(19) **Bundesrepublik Deutschland**
Deutsches Patent- und Markenamt

(10) **DE 696 30 165 T2** 2004.07.22

(12) # Übersetzung der europäischen Patentschrift

(97) **EP 0 747 857 B1**
(21) Deutsches Aktenzeichen: **696 30 165.2**
(96) Europäisches Aktenzeichen: **96 107 379.8**
(96) Europäischer Anmeldetag: **09.05.1996**
(97) Erstveröffentlichung durch das EPA: **11.12.1996**
(97) Veröffentlichungstag der Patenterteilung beim EPA: **01.10.2003**
(47) Veröffentlichungstag im Patentblatt: **22.07.2004**

(51) Int Cl.[7]: **G06T 15/10**

(30) Unionspriorität:
465846 **06.06.1995** **US**

(84) Benannte Vertragsstaaten:
DE, FR, GB

(73) Patentinhaber:
Hewlett-Packard Co.(a Delaware Corporation), Palo Alto, Calif., US

(74) Vertreter:
Schoppe, Zimmermann, Stöckeler & Zinkler, 82049 Pullach

(72) Erfinder:
Alcorn, Byron A., Fort Collins, US; Emmot, Darel N., Fort Collins, US; Tucker, Steven Paul, Fort Collins, US

(54) Bezeichnung: **Herunterladen von Texturdaten unter Umgehung der 3D-Schaltungen**

Anmerkung: Innerhalb von neun Monaten nach der Bekanntmachung des Hinweises auf die Erteilung des europäischen Patents kann jedermann beim Europäischen Patentamt gegen das erteilte europäische Patent Einspruch einlegen. Der Einspruch ist schriftlich einzureichen und zu begründen. Er gilt erst als eingelegt, wenn die Einspruchsgebühr entrichtet worden ist (Art. 99 (1) Europäisches Patentübereinkommen).

Die Übersetzung ist gemäß Artikel II § 3 Abs. 1 IntPatÜG 1991 vom Patentinhaber eingereicht worden. Sie wurde vom Deutschen Patent- und Markenamt inhaltlich nicht geprüft.

Beschreibung

Gebiet der Erfindung

[0001] Die vorliegende Erfindung bezieht sich allgemein auf ein Texturabbildungscomputergraphiksystem und insbesondere auf ein Cachespeichersystem zum Speichern von Texturabbildungsdaten.

Hintergrund der Erfindung

[0002] Computergraphiksysteme werden im allgemeinen zum Anzeigen graphischer Darstellungen von Objekten auf einem zweidimensionalen Anzeigebildschirm verwendet. Aktuelle Computergraphiksysteme können hochdetaillierte Darstellungen liefern und werden bei einer Vielzahl von Anwendungen verwendet.

[0003] Bei einem typischen Computergraphiksystem wird ein Objekt, das auf dem Anzeigebildschirm dargestellt werden soll, in eine Mehrzahl von Graphikgrundelementen unterteilt. Grundelemente sind Grundkomponenten eines Graphikbildes und können Punkte, Linien, Vektoren und Polygone, wie z. B. Dreiecke, umfassen. Typischerweise ist ein Hardware-/Softwareschema implementiert, um die Graphikgrundelemente, die die Ansicht von einem oder mehreren Objekten darstellen, das/die auf dem Bildschirm dargestellt wird, auf dem zweidimensionalen Anzeigebildschirm aufzubereiten oder zu zeichnen.

[0004] Typischerweise werden die Grundelemente, die das dreidimensionale Objekt definieren, das aufbereitet werden soll, von einem Hostcomputer geliefert, der jedes Grundelement bezüglich Grundelementdaten definiert. Wenn das Grundelement beispielsweise ein Dreieck ist, kann der Hostcomputer das Grundelement bezüglich der xyz-Koordinaten seiner Scheitelpunkte, und auch der RGB-Farbwerte jedes Scheitelpunkts.

[0005] Die Aufbereitungshardware interpoliert die Grundelementdaten, um die Anzeigebildschirmpixel zu berechnen, die eingeschaltet werden, um jedes Grundelement darzustellen, und die RGB-Werte für jedes Pixel.

