixelmaske

Die zentrale Datenstruktur des Subpixel-Verfahrens ist die Subpixelmaske. Jedes Pixel im Framebuffer verfügt über einen solchen Eintrag.

Die Subpixelmaske repräsentiert eine quadratische Anordnung von $4*4$ Subpixeln, wobei jedem Subpixel genau ein Bit zugeordnet ist. Die Subpixelmaske kann damit als vorzeichenloses 16 Bit Speicherwort gespeichert werden. Die Zuordnung zwischen Subpixeln und Bitpositionen wird im Abschnitt *'Nachbearbeitung'* vorgestellt.

Jedes Subpixel verweist auf die in Abbildung 3.1 dargestellte Weise auf sein nächstes oder seine zwei nächsten Nachbarpixel.

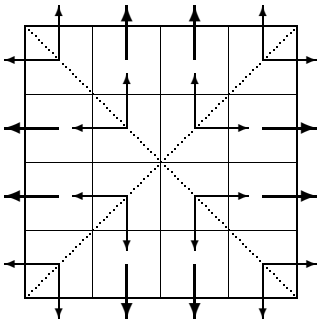


Abbildung 3.1: Jedes Subpixel verweist auf ein (dicker Pfeil) oder zwei (dünne Pfeile) Nachbarpixel. Ist ein Subpixel mit einem dicken Pfeil gesetzt, wird $\frac{1}{16}$ der Pixelfarbe von dem Nachbarpixel zugemischt, auf das der Pfeil zeigt. Ist ein Subpixel mit zwei dünnen Pfeilen gesetzt, wird von beiden Nachbarpixeln je $\frac{1}{32}$ zugemischt.

Der Wert des einem Subpixel zugeordneten Bits bestimmt dessen Farbe:

Wert	**Bedeutung**
0	Die Farbe des Subpixels ist gleich dem Farbeintrag im eigenen Pixel.
1	Die Farbe des Subpixels ist gleich der Farbe in dem Farbeintrag des Nachbarpixels, bzw. dem arithmetischen Mittel der Farbeinträge der beiden Nachbarn.

Jedes Subpixel bestimmt damit $\frac{1}{16}$ der Farbe des Pixels. Wenn das Subpixel auf zwei Nachbarpixel verweist, dann teilt sich dieser Anteil in zwei $\frac{1}{32}$ auf. Die Abbildungen 3.2 und 3.3 zeigen Beispiele.

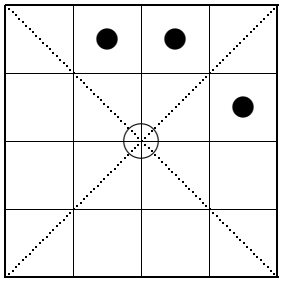


Abbildung 3.2: Beispiel: $\frac{1}{16}$ der Farbe des Pixels kommt vom rechten Nachbarn, $\frac{2}{16}$ vom oberen, der Rest vom Pixel selbst.

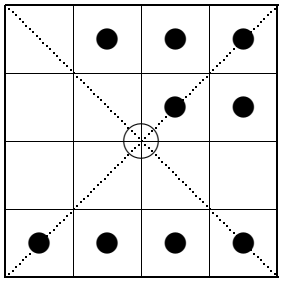


Abbildung 3.3: $\frac{6}{32}$ von oben, $\frac{5}{32}$ von rechts, $\frac{6}{32}$ von unten, und $\frac{1}{32}$ von links. Die restlichen $\frac{14}{32}$ vom Pixel selbst.

In Abbildung 3.4 ist ein Bildausschnitt mit den zugehörigen Subpixelmasken gezeigt. Jedes $4 * 4$-Schema repräsentiert ein Pixel mit seinen Subpixeln. Ein schwarzer Kreis • in einem Subpixel zeigt an, daß dieses Subpixel auf ‘1’ gesetzt ist, also die Farbe der entsprechenden Nachbarpixel repräsentiert. Pixel, die beim Rendern des dargestellten Dreiecks gesetzt wurden, sind so markiert, daß sich ein Kreis ∘ in ihrem Mittelpunkt befindet.

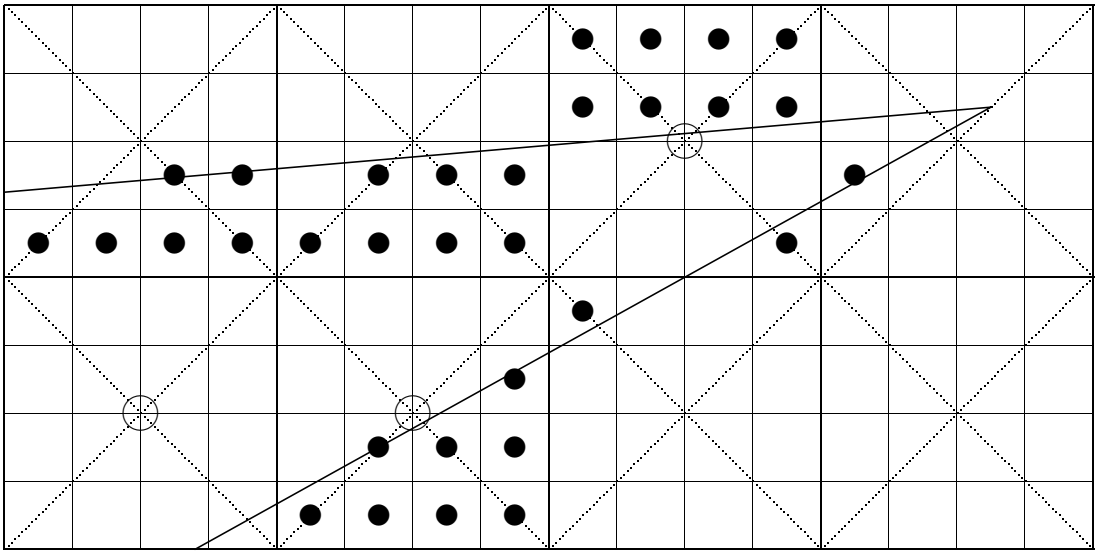


Abbildung 3.4: Beispiel: Ausschnitt eines Dreiecks mit den dazugehörigen Subpixelmasken. Die drei Pixel mit umkreistem Mittelpunkt wurden beim Rendern des Dreiecks gesetzt. An der zweiten Maske von links in der oberen Zeile erkennt man, daß das Verfahren zur Berechnung der exakten Überdeckung verwendet wurde (siehe Abschitt 2.6). Die Anzahl der gesetzten Subpixel in diesem Pixel repräsentiert die überdeckte Fläche, nicht die Anzahl der überdeckten Subpixel-Mittelpunkte.

3.3 Ablauf

In den folgenden Diagrammen wird der Ablauf des Subpixel-Verfahrens schematisch dargestellt. Die Symbole in der ersten Spalte haben dabei folgende Bedeutung:

Symbol	Bedeutung
$+$	Der Punkt wird in einer der nächsten Boxen weiter untergliedert
$(\bullet)$	Eine detaillierte Beschreibung befindet sich im nächsten Abschnitt
$\bullet$	keine weitere Untergliederung

Es sei an dieser Stelle darauf hingewiesen, daß sich der hier dargestellte Ablauf an der Software-Version orientiert. Der Ablauf des Teils *'Span rendern'* fällt in der Hardware-Version deutlich einfacher aus, da bestimmte Optimierungs-Techniken der Software zugunsten des auf Parallelität ausgelegten Hardwareansatzes entfallen. Der entsprechende Ablauf der Hardware-Version ist im Kapitel *'Hardware-Realisierung des Vier-Zeiger-Verfahrens'* wiedergegeben.[1]

[1] Der Ablauf der Hardware-Version des Subpixel-Verfahrens unterscheidet sich nicht wesentlich von dem des Vier-Zeiger-Verfahrens.

Gesamtablauf

- • Speicher für Subpixelmasken und temporäre Arrays reservieren.
- (•) Vorberechnen der Tabellen für die Maskenberechnung (Siehe Abschnitt *'Berechnung der Subpixelmasken'* ab Seite 43)
- \+ Für alle Bilder: **Bild berechnen**
- • Speicher freigeben

Bild berechnen

- • Initialisieren des Subpixelmasken-Arrays
- \+ Für alle Dreiecke: **Dreieck rendern**
- \+ **Nachbearbeiten**

Dreieck rendern

- • Initialisieren der Dreiecksdaten und der Kantenstrukturen
- \+ Für alle Spans: **Span rendern**
- • Löschen verbleibender Einträge in temporären Arrays.

Span rendern

- • Pointsampling des Spans. Merken, welche Pixel gesetzt wurden.[2]
- (•) Falls der Span Eckpunkte des Dreiecks enthält: Erzeugen spezieller Masken für die Eckpunkte, Entfernen der Eckpunkte von den Kanten. (Siehe Abschnitt *'Spezielle Behandlung der Eckpixel'* ab Seite 32)
- (•) Für alle Kanten des Dreiecks, die den Span berühren: Erzeugen der Masken und Verknüpfen mit den Masken vorher gerenderter Kanten. (Siehe Abschnitt *'Verknüpfung von Masken'* ab Seite 28)
- (•) Zurückschreiben sichtbarer neuer Masken in den Framebuffer (Siehe Abschnitte *'Visibilität'* und *'Zurückschreiben neuer Masken'* ab Seite 36)
- • Zurücksetzen aller Subpixel im Span, die Verweise innerhalb des neuen Dreiecks repräsentieren. (Siehe Abschnitt *'Zurücksetzen 'interner' Subpixel'*, ab Seite 42)
- • Löschen temporärer Arrays

Nachbearbeiten

- • Entferne alle Subpixel, die aus dem Framebuffer heraus zeigen.
- (•) Für alle Pixel des Framebuffers: Mische die Farbe des Pixels anhand seiner Maske zusammen. Nutze dabei die Farbeinträge der Nachbbarpixel mit. (Siehe Abschnitt *'Nachbearbeitung'* ab Seite 22)

[2]Das eigentliche Rendern des Spans ist natürlich nicht Bestandteil des Antialiasings. Die Software-Implementierung des Verfahrens setzt auf dem Renderer Da*Render* auf, dessen Funktionsweise in [AH92] nachzulesen ist.

3.4 Details des Verfahrens

3.4.1 Notwendigkeit zweier separater Durchgänge

Das Linien-Antialiasing-Verfahren nach Gupta und Sproull [GS81] berechnet die für das Antialiasing benötigten Zwischenfarben bereits während des Renderns und schreibt die Farben sofort in den Bildspeicher. Durch diesen Ansatz kann es an Kreuzungen zweier Linien zu Artefakten kommen und zwar umso stärker, je spitzer der Winkel zwischen den beiden Linien ist. Da die Information über die Lage der zuerst gerenderten Linie bereits verloren ist, gibt es keine Möglichkeit herauszufinden, wie die neue zu der alten Linie in Bezug steht. Die Zwischenfarben können deshalb nicht sinnvoll aufeinander aufbauen. Handelt es sich um eine Darstellung, bei der ausschließlich gleichfarbige Linien vor schwarzem Hintergrund erzeugt werden, könnte man die Farben addieren, doch dadurch würde das Ergebnis zu intensiv, da Anteile addiert werden, die einander sonst verdecken würden. Überschreibt man die Farben, wird das Ergebnis nicht intensiv genug, da Anteile verloren gehen. Auch wenn die Maximumsbildung als Verknüpfungsoperation verwendet wird, ist das Ergebnis nicht intensiv genug. Das Problem vergrößert sich noch, wenn Linien und Hintergrund beliebige Farben und/oder sogar Strukturen aufweisen dürfen.

Was bei ausschließlich für Linien bestimmten Verfahren unter Umständen noch tragbar ist, ist für polygonbasierte Systeme nicht mehr akzeptabel. Es kommen nämlich zwei neue Problemdimensionen hinzu:

1. Die Sichtbarkeit von Polygonen muß berechnet werden (Hidden-Surface-Removal).
2. Das direkte Angrenzen von Polygonen über die Länge einer ganzen Kante tritt extrem häufig auf.

Als Beispiel zeigt Abbildung 3.5 zwei aneinandergrenzende Dreiecke und das bei ihrer Berechnung entstehende Artefakt.

Die Bildung dieser Artefakte wird durch die Aufspaltung des Verfahrens in zwei separate Durchgänge vermieden. Dadurch, daß erst nach Abschluß des Rendervorgangs gemischt wird, ist die zugemischte Farbe stets die Farbe des *am Ende* sichtbaren Nachbarn. Die Hintergrundfarbe kann nicht durchscheinen.

3.4.2 Nachbearbeitung

Am Ende des Render-Durchgangs unterscheidet sich der Inhalt der Farbeinträge nicht vom Ergebnis eines normales Pointsamplings. Erst jetzt, im Nachbearbeitungs-Durchgang, nutzen wir die während des Renderns in den Subpixelmasken gespeicherte Information, um die Pixelfarben zu mischen.

Die Nachbearbeitung läuft schematisch wie folgt ab

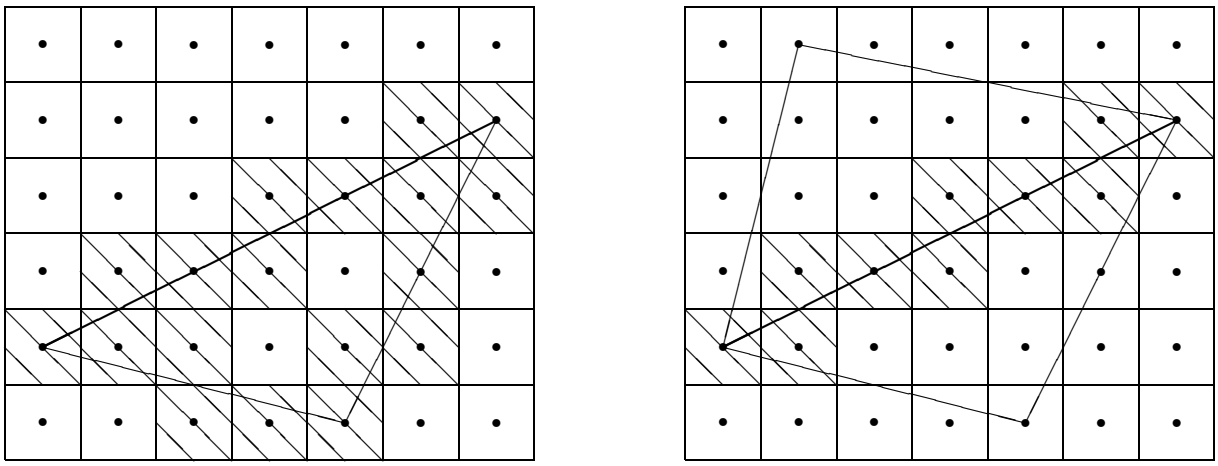


Abbildung 3.5: Problem bei Verfahren ohne getrennten Nachbearbeitungs-Durchgang. Beim Rendern des ersten Dreiecks werden bereits Farbe und Hintergrund vermischt (schraffierte Pixel links). Beim Rendern des zweiten Dreieck wird mit den bereits vermischten Farben weitergearbeitet. Da sich die Hintergrundfarbe in den schraffierten Pixeln nicht mehr von der Farbe des ersten Dreiecks trennen läßt, scheint sie an der Kante zwischen den angrenzenden Dreiecken hindurch. Das Ergebnis ist ein linienförmiges Artefakt in Hintergrundfarbe (schraffierte Pixel rechts).

```
for (x = 0; x < Spalten; x++)
  for (y = 0; y < Spalten; y++)
    if (Maske[x,y] == 0)
      Farbe_neu[x,y] = Farbe_alt[x,y];
    else
      Farbe_neu[x,y] = MischeFarben(Maske[x,y],Farbe_alt[x,y]
                                    Farbe_alt[x,y-1],Farbe_alt[x+1,y],
                                    Farbe_alt[x,y+1],Farbe_alt[x-1,y]);
```

Farbe_neu und Farbe_alt können dabei zu zwei verschiedenen Framebuffern gehören. Beim Farbmischen wird dann das Umkopieren in den anderen Buffer gleich miterledigt. Liegen Farbe_neu und Farbe_alt hingegen im selben Framebuffer, dann ist es notwendig, mindestens die letzte Zeile und das letzte Pixel in einem Zwischenspeicher zu halten. Dadurch wird sichergestellt, daß beim Mischen immer die ursprünglichen und nicht die bereits vermischten Farben verwendet werden. Unterlassen des Zwischenspeicherns, kann zum ‘Auslaufen’ der Farben nach rechts und unten führen, da Farbe von einem bereits bearbeiteten Pixel weitergegeben wird.

Beim Zusammenmischen der neuen Farbe ist es notwendig, gesetzte Subpixel der vier Mischrichtungen zu zählen, um das Mischungsverhältnis entsprechend zu bestimmen. Durch die in Abbildung 3.6 dargestellte spezielle Zuordnung zwischen Subpixelpositionen und Bitpositionen läßt sich das Zählen der Subpixel effizient unter Zuhilfenahme einer vorberechneten Tabelle durchführen.

Die Abbildungen 3.7 und 3.8 sollen den regelmäßigen Aufbau der Maske verdeutlichen. Die sechs Subpixel einer Richtung sind so angeordnet, daß die ‘halben’ Subpixel, also die mit zwei

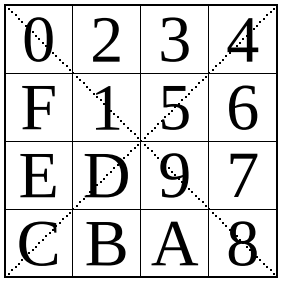


Abbildung 3.6: Anordnung der Subpixel im Speicherwort. Die hexadezimalen Ziffern in den Subpixeln geben deren Bit-Positionen innerhalb des Speicherwortes an. Subpixel, die auf das gleiche Nachbarpixel verweisen, liegen auch im Speicherwort benachbart (man stelle sich das Speicherwort als zyklisch geschlossen vor).

Nachbarn, jeweils außen liegen.

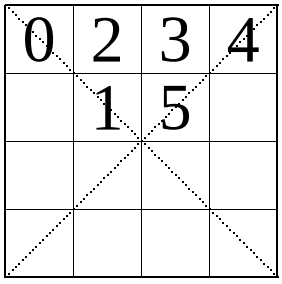


Abbildung 3.7: Die Bits der Subpixel, die auf das obere Nachbarpixel verweisen, liegen benachbart auf den Positionen 0–5.

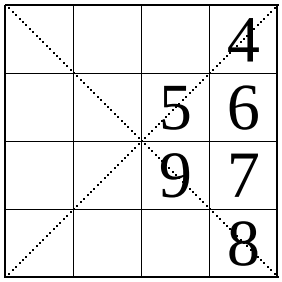


Abbildung 3.8: Die Bits der Subpixel, die auf das rechte Nachbarpixel verweisen, liegen benachbart auf den Positionen 4–9.

Wenn wir die jeweils sechs Bits, deren zugehörige Subpixel auf dasselbe Nachbarpixel verweisen, entsprechend verschieben, können wir mit ihnen die im folgenden wiedergegebene Tabelle adressieren. Wegen der Subpixel mit zwei Nachbarn liefert die Tabelle die Subpixelanzahl in *halben* Subpixeln, also in $\frac{1}{32}$ Pixeln zurück.

0, 1, 1, 2, 2, 3, 3, 4, 2, 3, 3, 4, 4, 5, 5, 6,
1, 2, 2, 3, 3, 4, 4, 5, 3, 4, 4, 5, 5, 6, 6, 7,
1, 2, 2, 3, 3, 4, 4, 5, 3, 4, 4, 5, 5, 6, 6, 7,
2, 3, 3, 4, 4, 5, 5, 6, 4, 5, 5, 6, 6, 7, 7, 8

Das Zählen der Subpixel gestaltet sich nun überaus einfach, wie das folgende Codefragment zeigt:

```
UpBits    = SubpixelZaehltabelle[ Mask & UP_BITS];
RightBits = SubpixelZaehltabelle[(Mask & RIGHT_BITS) << 4];
```

```
DownBits  = SubpixelZaehltabelle[(Mask & DOWN_BITS)  << 8];
LeftBits  = SubpixelZaehltabelle[(Mask & LEFT_BITS)  << 12
                                |(Mask & LEFT_BITS)  >> 4];
OwnBits   = 32 - UpBits - RightBits - DownBits - LeftBits;
```

In der Software-Implementierung des Subpixel-Verfahrens befindet sich der Nachbearbeitungs-Prozeß im Modul *XpAAPost.c*.

3.4.3 Traversierung der Kanten

In [Sch91, Seite 139] ist ein auf Pineda [Pin88] zurückgehendes Verfahren vorgestellt, mit dem die Distanz zwischen einem Punkt und einer Kante berechnet werden kann. Die Distanzfunktion, in diesem Fall Errorterm genannt, ist definiert als

$$\mathrm{E}(x, y) = (x - X) * de_x + (y - Y) * de_y$$

(X, Y) ist dabei ein beliebiger Punkt auf der Kante, de_x und de_y sind definiert als

$$de_x = \frac{\Delta Y}{|\Delta X| + |\Delta Y|}$$

$$de_y = \frac{-\Delta X}{|\Delta X| + |\Delta Y|}$$

Diese Distanzfunktion hat einige bemerkenswerte Eigenschaften:

1. Im Nenner der Definition von de_x und de_y wurde die sog. Manhatten-Distanz ($||_1$-Norm) anstelle der normalen, euklidischen Distanz ($||_2$-Norm) verwendet. Der Errorterm $\mathrm{E}(x, y)$ gibt deshalb nicht die euklidische Distanz zwischen dem Punkt (x, y) und der Kante an, sondern eine Art 'quadratische' Distanz. Abbildung 3.9 verdeutlicht diese Eigenschaft. In den beiden Diagrammen sind auf einem Pixelraster jeweils einige Geraden eingetragen, die dieselbe Distanz zum Mittelpunkt haben.

   Die Verwendung der $||_2$-Norm hat zwei Vorteile:

   (a) Man spart sich die Berechnung der Quadratwurzel, wie sie bei der euklidischen Distanz auftritt.

   (b) Die entstandene Distanzfunktion läßt sich einsetzen, um den Abstand einer Kante zu einem *quadratischen* Pixel zu berechnen.

   Es ergibt sich die einfache Formel, daß die durch E beschriebene Kante das Pixel bei (x,y) genau dann schneidet, wenn

   $$|\mathrm{E}(x, y)| < \frac{1}{2} \tag{3.1}$$

   Das sind genau die Pixel, die durch das Subpixel-Verfahren bearbeitet werden müssen.

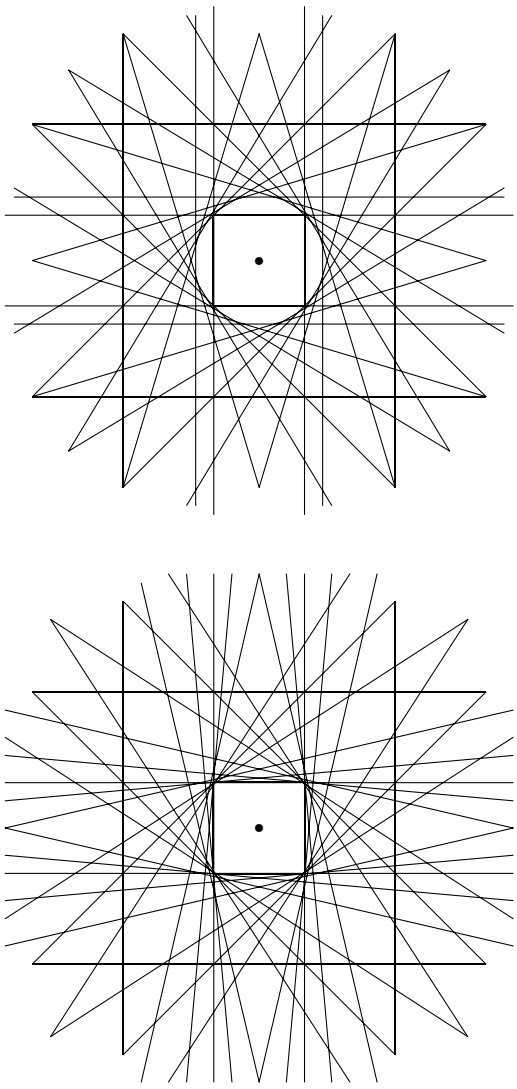


Abbildung 3.9: Zirkuläre und quadratische Distanz. Die Linien mit gleichem Abstand zum Mittelpunkt sind genau die Tangenten an einen Kreis mit Radius E im Fall der zirkulären Distanz. Bei der quadratischen Distanz ergeben sich Tangenten an ein Quadrat mit Kantenlänge $E\sqrt{2}$.

2. Wenn alle Kanten so gerichtet werden, daß sie das Dreiecksinnere im Uhrzeigersinn umlaufen, dann läßt sich die Zugehörigkeit eines Pixels zum Dreieck auf einfache Weise

testen.[3] Das Pixel gehört genau dann zum Dreieck, wenn die Werte von $\mathrm{E}(x, y)$ positiv sind,[4] und zwar für alle drei Kanten. Dieser Ansatz wird von Schilling in [Sch91] verwendet und könnte auch in einer Hardware-Realisierung des Subpixel-Verfahrens Verwendung finden.

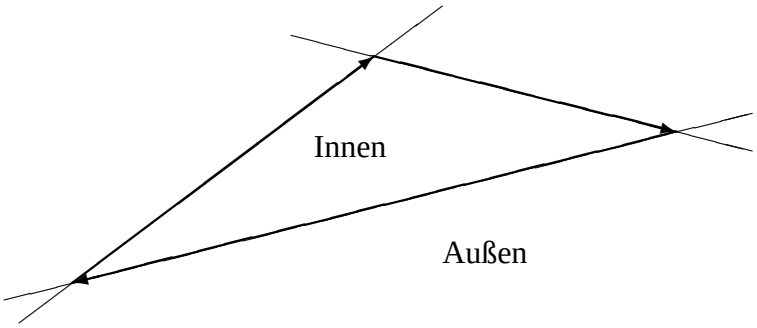


Abbildung 3.10: Bestimmung der Zugehörigkeit von Pixeln zum Dreieck. Die Kanten werden als Vektoren betrachtet, die das Dreieck im Uhrzeigersinn umlaufen. Das Innere des Dreiecks befindet sich dann immer rechts von allen drei Kante. Diese Eigenschaft läßt sich einfach berechnen. Ein Pixel liegt genau dann im Inneren des Dreiecks, wenn sein Errorterm unter allen Kanten positiv ist.

3. Aufgrund der Linearität von E lassen sich alle Werte entlang einer Kante inkrementell berechnen.

$$\mathrm{E}(x+1, y) = \mathrm{E}(x, y) + de_x \tag{3.2}$$
$$\mathrm{E}(x-1, y) = \mathrm{E}(x, y) - de_x \tag{3.3}$$
$$\mathrm{E}(x, y+1) = \mathrm{E}(x, y) + de_y \tag{3.4}$$
$$\mathrm{E}(x, y-1) = \mathrm{E}(x, y) - de_y \tag{3.5}$$

Multiplikationen und Divisionen werden also nur einmal und zwar bei der Initialisierung der Kante benötigt.

Die Funktion $\mathrm{E}(x, y)$ ist also eine geeignete Funktion, um die vom Subpixel-Verfahren benötigte Kantentraversierung durchzuführen. Die sich dabei ergebende Traversierungsordnung, die alle Kantenpixel besucht, entspricht der des Symmetrischen DDA (Digital Differential Analyzer)

[3] Bei dem für die Software-Implementierung verwendeten Da*Render* sind Kanten immer in Richtung aufsteigender Y-Werte gerichtet. Die Bestimmung der umzudrehenden Kante(n) gestaltet sich jedoch recht einfach. Die Variable n, die aus dem Vektorprodukt der beiden Kanten durch den obersten Punkt entsteht, gibt an, auf welcher Seite der Hauptkante sich der mittlere Punkt befindet.

$$n = d_{12}.x * d_{13}.y - d_{12}.y * d_{13}.x$$

Ist n negativ, liegt der mittlere Punkt links von der Hauptkante (Der Ursprung des Da*Render*-Koordinatensystems liegt in der linken oberen Ecke, Angaben wie rechts, links etc. sind deshalb gerade umgedreht). In diesem Fall muß die Orientierung der Hauptkante negiert werden. Ist n positiv, ändern wir die Orientierung der anderen beiden Kanten. Diese Berechnungen befinden sich im Quellcodemodul *xpsftria.c*.

[4] Ein Ansatz, Fälle mit $\mathrm{E}(x, y) = 0$ zu behandeln, kann in [AH92] nachgelesen werden

Algorithmus [ES88, Seiten 57–60]. Im Gegensatz zum bekannteren Bresenham-Algorithmus [Bre65][ES88, Seite 60–63] liegen dabei sowohl in Zeilen als auch in Spalten mehrere besuchte Pixel (Abb. 3.11).

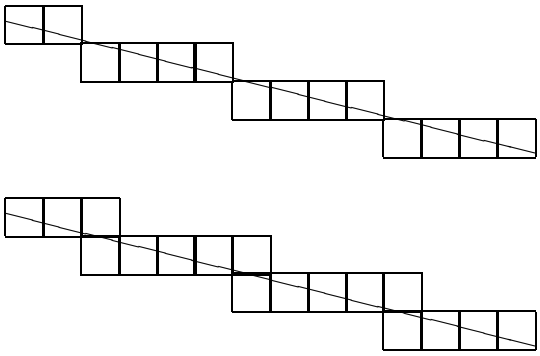


Abbildung 3.11: Traversierungsordnung von Bresenham- und symmetrischem DDA-Algorithmus. Der Bresenham Algorithmus (links) erzeugt eine Linie minimaler Dicke. Bei Linien mit einer Steigung unter 45^{o} liegt in jeder Spalte nur ein besuchtes Pixel, bei Linien mit einer Steigung über 45^{o} in jeder Zeile. Beim symmetrischen DDA-Algorithmus (rechts) entsteht ein durchgehender Pfad aus aneinander direkt angrenzenden Pixeln.

Die Gesamtzahl der zu bearbeitenden Pixel erhöht sich durch die Kantenbearbeitung. Der Anteil der Kantenpixel ist umso höher, je schmaler das bearbeitete Dreieck ist. Abbildung 3.12 zeigt ein Beispiel.

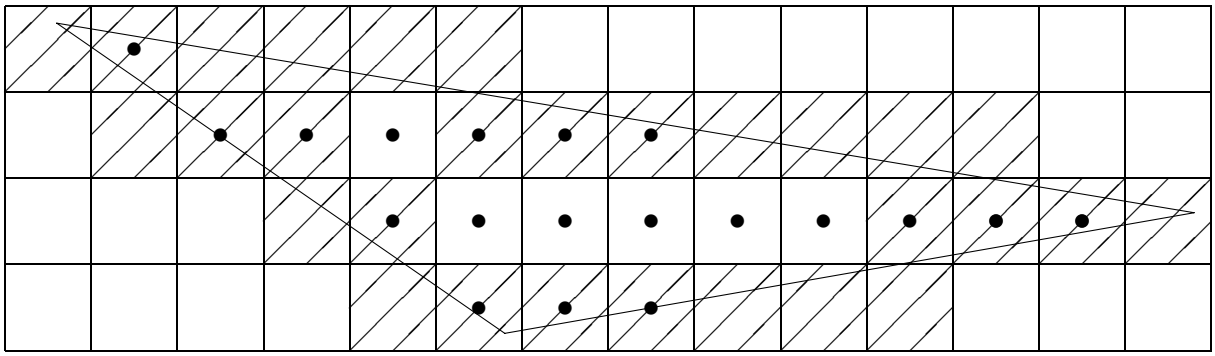


Abbildung 3.12: Bei kleinen oder schmalen Dreiecken werden u.U. mehr Pixel durch die Kantentraversierung besucht (///), als durch den eigentlichen Rendervorgang (•). Bei großen Dreiecken fällt die Anzahl der Kantenpixel dagegen nicht so stark ins Gewicht.

3.4.4 Verknüpfung von Masken

In der Software-Implementierung des Subpixel-Verfahrens erfolgt die Generierung der Subpixelmasken eines Spans kantenweise, nicht pixelweise. Als Konsequenz müssen die von den einzelnen Kanten generierten Masken in einem temporären Array von der Breite eines Spans aufbewahrt und bei Bedarf dort miteinander verknüpft werden. Erst wenn beide bzw. alle drei Kanten ihre Masken in das temporäre Array geschrieben haben, wird die Routine aufgerufen,

die die verknüpften neuen Masken zum Frame zurückschreibt (siehe Abschnitt *'Zurückschreiben neuer Masken'*).

In der Hardware-Version entfällt das temporäre Array, die Verknüpfung der neuen Masken wird sofort und auf Pixelbasis abgewickelt. Das Verknüpfen der Masken verläuft jedoch völlig analog zu der Software-Version.

In Bereichen, in denen das gerade gerenderte Dreieck schmal ist, kann es durchaus vorkommen, daß zwei oder gar alle drei Kanten ein Pixel berühren. Die Masken der einzelnen Kanten dürfen einander dabei nicht überschreiben, sondern müssen verknüpft werden. Zur einfacheren Ausdrucksweise hier zwei vereinfachende Definitionen:

Def: Schwarze Maske

Eine Maske heißt 'schwarz' in Bezug auf eine Kante, wenn der Mittelpunkt des zugehörigen Pixels im Inneren der durch die Kante beschriebenen Halbebene liegt. Die Subpixel einer schwarzen Maske zeigen also nach 'außen'.

Def: Rote Maske

Eine Maske ist genau dann rot, wenn sie nicht schwarz ist. Die Subpixel einer roten Maske zeigen also in die Halbebene der Kante hinein.

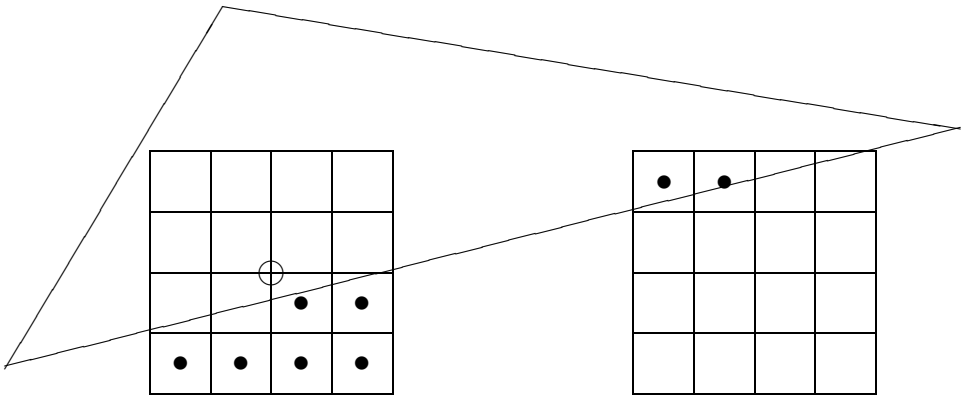


Abbildung 3.13: Die Maske links ist eine 'schwarze' Maske. Der Mittelpunkt des zugehörigen Pixels liegt innerhalb der Halbebene der betrachteten Kante. In diesem Fall, wie in der Mehrzahl der Fälle, liegt der Pixelmittelpunkt auch innerhalb des Dreiecks. Die rechte Maske ist eine 'rote' Maske. Der zugehörige Pixelmittelpunkt liegt außerhalb der Halbebene der Kante und damit auch außerhalb des Dreiecks.

Die Zugehörigkeit eines Pixels zu der Halbebene einer Kante bedeutet nicht zwangsläufig, daß das Pixel auch zum zum Dreieck gehört. Falls das Pixel in mehr als einer Kante liegt, ist es möglich, daß die Maske bei Betrachtung der einen Kante innen, bei Betrachtung einer anderen Kante aber außen liegt. Erst das Ergebnis der Verknüpfung aller beteiligten Masken sagt etwas über die Zugehörigkeit des Pixels zum Dreieck aus. Ist die endgültige Maske schwarz, dann liegt das Pixel im Dreieck, sonst nicht.

Einer Maske sieht man nicht an, ob sie 'schwarz' oder 'rot' ist, diese Eigenschaft ist nur implizit über den Errorterm der zugehörigen Kante definiert. Abbildung 3.14 zeigt zwei identische Masken, von denen eine schwarz, die andere rot ist.

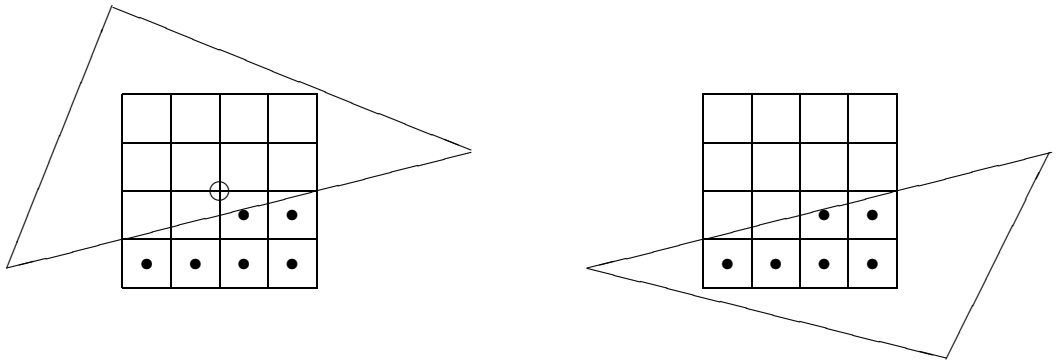


Abbildung 3.14: Rote und schwarze Masken lassen sich nicht durch ihren Inhalt unterscheiden. Die linke Maske ist schwarz, da das zugehörige Pixel in der Halbebene der betrachteten Kante liegt, die rechte Maske hingegen ist rot.

Wenn zwei Masken verknüpft werden, ergeben sich auf den ersten Blick die drei möglichen Kombinationen ‘beide schwarz’, ‘rot/schwarz’ und ‘beide rot’. Die Abbildungen 3.15–3.17 zeigen Beispiele für diese drei Fälle.

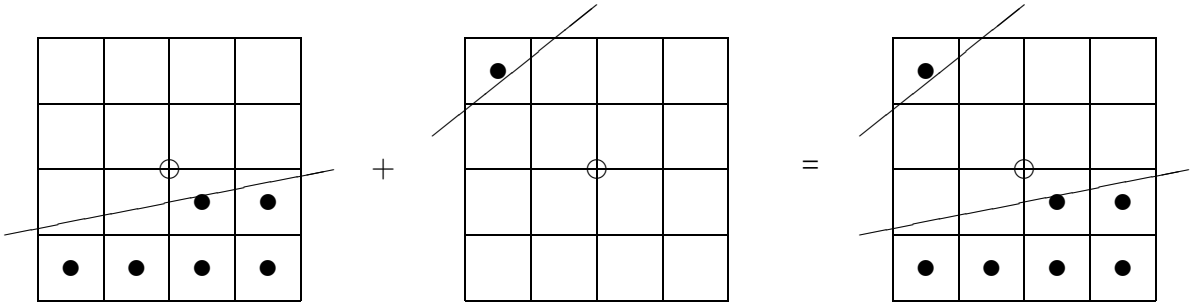


Abbildung 3.15: Verknüpfung zweier schwarzer Masken. Da der zugehörige Pixelmittelpunkt sowohl in der Halbebene der ersten, als auch in der Halbebene der zweiten Kante liegt, liegt er auch im Schnitt beider Halbebenen. Die Ergebnismaske ist deshalb ebenfalls schwarz.

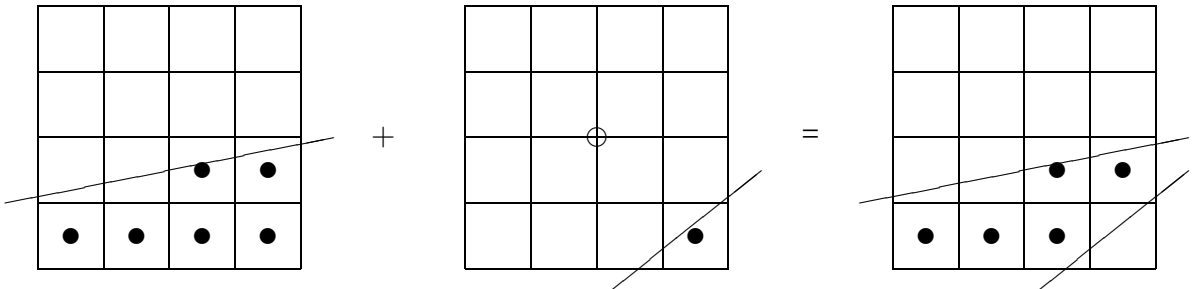


Abbildung 3.16: Verknüpfung einer roten mit einer schwarzen Maske. Der Mittelpunkt des zugehörigen Pixels liegt außerhalb der Halbebene der roten Maske, also auch außerhalb des Schnittes. Die Ergebnismaske ist deshalb rot.

Das Problem besteht nun darin, daß rote und schwarze Masken unterschiedliche Darstellungen verwenden: Subpixel, die die Farbe des neuen Dreiecks repräsentieren, werden in schwarzen

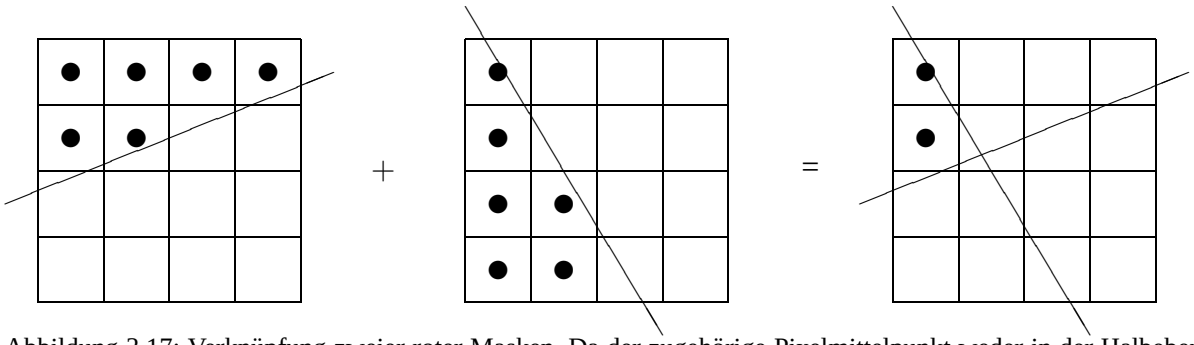


Abbildung 3.17: Verknüpfung zweier roter Masken. Da der zugehörige Pixelmittelpunkt weder in der Halbebene der ersten, noch in der Halbebene der zweiten Maske liegt, befindet er sich auch nicht im Schnitt der beiden Halbebenen. Die Ergebnismaske ist deshalb rot. Der Fall zweier roter Masken kann nur an einem Eckpixel auftreten.

Masken durch Nullen, in roten Masken durch Einsen dargestellt. Daraus ergibt sich die Notwendigkeit unterschiedlicher Verknüpfungsoperationen für die verschiedenen Maskentypen. Die folgende Tabelle zeigt die möglichen Kombinationen:

Typ $Maske_1$	Typ $Maske_2$	Berechnung der Ergebnismaske	Typ Erg.
rot	rot	$Maske_1$ AND $Maske_2$ NOT(NOT($Maske_1$) OR NOT($Maske_2$))	rot
schwarz	rot	NOT($Maske_1$) AND $Maske_2$ NOT($Maske_1$ OR NOT($Maske_2$))	rot
schwarz	schwarz	NOT(NOT($Maske_1$) AND NOT($Maske_2$)) $Maske_1$ OR $Maske_2$	schwarz

Die Darstellung läßt sich offensichtlich auf einen Junktor-Typ vereinheitlichen, wenn entweder alle roten oder alle schwarzen Masken vor der Verknüpfung negiert werden. Die Ergebnismaske muß aber, wie die Tabelle ebenfalls zeigt, nochmals negiert werden, wenn sie von dem zuvor negierten Typ ist.

Noch einfacher wird die Verknüpfung, wenn wir das Exklusiv-Oder verwenden. Die Voraussetzung für dessen Einsatz ist jedoch, daß sich die Kanten der beiden Masken nicht im Pixel kreuzen. Ist das nämlich der Fall, entstehen zusätzliche gesetzte Pixel jenseits des Schnittpunktes. Abbildung 3.18 zeigt ein Beispiel für das entstehende Artefakt.

Ansonsten realisiert das XOR die gewünschte Verknüpfung, insbesondere ist das Ergebnis wieder vom richtigen Typ, wie man sich an den Beispielen in den Abbildungen 3.15 und 3.16 klarmachen kann.

Aufgrund der Assoziativität und Kommutativität des XOR-Operators ergeben sich aus der Verknüpfung mit einer dritten Maske keine Unterschiede. Die Abbildungen 3.19 und 3.20 geben hierfür Beispiele.

Wie die Eckpixel bearbeitet werden, wird im folgenden Abschnitt beschrieben.

In der Software-Implementierung des Subpixel-Verfahrens befindet sich die Maskenverknüpfung im Modul *XpAANMsk.c*.

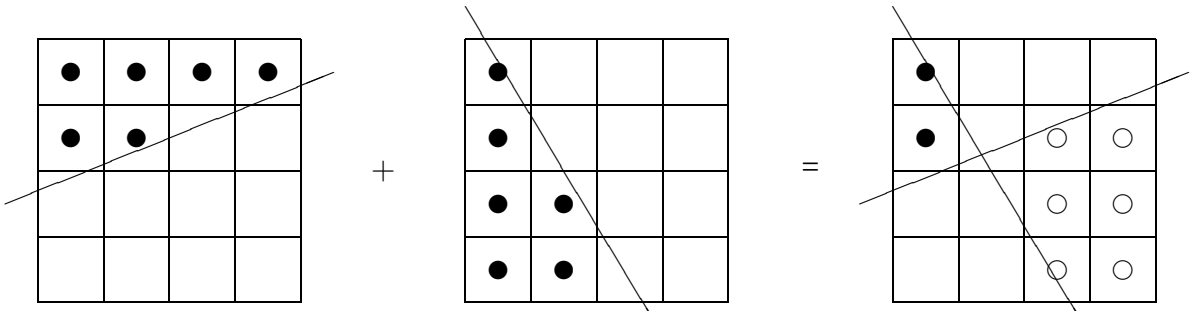


Abbildung 3.18: XOR-Artefakt an einem Eckpixel: Dadurch, daß sich die beiden Kanten innerhalb des Pixels überkreuzen, werden die mit ○ markierten Subpixel fälschlicherweise gesetzt. Die Masken von Eckpixeln dürfen deshalb nicht XOR-verknüpft werden. Die Verknüpfung muß mit Hilfe einer Fallunterscheidung und der logischen Kon- bzw. Disjunktion berechnet werden.

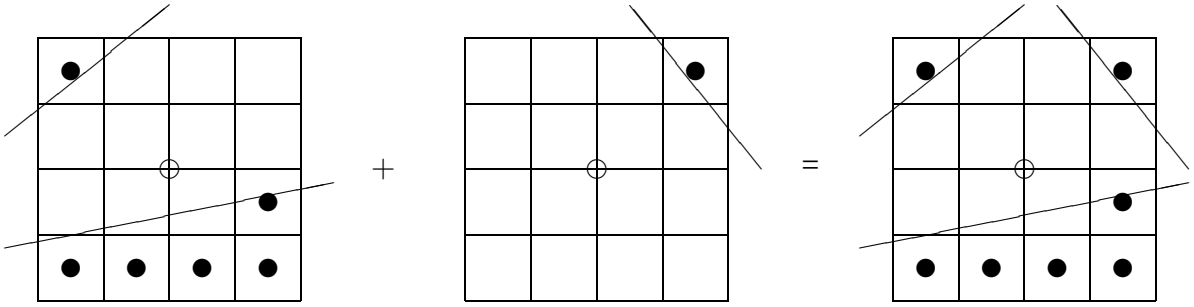


Abbildung 3.19: Schwarzes Pixel mit drei Kanten: Die Verknüpfung der ersten beiden Masken (hier nicht dargestellt) ergibt wieder eine schwarze Maske. Deren Verknüpfung mit der dritten schwarzen Maske verhält sich wie erwartet.

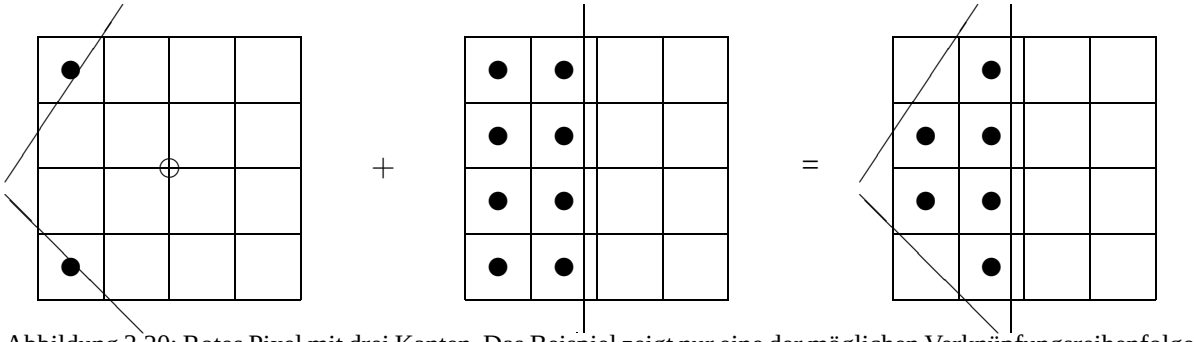


Abbildung 3.20: Rotes Pixel mit drei Kanten. Das Beispiel zeigt nur eine der möglichen Verknüpfungsreihenfolgen. Aufgrund von Assoziativität und Kommutativität des XOR-Operators ist die Verknüpfungsreihenfolge beliebig.

3.4.5 Spezielle Behandlung der Eckpixel

Wie im vorhergehenden Abschnitt erklärt wurde, bedürfen die Eckpixel eines Dreiecks besonderer Sorgfalt. Die Verknüpfung der Masken, die im Fall einer normalen Kante sehr effizient mit Hilfe der XOR-Verknüpfung bewältigt werden konnte, muß im Fall eines Eckpixels explizit durchgeführt werden.

In der Hardware-Realisierung entfällt die gesonderte Behandlung von Eckpixeln, weil es durch die Möglichkeit zur Parallelität keinen zusätzlichen Aufwand verursacht, alle Pixel einheitlich

so zu bearbeiten, daß auch Eckpixel korrekt gelöst werden.

Wie im letzten Abschnitt erklärt wurde, läßt sich die Verknüpfung von Masken auf einen einzigen Junktor-Typ vereinheitlichen, wenn entweder alle roten oder alle schwarzen Masken negiert werden. Im folgenden sollen das willkürlicherweise die schwarzen Masken sein. Für die Ein- und Ausgabeparameter der Verknüpfung wird folgende Struktur verwendet, in der die Farbe der Masken explizit mitgeführt wird:

```
typedef struct {
  Mask Mask;
  tFlag IsBlack;
} tPAAMaskAndColor;
```

Die Verknüpfung läßt sich nun für alle möglichen Fälle einheitlich mit folgendem Codestück durchführen:

```
if (MnC1.IsBlack) MnC1.Mask ^= 0xffff;
if (MnC2.IsBlack) MnC2.Mask ^= 0xffff;
Result->IsBlack = MnC1.IsBlack && MnC2.IsBlack;
Result->Mask    = (MnC1.Mask & MnC2.Mask);
if (Result->IsBlack) Result->Mask ^= 0xffff;
```

Wie die vorletzte Zeile zeigt, wird hier die Konjunktion anstelle des Exklusiv-Oders verwendet.

Sobald die Eckpixel bearbeitet wurden, können sie von den beiden zugehörigen Kanten entfernen werden. Dadurch wird sichergestellt, daß die korrekt berechneten Masken nicht durch die falschen Ergebnisse der XOR-Methode überschrieben werden. Wie die Eckpixel entfernt werden müssen, hängt davon ab, ob es sich um den Anfang oder um das Ende der Kante handelt.[5]

Oberes Ende: Fortschalten der Kante um ein Pixel

Unteres Ende: Vorziehen der Zielspalte der Kante um ein Pixel. Dieses Abschneiden darf erst durchgeführt werden, wenn das Verfahren bereits in der Zeile des Eckpixels angelangt ist, ansonsten könnten im Falle steiler Kanten andere Kantenmasken aus derselben Spalte mit abgeschnitten werden.

Der nächste Schritt besteht darin, festzustellen ob und, wenn ja, welche Ecken in einem Pixel zusammenfallen. Ist das der Fall, muß die dritte Kante ebenfalls in die Verknüpfung mit einbezogen und entsprechend gekürzt werden.[6] Um redundante Abfragen bei der Verarbeitung der

[5] Bei Da*Render* findet das Rendern immer in der Richtung zunehmender Y-Werte statt. Der Anfang ist also das Pixel mit der kleinsten, das Ende das Pixel mit der größten Y-Komponente.

[6] Der Fall, daß die dritte Kante ein Eckpixel durchläuft, ohne daß sich in diesem Eckpixel ein doppelter Eckpunkt befindet, stellt kein Problem dar: Das Eckpixel kann in den temporären Zeilenpuffer zurückgeschieben werden, die dritte Kante wird dann im Zeilenpuffer mit der Maske des Eckpixels ganz normal XOR-verknüpft. Das Problem der überkreuzenden Kanten besteht zu diesem Zeitpunkt nicht mehr, der XOR-Operator arbeitet also korrekt.

drei Ecken zu vermeiden, bietet es sich an, die notwendigen Analysen nur einmal und zwar bei der Initialisierung des Dreiecks durchzuführen. Der Test, welche Eckpixel zusammenfallen, kann sehr übersichtlich durchgeführt werden, wenn man das sog. Zentrums-Pixel mit einbezieht.

Def: Zentrumspixel

Das Zentrums-Pixel eines Dreiecks befindet sich in dessen mittlerer Zeile in der mittlerer Spalte. Die mittlere Zeile ist die mittlere der drei Zeilen, die ein Eckpixel enthalten. Die mittlere Spalte ist die mittlere der drei Spalten, die ein Eckpixel enthalten.

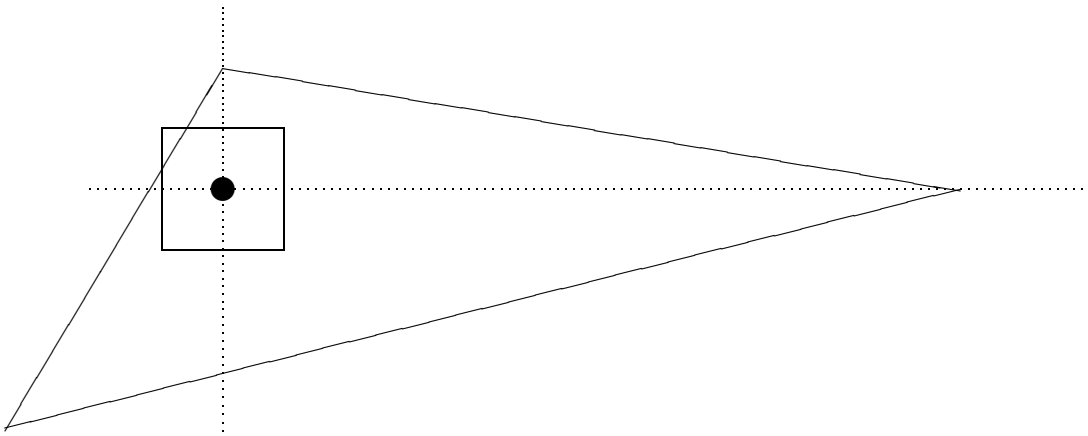


Abbildung 3.21: Das Zentrums-Pixel befindet sich in der mittlere Zeile in der mittleren Spalte eines Dreiecks.

Die Berechnungen beim Vergleich der Eckpixel basieren auf der folgender Beobachtung: Wenn zwei Eckpunkte in dasselbe Pixel fallen, dann fallen sie auch mit dem Zentrums-Pixel zusammen. Beweis: Wenn zwei Eckpunkte in dasselbe Pixel fallen, dann haben sie jeweils gleiche Werte in Zeile und Spalte. Wenn sich in einer Menge von drei Elementen zwei identische befinden, dann ist der Median ebenfalls identisch mit diesen beiden Elementen. Das Zentrums-Pixel aber befindet sich genau auf dem Median von Zeile und Spalte.

Betrachten wir die möglichen Fälle, in denen Eckpixel und Zentrums-Pixel zusammenfallen können, so ergeben sich insgesamt acht verschiedene Kombination. Die bei einem bestimmten Dreieck vorliegende Kombination soll dessen Dreieckstyp genannt werden. Die Menge der möglichen Dreieckstypen ist in Abbildung 3.22 dargestellt. In Abbildung 3.23 werden Beispiele für alle acht Fälle gezeigt. Die räumliche Anordnung der Fälle ist in beiden Abbildungen identisch.