[0006] Frühe Graphiksysteme waren nicht in der Lage, Bilder auf eine ausreichend realistische Weise anzuzeigen, um komplexe dreidimensionale Objekte darzustellen oder zu modellieren. Die Bilder, die durch solche Systeme angezeigt wurden, zeigten äußerst glatte Oberflächen ohne Texturen, Höcker, Kratzer, Schattierungen und andere Oberflächeneinzelheiten, die bei dem Objekt, das modelliert wurde, vorliegen.

[0007] Als Folge wurden Verfahren zum Anzeigen von Bildern mit verbesserten Oberflächeneinzelheiten entwickelt. Texturabbildung ist ein solches Verfahren, das das Abbilden eines Quellbildes, das als eine Textur bezeichnet wird, auf eine Oberfläche eines dreidimensionalen Objekts umfaßt, und danach das Abbilden des texturierten dreidimensionalen Objekts auf den zweidimensionalen Graphikanzeigebildschirm zum Anzeigen des resultierenden Bildes. Oberflächeneinzelheitsattribute, die üblicherweise texturabgebildet werden, umfassen Farbe, Spiegelreflexion, Vektorstörung, Spiegelung, Transparent, Schatten, Oberflächenunregelmäßigkeiten und Abstufung.

[0008] Texturabbildung umfaßt das Anlegen von einem oder mehreren Punktelementen (Texeln) einer Textur an jedes Punktelement (Pixel) des angezeigten Abschnitts des Objekts, auf das die Textur abgebildet wird. Texturabbildungshardware wird herkömmlichen mit Informationen versehen, die die Art und Weise anzeigen, auf die die Texel in einer Texturabbildung den Pixeln auf dem Anzeigebildschirm entsprechen, die das Objekt darstellen. Jedes Texel in einer Texturabbildung ist durch S- und T-Koordinaten definiert, die die Position desselben in der zweidimensionalen Texturabbildung identifizieren. Für jedes Pixel wird auf das entsprechende Texel oder die Texel, die auf dasselbe abbilden, von der Texturabbildung zugegriffen, und in die End-RGB-Werte aufgenommen, die für das Pixel erzeugt werden, um das texturierte Objekt auf dem Anzeigebildschirm darzustellen.

[0009] Es sollte klar sein, daß jedes Pixel in einem Objektgrundelement nicht für jede Ansicht des Objekts in einer Einszu-Eins-Entsprechung auf ein einzelnes Texel in der Texturabbildung abbildet. Beispielsweise, je näher das Objekt zu dem Ansichtstor ist, das auf dem Anzeigebildschirm dargestellt wird, um so größer erscheint das Objekt. Da das Objekt auf dem Anzeigebildschirm größer erscheint, wird die Darstellung der Textur detaillierter. Wenn somit das Objekt einen relativ großen Teil des Anzeigebildschirms verbraucht, wird eine große Anzahl von Pixeln verwendet, um das Objekt auf dem Anzeigebildschirm darzustellen, und jedes Pixel, das das Objekt darstellt, kann in einer Einszu-Eins-Entsprechung auf ein einziges Texel in der Texturabbildung abbilden, oder ein einziges Texel kann auf mehrere Pixel abbilden. Wenn das Objekt jedoch einen relativ kleinen Teil des Anzeigebildschirms ausfüllt, wird eine viel kleinere Anzahl von Pixeln verwendet, um das Objekt darzustellen, was dazu führt, daß die Textur weniger detailliert dargestellt ist, so daß jedes Pixel auf mehrere Texel abbilden kann. Jedes Pixel kann auch auf mehrere Texel abbilden, wenn eine Textur auf einen kleinen Teil eines Objekts abgebildet ist. Resultierende Texeldaten werden für jedes Pixel berechnet, das auf mehr als ein Texel abbildet, und stellen typischerweise einen Mittelwert der Texel dar, die auf das Pixel abbilden.

[0010] Texturabbildungshardwaresysteme umfassen typischerweise einen lokalen Speicher, der Daten speichert, die eine Textur darstellen, die dem Objekt, das aufbereitet wird, zugeordnet ist. Wie es oben erörtert wurde, kann ein Pixel auf mehrere Texel abbilden. Falls es für die Texturabbildungshardware notwendig wäre, eine große Anzahl von Texeln, die

auf ein Pixel abbilden, von dem lokalen Speicher zu lesen, um einen Mittelwert zu erzeugen, dann wäre eine große Anzahl von Speicherlesevorgängen und die Mittelwertbildung von vielen Texelwerten erforderlich, was zeitaufwendig wäre und die Systemleistungsfähigkeit verschlechtern würde.