Um Tests während der Eckpixelbearbeitung zu vereinfachen, bietet es sich an, jeden Dreieckstyp durch ein eigenes bestimmtes Bit zu definieren. Jeder mögliche Test auf Zugehörigkeit des Dreieckstyps zu einer bestimmten Menge läßt sich dann auf eine einzige Konjunktion reduzieren. Das folgende Codefragment realisiert die Berechnung des Dreieckstyps:

```
if (Ecke[1].x == Zentrum.x && Ecke[1].y == Zentrum.y)
```

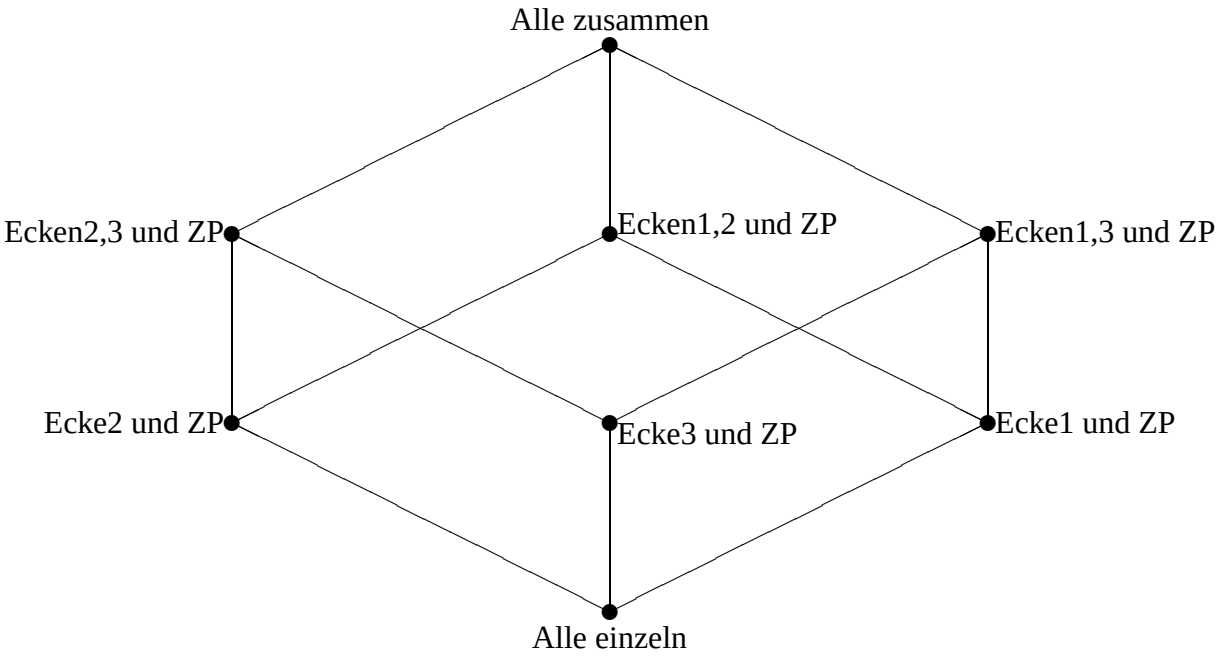


Abbildung 3.22: Die Menge aller Dreieckstypen bildet zusammen mit den Operationen Schnitt und Vereinigung eine Verbandsstruktur.

```
  DTyp |= 1;
if (Ecke[2].x == Zentrum.x && Ecke[2].y == Zentrum.y)
  DTyp |= 2;
if (Ecke[3].x == Zentrum.x && Ecke[3].y == Zentrum.y)
  DTyp |= 4;
DreiecksTyp = 1 << DTyp;
```

Abschließend noch als Beispiel eine Übersicht über die Dreieckstyp-Verteilung bei dem Demo-Bild ‘Sphere.c’, von dem sich mehrere Abbildungen im Anhang befinden. Wie man sieht, tritt das Zusammenfallen von Eckpixeln nur extrem selten auf, im Beispiel von ‘Sphere.c’ nur in 2 von 840 Fällen.

640/840	Alle Eckpixel einzeln
172/840	Zentrums-Pixel mit Eckpixel1
14/840	Zentrums-Pixel mit Eckpixel2
12/840	Zentrums-Pixel mit Eckpixel3
1/840	Eckpixel1 mit Eckpixel2
1/840	Eckpixel2 mit Eckpixel3
0/840	Eckpixel1 mit Eckpixel3
0/840	Alle Eckpixel zusammen

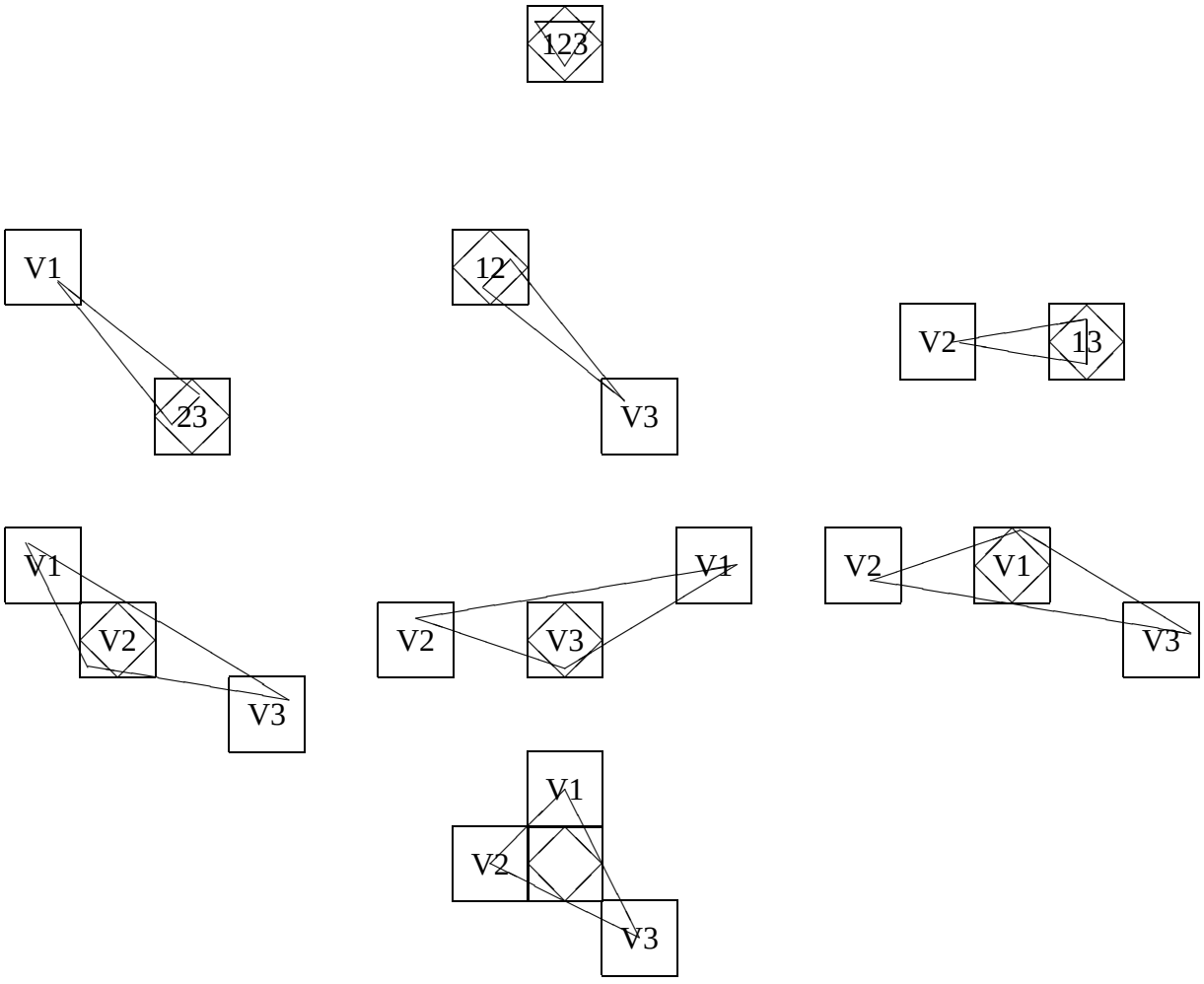


Abbildung 3.23: Die Menge aller unterschiedlicher Dreieckstypen. Der Dreieckstyp gibt an, welche Eckpixel mit dem Zentrumspixel (⋄) zusammenfallen. Die Ziffern in den Pixeln geben an, welche Ecken des Dreiecks in dieses Pixel fallen.

3.4.6 Visibilität

Die Berechnung, welche Bildteile sichtbar und welche verdeckt sind (Hidden-Surface-Removal), wird beim Z-Buffering auf die Sichtbarkeit der jeweiligen Pixelmittelpunkte reduziert. Ist das betrachtete Objekt am Pixelmittelpunkt sichtbar, d.h. hat es den kleinsten Z-Wert, dann nehmen wir an, daß es das gesamte Pixel überdeckt und daß es dabei auch vollständig sichtbar ist. Der Umstand, daß diese Annahme oft zu Unrecht getroffen wird, führt zu den deutlichen Aliasing-Effekten des Z-Buffers.

Das Subpixel-Verfahren baut auf dem Z-Buffering auf. Es ergibt sich die Frage, wie die Sichtbarkeit von Subpixeln und Subpixelmasken, sinnvoll approximiert werden kann, wenn verläßliche Informationen über die geometrische Situation nur an den Pixelmittelpunkten vorliegen. Folgende Regeln werden verwendet:

Schwarze Masken [7] sind sichtbar, wenn das zugehörige Pixel sichtbar ist. Eine schwarze Maske wird also genau dann in den Framebuffer geschrieben, wenn das zugehörige Pixel ebenfalls zurückgeschrieben wurde. Welche Pixel zurückgeschrieben werden wird vom Renderer bestimmt und ist nicht Teil des Antialiasing-Mechanismus. Informationen über Aufbau der Visibilitätsbestimmung von Da*Render*, der Plattform für die Software-Implementierung des Subpixel-Verfahrens befinden sich in [AH92].

Rote Masken sind genau dann sichtbar, wenn das bzw. die Nachbarpixel, auf die die Maske verweist, sichtbar sind, also vom Renderer gesetzt wurden.

Die Sichtbarkeit von roten Masken wird in Abschnitt 4.4 weiter verfeinert und auf subpixelgroße Fragmente erweitert.

Die beiden Regeln zur Visibilität von schwarzen und roten Masken führen zu folgendem Verhalten des Subpixel-Verfahrens in den unterschiedlichen Fällen:

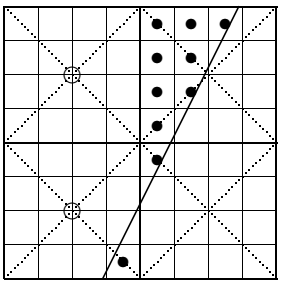
Aneinandergrenzende Objekte
Da beide Objekte voll sichtbar sind, wird die gemeinsame Kante doppelt bearbeitet, von jedem Objekt einmal. Die Masken in der Kantenregion, die beim Rendern des ersten Objektes gesetzt worden waren, werden beim Rendern des zweiten Objektes überschrieben. Dabei ändert sich nichts, da die von den beiden Objekten für die Kantenregion generierten Masken identisch sind (siehe Abbildung 3.24).

Einander überdeckende Objekte
Die sichtbare Kante zwischen den beiden Objekten ist nur bei der Bearbeitung eines der beiden Objekte, nämlich des überdeckenden bekannt. Da nur dieses Objekt sichtbar ist, wird bei seiner Bearbeitung die gemeinsame Kante geschrieben (siehe Abbildung 3.25). Die Masken des nicht sichtbaren Teils des überdeckten Objektes werden nicht zurückgeschrieben bzw. werden von dem sichtbaren Objekt überschrieben. Das Rendern eines Objektes direkt vor dem Hintergrund entspricht ebenfalls einer Überdeckung. Der Hintergrund übernimmt dabei die Rolle des überdeckten Objektes.

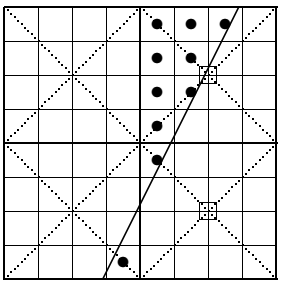
Einander durchdringende Objekte
Da die Kante zwischen den Objekten nur implizit existiert, kann sie von keinem der Objekte bearbeitet werden. Beim Rendern beider Objekte erhält die bearbeitende Routine — zu Recht — den Eindruck, das Objekt werde von einem anderen überdeckt. Bei der Bearbeitung beider Objekte wird deshalb die Annahme getroffen, die entstehende Kante werde beim Rendern des anderen Objektes bearbeitet. Da die Kante nicht bearbeitet wird, erscheint die Durchdringung im fertigen Bild als das Resultat eines normalen Pointsampling-Algorithmus ohne Nachbearbeitung. Abbildung 3.26 verdeutlicht das an einem Beispiel.

Das Subpixel-Verfahren, wie auch das Vier-Zeiger-Verfahren, ist also nicht in der Lage, Durchdringungen korrekt zu behandeln. Ich möchte an dieser Stelle eine Erweiterung des Verfahrens

[7] Begiffsdefinitionen siehe Seite 29



+



=

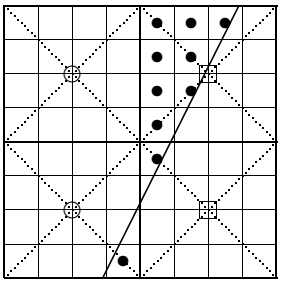


Abbildung 3.24: Angrenzende Dreiecke. Die Pixel des linken Dreiecks sind mit ○ markiert, die des rechten mit □. Die Masken der gemeinsamen Kante werden von beiden Dreiecken einmal geschrieben. Die von den beiden Dreiecken für ein Pixel erzeugten Masken sind identisch. Jede gemeinsame Maske ist übrigens aus der Sicht des einen Dreiecks rot, aus der des anderen schwarz.

skizzieren, um die Bildqualität an Durchdringungen zu verbessern. Dieser Ansatz ist jedoch z.Z. nicht Bestandteil der Software-Implementierung.

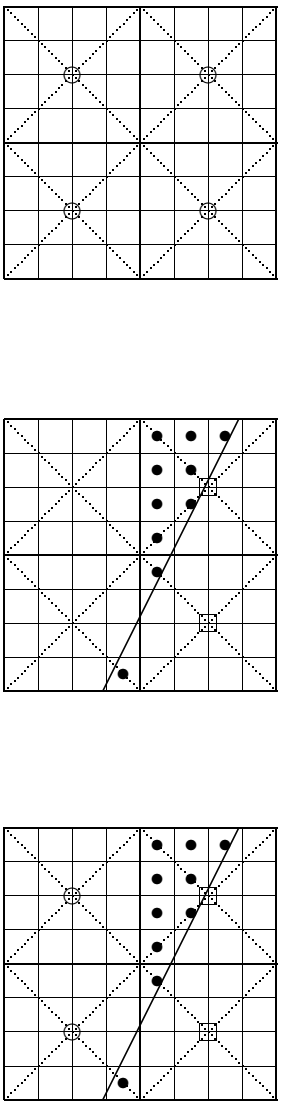


Abbildung 3.25: Einander überdeckende Dreiecke. Die Pixel des unteren Dreiecks sind mit ○ markiert, die des oberen mit □. Die Masken der gemeinsamen Kante werden von dem sichtbaren Dreieck erzeugt.

Behandlung von Durchdringungen
• Vergleiche während des Pointsamplings die Sichtbarkeit des letzen Pixels mit der Sichtbarkeit des aktuellen Pixels. Wenn beide sichtbar oder beide unsichtbar sind brich ab.
• Teste die Masken der beiden Pixel. Ist wenigstens eine von beiden gesetzt, dann wollen wir annehmen, es handele sich um eine Überdeckung durch ein anderes Dreieck. Brich ab.
(•) Approximiere die Schnittkante der beiden Dreiecke über das Verhältnis der Z-Wert Differenzen (siehe unten). Setze die Maske entsprechend.

Es liegt keine Information über die Steigung der Schnittkante vor. Es ist jedoch möglich, die

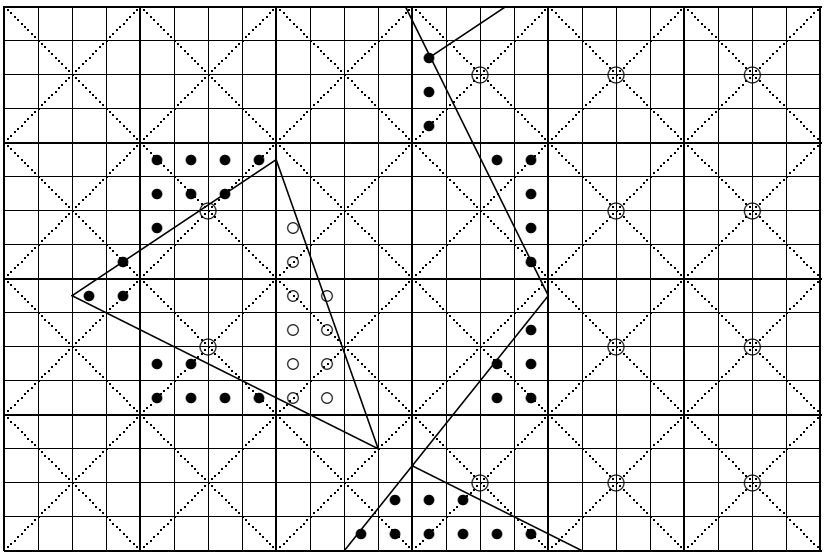


Abbildung 3.26: Einander durchdringende Dreiecke. Die ∘ markierten Subpixel hätten gesetzt werden müssen.

Stelle zu berechnen an der die Schnittkante die Gerade durch die Pixelmittelpunkte des Spans kreuzt. Für die Berechnung des Abstands D des Schnittpunkts zu dem aktuellen Pixel ist es erforderlich, alte und neue Z-Werte des Pixels und seines Vorgängers kurzfristig zu speichern.

$$D = (Z_{neu} - Z_{alt})/(ZVor_{alt} - ZVor_{neu}) \tag{3.6}$$

Abbildung 3.27 verdeutlicht die geometrische Situation.

Anhand des Wertes von D lassen sich Subpixelmasken für das aktuelle Pixel und/oder das Vorgängerpixel bestimmen. Wie üblich kann dabei eine Tabelle verwendet werden. Da die Steigung der aufgespürten Kante nicht mehr zu rekonstruieren ist, sind wir gezwungen heuristische Annahmen zu machen. Die Annahme einer stärker geneigten Geraden führt zu einer stärkeren Durchmischung der Farben und somit zu einer stärkeren Vergrauung im Bereich der Durchdringung. Gleichzeitig werden aber auch eventuelle Artefakte weniger auffällig.

Die vorgestellte Methode zur Approximation der Subpixelmasken an Durchdringungen funktioniert nur in Spanrichtung. Nahezu horizontale Schnittkanten werden nur schlecht erfaßt. Um alle Möglichkeiten auszuschöpfen wäre es daher notwendig, die Methode auf den zweidimensionalen Fall auszudehnen und auch die Z-Werte des vorhergehenden Spans temporär zu speichern.

In der Software-Implementierung des Subpixel-Verfahrens befindet sich das Testen der Sichtbarkeit zusammen mit dem Zurückschreiben sichtbarer Masken im Modul *XpAAWrit.c*.

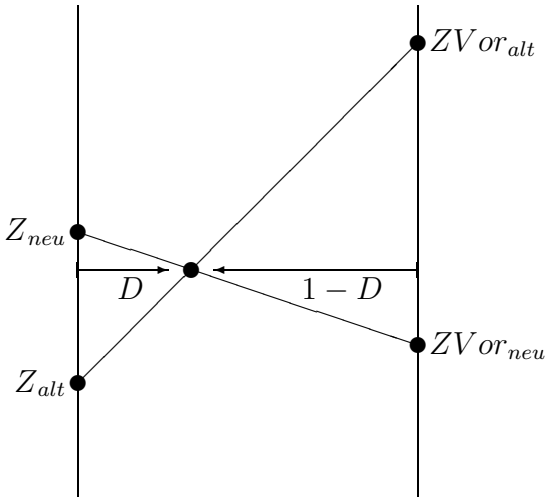


Abbildung 3.27: Schnittpunktberechnung an Durchdringung: Der Punkt in der Mitte der Skizze befindet sich an der Stelle, an der die Durchdringung die Mitte des Spans kreuzt. Über das Verhältnis der vier eingetragenen Z-Werte läßt sich der Abstand D zum Mittelpunkt des aktuellen Pixels bestimmen. Mit diesem Abstand läßt sich eine Subpixelmaske erzeugen, die die Verhältnisse an der Durchdringung näherungsweise beschreibt.

3.4.7 Zurückschreiben neuer Masken

Die folgenden Betrachtungen beziehen sich in erster Linie auf die Software-Version des Subpixel-Verfahrens. Bei einer Hardware-Realisierung lassen sich die entstehenden Probleme einfacher lösen, was im Kapitel *'Hardware-Realisierung des Vier-Zeiger-Verfahrens'* vorgestellt wird.

Bevor eine Maske zum Framebuffer zurückgeschrieben werden kann, muß ihre Sichtbarkeit überprüft werden. Im Fall einer schwarzen Maske kann der Test auf Sichtbarkeit sofort durchgeführt werden, da sich der Test ausschließlich auf das aktuelle Pixel bezieht. Im Fall einer roten Maske ist das unter Umständen anders. Falls das Nachbarpixel, auf das verwiesen wird, in der nächsten Zeile liegt, kann der Sichtbarkeitstest noch nicht durchgeführt werden, weil diese Zeile noch nicht gerendert wurde. Wir wissen also noch nicht, welche der darin liegenden Pixel sichtbar sein werden.[8] In diesem Fall müssen wir das Pixel, bzw. die gesamte Kante zu der das Pixel gehört, 'verzögert', das heißt nach dem Rendern der nächsten Zeile, bearbeiten.

An welchen Kanten solche Fälle eintreten werden, läßt sich bereits zu Beginn der Bearbeitung eines Dreiecks anhand der Steigung der Kanten feststellen. Wir können deshalb jeder Kante eine boolesche Variable zuweisen, die aussagt, ob die Kante sofort oder verzögert bearbeitet werden soll. Wir werden uns immer dann für eine verzögerte Bearbeitung entscheiden, wenn sich die Halbebene der Kante *unter* der Kante befindet.[9] Genau dann kann es nämlich passieren, daß rote Masken auf die nächste Zeile verweisen. Abbildung 3.28 soll dies verdeutlichen.

[8] Wie im Abschnitt *'Visibilität'* erklärt wurde, wird die Sichtbarkeit der Masken auf die Sichtbarkeit von Pixeln zurückgeführt. Am Ende ist es also immer der Renderer, der die Sichtbarkeitsentscheidungen fällt.

[9] Vorrausgesetzt wir rendern von oben nach unten,wie das bei Da*Render* zum Beispiel der Fall ist.

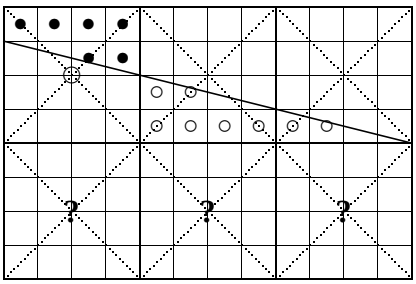


Abbildung 3.28: Kante mit verzögerter Bearbeitung. Das Dreiecksinnere befindet sich unter der Kante. Die mit ◦ markierten Masken können erst verzögert zurückgeschrieben werden, da die Sichtbarkeit der **?**-markierten Pixel noch nicht bekannt ist.

Um Sichtbarkeitstests für alle Arten roter Masken durchführen zu können, ist es also notwendig, daß wir folgende Arrays mitführen:

Name	**Inhalt**
Up. Pixel	Information, welche Pixel der vorhergehenden Zeile sichtbar waren
This. Pixel	Information, welche Pixel der aktuellen Zeile sichtbar sind
Up. Mask	Noch nicht zurückgeschriebene rote Masken der vorhergehenden Zeile
This. Mask	Alle Masken der aktuellen Zeile

Das Zurückschreiben von Masken zum Framebuffer kann dann wie folgt durchgeführt werden:

1. Zurückschreiben aller ‘schwarzen Masken’, deren zugehöriges Pixel gesetzt ist.
2. Zurückschreiben aller ‘roten Masken’ des aktuellen Spans, deren Kante nicht ‘verzögert’ ist und deren Zielpixel gesetzt ist.
3. Zurückschreiben aller ‘roten Masken’ des vorhergehenden Spans, deren Kante ‘verzögert’ ist und deren Zielpixel gesetzt ist.

In der Software-Implementierung des Subpixel-Verfahrens befindet sich Zurückschreiben sichtbarer Masken im Modul *XpAAWrit.c*.

3.4.8 Zurücksetzen ‘interner’ Subpixel

Im vorhergehenden Abschnitt haben wir uns damit beschäftigt, wann und wie neue Masken in den Framebuffer geschrieben werden. Es kommt aber auch vor, daß wir Masken aus dem Framebuffer löschen müssen, und zwar dann, wenn ein neues Objekt ein bereits im Framebuffer abgelegtes überdeckt.

Bei einfarbigen Dreiecken bleiben die alten Masken im Dreiecksinneren ohne Auswirkungen, da immer nur mit einem Pixel identischer Farbe gemischt wird. Ist das neue Dreieck jedoch

texturiert oder schattiert, dann führen die nicht gelöschte Masken zu schwachen, aber dennoch sichtbaren Schlieren. Wir sollten deshalb ‘interne’ Subpixelmasken, also Subpixelmasken im Inneren des Dreiecks, sicherheitshalber löschen.

Beim Löschen dürfen wir jedoch nicht maskenweise vorgehen, wie Abbildung 3.29 zeigt.

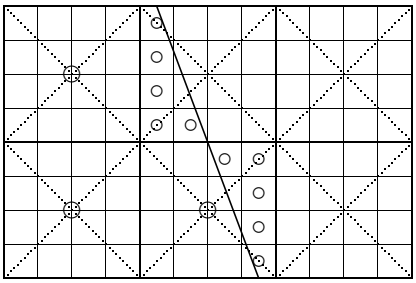


Abbildung 3.29: Die Fläche links von der Geraden sei ein Dreieck, das unter das Dreieck in der rechten Hälfte geschoben wurde. Die Masken in der mittleren Spalte dürfen beim Rendern des linken, untergeschobenen Dreiecks nicht gelöscht werden, denn sie sind korrekt und notwendig. Der Umstand, daß die beiden Pixel links unten gesetzt wurden, legitimiert lediglich das Löschen der Teilmasken zwischen diesen beiden Pixeln, nicht das Löschen der gesetzten Subpixel in der mittleren Spalte.

Wenn wir mit dem im vorhergehenden Abschnitt eingeführten Zwei-Zeilenpuffer auskommen wollen, müssen wir die Masken stückweise löschen:

1. Für alle Pixel der Zeile: Wenn zwei benachbarte Pixel sichtbar sind, dann lösche die linke Teilmaske des rechten und die rechte Teilmaske des linken Pixels.
2. Verfahre analog für vertikale Paare, setze dabei untere und obere Teilmasken zurück.
3. Setze Teilmasken, die auf zwei Nachbarn verweisen, zurück, wenn beide Nachbarn gesetzt wurden.

Das Verfahren des Löschens interner Masken wird im Rahmen der Hardware-Version deutlich beschleunigt werden. Siehe dazu das Ablaufschema auf Seite 84.

In der Software-Implementierung des Subpixel-Verfahrens befinden sich die Funktionen zum Löschen interner Masken im Modul *XpAARset.c*.

3.4.9 Berechnung der Subpixelmasken

Die Subpixelmaske eines Pixels ergibt sich aus der Verknüpfung der Masken der drei Kanten. Die einzelnen Masken können dabei zunächst separat erzeugt werden.

Die Generierung der Subpixelmasken ist an das von Andreas Schilling entwickelte Verfahren zur Berechnung der exakten Überdeckung angelehnt[Sch91] (Siehe Abschnitt *'Exakte Überdeckung'* auf Seite 16). Im Rahmen der Software-Implementierung des Subpixel-Verfahrens wurde die Maskengenerierung dabei als zweistufiger Prozeß verwirklicht:

1. Berechnung der Überdeckung des Pixels durch die Halbebene an der Kante
2. Bestimmung der Subpixelmaske

Beide Teilaufgaben können effektiv mit Hilfe je einer vorberechnete Tabellen gelöst werden. Die Erstellung der Tabellen wird in den folgenden Abschnitten dargestellt.

3.4.9.1 Berechnung der Überdeckung

Die überdeckte Fläche A eines Pixels durch eine Halbebene kann als Funktion Cov[10] von Erroterm und Winkel der Kante berechnet werden.

$$A = \text{Cov}(E, \alpha)$$

Wir werden hier zunächst nur die Fälle betrachten, bei denen

$$E \,\epsilon\, [0, \frac{1}{2}] \qquad \text{und} \qquad \alpha \,\epsilon\, [0, \frac{\pi}{4}]$$

Alle anderen Fälle lassen sich durch Spiegelungen in dieses Intervall überführen.

Offensichtlich läßt sich $\text{Cov}(E, \alpha)$ in zwei Fälle unterteilen:

1. Die überdeckte Fläche ist trapezförmig (Abb. 3.30 links)
2. Die überdeckte Fläche ist dreieckig (Abb. 3.30 rechts)

Entsprechend definieren wir die Überdeckungsfunktion $\text{Cov}(E, \alpha)$ als

$$\text{Cov}(E, \alpha) = \begin{cases} \text{Cov}_{Trapez}(E, \alpha) & : \quad \alpha <= \text{Bound}(E) \\ \text{Cov}_{Dreieck}(E, \alpha) & : \quad \alpha > \text{Bound}(E) \end{cases} \tag{3.7}$$

$\alpha = \text{Bound}(E)$ gilt dabei für genau die Kanten, die exakt durch den linken unteren Eckpunkt verlaufen.

Zunächst die Herleitung von $\text{Cov}_{Dreieck}(E, \alpha)$. Um die folgenden Berechnungen übersichtlicher zu gestalten führen wir einige Hilfsgrößen ein, deren Definition sich leicht aus der Skizze in Abbildung 3.31 ablesen lassen. Die Kantenlänge eines Subpixels beträgt $\frac{1}{4}$.

[10] Der Variablenname 'Cov' steht für Coverage, zu deutsch Überdeckung.

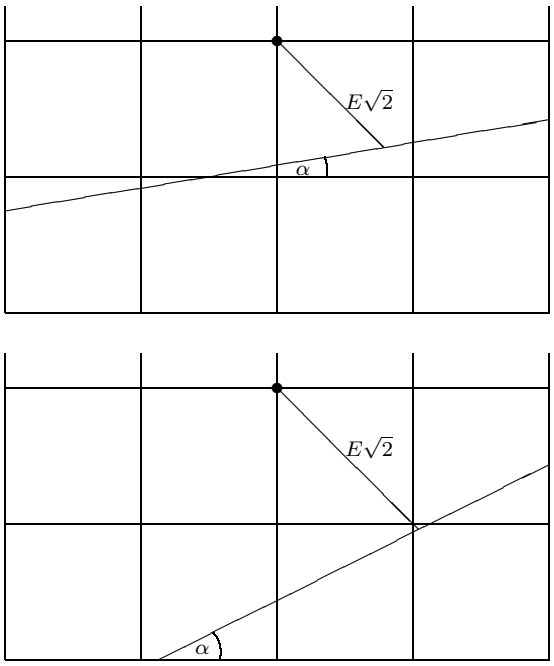


Abbildung 3.30: Trapezförmige (links) und dreieckige Überdeckung (rechts). Die Skizzen zeigen jeweils die untere Hälfte eines in $4 * 4$ Subpixel unterteilten Pixels. Der Term $E\sqrt{2}$ gibt die Distanz zwischen Kante und Pixelmittelpunkt an. Die von den Halbebenen der Kanten überdeckten Flächen liegen unterhalb der Kanten.

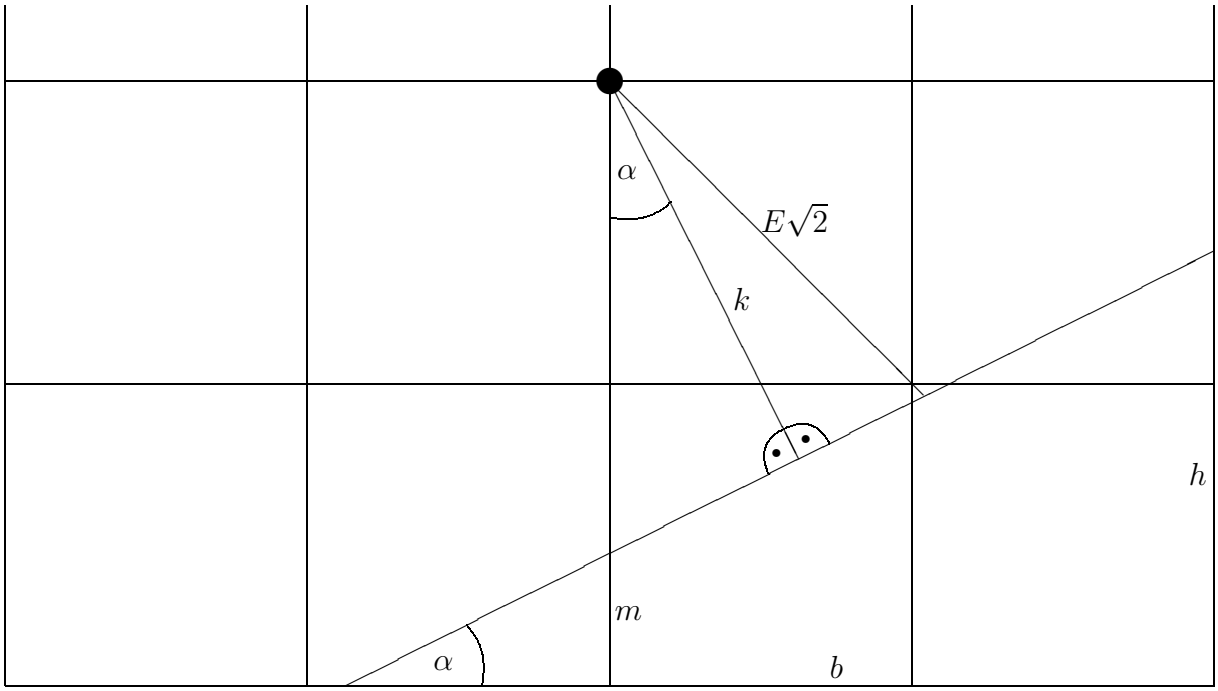


Abbildung 3.31: Schnitt zwischen Pixel und Halbebene. Die Skizze zeigt die untere Hälfte eines in $4 * 4$ Subpixel unterteilten Pixels. Die von der Halbebene der Kante überdeckte Fläche liegt unterhalb der Kante.

$$k = E\sqrt{2}\cos(\frac{\pi}{4} - \alpha) \tag{3.8}$$

$$m = \frac{1}{2} - \frac{K}{\cos\alpha} \tag{3.9}$$

$$b = \frac{1}{2} + \frac{m}{\tan\alpha} \tag{3.10}$$

$$h = b\tan\alpha \tag{3.11}$$

Nun können wir $\text{Cov}_{Dreieck}(E,\alpha)$ unter Zuhilfenahme der neuen Variablen direkt angeben mit

$$\text{Cov}_{Dreieck}(E,\alpha) = \frac{1}{2} * b * h \tag{3.12}$$

Durch Anwendung der Formel

$$\cos(\frac{\pi}{4} - \alpha) = \frac{1}{2}\sqrt{2}(\cos\alpha + \sin\alpha)$$

läßt sich die Definition von k vereinfachen zu

$$k = E(\cos\alpha + \sin\alpha) \tag{3.13}$$

Sukzessives Einsetzen ergibt dann

$$m = \frac{1}{2} - E(1 + \tan\alpha) \tag{3.14}$$

$$b = \frac{1}{2}(1-2E)(\cot\alpha + 1) \tag{3.15}$$

$$h = \frac{1}{2}(1-2E)(\tan\alpha + 1) \tag{3.16}$$

$$\tag{3.17}$$

und schließlich

$$\text{Cov}_{Dreieck}(E,\alpha) = \frac{1}{8}(2E-1)^2(\cot\alpha + \tan\alpha + 2) \tag{3.18}$$

Die obige Skizze läßt sich auch für die Berechnung von $\text{Cov}_{Trapez}(E,\alpha)$ verwenden. Die Definitionen von b und h werden hierfür nicht mehr benötigt. In Bezug auf die folgenden Berechnungen von de_x sei jedoch angemerkt, daß die Berechnungen von b und h sich im Falle einer trapezförmigen Überdeckung auf den Schnittpunkt der verlängerten Geraden mit der verlängerten Unterkante des Pixels beziehen. Mit dieser Betrachtungsweise können b und h auch im Fall der trapezförmigen Überdeckung als Repräsentationen von ΔX und ΔY verwendet werden können.

Die Trapezfläche ergibt sich zu

$$\text{Cov}_{Trapez}(E,\alpha) = m \tag{3.19}$$

was wir bereits als

$$\text{Cov}_{Trapez}(E,\alpha) = \frac{1}{2} - E(1 + \tan\alpha) \tag{3.20}$$

ausgerechnet hatten.

Nun benötigen wir noch die Funktion der Schnittgeraden Bound(E). Da, wie später noch gezeigt wird, beide Teilfunktionen $\text{Cov}_{Dreieck}(E,\alpha)$ und $\text{Cov}_{Trapez}(E,\alpha)$ monoton sind, läßt sich Bound(E) durch einen einfachen Schnittansatz bestimmen. Wir setzen

$$\text{Cov}_{Dreieck}(E, \text{Bound}(E)) = \text{Cov}_{Trapez}(E, \text{Bound}(E)) \tag{3.21}$$

und erhalten durch Auflösung der entstehenden Gleichung[11] nach α

$$\text{Bound}(E) = -\arctan\left(\frac{2\text{E}-1}{2\text{E}+1}\right) \tag{3.22}$$

Somit erhalten wir

$$\text{Cov}(E,\alpha) = \begin{cases} \frac{1}{2} - E(1+\tan\alpha) & : \quad \alpha <= -\arctan\left(\frac{2\text{E}-1}{2\text{E}+1}\right) \\ \frac{1}{8}(2E-1)^2(\cot\alpha + \tan\alpha + 2) & : \quad \alpha > -\arctan\left(\frac{2\text{E}-1}{2\text{E}+1}\right) \end{cases} \tag{3.23}$$

Wir sind bis jetzt immer von dem Winkel α als Eingabeparameter ausgegangen. Tatsächlich liegt α aber gar nicht explizit vor. Statt dessen verfügen wir unter anderem über die α-abgeleiteten Variablen de_x und de_y. Aus der Definition von de_x

$$de_x = \frac{\Delta Y}{|\Delta X| + |\Delta Y|}$$

erhalten wir durch Einsetzen von $\Delta Y = b$ und $\Delta X = h$

$$de_x = \frac{h}{b * h} \tag{3.24}$$

Nach Einsetzen von (3.10) und (3.11) erhalten wir

$$de_x = \frac{\tan\alpha}{1+\tan\alpha} \tag{3.25}$$

Auflösen nach α ergibt

$$\alpha = -\arctan(\frac{de_x}{de_x - 1}) \tag{3.26}$$

Um die neuen, von de_x statt von α abhängigen Formeln zu erzeugen, brauchen wir lediglich (3.26) in die bestehenden Gleichungen einzusetzen. Die neue Funktion $\text{DCov}(E, de_x)$ hat den Definitionsbereich $[0, \frac{1}{2}]$ im zweiten Parameter, im Gegensatz zu $\text{Cov}(E,\alpha)$, die im zweiten Parameter im Intervall $[0, \frac{\pi}{4}]$ definiert war. Wir erhalten

$$\text{DCov}(E, de_x) = \begin{cases} \text{DCov}_{Trapez}(E, de_x) & : \quad de_x <= \text{DBound}(E) \\ \text{DCov}_{Dreieck}(E, de_x) & : \quad de_x > \text{DBound}(E) \end{cases} \tag{3.27}$$

[11] Hätten wir, was genauso gut möglich ist, $\text{Cov}_{Dreieck}(\text{Bound}'(E),\alpha) = \text{Cov}_{Trapez}(\text{Bound}'(E),\alpha)$ als Schnittansatz gewählt, so hätten wir $\text{Bound}'(\alpha) = \frac{1}{2}\frac{\cos\alpha - \sin\alpha}{\cos\alpha + \sin\alpha}$ erhalten.

Einsetzen ergibt

$$\mathrm{DCov}(E, de_x) = \begin{cases} \frac{1}{2} - \frac{E}{1-de_x} & : \quad de_x <= \frac{1}{2} - E \\ (1-2E)^2 \frac{1}{8de_x(1-de_x)} & : \quad de_x > \frac{1}{2} - E \end{cases} \tag{3.28}$$

Vergleicht man die Ableitungen von $\mathrm{Cov}(E, \alpha)$ und $\mathrm{DCov}(E, de_x)$ stellt man übrigens fest, daß $\mathrm{DCov}(E, de_x)$ nicht nur leichter zu berechnen ist, sondern auch in α-Richtung weniger stark gekrümmt ist. Das hat den Vorteil, daß, bei gleicher Ergebnisgenauigkeit, die benötigte Auflösung der Überdeckungstabelle geringer ausfällt.

3.4.9.2 Erstellung der Überdeckungstabelle

Bevor die Überdeckungstabelle erstellt werden kann, muß die benötigte Auflösung der Tabelle bestimmt werden. Diese ist abhängig von der maximalen Steigung der Funktion $\mathrm{DCov}(E, de_x)$. Wir müssen also zunächst die zweiten partiellen Ableitungen berechnen, um damit das Maximum der ersten partiellen Ableitungen in dem gegebenen Intervall zu bestimmen. Das Berechnen von Ableitungen und deren Maxima kann für $\mathrm{DCov}_{Dreieck}(E, de_x)$ und $\mathrm{DCov}_{Trapez}(E, de_x)$ getrennt durchgeführt werden.

$$\frac{\partial^2 \mathrm{DCov}_{Trapez}(E, de_x)}{\partial^2 E} = 0 \tag{3.29}$$

$$\frac{\partial^2 \mathrm{DCov}_{Dreieck}(E, de_x)}{\partial^2 E} = \frac{1}{de_x(1-de_x)} \tag{3.30}$$

$$\frac{\partial^2 \mathrm{DCov}_{Trapez}(E, de_x)}{\partial de_x \partial E} = -\frac{1}{(1-de_x)^2} \tag{3.31}$$

$$\frac{\partial^2 \mathrm{DCov}_{Dreieck}(E, de_x)}{\partial de_x \partial E} = \frac{(1-2E)(1-2de_x)}{(2de_x^2(1-de_x)^2)} \tag{3.32}$$

$$\frac{\partial^2 \mathrm{DCov}_{Trapez}(E, de_x)}{\partial^2 de_x} = -\frac{2E}{(1-de_x)^3} \tag{3.33}$$

$$\frac{\partial^2 \mathrm{DCov}_{Dreieck}(E, de_x)}{\partial^2 de_x} = \frac{(1-2E)^2(3de_x^2 - 3de_x + 1)}{(4de_x^3(1-de_x)^3)} \tag{3.34}$$

Die Ableitungen sind bewußt so notiert, daß alle Teilterme in dem betrachteten Definitionsbereich positiv sind. Der einzige Teilterm, bei dem das nicht sofort offensichtlich ist, ist $(3de_x^2 - 3de_x + 1)$ aus Gleichung (3.34). Nullsetzen des Terms zeigt jedoch, daß auch er keine Nullstellen im betrachteten Intervall aufweist und für alle de_x größer 0 ist.

Außer den Ableitung in (3.29) hat keine der partiellen zweiten Ableitungen eine Nullstelle im *offenen* betrachteten Intervall, weshalb die ersten partiellen Ableitungen monoton sind. Bei Gleichung (3.29) sind aber offensichtlich auch alle nachfolgenden Ableitungen gleich 0, so daß die zugehörige erste Ableitung kein Extremum aufweist. Die Extremwerte von $\mathrm{DCov}(E, de_x)$ können sich damit nur auf den Grenzen des Intervalls, bzw. an der Grenze zwischen $\mathrm{DCov}_{Dreieck}(E, de_x)$ und $\mathrm{DCov}_{Trapez}(E, de_x)$ befinden.

Betrachtet man nun die ersten partiellen Ableitungen

$$\frac{\partial \text{DCov}_{Trapez}(E, de_x)}{\partial E} = -\frac{1}{1-de_x} \tag{3.35}$$

$$\frac{\partial \text{DCov}_{Dreieck}(E, de_x)}{\partial E} = -\frac{1-2E}{2de_x(1-de_x)} \tag{3.36}$$

$$\frac{\partial \text{DCov}_{Trapez}(E, de_x)}{\partial de_x} = -\frac{E}{(1-de_x)^2} \tag{3.37}$$

$$\frac{\partial \text{DCov}_{Dreieck}(E, de_x)}{\partial de_x} = -\frac{(1-2E)^2(1-2de_x)}{8de_x^2(1-de_x)^2} \tag{3.38}$$

stellt man fest, daß alle im gesamten Intervall negativ sind. Unsere Suche nach den betragsmäßigen Maxima der partiellen Ableitungen nach E und de_x entspricht damit der Suche nach dem jeweiligen Minimum.

Betrachten wir noch einmal die partiellen Ableitungen zweiten Grades, so stellen wir fest, daß alle Ableitungen von $\text{DCov}_{Trapez}(E, de_x)$ (3.29), (3.31) und (3.33) kleiner oder gleich 0 sind, während die entsprechenden Ableitungen von $\text{DCov}_{Dreieck}(E, de_x)$ (3.30), (3.32) und (3.34) größer oder gleich 0 sind. Die gesuchten Minima müssen also auf der Grenze zwischen $\text{DCov}_{Trapez}(E, de_x)$ und $\text{DCov}_{Dreieck}(E, de_x)$ liegen.

Die Berechnung der ersten partiellen Ableitungen der Funktionen $\text{DCov}_{Trapez}(E, de_x)$ und $\text{DCov}_{Dreieck}(E, de_x)$ entlang der Schnittkante ergibt

$$\frac{\partial \text{DCov}_{Trapez}(E, \text{DBound}(E))}{\partial E} = -\frac{1}{1-\frac{1}{2}-E} \tag{3.39}$$

$$\frac{\partial \text{DCov}_{Dreieck}(E, \text{DBound}(E))}{\partial E} = -\frac{1-2E}{2\frac{1}{2}-E(1-\frac{1}{2}-E)} \tag{3.40}$$

$$\frac{\partial \text{DCov}_{Trapez}(E, \text{DBound}(E))}{\partial de_x} = -\frac{E}{(1-\frac{1}{2}-E)^2} \tag{3.41}$$

$$\frac{\partial \text{DCov}_{Dreieck}(E, \text{DBound}(E))}{\partial de_x} = -\frac{(1-2E)^2(1-2\frac{1}{2}-E)}{8\frac{1}{2}-E^2(1-\frac{1}{2}-E)^2} \tag{3.42}$$

Die Ergebnisterme sind paarweise identisch, die Funktion hat also, wie zu erwarten war, keinen Knick am Übergang zwischen den beiden Teilfunktionen. Wir erhalten

$$\begin{aligned} \frac{\partial \text{DCov}_{Trapez}(E, \text{DBound}(E))}{\partial E} &= \\ \frac{\partial \text{DCov}_{Dreieck}(E, \text{DBound}(E))}{\partial E} &= -\frac{2}{2E+1} \end{aligned} \tag{3.43}$$

$$\begin{aligned} \frac{\partial \text{DCov}_{Trapez}(E, \text{DBound}(E))}{\partial de_x} &= \\ \frac{\partial \text{DCov}_{Dreieck}(E, \text{DBound}(E))}{\partial de_x} &= -\frac{4E}{(2E+1)^2} \end{aligned} \tag{3.44}$$

Die erste von beiden ist offensichtlich monoton steigend, die zweite hat ein globales Minimum bei $E = \frac{1}{2}$. Durch Einsetzen von 0 in die erste und $\frac{1}{2}$ in die zweite Formel erhalten wir schließlich die gewünschten Ergebnisse. Die betragsmäßig maximale Steigung in E-Richtung liegt also bei $(\frac{1}{2}, 0)$ und hat den Betrag -2, die betragsmäßig maximale Steigung in de_x-Richtung bei $(0, \frac{1}{2})$. Sie hat den Betrag $-\frac{1}{2}$.

Die berechneten betragsmäßig maximalen Steigungen sind übrigens in beiden Parametern genau doppelt so groß wie die Steigung einer entsprechenden linearen Funktion auf den betrachteten Intervallen. Wir benötigen also jeweils ein Bit zusätzlicher Auflösung, um die Krümmung zu kompensieren.

Abbildung 3.32 zeigt eine Skizze der Funktion DCov(E, de_x).

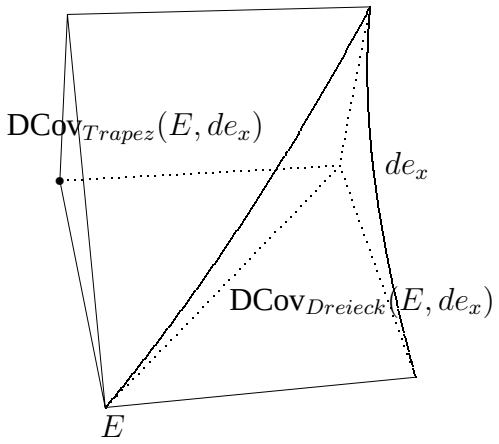


Abbildung 3.32: Die Überdeckungsfunktion DCov(E, de_x). Der Koordinatenursprung befindet sich hinten rechts. Die E-Achse zeigt zum Betrachter, die de_x-Achse nach rechts. Die Überdeckung nimmt mit zunehmendem E streng monoton, mit zunehmendem de_x monoton ab. Wegen ihrer geringen Krümmung, läßt sich die Funktion gut linear interpolieren.

Wenn wir uns bei einer Zerlegung der Pixel in 16 Subpixel für eine Genauigkeit der Tabelle von je einem halben Subpixel in de_x und E entscheiden, ergibt sich die Anzahl der Adressbits der Tabelle zu

Normale Auflösung in E	4 Bits
Extrabit für doppelte Auflösung in E	1 Bit
Normale Auflösung in de_x	2 Bits
Extrabit für doppelte Auflösung in de_x	1 Bit
Gesamt	8 Bits

wenn wir E auf Werte zwischen 0 und $\frac{1}{2}$ beschränken. Die andere Hälfte des Definitionsbereiches von E können wir durch die einfache Symmetrie

$$\text{DCov}(E, de_x) = 1 - \text{DCov}(-E, de_x) \tag{3.45}$$

erzeugen, was aber beim Subpixel-Verfahren zunächst gar nicht benötigt wird, da der Hauptanteil der Farbe immer vom Pixel selbst bestritten wird und nur der kleinere Anteil vom Nachbarn zugemischt wird. Die gewünschte Überdeckungstabelle ist also 256 Einträge groß. Da Subpixelanzahlen von 0–8 gespeichert werden sollen, benötigt jeder Eintrag mindestens vier Bit Speicher.

Die Extrabits für die Mantisse, die Schilling in seinem Ansatz [Sch91, Seite 136] zusätzlich auswertet, benötigen wir ausdrücklich *nicht*. Da wir keine Masken zusammenfügen, sondern nur überschreiben, besteht keine Notwendigkeit, sicherzustellen, daß die Masken zweier angrenzender Dreiecke exakt zusammenpassen. Das Mantissenbit führt auch nicht zu einer Erhöhung der Genauigkeit der Überdeckungsfläche. Es ist beim Subpixel-Verfahren also überflüssig.

Die Tabelle sei willkürlich so definiert, daß die untersten fünf Bits durch E bestimmt werden, die oberen durch de_x. Die Initialisierung der Tabelle kann dann durch folgende kurze Routine vorgenommen werden

```
for (dex = 0, dex < 8, dex++)
  for (E = 0, E < 32, E++)
    CoverageTab[i++] = (tCard8)(dex <= Bound(E)
                                ? Cov_Trapez(E,dex)
                                : Cov_Dreieck(E,dex));
```

3.4.9.3 Erstellung der Subpixeltabelle

Wie Schilling in [Sch91] feststellt, gibt es für jeden Oktanten nur vier mögliche Reihenfolgen, in denen die 16 Subpixel eines Pixels von einer Halbebene überdeckt werden können. Sie sind in den Abbildungen 3.33–3.36 dargestellt.

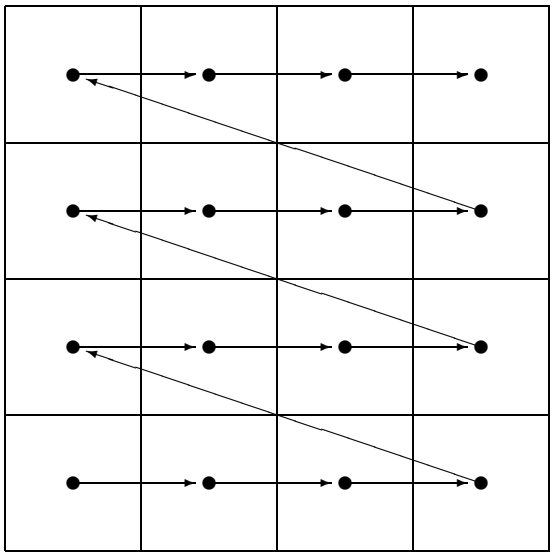


Abbildung 3.33: Ansprechordnung 1

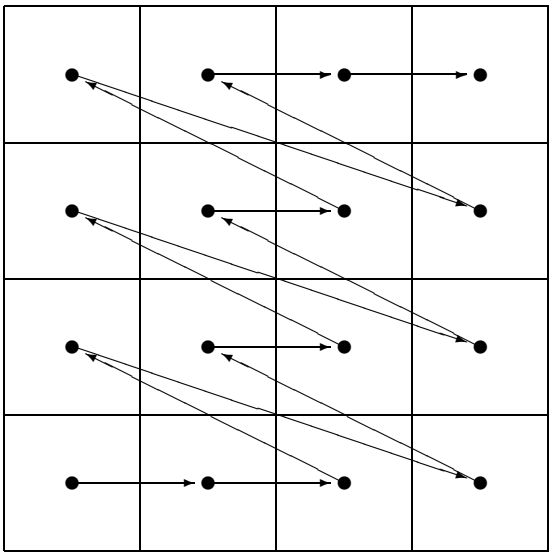


Abbildung 3.34: Ansprechordnung 2

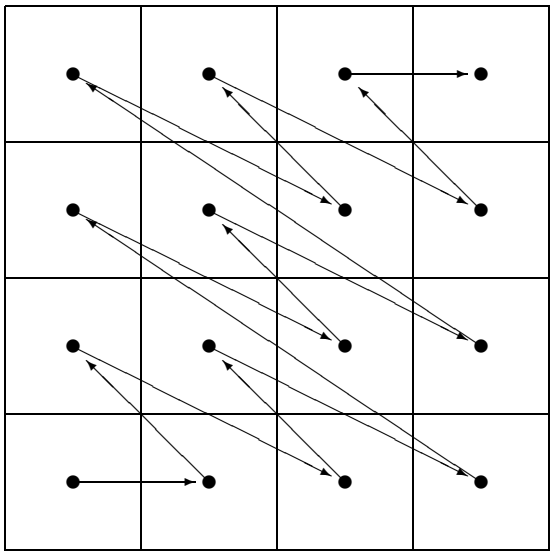


Abbildung 3.35: Ansprechordnung 3

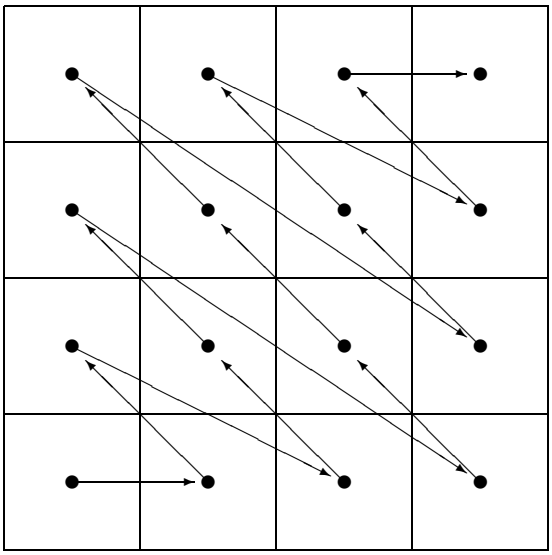


Abbildung 3.36: Ansprechordnung 4

Für die Anzahl der überdeckten Subpixel gibt es an sich 17 Werte, [0,16], wofür wir fünf Adressbits benötigen würden. Wir können aber zwei Reduktionen durchführen:

1. Wir benötigen eigentlich nur Werte aus [0,8], da die Hauptüberdeckung an *einer* Kante zunächst zum Hauptteil vom Pixel selbst gedeckt wird. Der Mittelpunkt des Pixels würde sonst zwangsläufig zu einem anderen Dreieck gehören. (Mit der gleichen Begründung hatten wir bereits Überdeckungstabelle auf die Hälfte reduziert.)

2. Der Fall, daß die Überdeckung 0 ist, läßt sich sehr einfach auch ohne Tabelle bewältigen. Die Maske ist in diesem Fall immer 0x0000, einheitlich für alle Oktanten und Steigungen.

Für die Subpixel erhalten wir damit die folgende Anzahl an Adressbits

Oktant	3 Bits
Ansprechreihenfolgen je Oktant	2 Bits
Anzahl der überdeckten Subpixel	3 Bits
Gesamt	8 Bits

Die Tabelle ist also 256 Einträge groß.

Wenn wir für die Tabelle willkürlich folgende Adressierung festlegen:

Bit 7	Setze Bits von links nach rechts $(de_x < 0)$
Bit 6	Setze Bits von oben nach unten $(de_y < 0)$
Bit 5	Steile Linie ($\|de_x\| > \|de_y\|$)
Bit 3,4	Steigungs-Sektor: 0 = flach, 3 = fast $45°$
Bit 0–2	Überdeckung in Anzahl gesetzter Subpixel - 1

dann können wir die Initialisierung der Tabelle durch die folgende Routine vornehmen. In TurnOnOrder[ ] sind dabei die ersten acht Subpixel der in Abbildungen 3.33 bis 3.36 dargestellten Reihenfolgen festgehalten.

```
for (left = 0, left < 2, left++)
  for (down = 0, down < 2, down++)
    for (steep = 0, steep < 2, steep++)
      for (sector = 0, sector < 4, sector++)
        for (coverage = 0, mask = 0, coverage < 8, coverage++)
         {
          NewSubpixel = TurnOnOrder[sector][coverage];
          if (steep)
            NewSubpixel = MirrorLeftDiagonal(NewSubpixel);
          if (left)
            NewSubpixel = MirrorLeftRight(NewSubpixel);
          if (down)
           NewSubpixel = MirrorUpDown(NewSubpixel);
          mask |= NewSubpixel;
          SubpixelMaskTab[i++] = mask;
         }
```

Die Mirror-Funktionen erledigen die Transformationen in die anderen Oktanten[12].

3.4.9.4 Generierung einer Subpixelmaske

Nachdem die Tabellen erstellt wurden, kann mit dem folgendem Codestück eine Maske erzeugt werden:

```
tPAAMask XpAASubpixelMaskGeneration(tPAAEdge *Edge)
 {
  tInt Slope    = Edge->SlopeForSubpixelmaskLookup;
  tInt Coverage = XpAACoverage(Edge);
  if (Coverage == 0)
    return(0x0000);
  if (Edge->E.fix > 0)
    Slope = ChangeSlopeDirection(Slope);
  return SubpixelMaskTab[Slope | Coverage-1];
 }
```

[12] In der Software-Implementierung des Subpixel-Verfahrens werden die Transformationen dadurch vereinfacht, daß ein intermediäres Format verwendet wird, in dem die Transformationen auf wenige boolesche Operationen zurückgeführt werden können. Die Berechnungen befinden sich im Quellcodemodul *XpAATab.c*.