[0011] Um dieses Problem zu lösen, wurde ein Schema entwickelt, das die Erzeugung einer Reihe von MIP-Abbildungen für jede Textur umfaßt, und das Speichern der MIP-Abbildungen der Textur, die dem Objekt, das aufbereitet wird, zugeordnet sind, in dem lokalen Speicher der Texturabbildungshardware. Eine MIP-Abbildung für eine Textur umfaßt eine Basisabbildung, die direkt der Texturabbildung entspricht, und auch eine Reihe von gefilterten Abbildungen, wobei jede aufeinanderfolgende Abbildung in jeder der zwei Texturabbildungsdimensionen um einen Faktor von 2 in der Größe reduziert ist. Ein darstellendes Beispiel eines Satzes von MIP-Abbildungen ist in **Fig.** 1 gezeigt. Die MIP-Abbildungen (MIP = multum in parvo = viele Dinge auf kleinem Raum) umfassen eine Basisabbildung **100**, die acht mal acht Texel groß ist, und auch eine Reihe von **Abb.** 102, 104 und 108, die jeweils vier mal vier Texel, zwei mal zwei Texel und ein Texel groß sind.

[0012] Die Vier-mal-Vier-Abbildung **102** wird durch Box-filtern (Dezimieren) der Basisabbildung **100** erzeugt, so daß jedes Texel in der **Abb.** 102 einem Mittelwert von vier Texeln in der Basisabbildung **100** entspricht. Beispielsweise ist das Texel **110** in der **Abb.** 102 gleich dem Mittelwert der Texel **112**–**115** in der **Abb.** 100, und die Texel **118** und **120** in der **Abb.** 102 sind entsprechend gleich den Mittelwerten der Texel **121**–**124** und **125**–**128** in der **Abb.** 100. Die Zwei-mal-Zwei-Abbildung **104** wird gleichermaßen durch Kästchenfiltern (Box-Filtern) der **Abb.** 102 erzeugt, so daß das Texel **130** in der **Abb.** 104 gleich dem Mittelwert der Texel **110** und **118**–**120** in der **Abb.** 102 ist. Das einzige Texel in der **Abb.** 108 wird durch Mittelwertbilden der vier Texel in der **Abb.** 104 erzeugt.

[0013] Herkömmliche Graphiksysteme laden im allgemeinen die voll- ständige Reihe von MIP-Abbildungen für jede Textur, die mit den Grundelementen, die auf dem Anzeigebildschirm aufbereitet werden sollen, verwendet werden soll, von dem Hauptspeicher des Hostcomputers zu dem lokalen Speicher der Texturabbildungshardware herunter. Somit kann die Texturabbildungshardware von jeder der Reihe von MIP-Abbildungen auf Texturdaten zugreifen. Die Bestimmung darüber, auf welche Abbildung zugegriffen wird, um die Texeldaten für ein spezielles Pixel zu liefern, basiert auf der Anzahl von Texeln, auf die das Pixel abbildet. Falls das Pixel beispielsweise in Eins-zu-Eins-Entsprechung mit einem einzigen Texel in der Texturabbildung abbildet, dann kann auf die Basisabbildung **100** zugegriffen werden. Falls das Pixel jedoch auf 4, 16 oder 64 Texel abbildet, wird jeweils auf die **Abb.** 102, 104 und 108 zugegriffen, weil diese Abbildungen jeweils Texeldaten speichern, die einen Mittelwert von 4, 16 und 64 Texeln in der Texturabbildung darstellen.