Die Variable *Edge->SlopeForSubpixelMaskLookup* muß bei der Kanten-Initialisierung mit den im vorherigen Abschnitt erklärten fünf Bits left, down, steep und sector (2 Bit) geeignet gesetzt werden. Die Funktion *ChangeSlopeDirection()* negiert lediglich die Bits für left und down, wodurch der Inhalt der Ergebnismaske negiert wird. Die Berechnung der Überdeckung wird durch die folgende Funktion erledigt:

```
tInt XpAACoverage(tPAAEdge *Edge)
 {
  tFix E = Edge->E;
  if (E.fix < 0)
    E.fix = -E.fix;
  if (E.fix == E_HALFPIXEL)
    return 0;
  E.fix <<= 6;
  return (int)CoverageTab[Edge->DexForCoverageLookup | E.fp.i];
 }
```

Die hierbei verwendete Variable *Edge->DexForCoverageLookup* beinhaltet die benötigten Bits der Variablen de_x, die bereits an die passende Position geschoben wurden. Auch diese Berechnung kann bereits bei der Initialisierung der Kante durchgeführt werden.

Die in diesem Abschnitt vorgestellten Funktionen befinden sich in dem Quellcodemodul *XpAA-Mask.c*.

3.5 Komplexität und Speicherbedarf

Komplexität und Speicherbedarf entsprechen den Werten des Vier-Zeiger-Verfahrens. Diese sind in den Abschnitten 4.6 und 4.7 wiedergegeben.

3.6 Grenzen des Verfahrens

Das Subpixel-Verfahren kann nicht alle Alias-Effekte beseitigen. Die endgültige Abhandlung über die verbleibenden Artefakte des Subpixel-Verfahrens soll jedoch erst im Abschnitt *'Grenzen des Verfahrens'* beim Vier-Zeiger-Verfahren auf Seite 71 erfolgen. An dieser Stelle soll nur auf die Schwäche des Subpixel-Verfahrens eingegangen werden, die zur Entwicklung des Vier-Zeiger-Verfahrens führte.

Das Hauptproblem des Subpixel-Verfahrens ist die feste Zuordnung zwischen Subpixeln und Nachbarn. Jedes gesetzte Subpixel soll das Beimischen von Farbe aus einem *anderen* Dreieck bewirken. Unter bestimmten geometrischen Gegebenheiten, wie der in Abbildung 3.37 gezeigten, schlägt diese Zuordnung fehl. Ein oder sogar zwei Nachbarpixel, auf die gesetzte Subpixel

verweisen, gehören demselben Dreieck wie das Pixel selbst an. Der tatsächliche Anteil zugemischter fremder Farbe ist dadurch geringer als beabsichtigt.

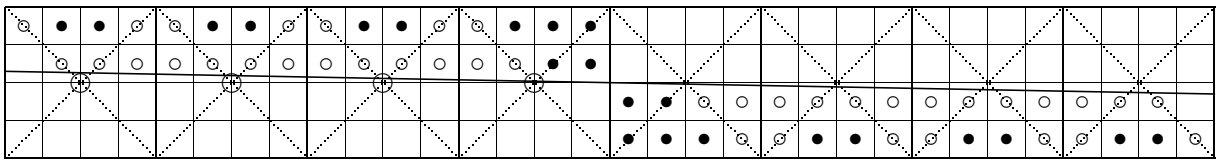


Abbildung 3.37: Fehler durch statische Zuordnung zwischen Subpixeln und Nachbarn. Die mit ∘ markierten Subpixel sollten eigentlich, genau wie die mit • markierten, zu einem Pixel auf der anderen Seite der Kante 'zeigen'. Durch die Flachheit der Linie liegen die entsprechende Nachbarpixel aber immer noch auf der selben Seite der Kante, so daß die mit ∘ markierten Subpixel nicht die Farbe des Dreiecks auf der anderen Seite der Kante, sondern die Farbe des eigenen Dreiecks repräsentieren. Da der Effekt bei den beiden mittleren Pixeln in entgegengesetzter Form auftritt, kommt es zu einer sprunghaften Intensitätsänderung entlang der Kante, die im fertigen Bild als Knick empfunden wird.

Die Abbildung 3.38 zeigt eine ähnliche Situation wie Abbildung 3.37, diesmal beträgt die Steigung der Linie rund 45^o. Daran daß die Zuordnung zwischen Subpixeln und Nachbarpixeln in diesem Beispiel annähernd korrekt ist, erkennt man, daß der beschriebene Effekt unter anderem von der Steigung der Kante abhängig ist.

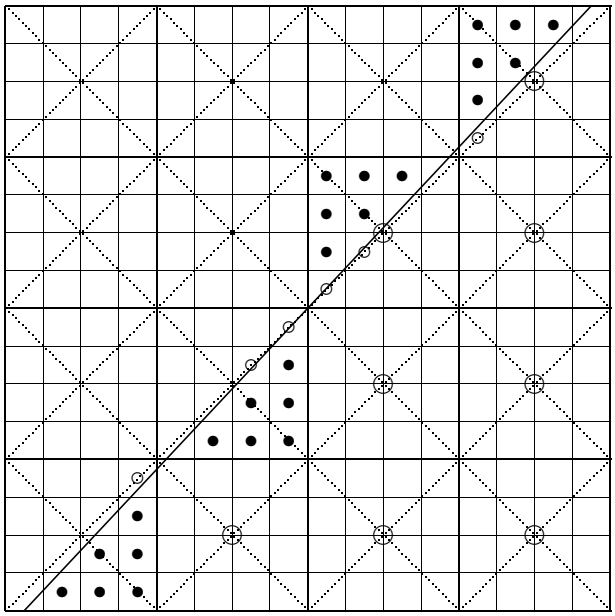


Abbildung 3.38: Der Fehler durch die statische Zuordnung ist bei einer Kante mit einer Steigung von rund 45^o am geringsten.

Die Bemühung, diesen Effekt zu vermeiden, führt zum Vier-Zeiger-Verfahren.

Kapitel 4

Das Vier-Zeiger-Verfahren

Wie im letzten Abschnitt erläutert wurde, besteht das Hauptproblem des Subpixel-Verfahrens in der statischen Zuordnung zwischen Subpixeln und Nachbarn. Diese Zuordnung soll jetzt mit einem neuen Konzept flexibler gestaltet werden.

4.1 Aufbau der Vier-Zeiger-Maske

Der Unterschied zwischen Subpixel-Verfahren und Vier-Zeiger-Verfahren besteht in einer Veränderung der zugrundeliegenden Maske. Die 'Vier-Zeiger-Maske' ist eine Art Subpixelmaske, bei der je vier Subpixel zu einem 'Meta-Subpixel' zusammengefaßt wurden. Zur Realisierung des Meta-Subpixels stehen uns damit vier Bit zur Verfügung, wenn wir mit der gleichen Speichermenge auskommen wollen. Diese vier Bit werden als vorzeichenlose Zahl interpretiert und dazu benutzt, die Überdeckung des Meta-Subpixels anzugeben. Das Meta-Subpixel selbst ist formlos und enthält im Gegensatz zu den Subpixeln des Subpixel-Verfahrens keine Information über die räumliche Lage der überdeckten Fläche.

Die Idee des Vier-Zeiger-Verfahrens besteht darin, die Überdeckung so zu skalieren, daß die Summe von vier maximal überdeckten Meta-Subpixeln die eigentliche Gesamtfläche eines Pixels übersteigt. Wenn eine 50 prozentige Überdeckung durch ein einziges Meta-Subpixel repräsentiert werden kann, versetzt uns das in die Lage, das Problem-Beispiel des Subpixel-Verfahrens korrekt zu lösen. Abbildung 4.1 zeigt, wie der Fall einer sehr flachen Kante bei Verwendung des Vier-Zeiger-Verfahrens gelöst wird.

Mit den jeweils vier Bit können wir Zahlen von 0 bis 15 darstellen, was wie gesagt einer Überdeckung von 50 Prozent entsprechen soll. Die korrekte Skalierung wäre also erreicht, würden wir jeder Überdeckungs-Einheit den Wert $50\% * \frac{1}{15} = \frac{1}{30}$ zuweisen. Dieser Wert ist in der Praxis eher unhandlich, weshalb in der Software-Implementierung des Vier-Zeiger-Verfahrens eine Überdeckungseinheit als $\frac{1}{32}$ definiert wurde. Diese Definition kommt der Architektur einer binär rechnenden Maschine eher entgegen. Außerdem ist es dadurch möglich, die Überdeckungstabellen des Subpixel-Verfahrens mitzuverwenden. Ist man bereit, den erhöhten Aufwand bei der

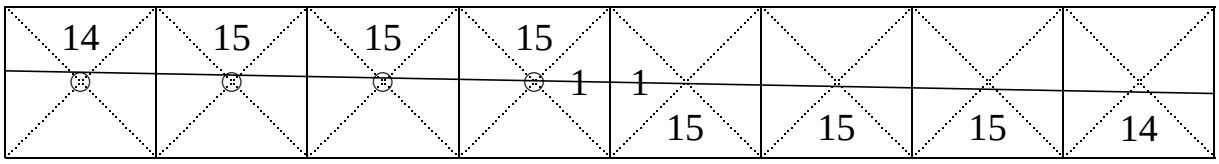


Abbildung 4.1: Korrekte Behandlung der flachen Kante beim Vier-Zeiger-Verfahren. Die Zahlen in den Quadranten der Pixel geben an, wieviel Farbe von dem entsprechenden Nachbarpixel zugemischt werden soll. Befindet sich in einem Quadranten keine Zahl, dann ist der Eintrag in dem entsprechenden Meta-Subpixel gleich 0, es wird also durch über dieses Meta-Subpixel keine Farbe zugemischt. An Stellen, wo nur ein einziges Nachbarpixel auf der anderen Seite der Kante liegt, wird die gesamte Überdeckung durch dieses eine Pixel bezogen. An anderen Stellen, an denen mehrere geeignete Nachbarpixel zur Verfügung stehen, wird die Überdeckung zwischen diesen Nachbarpixeln aufgeteilt. Das Aufteilung kann dabei so erfolgen, wie das durch eine Subpixelmaske implizit vorgegeben wäre, es sind aber auch andere Aufteilungen möglich. Jede Einheit eines Meta-Subpixels in der Abbildung hat den Wert $\frac{1}{32}$. Die Masken in der Mitte repräsentieren also eine Überdeckung von je 50%.

Nachbearbeitung zu tragen, kann selbstverständlich auch mit dem Wert $\frac{1}{30}$ gearbeitet werden. Die Verwendung des Wertes $\frac{1}{32}$ hat lediglich den Nachteil, daß in dem Fall einer waagerechten Kante, die genau durch den Pixelmittelpunkt geht, $\frac{1}{32}$ der Farbzumischung fehlt, da der Eintrag in dem Meta-Subpixel den Wert 16 hätte tragen müssen, dieser Wert aber nicht darstellbar ist. Als Ausgleich erhält man eine etwas erhöhte Auflösung im restlichen Bereich.

Als Datenstruktur für die neue Maske erhalten wir also wieder ein 16 Bit Wort, das in diesem Fall aus vier Meta-Subpixeln, auch Zeiger genannt, à vier Bit besteht.[1] Die Anordnung der Meta-Subpixel im Speicherwort ist unerheblich. Im Rahmen der Software-Implementierung wurde sie willkürlich festgelegt als {oben, rechts, unten, links}, wobei ‘oben’ die niederwertigsten Bits belegt.

4.2 Unterschiede gegenüber dem Subpixel-Verfahren

Die Unterschiede zwischen Subpixel- und Vier-Zeiger-Verfahren sind gering. Lediglich bei Berechnungen, die direkt einzelne Masken manipulieren, ergeben sich Abweichungen. Diese Abweichungen sollen in diesem Abschnitt erklärt werden. Der im vorherigen Kapitel vorgestellte Ablaufplan für das Subpixel-Verfahren (Seite 20) gilt ansonsten auch für das Vier-Zeiger-Verfahren.

[1] Der Vollständigkeit halber soll hier erwähnt werden, daß frühe Ansätze des Vier-Zeiger-Verfahrens auf einer etwas anderen Datenstruktur basierten. Anstelle der vier Meta-Subpixel wurden nur eine bzw. zwei solche Einheiten verwendet, zusammen mit jeweils einem Eintrag, auf welchen Nachbarn sie sich beziehen sollten. Wegen der Fähigkeit dieser Einheiten, eine variable Zuordnung zu realisieren, wurde der Begriff ‘Zeiger’ eingeführt — ein Begriff, der für die endgültige Version des Vier-Zeiger-Verfahrens eigentlich nicht mehr passend ist. Die Versuche mit ein oder zwei Zuordnungseinheiten erwiesen sich in vielen Situationen als unzureichend und führten zu einer Fülle von Fallunterscheidungen. Dieser Ansatz wurde deshalb nicht weiterverfolgt.

4.2.1 Nachbearbeitung

Die Nachbearbeitung beim Vier-Zeiger-Verfahren unterscheidet sich von der Subpixel-Version lediglich durch die Subpixel-Zählfunktion. Diese vereinfacht sich zu:

```
UpBits    = (Mask & UP_BITS)    << UP_SHIFT;
RightBits = (Mask & RIGHT_BITS) << RIGHT_SHIFT;
DownBits  = (Mask & DOWN_BITS)  << DOWN_SHIFT;
LeftBits  = (Mask & LEFT_BITS)  << LEFT_SHIFT;
OwnBits   = 32 - UpBits - RightBits - DownBits - LeftBits;
```

4.2.2 Verknüpfung von Masken

Auch beim Vier-Zeiger-Verfahren unterscheiden sich Hard- und Software-Implementierung was das Verknüpfen der Masken betrifft. Da bei der Hardware-Version das Verknüpfen von Masken auf Pixelbasis abgewickelt wird, entfällt der Großteil der hier vorgeführten Berechnungen (siehe Kapitel 6).

Die Verknüpfung der Masken der drei Kanten eines Dreiecks, die sich beim Subpixel-Verfahren mit Hilfe der XOR-Funktion so elegant lösen ließ, erfordert beim Vier-Zeiger-Verfahren zusätzlichen Aufwand. Zunächst müssen wir herausfinden, von welchem Typ die zu verknüpfenden Masken sind, dann müssen die Masken nach Typen unterschieden verknüpft werden.

Wie bereits erwähnt wurde, werden die Masken eines Spans kantenweise erzeugt. Eine Kante erzeugt dabei gleich alle Masken für den aktuellen Span auf einmal und speichert sie in dem sog. Verknüpfungs-Array zwischen. Dort werden die Masken sukzessive mit den Masken der anderen Kanten verknüpft. An der Verknüpfung sind also immer zwei Masken beteiligt: Eine neue Maske und eine, die sich bereits in dem Verknüpfungsarray befindet. In den meisten Fällen wird die Maske in dem Verknüpfungsarray leer sein, so daß sich die Verknüpfung erübrigt und die neue Maske direkt in das Verknüpfungsarray geschrieben werden kann.

Ist die Maske in dem Verknüpfungsarray nicht leer, müssen wir zunächst die Typen der beiden Masken bestimmen, um die richtige Verknüpfung auswählen zu können. Mit Typ ist wieder gemeint, ob die Maske ‘schwarz’ oder ‘rot’ ist. Analog der Definition auf Seite 29 wollen wir von einer ‘schwarzen Maske’ sprechen, wenn der Mittelpunkt des Pixels im Inneren der Halbebene der Kante liegt, im anderen Fall von einer ‘roten Maske’. Der Typ der neuen Maske läßt sich wie üblich mit Hilfe des Errorterms bestimmen. Die neue Maske ist schwarz, wenn ihr Errorterm $E > 0$ ist.[2]

Die einfachste Methode, den Typ der Maske im Verknüpfungsarray zu bestimmen, funktioniert so, daß jede Kante den Typ der von ihr erzeugten Masken zwischenspeichert. Dabei ist es nicht

[2]Im Fall $E = 0$ bleibt die Entscheidung dem Renderer überlassen. Will man sicherstellen, daß die Pixel von Dreiecken mit zwei gemeinsamen Eckpunkten wirklich lückenlos aneinandergrenzen, ohne daß Pixel mehrfach gesetzt werden, dann ist eine detailliertere Regelung unvermeidbar. Im Fall von Da*Render* wird beispielsweise ein Pixel mit $E = 0$ nur genau dann gesetzt, wenn sich die Halbebenen aller Kanten nach rechts erstrecken.

erforderlich, die Typen aller Masken einzeln zu speichern. Wie Abbildung 4.2 zeigt, zerfällt die Menge der Masken, die eine Kante für einen Span generiert, in zwei Intervalle. Es genügt somit, die Intervallgrenzen zu speichern.

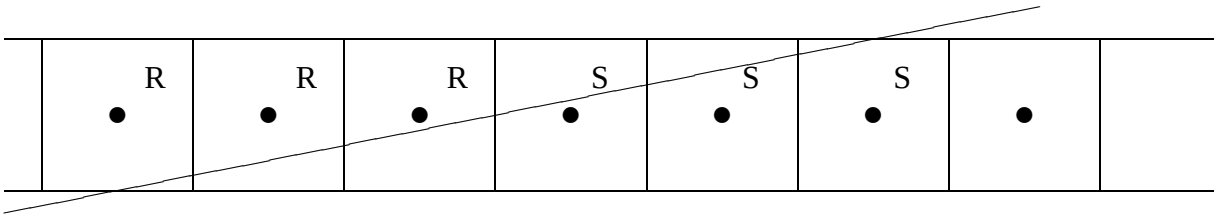


Abbildung 4.2: Eine den Span kreuzende Kante erzeugt genau ein Intervall von roten (R) und ein Intervall von schwarzen (S) Masken. Die beiden Intervall grenzen direkt aneinander an. In der Skizze befindet sich die Halbebene, in der das Dreieck liegt, unterhalb der Kante.

Wenn der Typ der beiden Masken bestimmt ist, können sie wie folgt verknüpft werden:

Typ Maske_1	Typ Maske_2	ErgebnisMaske
schwarz	schwarz	$\text{Maske}_1 + \text{Maske}_2$
rot	schwarz	$\text{Maske}_1 \ominus \text{Maske}_2$

Der Fall rot/rot tritt, wie bereits erwähnt wurde, nur an Eckpixeln auf und wird dort gesondert behandelt.

Der $\ominus$-Operator steht für eine modifizierte Subtraktion. Wie in Abschnitt 4.2.5 erklärt wird, werden manche Masken bei ihrer Generierung ‘gesplittet’ andere nicht. Splitten bedeutet das Aufteilen der Gesamtüberdeckung auf zwei Zeiger innerhalb der Maske. Wird eine gesplittete Maske von einer nicht gesplitteten subtrahiert oder umgekehrt, kann es zu einem Unterlauf kommen. Um das zu verhindern, müssen die Masken vor der Subtraktion gegebenenfalls durch das Zusammenfassen von Zeigern vorbereitet werden.

Wie bereits angemerkt wurde, ist die Maske im Verknüpfungs-Array in den meisten Fällen leer. Das ist darauf zurückzuführen, daß das Zusammentreffen mehrerer Kanten in einem Pixel überhaupt nur in den schmalen Bereichen z.B. im Bereich um die Eckpixel auftreten kann. In der Software-Implementierung beginnt die Bearbeitung daher damit, diesen Fall zu testen. Im Fall des Demobildes ‘Sphere.c’, das im Anhang wiedergegeben ist, werden weit über 90% aller Maskenverknüpfungen durch diese Abfrage erledigt.

Funktionen zur Masken-Arithmetik im Vier-Zeiger-Modell befinden sich im Quellcode-Modul *XpAAArit.c*, die beschriebenen Berechnungen zur Fallunterscheidung im Modul *XpAANMsk.c*.

4.2.3 Spezielle Behandlung der Eckpixel

Um die Masken von Kantenpixeln zu verknüpfen, ist die topologische Information einer Vier-Zeiger-Maske gerade noch ausreichend. Nicht so im Fall eines Eckpixels. Die Erzeugung einer

Eckpixelmaske kann nicht, wie das bei Subpixelmaske möglich war, durch sukzessives Verknüpfen einzelner Masken erfolgen, da die nötige Information über die Lage der Kanten nicht mehr in den Masken enthalten ist. Folgende Ansätze zur Erzeugung der Eckpixelmasken sind denkbar:

Tabelle: Die Vier-Zeiger-Maske wird einer Tabelle entnommen, die als Eingabe Position und Steigung der beteiligten Kanten erhält. Nachteil dieses Ansatzes ist die (zu) große Anzahl an Freiheitsgraden. Die Anzahl der Freiheitsgrade nimmt nochmals zu, wenn die Möglichkeit mit in Betracht gezogen wird, daß mehrere Ecken in ein und dasselbe Pixel fallen.

Näherungslösung: Die Masken werden so verknüpft als handele es sich um Kantenpixel. Etwaig auftretende Fehler werden in Kauf genommen (Abbildung 3.18 auf Seite 32 illustriert die Art des gemachten Fehlers). Der Fehler läßt sich durch eine geringe Anzahl von Fallunterscheidungen etwas einschränken.

Subpixelmasken: Alle beteiligten Masken werden zunächst als Subpixelmasken erzeugt und, wie auf Seite 32 erklärt, konjunktiv verknüpft. Erst dann wird das Ergebnis in eine Vier-Zeiger-Maske umgewandelt.

Die Software-Implementierung des Vier-Zeiger-Verfahrens verwendet den letzten der genannten Ansätze. Das hatte sich angeboten, da das Subpixel-Verfahren ebenfalls im Rahmen dieser Implementierung verwirklicht worden war, der benötigte Programmcode also bereits vorlag.

4.2.4 Zurücksetzen ‘interner’ Zeiger

Das Zurücksetzen von Maskenteilen, die vollständig innerhalb des neuen Dreiecks liegen, entspricht prinzipiell dem Vorgehen beim Subpixel-Verfahren, es unterscheiden sich lediglich die beim Löschen verwendeten Bitmasken. Außerdem spart man das separate Löschen von ‘Subpixeln mit zwei Nachbarn’, da diese beim Vier-Zeiger-Verfahren nicht mehr vorkommen.

4.2.5 Generierung der Vier-Zeiger-Masken

Wie beim Subpixel-Verfahren teilt sich die Generierung der Masken in die Berechnung der Überdeckung und die Erzeugung der Maske auf.

4.2.5.1 Auflösung der Überdeckungstabelle

Die Berechnung der Überdeckung ist identisch mit der entsprechenden Berechnung beim Subpixel-Verfahrens. Die einzigen Abweichung besteht darin, daß das Vier-Zeiger-Verfahren über

ein Bit mehr darstellbarer Auflösung verfügt[3], also von einer höheren Tabellenauflösung profitieren kann. Da die Tabelle zweidimensional ist, bedeutet das, daß die vergrößerte Tabelle viermal soviele, und zwar 1024 Einträge belegt. Darüber hinaus sind die Einträge ein Bit länger. Vergleicht man allerdings die Genauigkeit von Subpixel-Verfahren und Vier-Zeiger-Verfahren bei Verwendung von Tabellen gleicher Größe, dann schneidet das Vier-Zeiger-Verfahren allein schon wegen der höheren Präzision der Tabelleneinträge besser ab. Die Vergrößerung der Tabellenauflösung sollte deshalb nur dann durchgeführt werden, wenn dies zu sichtbaren Verbesserungen führt und einfach zu realisieren ist. Wegen der 'Gutartigkeit' der Überdeckungsfunktion, kann die höhere Auflösung auch durch lineare Interpolation erreicht werden.

4.2.5.2 Splittung von Vier-Zeiger-Masken

Die Generierung einer Vier-Zeiger-Maske gestaltet sich deutlich einfacher als die Generierung einer Subpixelmaske, da die Vier-Zeiger-Maske nur über wenig interne Struktur verfügt. Prinzipiell wäre es möglich, die errechnete Überdeckung durch einen einzigen der vier Zeiger zu repräsentieren. Die Maske wäre dann nichts weiter, als das Ergebnis der Überdeckungs-Berechnung an die entsprechende Bitposition innerhalb des Speicherwortes geschoben. Um die Maske robuster gegen Problemfälle wie schmale Zwischenräume und schmale Überdeckungen zu machen, ist es jedoch sinnvoller, die Gesamtüberdeckung auf zwei Zeiger aufzuteilen, sofern dies möglich ist. Dieser Vorgang soll im weiteren 'splitten' genannt werden. In welchem Verhältnis die Überdeckung auf die zwei Zeiger aufgeteilt wird, hängt von der Steigung der Kante ab.

Um die weiteren Betrachtungen begrifflich besser fassen zu können, sollen zwei der vier Zeiger-Richtungen gesondert benannt werden. Abbildung 4.3 zeigt eine Skizze, die die Begriffe verdeutlichen soll.

Def: Haupt- und Neben-Mischrichtung

Die Haupt-Mischrichtung ist die Richtung, die ins Innere der Halbebene zeigt und dabei möglichst senkrecht zu der Kante steht. Die Neben-Mischrichtung ist die Richtung, die ins Innere der Halbebene zeigt und dabei einen möglichst spitzen Winkel mit der Kante bildet.

Beim Splitten einer roten Maske wird die Gesamtüberdeckung auf Haupt-Mischrichtung und Neben-Mischrichtung aufgeteilt, beim Splitten einer schwarzen Maske auf die dazu entgegengesetzten Richtungen. Der Hauptanteil entfällt stets auf die Haupt-Mischrichtung (bzw. deren Komplement). Beträgt die Steigung der betrachteten Kante ein Vielfaches von 90^{o}, dann wird die gesamte Überdeckung der Haupt-Mischrichtung zugewiesen, beträgt sie $n * 90^{\mathrm{o}} + 45^{\mathrm{o}}$ sind die Anteile von Haupt- und Neben-Mischrichtung gleich groß. Abbildung 4.4 zeigt die entsprechenden Skizzen.

[3] Der Wert eines Subpixels beim Subpixel-Verfahren beträgt $\frac{1}{16}$, der Wert einer Einheit im Meta-Subpixel $\frac{1}{30}$ bzw. $\frac{1}{32}$.

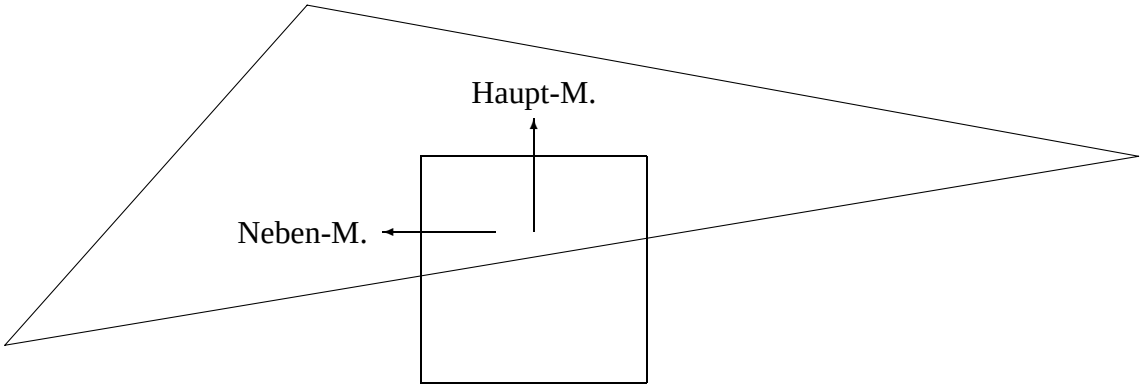


Abbildung 4.3: Haupt- und Neben-Mischrichtung in Bezug auf die untere Kante. Die Haupt-Mischrichtung ist die Richtung, die direkt ins Innere der Halbebene zeigt. Die Neben-Mischrichtung zeigt ebenfalls in Richtung des Inneren der Halbebene, bildet jedoch einen spitzen Winkel zur Kante.

Der Anteil der Neben-Mischrichtung sollte bei Winkeln zwischen 45^{o} und 90^{o} stetig und monoton abnehmen, es sind aber prinzipiell verschiedene Funktionen möglich, die den Verlauf beschreiben. Beispielsweise ließe sich die Charakteristik des Subpixel-Modells approximieren. Da die genaue Form der Funktion nur wenig Einfluß auf die resultierende Bildqualität besitzt, wurde in der Software-Version des Vier-Zeiger-Verfahrens ein diskretes Modell gewählt, das nur vier Werte enthält. Die Aufteilung kann dabei durch je eine Schiebeoperation und eine Subtraktion berechnet werden.

4.2.5.3 Wann wird gesplittet?

Würden wir alle Masken splitten, wären wir genauso weit wie beim Subpixel-Verfahren. Es sind nämlich genau die Neben-Mischrichtungen bzw. ihre Komplemente die u.U. fälschlicherweise auf Pixel zeigen, die sich auf der selben Seite der Kante befinden. Genau diese Fälle müssen wir abfangen, um tatsächlich eine Verbesserung gegenüber dem Subpixel-Verfahren zu erhalten. Abbildung 4.5 zeigt das Beispiel aus Abbildung 4.1 auf Seite 57. Diesmal wurden jedoch alle Masken gesplittet, was zu einer fehlerhaften Behandlung der Kante führt. Der Effekt ist bei Kanten mit größerer Steigung noch deutlich ausgeprägter, da der Betrag der Zumischung aus der Neben-Mischrichtung dabei entsprechend größer ist.

Die Berechnung, welche Masken gesplittet werden dürfen und welche nicht, gestaltet sich aufgrund der Linearität der Kantenfunktion recht einfach. Um festzustellen, ob das Pixel in Neben-Mischrichtung innerhalb der Halbebene der aktuellen Kante liegt oder nicht, genügt eine Addition. Der Errorterm des aktuellen Pixels ist bekannt, wir brauchen also lediglich $\pm de_x$ bzw. $\pm de_y$ zu addieren, um den Errorterm des Pixels in Neben-Mischrichtung zu bestimmen. Ob dieses Pixel zur Halbebene gehört, ergibt sich wie üblich aus dem Vorzeichen seines Errorterms. Stellen wir nun fest, daß der Errorterm im Nachbarpixel und der Errorterm des aktuellen Pixels das gleiche Vorzeichen haben, dann wissen wir, daß wir nicht splitten dürfen, da das Pixel in

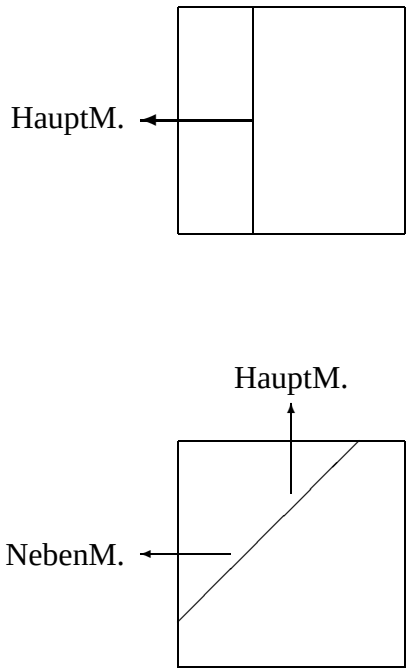


Abbildung 4.4: Haupt-Mischrichtung und Neben-Mischrichtung bei 90^o und bei 45^o. Bei einer Kantensteigung von 90^o muß die gesamte Überdeckung einem einzigen Zeiger zugewiesen werden. Stellt man sich die Kante in der linken Skizze in beide Richtungen verlängert vor, sieht man, daß das linke Nachbarpixel das einzige geeignete ist. Bei einer Kantensteigung von 45^o sind immer zwei Nachbarpixel geeignet. Das gleichmäßige Aufteilen der Überdeckung minimiert die Artefakte, die auftreten können, wenn der Mittelpunkt eines der beiden Nachbarpixel durch ein sehr schmales Dreieck überdeckt wird.

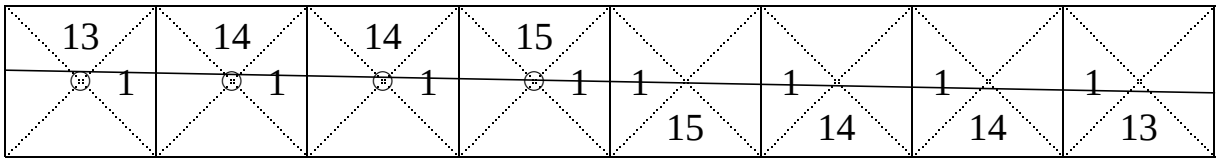


Abbildung 4.5: Inkorrekte Behandlung einer Kante bei Splittung aller Masken. Die sechs äußeren Masken hätten nicht gesplittet werden dürfen, da die Neben-Mischrichtung auf ein Pixel zeigt, das auf der selben Seite der Kante liegt. Die korrekte Lösung ist in Abbildung 4.1 wiedergegeben.

Neben-Mischrichtung auf der gleichen Seite der Kante liegt.

Die verwendeten Größen Haupt- und Neben-Mischrichtung, der Faktor mit dem die Gesamtüberdeckung aufgeteilt wird, sowie die Werte, mit denen die Überdeckungswerte an ihre Positionen innerhalb der Vier-Zeiger-Masken geschoben werden sind allesamt über eine Kante konstant und können bei der Initialisierung des Dreiecks bzw. der Kante vorberechnet werden.

4.3 Erweiterung: Zeigerumlenkung

Eines der Hauptprobleme des Pointsamplings und damit auch des Vier-Zeiger-Verfahrens sind schmale Dreiecke (siehe auch Abschnitt *'Grenzen des Verfahrens'*). Abbildung 4.6 zeigt ein solches Dreieck, von dem aufgrund der Unterabtastung ein Teil nicht sichtbar ist. Das Dreieck ist so schmal, daß es über einen Bereich von mehreren Pixeln hinweg keinen Pixelmittelpunkt trifft.

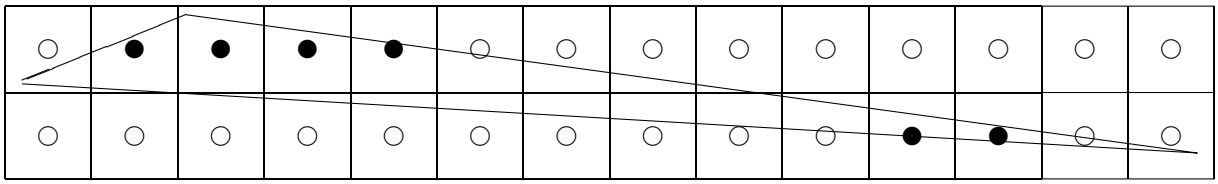


Abbildung 4.6: Schmales Dreieck mit 'Löchern'. In dem mittleren Bereich überdeckt das Dreieck keine Pixelmittelpunkte, es werden deshalb in diesem Bereich keine Pixel mit der Farbe des Dreiecks geschrieben. Da das Vier-Zeiger-Verfahren Farbe immer nur zwischen benachbarten Pixeln übertragen kann, ist seine Reichweite auf einen Umkreis von einem Pixel beschränkt. Die Pixel in den drei mittleren Spalten des Lochs können also nicht korrigiert werden, weil keines von ihnen über einen Nachbarn mit der Farbe des Dreiecks verfügt. Die beiden Pixel am Rand des Lochs können jedoch mit Hilfe der 'Zeigerumlenkung' korrigiert werden.

Da das Vier-Zeiger-Verfahren nur lokale Nachbearbeitungen durchführt, nämlich Verknüpfungen mit den direkten Nachbarn, kann es dieses Problem nur teilweise lösen. Trotzdem können wir die Anzahl der korrekten Pixel maximieren, indem wir das Verfahren so modifizieren, daß wenigstens die Pixel mit mindestens einem gesetztem Nachbarpixel vollständig erfaßt werden. Um das zu bewerkstelligen müssen wir Zeiger 'umlenken'. In Abbildung 4.7 ist ein solcher Fall dargestellt.

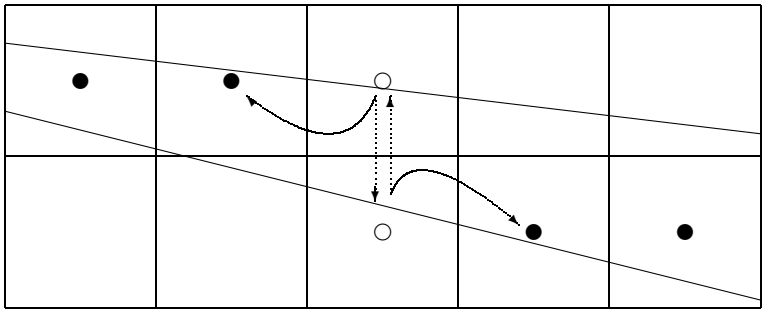


Abbildung 4.7: Umlenkung von Zeigern. Die beiden Pixel in der mittleren Spalte würden normalerweise ein sichtbares 'Loch' verursachen, da das Nachbarpixel in der jeweiligen Haupt-Mischrichtung nicht gesetzt wurde. Die Überdeckung kann jedoch von dem linken bzw. rechten Nachbarpixel gedeckt werden, indem die Zeiger entsprechend umgelenkt werden.

Die schraffierten roten Zeiger im Bild würden normalerweise verworfen, da das Pixel, auf das sie zeigen, nicht gesetzt wurde. Genau das war nämlich das im Abschnitt *'Visibilität'* (siehe

Seite 36) definierte Kriterium dafür, daß eine rote Maske nicht sichtbar ist. Wir wollten annehmen, daß das neue Dreieck einem bereits dargestellten Dreieck untergeschoben wurde und die Bearbeitung der Kante bereits von dem anderen Dreieck erledigt worden war. Diese Annahme ist in diesem Fall jedoch nicht gerechtfertigt, die Maske wird sehr wohl benötigt, nur befindet sich in der vorgesehenen Richtung kein geeignetes Pixel, um die gesuchte Farbinformation zu beziehen. Wir werden die Maske also trotzdem zurückschreiben, nur muß der Zeiger vorher auf ein geeignetes Pixel umgelenkt werden. Nur wenn kein geeignetes Pixel zu finden ist, werden wir die Maske verwerfen.

Ab dem Punkt, an dem wir feststellen, daß das Pixel in der Haupt-Mischrichtung der roten Maske nicht gesetzt ist, ergibt sich folgender Ablauf:

Zeigerumlenkung

- Stelle fest, ob das Pixel in Haupt-Mischrichtung zum Dreieck gehört. Falls ja, brich ab, denn es handelt sich um eine Überdeckung.
- Stelle fest, ob es ein anderes Pixel gibt, auf das umgelenkt werden kann. Dieses Pixel muß sich in der Neben-Mischrichtung der betrachteten Kante befinden. Falls dieses Pixel nicht gesetzt wurde, brich ab.
- Mache eine eventuelle Splittung rückgängig durch Aufsummieren der umzulenkenden Maske.
- Schiebe die erhaltene Überdeckung so, daß sie eine Maske ergibt, die auf das geeignete Nachbarpixel zeigt.
- Schreibe die Maske in den Framebuffer.

Demobilder, die den Effekt der Zeigerumlenkung belegen, befinden sich im Anhang.

Alle sich aus dem Umlenkungs-Ansatz ergebenden Modifikationen sind auf die Routine beschränkt, die die Masken zum Framebuffer zurückschreibt. In der Software-Implementierung des Vier-Zeiger-Verfahrens befindet sich diese Funktion im Sourcemodul *XpAAWrit.c*. Die Zeigerumlenkung läßt sich, wie auf Seite 76 beschrieben, mit einem Präprozessorkommando an- und abschalten.

4.4 Erweiterung: Akkumulation

Das im Abschnitt *'Visibilität'* vorgestellte Entscheidungsverfahren zur Sichtbarkeit von Subpixelmasken bzw. Vier-Zeiger-Masken kommt schnell an seine Grenzen, wenn die betrachteten Dreiecke sehr klein werden. Wie die Zeigerumlenkung ist auch die Akkumulation eine Erweiterung des Vier-Zeiger-Verfahrens, um Dreiecksfragmente, die sonst unsichtbar wären, sichtbar zu machen. Das Prinzip der Akkumulation ist wesentlich weitreichender, hat jedoch einen stark heuristischen Charakter.

Die Idee ist folgende: Dreiecksfragmente, bei denen der Umzulenkversuch erfolglos war, da es kein geeignetes Nachbarpixel gab, werden der Maske des nächstbesten Dreiecks zugeschlagen.

Wir tun dies in der Erwartung, das Fragment gehöre zu demselben ‘Objekt’ auf Modellierungsebene wie das Dreieck und habe somit die gleiche oder zumindest eine sehr ähnliche Farbe. Das Objekt könnte beispielsweise eine Freiformfläche sein, die beim Rendern in sehr viele, sehr kleine Dreiecke zerlegt wurde. In dem Kantenbereich zwischen Freiformfläche und Hintergrund ist es dann wesentlich günstiger, die Fragmente einem anderen Dreieck der Freiformfläche zuzuschlagen, als sie zu verwerfen. Ein Fragment zu verwerfen ist gleichbedeutend damit, es dem falschen Objekt, in diesem Fall dem Hintergrund zuzuschlagen.

Beim Zurückschreiben eines Dreiecksfragments zum Framebuffer müssen folgende Fälle unterschieden werden:

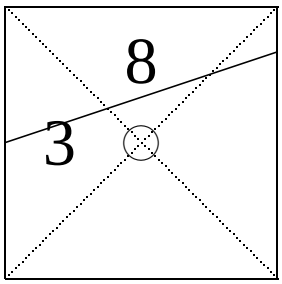
- Befindet sich in dem Pixel bereits eine positive Maske, dann wird das Fragment dieser Maske zugeschlagen.
- Ist der Wert der Maske des Pixels gleich 0, wird das Fragment negiert und zurückgeschrieben. Die Negativität der Maske markiert diese als ‘herrenloses’ Dreiecksfragment, das der nächsten Maske, die in dieses Pixel geschrieben wird, zugeschlagen werden soll.
- Befindet sich in dem Pixel eine negative Maske, dann handelt es sich um ein anderes Fragment. Nachdem das Fragment in der Maske wieder positiv gemacht wurde, werden beide Fragmente addiert, negiert und zurückgeschrieben.

Darüber hinaus müssen natürlich auch die Routinen zum Schreiben der normalen Masken geändert werden. Wann immer eine Maske zurückgeschrieben wird, muß die eventuell vorhandene negative Maske der neuen Maske zugeschlagen werden.

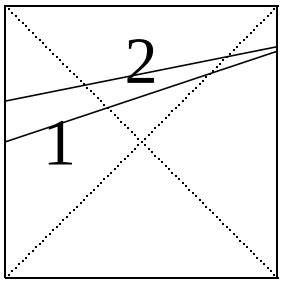
Als nächstes müssen wir den Ausdruck *‘ein Fragment einer Maske zuschlagen’* genauer definieren. Im Fall einer schwarzen Maske bedeutet dies, das Fragment zu subtrahieren, im Falle einer roten Maske zu addieren. Die Abbildungen 4.8 und 4.9 zeigen Beispiele für diese Verknüpfungen.

Falls wir ein Fragment auf eine Maske im Framebuffer zurückschreiben, besteht dabei das Problem, den Typ der Maske im Frame zu bestimmen. Die Software-Implementierung des Vier-Zeiger-Verfahrens verfügt über eine Routine, die mit Hilfe verscheidener heuristischer Analysen recht zuverlässige Vorhersagen macht. Um eine in allen Fällen korrekte Bearbeitung sicherzustellen, ist es jedoch notwendig, ein zusätzliches Bits mit der entsprechenden Information in den Framebuffer einzufügen.

Bereits bei der Sichtbarkeitsbetrachtung der roten Kanten hatten wir über die Gültigkeit des Z-Buffers hinaus extrapoliert. Nun geht es um die Sichtbarkeit von Fragmenten, die unter Umständen überhaupt keinen Pixelmittelpunkt überdecken, von denen also gar keine Sichtbarkeitsinformation vorliegt. Im Rahmen der Software-Implementierung wurden mit folgendem, besonders simplen Ansatz gute Erfahrungen gemacht. In einem globalen Flag wird gespeichert, ob die letzte zurückgeschriebene Maske sichtbar war oder nicht. War sie sichtbar, nehmen wir an, das aktuelle Fragment sei auch sichtbar, im anderen Fall nicht. Es kann dabei sehr gut



−



=

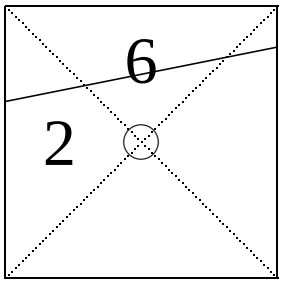
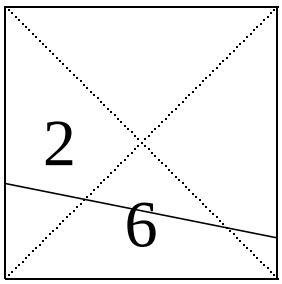


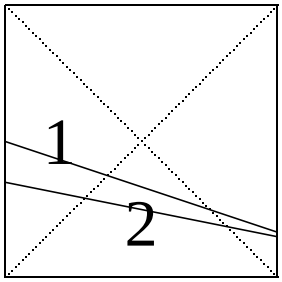
Abbildung 4.8: Eine schwarze Maske und ein Fragment werden subtrahiert. Durch die Subtraktion verringern sich die Werte der Zeiger der schwarzen Maske, es wird also umso mehr Farbe vom Pixel selbst bestritten. Das entspricht dem Hinzufügen des Fragments zu der Farbe der schwarzen Maske.

vorkommen, daß die letzte Maske nicht zu dem aktuellen Dreieck, sondern zu dem vorherigen Dreieck gehörte. Das Verfahren funktioniert trotzdem gut, da der zugrundeliegende Renderer Objektfragmente mit einem gewissen Maß an Ordnung erzeugt. Zwei aufeinanderfolgende Dreiecke liegen dabei auch im Objektraum meist benachbart, so daß die Visibilität der beiden Dreiecke eine hohe Korrelation aufweist. In Verwendung mit einem anderen Renderer mag die hier beschriebene Sichtbarkeitsdetermination schlechtere Ergebnisse liefern.

Negierte Masken, die zu Beginn des Nachbearbeitungs-Durchgangs im Framebuffer stehen, könnten an sich gelöscht werden. Im Rahmen der Software-Implementierung wurde jedoch der Ansatz verfolgt, sie wieder positiv zu machen und wie eine normale rote Maske nachzubearbeiten. Der dabei entstehende durchschnittliche Fehler kann nicht größer werden als der Fehler, der



+



=

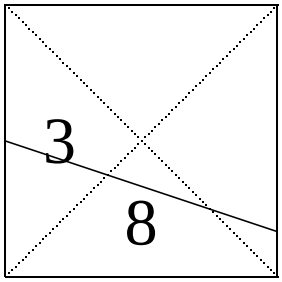


Abbildung 4.9: Eine rote Maske und ein Fragment werden addiert. Durch die Addition vergrößern sich die Werte der Zeiger der roten Maske, es wird also mehr Farbe von dem Nachbarpixel zugemischt. Das entspricht dem Hinzufügen des Fragments zu dem Nachbarpixel.

beim Löschen der Masken entsteht, in einiger Fällen wird sich das Resultat jedoch verbessern.

Demobilder, die den Effekt der Akkumulation belegen, befinden sich im Anhang.

Alle sich aus dem Akkumulations-Ansatz ergebenden Modifikationen sind auf die Sourcemodule *XpAAWrit.c* (Zurückschreiben der Masken) und *XpAAPost.c* (Nachbearbeitung) beschränkt. Wie die Zeigerumlenkung läßt sich auch die Akkumulation mit einem Präprozessorkommando an- und abschalten (siehe Seite 76).

4.5 Erweiterung: Maskenfusion

Sowohl beim Subpixel-Verfahren als auch beim Vier-Zeiger-Verfahren entsteht ein geringer Fehler an schattierten oder texturierten Objekten. Der Fehler beruhrt darauf, daß die zugemischte Farbe vom Nachbarpixel nicht mit der Farbe identisch ist, die das Dreieck in diesem Pixel hätte. Siehe dazu Abbildung 4.10.

Abbildung 4.10: Die zugemischte Farbe bei schattierten Objekten ist nur annähernd korrekt. Das linke Bild zeigt einen Ausschnitt eines schattierten Dreiecks. Da der Mittelpunkt des rechten Pixels nicht mehr im Dreieck liegt, wird das Pixel nicht gesetzt. Stattdessen wird die Zumischung in einem entsprechenden Zeiger vermerkt (Bild mitte). Nach Vollendung des Nachbearbeitungsprozesses (Bild rechts) beinhaltet das Kantenpixel eine Mischung aus seinem linken Nachbarn und dem Hintergrund. Die Farbe des Nachbarpixels ist jedoch etwas heller, als die Farbe des nicht gesetzten Fragmentes, so daß die endgültige Farbe des rechten Pixels etwas zu hell aufällt.

Im Inneren eines schattierten Objektes können durch die Nachbearbeitung schwach sichtbare Schlieren auftreten. Die Stärke des Effektes ist proportional zu dem Farbunterschied an der Kante. Außer bei hochfrequenten Texturen sind die Farbunterschiede jedoch zu gering, um vom Betrachter wahrgenommen zu werden.

Eine mögliche Lösung des Problems besteht darin, nur an Kanten zwischen ‘Objekten’ zu mischen, nicht aber an Kanten innerhalb eines ‘Objektes’. Ein Objekt ist dabei eine Einheit auf Modellierungsebene, deren Teile gleiche Farbe und Oberflächenbeschaffenheit aufweisen und komplanar und nahtlos aneinander angrenzen.[4] Innerhalb eines Objektes existieren also keine sichtbaren Kanten.

Wie ist es aber möglich, Kanten innerhalb eines Objektes von solchen zwischen Objekten unterscheiden? Der Maskenfusions-Ansatz geht folgendermaßen vor: Wenn beim Schreiben einer Maske festgestellt wird, daß sie nahtlos zu der Maske im Framebuffer paßt, liegt die Vermutung nahe, daß es sich auch im dreidimensionalen um angrenzende Dreiecke handelt. Jetzt müßte getestet werden, ob die beiden Dreiecke zu ein und demselben Objekt gehören, was leider nicht möglich ist, da derartige Informationen auf der unteren Renderingebene im allgemeinen nicht

[4] Sinnvollerweise sollte man weiterhin fordern, daß sich das Objekt nicht selbst überdecken oder durchdringen darf.

mehr verfügbar sind.[5] Da es uns aber lediglich um das Vermeiden der Schlieren zwischen schattierten Dreiecken eines Objektes geht, reicht es, die Bedingung

$$\Delta\text{Farbe} < \epsilon$$

zu prüfen. Passen die Masken zusammen *und* ist die Farbdifferenz hinreichend klein, wollen wir die beiden Dreiecke zu einem zusammenfügen, indem wir die Masken an der Schnittkante löschen. Die aufgrund dieser Berechnungen erzeugten Annahmen über Objektzusammengehörigkeiten mögen natürlich im Einzelfall falsch sein, was aber aufgrund der Kleinheit von ϵ zu keinen nennenswerten sichtbaren Fehlern führt. Das Verfahren ließe sich u.U. durch die Ausnutzung von Kohärenzen innerhalb einer Kante noch verbessern.

Um das Zusammenpassen der Masken eines Objektes auch tatsächlich zu gewährleisten, ist es notwendig, die Maskengenerierung auf die von Schilling [Sch91] beschriebene Art und Weise zu erweitern. Bei dem von Schilling verfolgten Ansatz werden zusätzliche Bits aus der Mantisse des Errorterms ausgewertet, um sicherzustellen, daß die Überdeckungen von zusammengehörigen Dreiecksfragmenten sich immer zu eins ergänzen. Die Tabellen zur Überdeckungsberechnung werden dadurch entsprechend größer.

Aufgrund der extrem geringen Auffälligkeit der beschriebenen Fehler, wurde die Maskenfusion nicht in die Software-Version aufgenommen.

4.6 Komplexität

Die einzelnen Teilaufgaben und ihre Komplexitäten

Komplexität	**Prozeß**
$\mathrm{O}(n*(b+h))$	Generieren, Verknüpfen und Zurückschreiben der Masken
$\mathrm{O}(n*b*h)$	Löschen der Masken im Dreiecksinneren
$\mathrm{O}(x*y)$	Nachbearbeitung aller Pixel im Framebuffer

Mit h, b und n Höhe, Breite und Anzahl der Dreiecke, x und y horizontale und vertikale Auflösung des Bildes. Das Rendern der Pixel ist von der gleichen Komplexität wie das Löschen der Masken im Dreiecksinneren. Als Gesamtkomplexität ergibt sich

$$\begin{aligned} & \mathrm{O}(\max(n*\max(b*h, b+h), x*y)) \\ = \ & \mathrm{O}(\max(n*(b+1)*(h+1), x*y)) \\ = \ & \mathrm{O}(\max(n*b*h, x*y)) \end{aligned} \tag{4.1}$$

Das Subpixel-Verfahren fällt also in eine Komplexitätsklasse, die kleiner ist als die von Supersampling ($\mathrm{O}(n*b*h*p)$ mit p = Anzahl der Durchgänge) oder dem A-Buffer von Carpenter

[5] Das A-Buffer-Verfahren verwendet zu genau diesem Zweck sogenannte Object-Tags, also Objekt-Bezeichnungs-Codes, mit deren Hilfe Objektteile beim Rendern bereits wieder zusammengefaßt werden können.

(O$(max(n\log(n) * b * h, x * y))$). Der logarithmische Anteil beim A-Buffer entsteht durch das Bearbeiten der verketteten Listen, vorausgesetzt, diese werden in einer günstigen Datenstruktur verwaltet.

4.7 Speicherbedarf

Die Software-Implementierung hat folgenden Speicherbedarf:

Element	**Größe**
Array der Vier-Zeiger-Masken	$x * y * 16$Bit
Array zum Zwischenspeichern von Farbeinträgen bei der Nachbearb.	$x * 24$Bit
Boolesche Arrays zum Vermerken, welche Pixel geschrieben wurden	$2x * 1$Bit
Array zum Verknüpfen von Vier-Zeiger-Masken	$x * 16$Bit
Array zum Aufbewahren verzögerter roter Vier-Zeiger-Masken	$x * 16$Bit

Mit x und y horizontale und vertikale Auflösung des Bildes. Das zweite Array wird nicht gleichzeitig mit den drei letzten Arrays benötigt, es kann also in denselben Speicherbereich gelegt werden.

Bei der Hardware-Realisierung entfallen die letzten zwei Arrays, da alle Verknüpfungen auf Masken- und nicht auf Spanbasis durchgeführt werden (siehe Kapitel *'Hardware-Realisierung des Vier-Zeiger-Verfahrens'*).

4.8 Grenzen des Verfahrens

Im Gegensatz zu einem analytischen Verfahren, wie dem Areasampling nach Catmull, gelingt es mit dem Vier-Zeiger-Verfahren nicht, alle Artefakte zu beseitigen. Im folgenden sollen die Situationen aufgezählt werden, in denen das Verfahren fehlerhafte Ergebnisse liefert:

1. Bei schattierten oder texturierten Dreiecken ist die Farbe der Kantenpixel nicht völlig korrekt. Im Abschnitt 4.5 findet sich eine nähere Beschreibung des Effektes. Da der Fehler praktisch nicht wahrnehmbar ist, ist keine Korrektur erforderlich.

2. Schmale Dreiecke und schmale Zwischenräume können durch das zugrundeliegende Pointsampling verloren gehen, bzw. Löcher aufweisen. Dieses Problem wurde im Abschnitt 4.3 bereits beschrieben. Lösungsansätze wurden in den Abschnitten *'Zeigerumlenkung'* und *'Akkumulation'* vorgestellt. Isolierte lange schmale Objekte, wie beispielsweise ein Fahnenmast, können jedoch nicht vollständig korrigiert werden, wenn sie größere Bereiche enthalten, in denen kein Pixel gesetzt wurde.

3. Das Verfahren hat nur eine endliche Auflösung. Da Farbinformation immer nur zwischen direkt benachbarten Pixeln ausgetauscht werden kann, stehen für ein Pixel nie mehr als

fünf Farbeinträge zur Verfügung. Pixel, die aus mehr als fünf verschiedenfarbigen Fragmenten zusammengesetzt werden, werden also mit Sicherheit nicht korrekt wiedergegeben. In vielen Fällen werden aber bereits Pixel mit deutlich weniger als fünf Fragmenten nicht mehr korrekt sein. Das Verfahren wird damit im Bereich der Kanten großer Polygone sehr gute Ergebnisse liefern, nicht aber an der Schnittkante zweier hochaufgelöster Freiformflächen.

4. Durch das zugrundeliegende Z-Buffering wird die Frage der Visibilität im Subpixelbereich nicht immer korrekt gelöst. Schilling stellt in dem Artikel über das EXACT-Verfahren [SS93] einen Lösungsansatz zu diesem Problem vor, der jedoch in Relation zu dem auf Effizienz ausgerichteten Vier-Zeiger-Verfahren unverhältnismäßig viel Aufwand erfordert.

5. Durchdringungen werden nicht nachbearbeitet. Dieser Effekt beruht darauf, daß Dreiecke nur an ihrer Umrandung nachbehandelt werden, die durch Durchdringungen entstehenden Kanten aber nicht zu den Umrandungen der Dreiecke gehören. In Abschnitt 3.4.6 wird ein Ansatz zur Lösung dieses Problems vorgestellt.

6. Bei der Akkumulation entstehen Fehler, wenn der Typ der Maske im Framebuffer falsch erkannt wird (siehe Seite 4.4). Dieses Problem kann mit Hilfe eines zusätzlichen Bits in allen Einträgen des Framebuffers gelöst werden, in dem der Typ der zuletzt geschriebenen Maske explizit vermerkt wird.

Kapitel 5

Implementierung des Vier-Zeiger-Verfahrens in Software

Sowohl das Subpixel-Verfahren als auch das Vier-Zeiger-Verfahren wurden im Rahmen dieser Diplomarbeit in Software implementiert. Als Basis diente dabei der PHIGS-Renderer Da*Render*. In diesem Kapitel sollen implementierungsspezifische Details wie der Umgang mit Compiler-Optionen und Quellcodemodulen erläutert werden.

5.1 Modul-Übersicht

Abbildung 5.1 zeigt den Zusammenhang zwischen den Quellcode-Modulen. Ein Pfeil von einem Modul X zu einem Modul Y hat dabei die Bedeutung ‘Funktionen in X rufen Funktionen in Y auf’.