[0014] Es kann sein, daß ein Pixel nicht direkt auf irgendein Texel in der ausgewählten Abbildung abbildet und zwischen zwei oder mehr Texel fällt. Einige Graphiksysteme verwenden eine bilineare Interpolation zum genauen Erzeugen von Texeldaten, wenn dies auftritt. Falls ein Pixel in eine MIP-Abbildung zwischen zwei oder mehr Texeleinträgen abbildet, dann sind die verwendeten resultierenden Texeldaten ein gewichteter Mittelwert der nächsten Texeleinträge. Somit können die Texeldaten, die jedem Pixel entsprechen, der gewichtete Mittelwert von bis zu vier Texeleinträgen in einer einzigen Abbildung sein. Falls beispielsweise ein Pixel auf eine Position in der **Abb.** 102 abbildet, die bei **132** angezeigt ist, wäre die resultierende Texeldatenabbildung auf dieses Pixel der gewichtete Mittelwert der Pixel **110** und **118**–**120**.

[0015] Es kann auch sein, daß Pixel nicht direkt auf eine der Abbildungen in der Reihe von MIP-Abbildungen abbilden, und dieselben zwischen zwei Abbildungen fallen. Beispielsweise kann ein Pixel auf eine Anzahl von Texeln in der Texturabbildung abbilden, die größer ist als 1, aber kleiner als 4. Einige Graphiksysteme adressieren diese Situation durch Interpolieren zwischen den beiden nächstliegenden MIP-Abbildungen, um die resultierenden Texeldaten zu erreichen. Für das oben gezeigte Beispiel, wo ein Pixel auf mehr als ein, aber weniger als vier Texel in der Texturabbildung abbildet, würden die Texeldaten, die durch die **Abb.** 100 und 102 geliefert werden, interpoliert, um die resultierenden Texeldaten für das Pixel zu erreichen. Wenn dieses Schema mit der oben beschriebenen Interpolation von mehreren Texeleinträgen in einer einzigen Abbildung kombiniert wird, ist dieses Schema als eine trilineare Interpolation bekannt und kann zu resultierenden Texeldaten für jedes eine Pixel führen, das als ein gewichteter Mittelwert von bis zu acht Texeln erzeugt wird, d. h. die vier nächstliegenden Texel in jeder der beiden nächstliegenden Abbildungen.

[0016] Wie es oben erörtert wurde, laden herkömmliche Texturabbildungssysteme die gesamten Reihen von MIP-Abbildungen für jede Textur herunter, die den Grundelementen zugeordnet ist, die durch das System aufbereitet werden sollen, selbst wenn auf einige der MIP-Abbildungen nicht zugegriffen wird. Das Herunterladen von MIP-Abbildungen, auf die nicht zugegriffen wird, und auch von Abschnitten von zugegriffenen Abbildungen, die nicht verwendet werden, ist eine Verschwendung der Systemressourcen und reduziert die Bandbreite desselben.

[0017] Ferner sind einige Texturabbildungssysteme pipelineartig, so daß verschiedene Operationen an unterschiedlichen Objektgrundelementen gleichzeitig durchgeführt werden. Eine Reihe von MIP-Abbildungen für eine Textur kann jedoch riesig sein. Die meisten Systeme verwenden einen lokalen Speicher, der in der Lage ist, nur eine solche große Reihe von MIP-Abbildungen zu einem Zeitpunkt zu speichern.

Wenn es daher eine Änderung bei der Textur gibt, die beim Aufbereiten der Grundelemente verwendet wird, muß das System eine neue Reihe von MIP-Abbildungen herunterladen. Typischerweise verläuft der Datenweg, der verwendet wird, um die neuen Texturdaten in den lokalen Speicher in der Texturabbildungshardware zu laden, durch die Grundelementaufbereitungspipeline des Systems. Wenn daher eine neue Textur abgebildet werden soll, muß es der Grundelementaufbereitungspipeline ermöglicht werden, sich zu leeren, bevor die neue Reihe von MIP-Abbildungen heruntergeladen werden kann. Sobald die Reihe von MIP-Abbildungen heruntergeladen ist, muß die Pipeline erneut gefüllt werden. Die Notwendigkeit des Entleerens der Grundelementaufbereitungspipeline, jedesmal, wenn eine neue Textur erforderlich ist, reduziert die Bandbreite des Systems.