Alle Module, deren Namen mit ‘XpAA’ beginnen, sind neue Dateien, die im Rahmen dieser Diplomarbeit Da*Render* hinzugefügt wurden. Das Präfix ‘XpAA’ setzt sich zusammen aus ‘X’, da es sich um eine Bearbeitung in horizontaler Richtung handelt, ‘p’ was bedeutet, daß die Funktionen dem Umfang von PHIGS unterstellt sind und ‘AA’ für Antialiasing.

Die drei Module direkt unter ‘Phigs.lib’ sind an sich Bestandteile der ursprünglichen PHIGS-Bibliothek, wurden aber im Rahmen der Einfügung der Antialiasing-Funktionen modifiziert. Sie bilden also sozusagen die Schnittstelle zwischen der ursprünglichen Da*Render*-Implementierung und den neu hinzugekommenen Antialiasing-Funktionen. Die drei großen gestrichelten Blöcke, zu denen jeweils eines der drei Module gehört, unterteilen das Projekt in die drei Teilaufgaben Initialisierung, Rendering und Nachbearbeitung.

Zu dem Rendering-Block gehören, über die neuen Module hinaus, noch die alten Module ‘xpsfedge.c’ und ‘xpsfspan.c’. Sie beinhalten die Funktionen zur Traversierung der Hauptkante bzw.

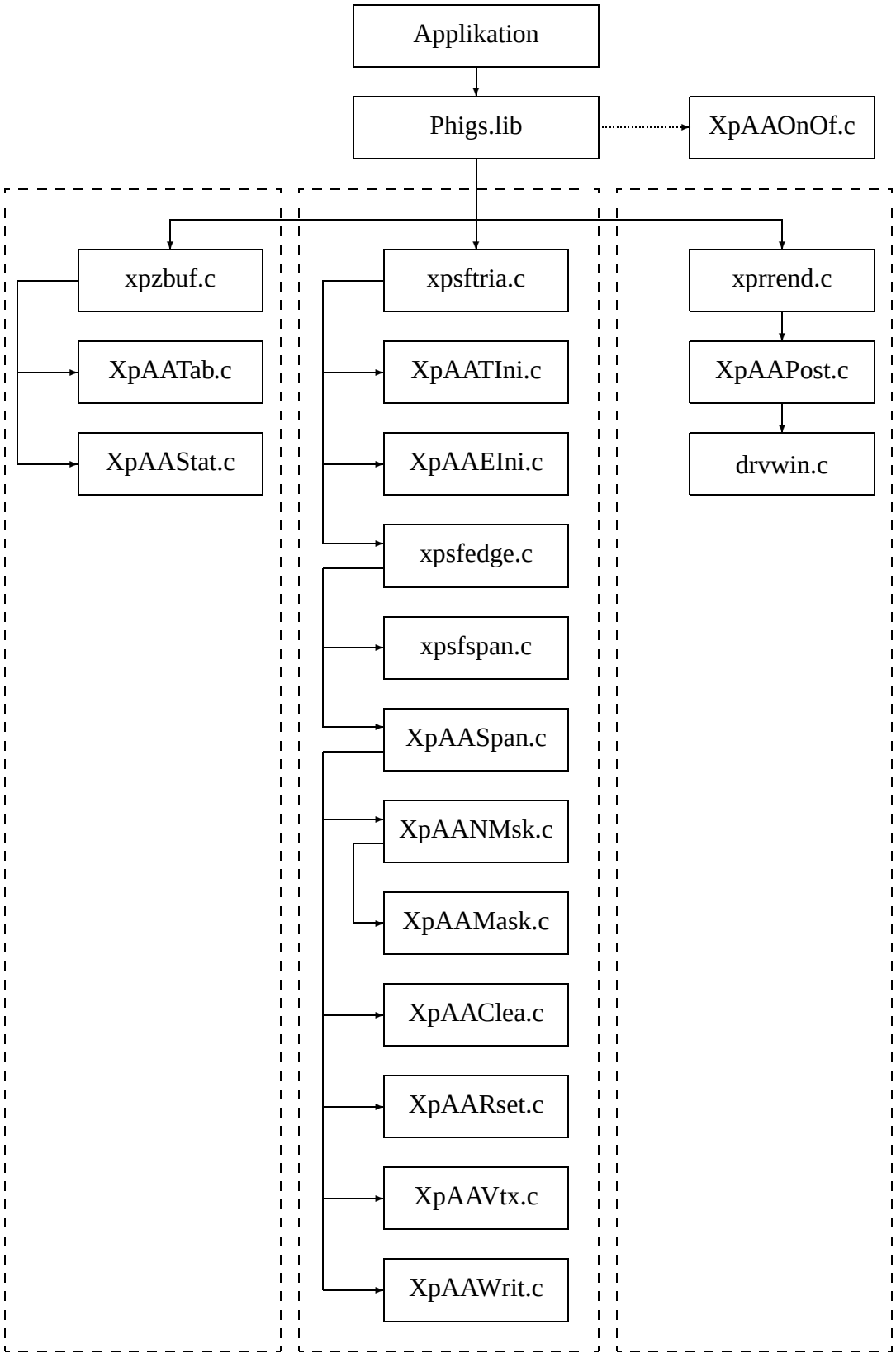


Abbildung 5.1: Aufruf-Hirarchie der Projektdateien

der Spans eines Dreiecks und wickeln Pointsampling und Z-Buffering ab.

Das Modul ‘drvwin.c’ stellt Funktionen zum Bearbeiten der Truecolor-Bitmap unter Windows zur Verfügung. Implementierungen unter anderen Betriebssystemen benötigen damit unter Umständen einen anderen Treiber.

5.2 Die Quellcodemodule

Neue Quellcodedateien	(Verzeichnis *source/dr/rnd*)
XpAAOnOf.c	An- und Abschalten des Antialiasings, Auswahl des Verfahrens
XpAATab.c	Erzeugen von Überdeckungs- und Subpixeltabellen
XpAAStat.c	Zurücksetzen der Statistik-Struktur
XpAATIni.c	Initialisierung eines Dreiecks
XpAAEIni.c	Initialisierung einer Kante
XpAASpan.c	Antialiasing-Bearbeitung eines Spans
XpAANMsk.c	Setzen der neuen Masken eines Spans einer Kante
XpAAMask.c	Generierung einer neuen Maske
XpAAClea.c	Löschen der temporären Arrays eines Spans
XpAARSet.c	Zurücksetzen ‘interner’ Masken
XpAAVtx.c	Spezielle Behandlung der Eckpixel
XpAAWrit.c	Zurückschreiben der Masken eines Spans zum Frame
XpAAPost.c	Nachbearbeitung des Bildes

Modifizierte alte Quellcodedateien	(Verzeichnis *source/dr/rnd*)
xpzbuf.c	Speicherreservierung für Framebuffer und temporäre Arrays
xpsftria.c	Initialisierung eines Dreiecks und Aufruf der AA-Initialisierung
xpsfedge.c	Traversierung der Hauptkante und Aufruf der Spanbearbeitung
xpsfspan.c	Rendern eines Spans und Vermerken von Renderinformation
xprrend.c	Da*Render*-Funktionen, Aufruf des Nachbearbeitungsprozesses

Includedateien	(Verzeichnis *include/dr/rnd*)
XpAA.h	Strukturen von Dreiecken und Kanten, globale Prototypen
XpAAMask.h	Macro-Namen für Maskenteile
XpAAStat.h	Struktur zur Speicherung statistischer Information zur Laufzeit

5.3 Installation

Zur Aufrüstung einer bestehenden Da*Render*-Implementierung mit der neuen Antialiasing-Funktionalität sind folgende Schritte erforderlich:

1. Kopieren der neuen beziehungsweise geänderten Sourcedateien in die Verzeichnisse *source/dr/rnd* und *include/dr/rnd*.
2. Ändern der Makedateien oder Verwendung eines Projektfiles.
3. Setzen der gewünschten Compiler-Optionen, mindestens der Option 'AntiAliasing' (siehe unten).
4. Compilieren der neuen Dateien und Linken.

5.4 An- und Ausschalten und Methodenauswahl

Die einzigen Funktionen, die vom Anwender bzw. einer Funktion der PHIGS-Bibliothek aufgerufen werden sollen, befinden sich im Modul *XpAAOnOf.c*.

void PrAntiAliasingOff(void)
Sofortiges Deaktivieren des laufenden Antialiasing-Verfahrens.

tFlag PrAntiAliasingOn(void)
Aktivierung des gewählten Antialiasing-Verfahrens ab dem nächsten Bild.

void XpAASetAAMethod(tPAAMethodEnum Method)
Auswahl des Antialiasing-Verfahrens. Als Parameter sind zur Zeit *aa_subpixel* und *aa_4pointers* zulässig.

Das An- und Abschalten der Erweiterungen Zeigerumlenkung und Akkumulation erfolgt zur Zeit über Compiler-Optionen (siehe nächster Abschnitt).

5.5 Präprozessor-Kommandos

Zur Steuerung der bedingten Compilierung können folgende Optionen gesetzt werden:

AntiAliasing: Wird diese Option *nicht* gesetzt, wird der gesamte neue Code von der Compilierung ausgeschlossen. Das Projekt verhält sich dann exakt wie eine ursprüngliche Version von Da*Render*. **Diese Option sollte also unbedingt gesetzt sein**.

AAStatistics: Das Setzen dieser Option veranlaßt das Führen einer internen Statistik über generierte Masken, kritische Fälle etc. Weitere Informationen befinden sich im nächsten Abschnitt.

AARedirect: Aktivierung der Zeigerumlenkung (siehe Seite 64). Diese läuft nur unter dem Vier-Zeiger-Verfahren, unter dem Subpixel-Verfahren bleibt die Option wirkungslos.

AAAccumulate: Aktivierung der Akkumulation (siehe Seite 65). Diese läuft nur unter dem Vier-Zeiger-Verfahren, unter dem Subpixel-Verfahren bleibt die Option wirkungslos.

NDEBUG: Diese Option wurde nicht im Rahmen der neuen Dateien eingeführt, sondern gehört zu der Standard-C Funktion *assert()*. Durch das Setzen der Option wird der zusätzliche Code aller *assert*-Anweisungen im Programm von der Compilierung ausgeschlossen.

Das Aktivieren der jeweiligen Option kann per Compiler-Option oder mit einem

`#define` *identifier* `1`

in der Includedatei *XpAA.h* durchgeführt werden.

5.6 Erzeugen statistischer Information zur Laufzeit

Wenn der Bezeichner *AAStatistics* gesetzt wurde, wird an diversen Stellen in den neuen Modulen statistische Information gesammelt. Diese Information hat zwei primäre Verwendungszwecke:

1. Überwachung der korrekten Funktion des laufenden Verfahrens (Debugging)
2. Analyse der auftretenden Fälle zur Bewertung der Leistungsfähigkeit des laufenden Verfahrens

Die Struktur *AAStatistic*, in der die Daten gesammelt werden, ist im Modul *XpAAStat.h* definiert. Die einzelnen Eintäge teilen sich in folgende Bereiche auf:

- Maskengenerierung
- Maskenverknüpfungen
- Visibilitäts-Analyse
- Zeigerumlenkung
- Akkumulation
- Nachbearbeitung

Da es sich lediglich um interne Information handelt, wird der Inhalt der *AAStatistic*-Struktur nicht am Bildschirm ausgegeben, sondern kann nur mit einem Debugger verfolgt werden. Um die abschließenden Daten eines Bild zu erhalten, bietet es sich an, am Ende der Funktion *XpAAPostProcessing()* im Modul *XpAAPost.c* einen Breakpoint zu setzen und die Daten einzusehen.

Kapitel 6

Vorschlag zur Realisierung des Vier-Zeiger-Verfahrens in Hardware

Viele der bisher vorgestellte Berechnungen waren auf eine Software-Implementierung zugeschnitten. Alle Berechnungsteile waren auf eine sequentielle Abarbeitung mit einem gebräuchlichen Prozessor ausgelegt, auf Möglichkeiten zur Parallelisierung wurde nicht geachtet.

In diesem Kapitel sollen nun konzeptuelle Änderungen vorgestellt werden, die sich ergeben, wenn die gleichen Algorithmen auf einer integrierten Schaltung realisiert werden. Soweit nicht anders vermerkt, beziehen sich alle Angaben auf das Vier-Zeiger-Verfahren. Der Chip soll Dreiecks-Renderer und Antialiasing auf sich vereinen, im folgenden werden jedoch nur die Antialiasing-spezifischen Teile behandelt.

6.1 Änderungen gegenüber der Software-Implementierung

6.1.1 Nachbearbeitung

Beim Nachbearbeitungs-Durchgang ergeben sich keine wesentlichen Unterschiede im Ablauf zwischen Soft- und Hardware. Mögliche Beschleunigungen der Hardware lassen sich erreichen durch:

Mischen: Die zwölf 4 Bit $*$ 8 Bit-Multiplikationen (vier Nachbarpixel à drei Farbkomponenten) können parallel ausgeführt werden. Die Farben der Nachbarpixel sollten dazu in einem Cache gehalten werden. Der Cache muß dabei groß genug sein, um zwei, besser drei Bildschirmzeilen mit Farbeinträgen aufzunehmen.

Umkopieren: Falls für das Rendern und das Anzeigen des Bildes zwei verschiedene Frames benutzt werden, kann das Umkopieren in Einheit mit der Nachbearbeitung erfolgen. Die verschiedenen Modelle für eine Rendering-Pipeline werden in Abschnitt 6.3 vorgestellt.

Parallelität: Die Nachbearbeitung ist wegen der Lokalität aller Operationen beliebig parallelisierbar.

6.1.2 Parallele Berechnung der Masken

Im Gegensatz zur Software-Implementierung können in Hardware die Masken aller drei Kanten parallel, durch jeweils eine eigene Recheneinheit erzeugt werden. Durch das gleichzeitige Vorliegen der Masken erübrigt sich die Verwendung eines temporären Arrays, da die Masken direkt verknüpft werden können.

Die Parallelität der Berechnungen bringt noch weitere Möglichkeiten mit sich. Im Gegensatz zur Software-Version besteht kein Vorteil mehr darin, die Anzahl der zu generierenden Masken zu minimieren. Es verursacht keinen Anstieg der Rechenzeit, wenn die vorhandenen Recheneinheiten ständig in Benutzung sind. Wir können deshalb die Bearbeitung aller Pixel dadurch vereinheitlichen, daß wir stets die Masken aller drei Kanten erzeugen und verknüpfen, selbst für Pixel die im Inneren des Dreiecks liegen. Mit diesem Ansatz wird es möglich, Maskengenerierung und Rendervorgang zu verschmelzen. Schilling verwendet in [Sch91] eine entsprechende Vorgehensweise.

Um Fallunterscheidungen zu vermeiden, wird eine Normierung der Maskenverknüpfung vorgenommen, ähnlich der in Abschnitt 3.4.5 vorgestellten. Jede Kanten-Einheit liefert dabei für jedes Pixel eine Maske und zusätzlich ein sog. Inklusionsbit zurück. Das Inklusionsbit ist genau dann gesetzt, wenn die Maske schwarz ist, der Pixelmittelpunkt also in der Halbebene der Kante liegt. Die Entscheidung, ob das Pixel zum Dreieck gehört, läßt sich dann auf die Und-Verknüpfung der drei Inklusionsbits reduzieren. Wie die Masken zu verknüpfen sind, wird im folgenden Abschnitt besprochen.

6.1.3 Verknüpfung der Masken und Eckpixelbehandlung

Wie bereits bemerkt, benötigt das Vier-Zeiger-Verfahren eine spezielle Behandlung der Eckpixel, da sich die für die Verknüpfung benötigte räumliche Information nicht mehr aus den Masken rekonstruieren läßt. Um maximalen Geschwindigkeit zu erzielen, ist es im Rahmen einer Hardware-Realisierung jedoch wünschenswert, alle Pixel gleich zu behandeln. Um diesem Anspruch zu genügen, bietet es sich an, Teile des Subpixel- und des Vier-Zeiger-Verfahrens in einem neuen Ansatz zu vereinen:

- Das Subpixel-Verfahren verfügt über die räumliche Information, um beliebige Masken korrekt zu verknüpfen. Die Maskenverknüpfung sollte also auf der Basis von Subpixelmasken stattfinden.
- Das Vier-Zeiger-Verfahren führt zu einer korrekten Zuordnung der Farbanteile auf die Nachbarpixel. Die zum Frame zurückgeschriebenen Masken sollten also Vier-Zeiger-Masken sein.

Es ergibt sich damit folgender Ablauf:

Pixel dieses Spans rendern
+ Für alle drei Kanten: **Generierung von Maske und Inklusionsbit**
• Verknüpfung der drei Masken durch Konjunktion
• Wenn Konjunktion der Inklusionsbits wahr ist, dann Ergebnismaske negieren
• Umwandlung der Subpixelmaske in eine Vier-Zeiger-Maske
• Falls Maske sichtbar ist, Maske zurückschreiben

Generierung von Maske und Inklusionsbit
• Erzeugen der Maske als Subpixelmaske
• Erzeugen des Inklusionsbits: Schwarz = 1, rot = 0
• Negieren der Maske, falls Inklusionsbit = 1

Die temporären Subpixelmasken müssen dabei nicht zwangsläufig die Auflösung von $4 * 4$ Subpixeln haben. Höhere Auflösungen sind denkbar und würden zu höherer Genauigkeit der Endergebnisse führen, ohne zusätzlichen Speicher im Framebuffer zu erfordern.

6.1.4 Neue Verzahnung von Rendering und Maskengenerierung

Durch die sofortige Verknüpfung der Masken in der Hardware-Version hat sich das bei der Software-Version für diesen Zweck benötigte temporäre Array erübigt. Es bleibt aber noch ein weiteres temporäres Array, das in der Software-Version benötigt wurde, um rote Masken mit verzögerter Bearbeitung aufzubewahren. Die Verwendung temporärer Arrays ist für die Hardware-Realisierung ungünstig und soll mit Hilfe der im diesem Abschnitt vorgestellten Modifikation vermieden werden.

Die Notwendigkeit, manche rote Masken verzögert zu bearbeiten, hatte sich dadurch ergeben, daß Ergebnisse des Renderns der folgenden Zeile bei der Sichtbarkeitsanalyse der Masken benötigt wurden, aber noch nicht vorlagen (siehe Abschnitt 3.4.7). Das Problem läßt sich lösen, indem wir dem Render-Vorgang generell einen ‘Vorsprung’ von einer Zeile gegenüber der Masken-Generierung einräumen. Bei der Maskenbearbeitung liegen dann alle benötigten Informationen über die Sichtbarkeit der Nachbarpixel sofort vor, so daß sich die verzögerte Bearbeitung erübrigt.

Um die Information, welche Pixel gesetzt wurden, bei der Maskenbearbeitung verfügbar zu haben, müssen wir sie zwischenspeichern. Dazu benötigen wir, wie in der Software-Version, ein Array, das groß genug ist, zwei Bildschirmzeilen à ein Bit aufzunehmen (siehe Abbildung 6.1).[1]

[1] Da Rendering und Maskenbearbeitung mit dem neuen Modell versetzt ablaufen, muß nun von neuem sichergestellt werden, daß beide Teilprozesse voll ausgelastet sind. Um zu vermeiden, daß Maskenprozeß und Renderprozeß aufeinander warten müssen, könnte es sich als günstig erweisen, dem Renderprozeß einen Vorsprung variabler Größe zu geben. Dadurch könnten auch solche Fälle ohne Wartezeiten gelöst, in denen ein längerer Span auf einen kürzeren Span folgt, der Renderer also kurzzeitig mehr Arbeit als der Maskengenerator hat. Dieser Ansatz

Dieses Array ist im Gegensatz zu den Maskenarrays klein genug, um auf dem Chip gehalten zu werden. Die Bearbeitung der Masken vereinfacht sich nun ganz extrem. Beispielsweise läßt sich das Zurücksetzen interner Masken in einem Stück lösen (siehe Abschnitt *'Ablauf'*).

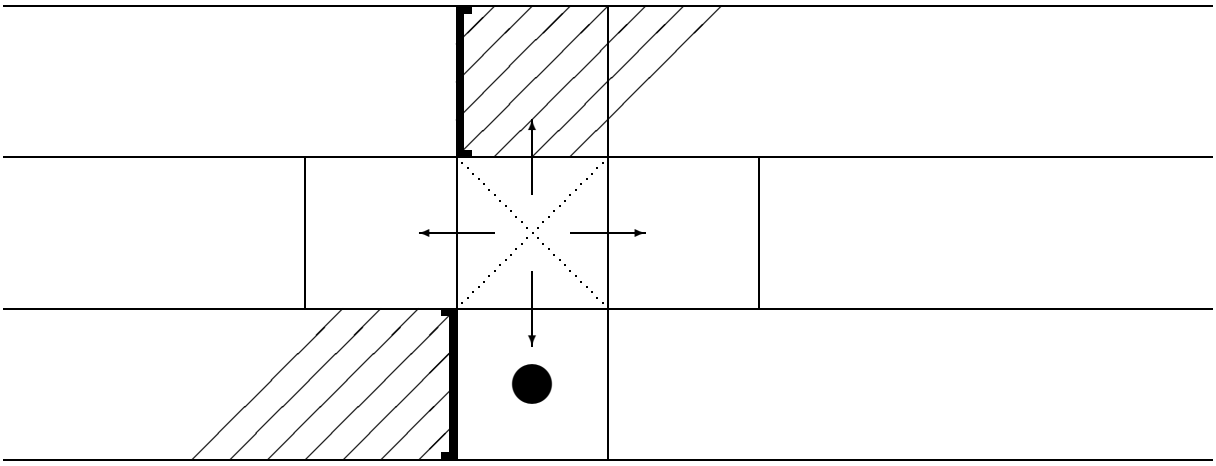


Abbildung 6.1: Zeitliches Zusammenspiel von Rendering und Maskengenerierung. Die Maske in der Mitte wird gerade generiert. Um die Sichtbarkeit der neuen Maske zu entscheiden, wird die Renderinginformation der vier direkten Nachbarn benötigt. Der Renderprozeß hat eine ganze Zeile Vorsprung und bearbeitet gerade das mit • markierte Pixel. Somit liegt die gesamte benötigte Renderinformation für die Maskenbearbeitung vor. Die Pixel in dem Bereich zwischen [und] müssen dazu zwischengespeichert werden.

6.1.5 Geänderte Traversierung der Kanten

In der Software-Version werden die Kanten auf besonders einfache Weise traversiert: Wir beginnen mit dem obersten Pixel (geringster Y-Wert) und schreiten dann in DDA-Traversierungsreihenfolge fort. Jeder Aufruf der Traversierungsfunktion erzeugt die Masken genau eines Spans.

Dieser Ansatz führt bei einer Hardware-Realisierung zu einem Problem, da einige Kanten ihre Masken in aufsteigender, andere in absteigender X-Wert-Reihenfolge produzieren. Da innerhalb eines Dreiecks an jedem Span zwei bis drei Kanten beteiligt sind, kann es passieren, daß die beiden entgegengesetzten Traversierungsrichtungen zusammen auftreten. Es ist dann nicht mehr möglich, den Span in einer einheitlichen Richtung zu erzeugen, wie das für die Hardware-Realisierung wünschenswert wäre.

Eine Lösungsmöglichkeit besteht darin, die Traversierungsreihenfolge von Da*Render* beizubehalten[2] und an jedem Spananfang die neuen Startwerte für die Errorterme der drei Kanten neu zu bestimmen. Bei den ersten zwei Zeilen wird dafür je eine Multiplikation benötigt. Der bei der zweiten Zeile bestimmte Wert kann dann für alle weiteren Zeilen wiederverwendet werden,

erfordert entweder einen größeren oder einen flexibleren Zwischenspeicher für Renderinformation. Die insgesamt aufzunehmende Speichermenge ist nie größer als eine Zeile, sie kann jedoch auf mehrere Zeilen verteilt sein.

[2]Traversierung der Zeilen entlang der Hauptkante, Traversierung der Spans von der Hauptkante ausgehend. Ein Span kann also sowohl in auf- als auch in absteigender X-Richtung traversiert werden. Innerhalb eines Dreiecks sind die Traversierungsrichtungen aller Spans jedoch gleich.

so daß sich wieder eine inkrementelle Vorgehensweise ergibt (Abbildung 6.2). Der Grund dafür, daß zwei Startwerte mit Hilfe einer Multiplikation bestimmt werden müssen, liegt darin, daß die oberste Zeile üblicherweise kürzer ist als die anderen, da sie einen Eckpunkt enthält.

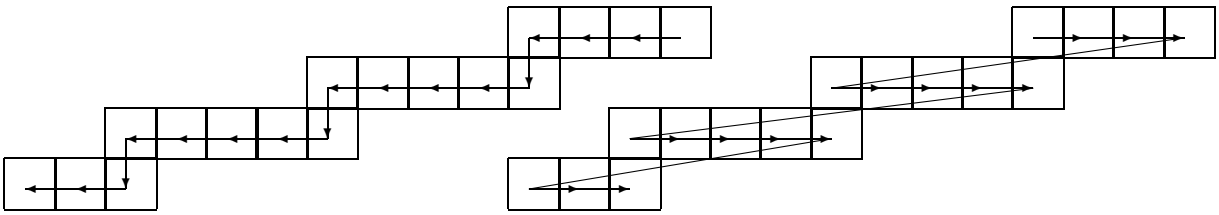


Abbildung 6.2: Kantentraversierung in einheitlicher Richtung. Kanten, deren natürliche Traversierungsrichtung der Span-Traversierungsrichtung entgegenläuft, werden Spanweise erzeugt (Bild rechts). Um den Offset zwischen den Zeilen zu bestimmen, ist zunächst eine Multiplikation erforderlich. Danach kann inkrementell weitergearbeitet werden.

6.1.6 Überdeckungs- und Subpixeltabelle

Es wäre vorteilhaft, die Repräsentation von Überdeckungs- und Subpixeltabelle mit auf dem Chip zu realisieren.

Die Überdeckungsfunktion zeichnet sich durch ihre 'Gutartigkeit' aus. Wie im Abschnitt *'Benötigte Auflösung der Überdeckungtabelle'* auf Seite 48 gezeigt wurde, ist die größte Steigung der partiellen Ableitungen nur gerade doppelt so hoch wie die der entsprechenden linearen Funktion. Lineare Interpolation kann also mit Gewinn eingesetzt werden.

Die Subpixeltabelle ist extrem einfach zu erzeugen. Wie im Abschnitt *'Bestimmung der Subpixelmaske'* auf Seite 51 erklärt wurde, gibt es bei einer $4 * 4$ Subpixelmaske nur vier verschiedene Reihenfolgen, die Subpixel anzusprechen. Die Größe der Tabelle entsteht lediglich aus den Symmetrietransformationen in die anderen sieben Oktanten und durch die acht verschiedenen Überdeckungsgrade. Diese Berechnungen lassen sich aber auch mit wenig Aufwand zur Laufzeit durchführen. Es reicht also aus, die vier Ansprechreihenfolgen explizit zu speichern.

6.2 Ablauf

Mit den in den obigen Abschnitten beschriebenen Änderungen ergibt sich folgender Ablauf:

Pixelbearbeitung	
+	**Pixel dieses Spans rendern.**
+	**Maske des vorherigen Spans rendern.**

Pixel dieses Spans rendern

- Wenn das Pixel nicht im Inneren des Dreiecks liegt, abbrechen,
- Z-Wert von neuem Pixel mit Z-Wert im Framebuffer vergleichen. Wenn neues Pixel nicht näher am Betrachter liegt, abbrechen.
- Pixel in den Framebuffer schreiben.
- In dem Visibilitätsbit dieses Pixels vermerken, daß das Pixel sichtbar war. Die Visibilitätsbits befinden sich in einem temporären Array auf dem Chip.
- Führen des Inklusionsintervalls: Vermerken der x-Koordinate des ersten und des letzten Pixels im Span. Die Speicherung der Intervallgrenzen reicht aus, da Dreiecke konvex sind.

Maske des vorherigen Spans rendern

- Zugehörigkeit der aktuellen Maske zum Inklusionsintervalls testen. Ergebnis im Inklusionsbit speichern.
- Falls das Inklusionsbit wahr, aber das Visibilitätsbit falsch ist, Bearbeitung der Maske abbrechen, da Maske unsichtbar ist.[3]

\+ Für alle drei Kanten: **Generierung der Maske als rote Subpixelmaske.**

- Verknüpfung der drei Masken durch Konjunktion.
- Falls das Inklusionsbit gesetzt ist, Ergebnismaske negieren (schwarze Maske).

\+ **Umwandlung der Subpixelmaske in eine Vier-Zeiger-Maske.**

- (Schwarze Maske) Falls Inklusionsbit gesetzt ist und Maske ungleich 0, dann Maske zurückschreiben.

\+ (Leere Maske) Falls Inklusionsbit gesetzt ist und Maske gleich 0, dann **Dreiecksinterne Zeiger löschen.**

- (Rote Maske) Falls Inklusionsbit nicht gesetzt ist und Maske ungleich 0, dann Maske zurückzuschreiben.