[0018] Ein herkömmlicher Lösungsansatz ist in der EP-A-0438195 beschrieben, die eine Anzeigevorrichtung offenbart, die eine Host-CPU umfaßt, die über einen Bus mit einer Hostplattenspeicherung und einem Hauptspeicher verbunden ist. Eine Anzeigeverarbeitungseinheit (DPU; DPU = display processing unit) ist zwischen der Host-CPU und dem Bus und einem Anzeigespeicher verbunden. Die Texturabbildungshardware, die einen Texturspeicher umfaßt, ist mit der DPU und mit dem Anzeigespeicher verbunden. Pixelwerte, die erforderlich sind, um ein Bild aufzubereiten, werden von dem Hostspeicher zu freien Positionen in dem Texturspeicher übertragen, durch die Host-CPU über den Bus und einen Weg, der getrennt von dem Weg über die DPU ist. Um ein Grundelement zu ziehen oder „aufzubereiten", bewirkt die CPU, daß Register in der DPU über den VME-Bus mit Werten geladen werden, die einziges Grundelement und dessen verschiedene Attribute definieren. Die DPU wandelt die Grundelemente abtastend um und speichert Pixeldaten im Rasterabtastformat in dem Anzeigespeicher. Die Pixelkoordinaten werden als Schreibadressen an den Anzeigespeicher angelegt, so daß ein Pixelfarbwert für jedes Pixel in den Anzeigespeicher geschrieben werden kann. Die Grundoberflächenfarbe des Grundelements wird mit Texturen moduliert, um Attribute einer Objektoberfläche realistisch widerzuspiegeln. Die DPU erzeugt gleichzeitig mit jedem Paar von Pixelkoordinaten ein Paar von Texturkoordinaten, so daß die Texturstruktur in dem Texturspeicher auf die Grundelementoberfläche abgebildet wird. Sollten die erforderlichen Texturwerte nicht in dem Texturspeicher gespeichert werden, werden sie zuerst über den getrennten Weg von der CPU erhalten. Die Texturkoordinaten werden in der Texturabbildungshardware verarbeitet und an den Texturspeicher angelegt, so daß für jede Pixelposition, die adressiert wird, ein Texturwert verfügbar ist. Die Texturwerte werden zusammen mit anderen Parametern in der DPU zum Modifizieren der Pixelfarbwerte verwendet.

[0019] Die EP-A-0631252 bezieht sich auf einen dreidimensionalen Graphikbeschleunigerabschnitt einer Grundelementaufbereitungspipeline. Die WO-A-9202923 bezieht sich auf ein Datenverarbeitungs- und Speichersystem. Gemäß diesem Dokument werden Unterbrechungen erzeugt, wenn ein Cachefehlschlag auftritt. Die JP 06028485 A bezieht sich auf spezifische Ausführungsbeispiele von Texturadreß- und Strukturgeneratoren.

[0020] Es ist die Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum Verwalten von Texturabbildungsdaten in einem Computergraphiksystem ohne unnötige Verzögerungen zu schaffen.

[0021] Diese Aufgabe wird durch ein Verfahren gemäß Anspruch 1 und ein System gemäß Anspruch 8 gelöst.

Kurze Beschreibung der Zeichnungen

[0022] Für ein besseres Verständnis der vorliegenden Erfindung wird auf die beiliegenden Zeichnungen Bezug genommen, die hierin durch Bezugnahme aufgenommen sind.

[0023] **Fig.** 1 ist eine graphische Darstellung eines Satzes von Textur-MIP-Abbildungen;

[0024] **Fig.** 2 ist ein Blockdiagramm eines Ausführungsbeispiels des Gesamtcomputergraphiksystems der vorliegenden Erfindung;

[0025] **Fig.** 2A ist ein Blockdiagramm eines weiteren Ausführungsbeispiels des Gesamtcomputergraphiksystems der vorliegenden Erfindung;

[0026] **Fig.** 3 ist ein Blockdiagramm der Texturabbildungshardware der vorliegenden Erfindung;

[0027] **Fig.** 4 ist ein detaillierteres Blockdiagramm des Parameterinterpolatorelements der Texturabbildungshardware der vorliegenden Erfindung;

[0028] **Fig.** 5 ist ein Blockdiagramm des Cachespeichers und ein Abschnitt der Texturabbildungshardware der vorliegenden Erfindung;

[0029] **Fig.** 6 stellt ein Beispiel der Art und Weise dar, wie Blöcke von Texturdaten organisiert werden, um eine Vier-Verschachtelungs-Implementierung des Cachespeichers der vorliegenden Erfindung auszunutzen;

[0030] **Fig.** 7 ist ein detailliertes Blockdiagramm der Organisation der Speicherchips, die den Cachespeicher der vorliegenden Erfindung bilden;

[0031] **Fig.** 8 ist ein detailliertes Blockdiagramm eines Abschnitts der Texturabbildungshardware der vorliegenden Erfindung;

[0032] **Fig.** 9 ist ein Diagramm und eine Tabelle, die ein Beispiel von Texeln darstellen, auf die von benachbarten MIP-Abbildungen für jeden Pixelstrom zugegriffen wird, gemäß einem Texturabbildungsschema der vorliegenden Erfindung;

[0033] **Fig.** 10 ist ein Diagramm von Texturabbildungshardwarepuffern und zugeordneten Dateneinträgen gemäß dem Beispiel von **Fig.** 9;

[0034] **Fig.** 11 ist ein Blockdiagramm einer Schaltung, die durch die Texturabbildungshardware der

vorliegenden Erfindung verwendet wird;

[0035] **Fig.** 12 ist ein Diagramm eines Beispiels eines Satzes von Textur-MIP-Abbildungen;

[0036] **Fig.** 13 ist ein Diagramm, das darstellt, wie die MIP-Abbildungen des Beispiels von **Fig.** 12 im Speicher gespeichert werden, gemäß einem Speicherspeicherungsschema der vorliegenden Erfindung;

[0037] **Fig.** 14 ist ein Diagramm einer MIP-Abbildung, die darstellt, wie die MIP-Abbildung gemäß einem Speicherspeicherungsschema der vorliegenden Erfindung unterteilt ist;

[0038] **Fig.** 15 ist ein detaillierteres Diagramm von Abschnitten der Abbildung, die in **Fig.** 14 gezeigt ist, das darstellt, wie die Abbildung gemäß einem Speicherspeicherungsschema der vorliegenden Erfindung weiter unterteilt ist;

[0039] **Fig.** 16 ist ein Diagramm, das die Art und Weise darstellt, auf die das Cacheblocktag erzeugt wird;

[0040] **Fig.** 17 ist ein Flußdiagramm, das ein Verfahren zum Bestimmen der Texeladresse mit einem entsprechenden Texturdatenblock von interpolierten Texeldaten darstellt;

[0041] **Fig.** 18 ist ein Flußdiagramm, das ein Verfahren zum Bestimmen darstellt, welcher Cacheblock ausgetauscht werden sollte, wenn ein Cachefehlschlag auftritt;

[0042] **Fig.** 19 ist ein Diagramm, das die Texeltorregister darstellt, die in dem Texturabbildungschip vorgesehen sind;

[0043] **Fig.** 20 ist ein Flußdiagramm, das ein Verfahren zum Warten von Cachefehlschlagunterbrechungen in dem Hostcomputer darstellt;

[0044] **Fig.** 21 ist ein Blockdiagramm des Cacheminiverzeichnisses;

[0045] **Fig.** 22 ist ein Blockdiagramm des Cachehauptverzeichnisses;

[0046] **Fig.** 23 ist ein Blockdiagramm einer Reihe von Komparatoren, die bereitgestellt werden, um Leistungsstrafen zu reduzieren, wenn ein Cachelesetag in dem Miniverzeichnis fehlschlägt; und

[0047] **Fig.** 24 ist ein Blockdiagramm einer darstellenden Implementierung des Cacheverzeichnisses der vorliegenden Erfindung.