Generierung der Maske als rote Subpixelmaske

- Berechnung der Überdeckung mit Hilfe einer Tabelle.
- Splitten der Überdeckung in Haupt- und Neben-Mischrichtung, bzw. deren Gegenrichtungen, falls das Inklusionsbit gesetzt ist,
- Falls das Inklusionsbit gesetzt ist, beide Überdeckungen negieren.
- Addieren der beiden Überdeckungen mit den Werten der Rechenwerke der anderen Kanten anhand der Richtung. Es ergeben sich vier Buckets, in die die Zumischungen der Richtungen oben, rechts, unten und links gesammelt werden.
- Erzeugen der Subpixelmaske aus der Kantensteigung de_x und der nicht gesplitteten Überdeckung.

[3] Die ersten beiden Schritte können sinnvollerweise zusätzlich bereits am Ende der Bearbeitung des vorhergehenden Pixels durchgeführt werden, so daß die noch rechtzeitig mit der Bearbeitung der folgenden Maske begonnen werden kann, falls die aktuelle Maske unsichtbar ist.

Umwandlung der Subpixelmaske in eine Vier-Zeiger-Maske

- Zählen der gesetzten Subpixel der Subpixelmaske.
- Berechnung der Summe der Bucketinhalte.
- Falls die Summe der Bucketinhalte ungleich der Anzahl der Subpixel ist ist, handelt es sich um ein Eckpixel. Bestimmen der Differenz und Verteilen der (negativen) Differenz auf die Buckets mit Status ‘valid’.
- Laden des Zustandes der vier direkt benachbarten Pixel aus dem Visibilitätsarray und den Inklusionsintervallen in ein Register.
- Zuordung von Zuständen zu den Nachbarn: Falls das Inklusionsbit des aktuellen Pixels gesetzt ist: $I \vee V$ ‘redirect’, $I \vee \neg V$ ‘drop’, $\neg I$ ‘valid’. Falls nicht: $I \vee V$ ‘valid’, $I \vee \neg V$ ‘drop’, $\neg I$ ‘redirect’.
- Falls keiner der Überdeckungsbuckets den Status ‘valid’ hat, abbrechen.
- Nullsetzen der Überdeckungsbuckets von Nachbarpixeln mit Status ‘drop’.
- Verteilen des Inhalts der Überdeckungsbuckets mit Status ‘redirect’ auf die Überdeckungsbuckets mit Status ‘valid’.
- Negieren des Inhalts der Buckets, falls das Inklusionsbit gesetzt ist.
- Die vier Buckets sind nun die vier Meta-Subpixel der neuen Vier-Zeiger-Maske.

Dreiecks-interne Zeiger löschen

- Setze Löschmaske auf 0xffff.
- Falls oberes Pixel gesetzt war, Löschmaske &= $\sim$UP_POINTERBITS.
- Falls rechtes Pixel gesetzt war, Löschmaske &= $\sim$RIGHT_POINTERBITS.
- Falls unteres Pixel gesetzt war, Löschmaske &= $\sim$DOWN_POINTERBITS.
- Falls linkes Pixel gesetzt war, Löschmaske &= $\sim$LEFT_POINTERBITS.
- Maske im Frame &= Löschmaske.

Das Ablaufschema realisiert das Vier-Zeiger-Verfahren ohne Akkumulation. Das Einfügen der Akkumulatuon unterscheidet sich jedoch nicht von der Software-Version (siehe 4.4).

6.3 Mögliche Konfigurationen der Renderingpipeline

Der Chip soll in einem Echtzeit-Renderingsystem eingesetzt werden. Um eine flimmerfreie Bildwiedergabe zu erreichen, benötigen wir mehrere Bildspeicher. Abbildung 6.3 zeigt eine mögliche Realisierung mit drei separaten Framebuffern.

Da der Nachbearbeitungsprozeß lediglich aus einer einfachen Traversierung des Framebuffers besteht, bietet es sich an, ihn mit der Bildwiedergabe zusammenzufassen. Dadurch läßt sich ein Framebuffer einsparen, so daß insgesamt zwei Framebuffer genügen. Abbildung 6.4 zeigt den entsprechenden Systemaufbau.

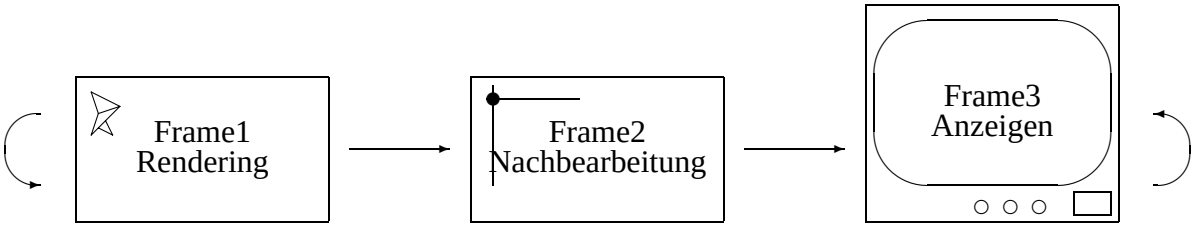


Abbildung 6.3: Realisierung der Renderingpipeline mit drei Framebuffern. Je ein Framebuffer wird für die drei Stufen Rendering, Nachbearbeitung und Anzeigen verwendet.

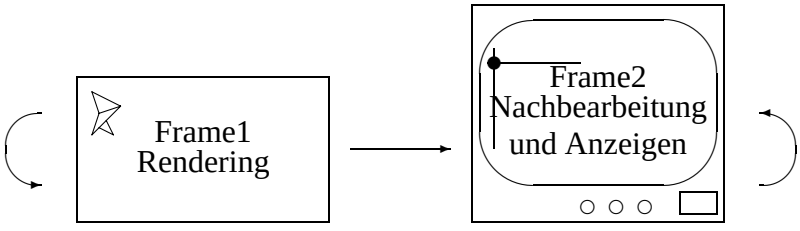


Abbildung 6.4: Realisierung mit zwei Framebuffern und Strahlsynchronisation. Wenn der Nachbearbeitungsprozeß schnell genug durchgeführt werden kann, ist es möglich, Nachbearbeitung und Bildwiedergabe in einem einzigen Framebuffer durchzuführen. Die Nachbearbeitung muß dabei in der gleichen Reihenfolge wie die Bildwiedergabe erfolgen und der Bildwiedergabe zeitlich etwas vorauseilen.

Kapitel 7

Zusammenfassung und Ausblick

Zur Auswertung wurde das entwickelte Vier-Zeiger-Verfahren nach den drei in der Zielsetzung definierten Gütekriterien Geschwindigkeit, Bildqualität und Kostengünstige Realisierbarkeit in Hardware untersucht:

Geschwindigkeit

In jedem Takt kann ein Pixel mit den dazugehörigen Masken gerendert werden. Die Geschwindigkeit eines Systems wird durch das Antialiasing-Verfahren nur dadurch verringert, daß über den Innenbereich der Dreiecke hinaus noch zusätzliche Kantenpixel bearbeitet werden müssen. Der Anteil der Kantenpixel ist jedoch nur bei sehr vielen kleinen Polygonen signifikant, einer Situation wie sie aus Leistungsgründen bei Echtzeitapplikationen generell vermieden wird.

Die Geschwindigkeit des Verfahrens beruht auf seiner hardwarenahen Architektur. Jedes Pixel erfordert die gleiche Anzahl an Speicherzugriffen. Die Adressen der benötigten Speicherzugriffe sind linear angeordnet, so daß Pipelining und Caching mit maximalem Gewinn eingesetzt werden können.

Bildqualität

Bei Objekten aus sehr schmalen Dreiecken gelingt es nicht immer, die Artefakte des zugrundeliegenden Pointsamplings vollständig zu unterdrücken. Der bei Echtzeitsystemen deutlich wichtigere Fall großer Polygone wird jedoch gut gelöst. Die durch die schmalen Polygone verursachten Artefakte sind für ein Echtzeitsystem vertretbar. Die Bildqualität wurde neben den theoretischen Überlegungen auch anhand von Testbildern überprüft (siehe Anhang).

Kostengünstige Realisierbarkeit

Die Hardware für den Rendering-Durchgang des Antialiasing-Verfahrens kann zusammen mit dem Dreiecks-Renderer auf einem Chip realisiert werden. Um Rendering und Nachbearbeitung parallel ausführen zu können, ist es erforderlich, die Nachbearbeitung auf einem separaten Chip zu realisieren. Das Verfahren benötigt lediglich 16 Bit zusätzli-

chen Speichers pro Pixel. Eine Rendering-Pipeline mit strahlsynchronisierter Nachbearbeitung kommt mit insgesamt zwei separaten Framebuffern aus.

Ein abschließender Vergleich mit den in Abschnitt 2.1 vorgestellten Antialiasing-Verfahren ergibt folgendes Ergebnis:

	Supersampling	Accumulationbuffer	A-Buffer	Vier-Zeiger-V.
Geschwindigkeit	$\ominus\ominus$	$\ominus\ominus$	$\odot$	$\oplus\oplus$
Bildqualität	$\oplus/\oplus\oplus$[1]	$\oplus/\oplus\oplus$[1]	$\oplus\oplus$	$\odot$
Speicherbedarf	$\ominus\ominus$	$\oplus$	$\odot$[2]	$\oplus$
Hardware-Aufwand	$\oplus$	$\oplus$	$\ominus\ominus$	$\oplus$

$\ominus\ominus$	sehr schlecht
$\ominus$	schlecht
$\odot$	befriedigend
$\oplus$	gut
$\oplus\oplus$	sehr gut

Die Software-Implementierung definiert die zu erreichende Funktionalität und Bildqualität. Kapitel 6 zeigt eine mögliche Hardware-Realisierung auf. Als nächster Schritt hin zu einer Hardware-Realisierung, erscheint eine Simulation der Ablaufsteuerung für das Verfahren und der Verzahnung mit dem Rendering-Prozeß sinnvoll. Dadurch ließe sich die Realisierung in Bezug auf das Pipelining und die gemeinsame Nutzung von Hardware-Ressourcen optimieren.

[1] Anhängig vom Supersamplingfaktor
[2] Nicht konstant

Anhang A

Demobilder

A.1 Objekte aus großen Polygonen

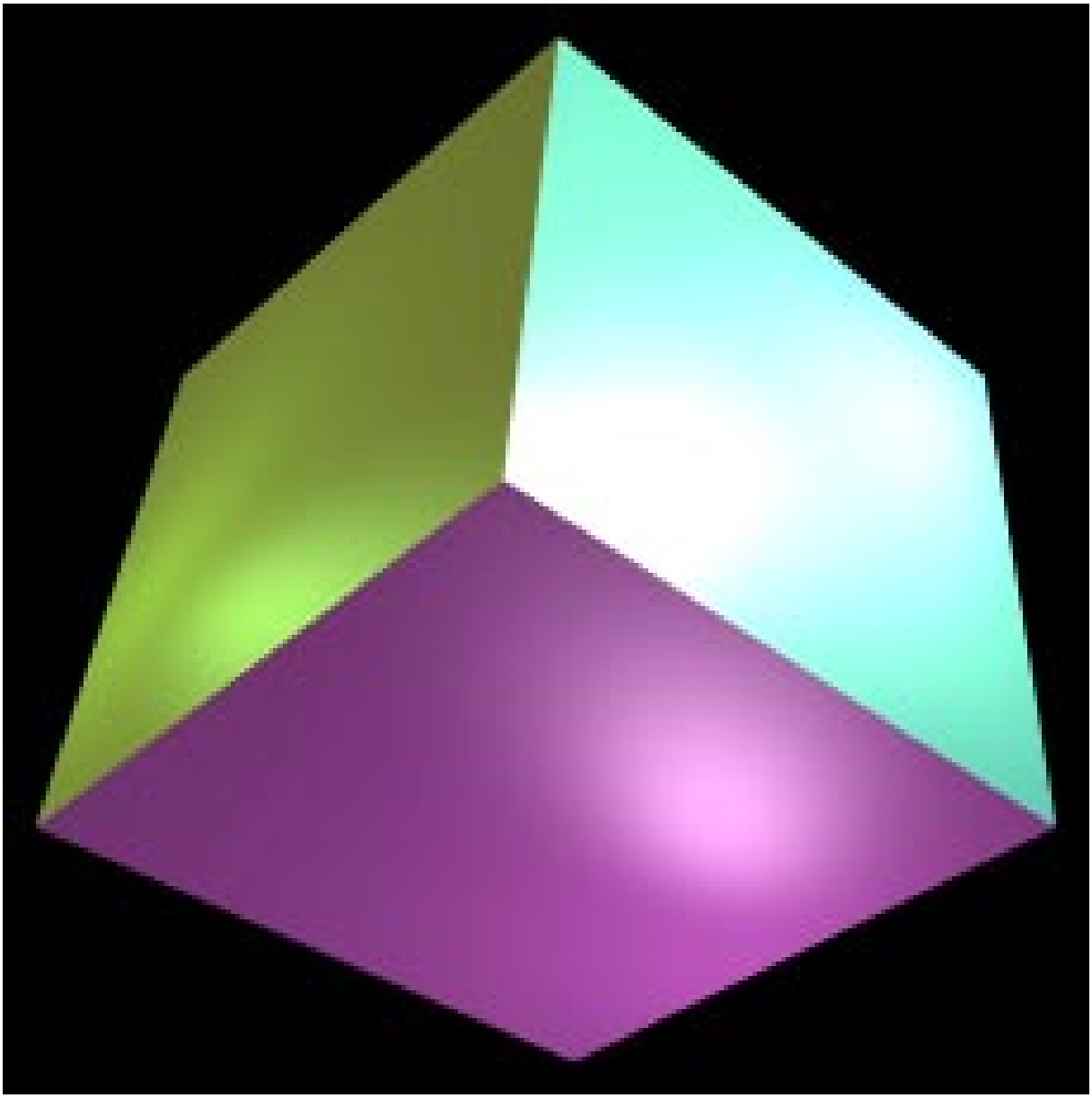


Abbildung A.1: Die Kanten großer Polygone werden beim Vier-Zeiger-Verfahren korrekt behandelt.

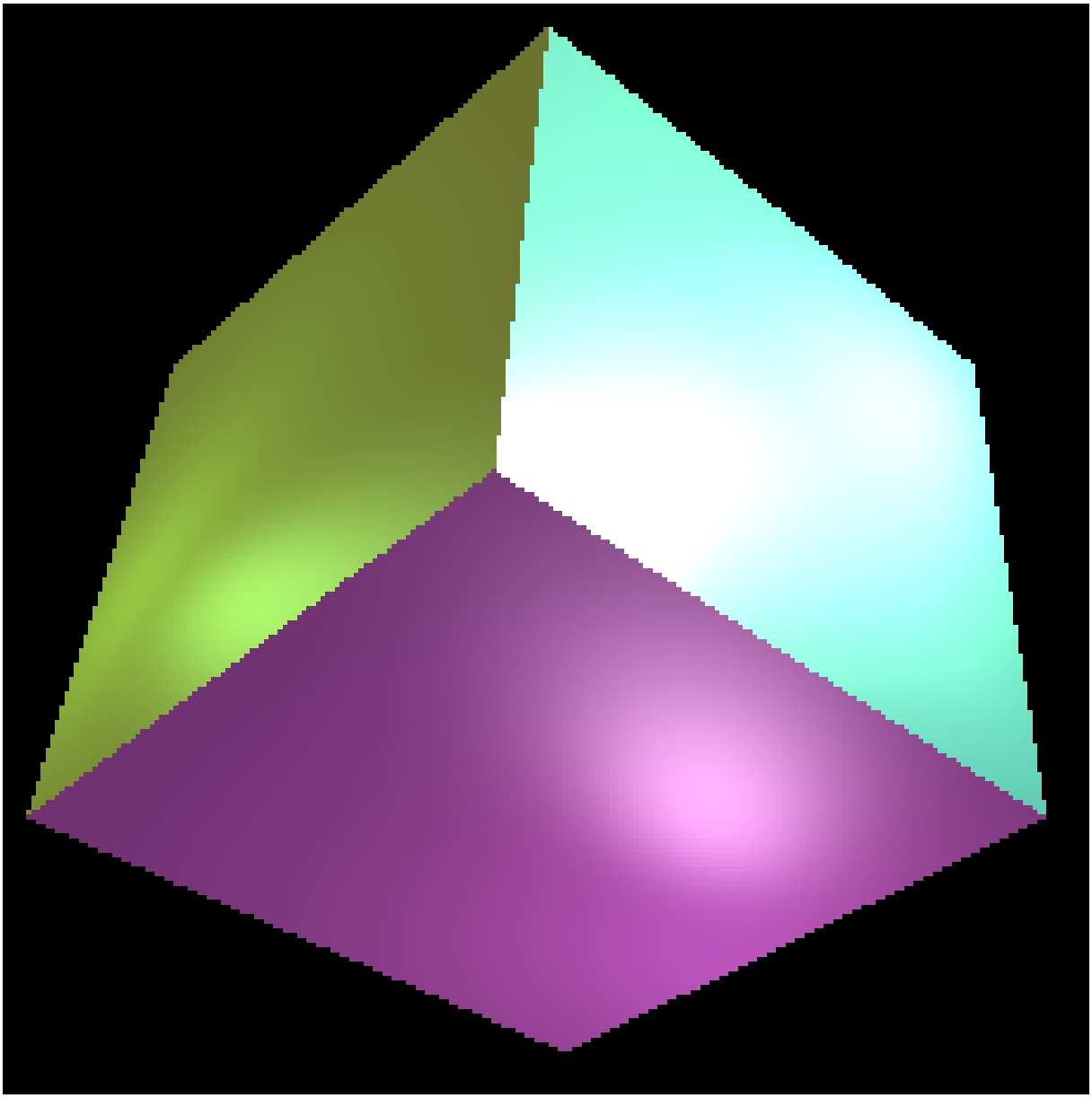


Abbildung A.2: Das gleiche Bild eines Würfels ohne Antialiasing.

A.2 Behandlung flacher Kanten

Abbildung A.3: Region um eine flache Polygonkante ohne Antialiasing.

Abbildung A.4: Die selbe Kante mit dem Subpixel-Verfahren nachbearbeitet. Die in Abschnitt 3.6 beschriebenen Intensitätssprünge sind gut zu erkennen. Auch die zu intensiven Pixel im Bereich der linken oberen Ecke sind auf die fehlerhafte Zuordnung von Subpixeln zu Nachbarpixeln zurückzuführen.

Abbildung A.5: Die selbe Kante mit dem Vier-Zeiger-Verfahren nachbearbeitet. Durch die korrekte Zuordnung der Zumischungen können keine Intensitätsprünge auftreten. Da das Vier-Zeiger-Verfahren nach dem Prinzip der exakten Überdeckung arbeitet, wird die volle Auflösung der Vier-Zeiger-Masken genutzt. Eine Kante kann damit bis zu 31 verschiedene Intensitätsstufen annehmen (siehe auch Abbildung A.11).

A.3 Effekt der Zeigerumlenkung

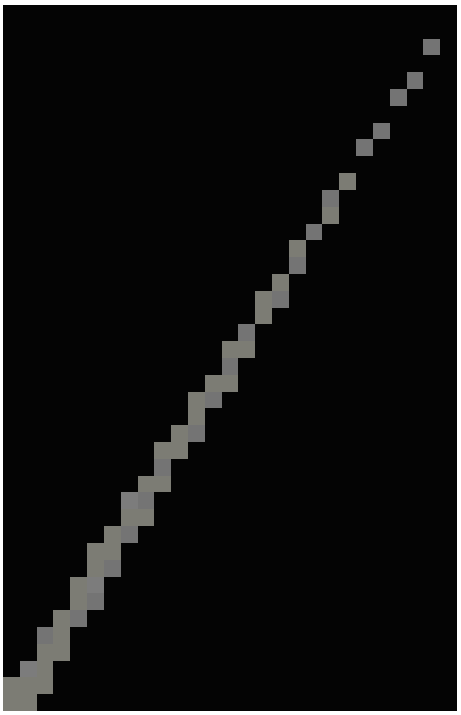
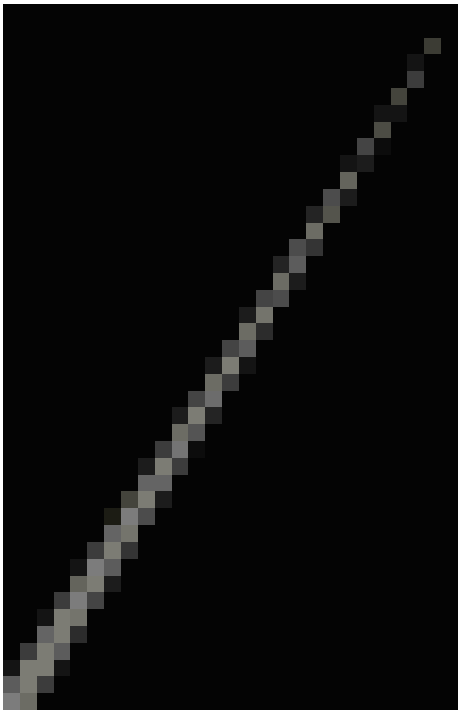


Abbildung A.6: (Bild links) Schmales Dreieck ohne Antialiasing. Da das Dreieck in der Nähe des Eckpunktes immer seltener Samplepoints überdeckt, weist es in diesem Bereich Löcher auf. Die Linie erscheint unterbrochen



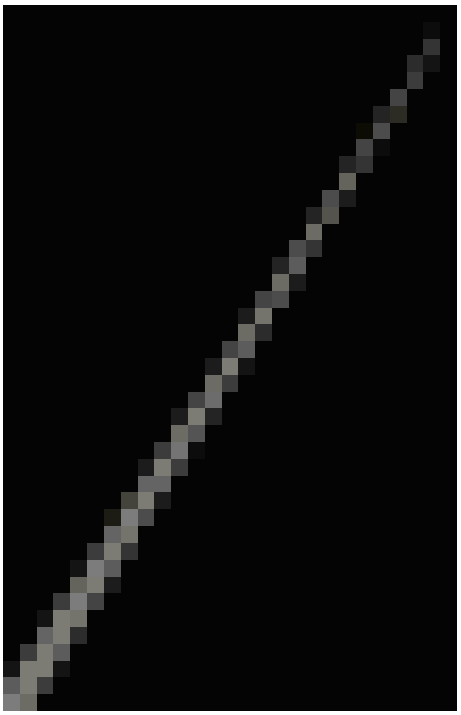


Abbildung A.7: (Bild oben) Das selbe Dreieck, dargestellt mit dem Vier-Zeiger-Verfahren aber ohne Zeigerumlenkung. Die Löcher sind verschwunden, doch das Dreieck weist weiterhin einige Pixel auf, die fälschlicherweise nicht gesetzt wurden.

Abbildung A.8: (Bild links) Das selbe Dreieck, diesmal mit Vier-Zeiger-Verfahren und Zeigerumlenkung erzeugt. Durch die Zeigerumlenkung werden zusätzliche Dreiecksfragmente in dem Bereich sichtbar gemacht, in dem das ungefilterte Dreieck Löcher aufweist. Das schmale Dreieck erscheint nun auch zum Eckpunkt hin vollständig.

A.4 Effekt der Akkumulation

Abbildung A.9: Das obere der drei Bilder zeigt ein schmales Dreieck ohne Antialiasing, das am rechten Bildschirmrand geclippt wurde. Um das Clipping effizient zu realisieren, wurde das Dreieck vom Renderer in zwei noch schmalere Dreiecke zerlegt. Zur Verdeutlichung zeigen die unteren beider Bilder die beiden entstandenen Dreiecke.

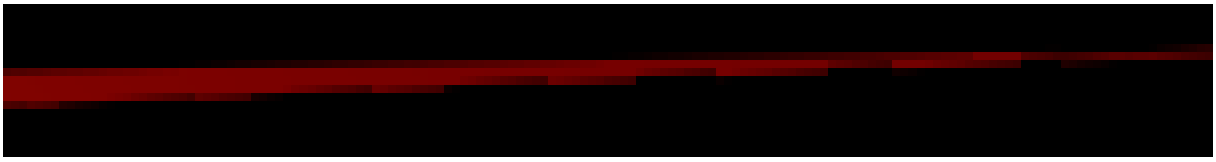


Abbildung A.10: Das Vier-Zeiger-Verfahren ohne Akkumulation kann das geclippte Dreieck nicht korrekt darstellen. Besonders das untere der beiden Dreiecke ist sehr schmal, so daß diverse Dreiecksfragmente, die keine Samplepoints überdecken, verloren gehen. Da das gesamte Dreieck lediglich als Überlagerung der beiden Teildreiecke erzeugt wird, ist auch das Gesamtobjekt ‘durchlöchert’. Die Auswirkungen der Löcher des unteren Dreiecks sind im Bild gut zu erkennen. Sie führen dazu, daß der untere Rand des Gesamtdreiecks schraffiert wirkt.

Abbildung A.11: Mit Hilfe der Akkumulation wird das Dreieck korrekt dargestellt. Die Dreiecksfragmente, die keine Samplepoints überdecken werden dem jeweils anderen Dreieck hinzugefügt. Die Summe beider Teildreiecke ist an allen Stellen breit genug, so daß das Gesamtobjekt wieder vollständig wiedergegeben werden kann.

A.5 Objekte aus vielen schmalen Dreiecken

Abbildung A.12: Durch die Projektion werden die Dreiecke am Rand der Kugel sehr schmal. Die Visibilität der schmalen Fragmente läßt sich schlecht bestimmen, so daß die heuristischen Vorhersagen der Akkumulation stellenweise fehlerhaft sind. Ein Fehler ist zum Beispiel am linken äußeren Rand der Kugel sichtbar. Die Kugel besteht aus 840 Dreicken.

Abbildung A.13: Die selbe Kugel mit dem Vier-Zeiger-Verfahren, aber ohne Akkumulation bearbeitet. Das Bild enthält im Randbereich eine Vielzahl von Fehlern.

Abbildung A.14: Die selbe Kugel ohne Antialiasing.

Abbildungsverzeichnis

Literaturverzeichnis

[AH92] Hans-Josef Ackermann and Christoph Hornung.
The triangle shading engine.
In A. Kaufmann, editor, *Advances in Computer Graphics Hardware V*, pages 157–174, Berlin, 1992. Springer-Verlag.

[AW85] Greg Abram and Lee Westover.
Efficient alias-free rendering using bit-masks and look-up tables.
Siggraph '85, 19(3):52–59, 1985.

[Bar91] Anthony C. Barkans.
Hardware-assisted polygon antialiasing.
IEEE Computer Graphics & Applications, 11(1):80.–88, Januar 1991.

[Bre65] J. E. Bresenham.
Algorithm for computer control of a digital plotter.
IBM System Journal, 4(1):25–30, 1965.

[Car84] Loren Carpenter.
The a-buffer, an antialiased hidden surface method.
ACM Computer Graphics, 18(3):103–108, Juli 1984.

[Cat78] Edwin Catmull.
A hidden-surface algorithm with anti-aliasing.
Computer Graphics, 12(3):6–11, 1978.

[Cro77] Franklin C. Crow.
The aliasing problem in computer-generated shaded images.
Communications ACM, 20(11):799–805, November 1977.

[Cro81] Franklin C. Crow.
A comparison of antialiasing techniques.
IEEE Computer Graphics & Applications, 1(1), Januar 1981.

[ES88] José Encarnação and Wolfgang Straßer.
Computer Graphics.
Datenverarbeitung. R. Oldenburg Verlag, München, third edition, 1988.

[FvDFH90] Foley, van Dam, Feiner, and Hughes.
Computer Graphics: Principles and Practice.
Addison Wesley, second edition, 1990.

[GS81] Satish Gupta and Robert F. Sproull.
Filtering edges for grey-scale displays.
ACM Computer Graphics, 15(3):1–5, August 1981.

[HA90] Paul Haeberli and Kurt Akeley.
The accumulation buffer: Hardware support for high-quality rendering.
ACM Computer Graphics, 24(4):309–317, August 1990.

[Jer77] A. J. Jerri.
The shannon sampling theorem — its various extensions and applications: A tutorial review.
Proc. IEEE, 65(11), November 1977.

[Kla93] R. Victor Klassen.
Increasing the apparent addressability of supersampling grids.
IEEE Computer Graphics & Applications, 13(5):74–77, September 1993.

[Lau94] Wing Hung Lau.
The anti-aliased approximation buffer.
Im Druck befindliche Arbeit, August 1994.

[Mam89] Abraham Mammen.
Transparency and antialiasing algorithms implemented with the virtual pixel maps technique.
IEEE Computer Graphics & Applications, 9(4):43–55, 1989.

[OS75] A. V. Oppenheim and R.W. Schafer.
Digital Signal Processing.
Prentice-Hall, Englewood Cliffs, N.J., 1975.

[Pin88] Juan Pineda.
A parallel algorithm for polygon rasterization.
ACM Computer Graphics, 22(4):17–20, August 1988.

[Sch91] Andreas Schilling.
A new simple and efficient antialiasing with subpixel masks.
ACM Computer Graphics, 25(4):133–141, Juli 1991.

[SH73] I. E. Sutherland and G. W. Hodgman.
Reentrant polygon clipping: A solution to the hidden surface problem.
Unveröffentlichte Arbeit, Oktober 1973.

[SS93] Andreas Schilling and Wolfgang Straßer.
Exact: Algorithm and hardware architecture for an improved a-buffer.
In *Computer Graphics Proceedings, Annual Conference Series*,
pages 85–91, 1993.

[WA77] K. Weiler and P. Atherton.
Hidden surface removal using area sorting.
Computer Graphics, 11(2):214–222, 1977.

[Wil83] L. Williams.
Pyramidial parametrics.
SIGGRAPH 83, pages 1–11, 1983.

Index