Detaillierte Beschreibung

I. Systemübersicht

[0048] **Fig.** 2 ist ein Blockdiagramm eines Ausführungsbeispiels eines Graphiksystems der vorliegenden Erfindung, das Texturabbildungshardware umfaßt, die einen Cachespeicher zum lokalen Speichern von Texturdaten aufweist. Es sollte klar sein, daß die gezeigte dargestellte Implementierung bezüglich der Anzahl von Platinen und Chips, der Art und Weise, wie dieselben unterteilt sind, der Busbreiten und der Datenübertragungsraten nur beispielhaft ist. Zahlreiche andere Implementierungen können verwendet werden. Wie es gezeigt ist, umfaßt das System eine Vorderendeplatine **10**, eine Texturabbildungsplatine **12** und eine Rahmenpufferplatine **14**. Die Vorderendeplatine kommuniziert mit einem Hostcomputer **15** über einen 52-Bit-Bus **16**. Die Vorderendeplatine empfängt Grundelemente, die von dem Hostcomputer über den Bus **16** aufbereitet werden sollen. Die Grundelemente werden durch xyz-Vektorkoordinatendaten, RGB-Farbdaten und ST-Koordinaten spezifiziert, alle für Teile der Grundelemente, wie z. B. für die Scheitelpunkte, wenn das Grundelement ein Dreieck ist. Daten, die die Grundelemente in drei Richtungen darstellen, werden dann durch die Vorderendeplatine **10** an die Texturabbildungsplatine **12** und über den 85-Bit-Bus **18** an die Rahmenpufferplatine **14** geliefert. Die Texturabbildungsplatine interpoliert die Grundelementdaten, die empfangen wurden, um die Bildschirmanzeigepixel zu berechnen, die das Grundelement darstellen, und bestimmt entsprechende resultierende Texturdaten für jedes Grundelementpixel. Die resultierenden Texturdaten werden über fünf 55-Bit-Busse **28** an die Rahmenpufferplatine übertragen, die in **Fig.** 2 als ein einziger Bus gezeigt sind, um die Figur deutlicher zu machen.

[0049] Die Rahmenpufferplatine **14** interpoliert auch die Grundelementdaten, die von der Vorderendeplatine **10** empfangen werden, um die Pixel auf dem Anzeigebildschirm zu berechnen, die jedes Grundelement darstellen werden, und um Objektfarbwerte für jedes Pixel zu bestimmen. Die Rahmenpufferplatine kombiniert dann auf einer Pixel-um-Pixel-Basis die Objektfarbwerte mit den resultierenden Texturdaten, die von der Texturabbildungsplatine geliefert werden, um resultierende Bild-RGB-Werte für jedes Pixel zu erzeugen. RGB-Farbsteuersignale für jedes Pixel werden jeweils über die RGB-Leitungen **29** geliefert, um die Pixel des Anzeigebildschirms (nicht gezeigt) zu steuern, um ein resultierendes Bild auf dem Anzeigebildschirm anzuzeigen, das das texturabgebildete Grundelement darstellt.

[0050] Die Vorderendeplatine **10**, Texturabbildungsplatine **12** und die Rahmenpufferplatine **14** sind jeweils pipelineartig angeordnet und bearbeiten mehrere Grundelemente gleichzeitig. Obwohl die Texturabbildungs- und Rahmenpufferplatine an Grundelementen arbeiten, die vorher durch die Vorderendeplatine geliefert wurden, bearbeitet die Vorderendeplatine weiterhin neue Grundelemente und liefert dieselben, bis die Pipelines in den Platinen **12** und **14** voll werden.

[0051] Die Vorderendeplatine **10** umfaßt einen Verteilerchip **30**, drei dreidimensionale(3D-)Geometriebeschleunigerchips **32A**, **32B** und **32C**, einen zweidimensionalen(2D-)Geometriebeschleunigerchip **34** und einen Konzentratorchip **36**. Der Verteilerchip **30** empfängt die XYZ-Koordinaten- und Farbgrundelementdaten über den Bus **16** von dem Rostcomputer und verteilt 3D-Grundelementdaten gleichmäßig zwischen den 3D-Geometriebeschleunigerchips **32A**, **32B** und **32C**. Auf diese Weise ist die Systemband-

breite erhöht, weil drei Gruppen von Grundelementen gleichzeitig bearbeitet werden. Daten werden über den 40-Bit-Bus **38A** an die 3D-Geometriebeschleunigerchips **32A** und **32B** geliefert, und über den 40-Bit-Bus **38B** an den Chip **32C**. Beide Busse **38A** und **38B** übertragen Daten bei einer Rate von 60 MHz und liefern eine ausreichende Bandbreite zum Unterstützen von zwei 3D-Geometriebeschleunigerchips. 2D-Grundelementdaten werden bei einer Rate von 40 MHz über einen 44-Bit-Bus **40** an den 2D-Geometriebeschleunigerchip **34** geliefert.

[0052] Jeder 3D-Geometriebeschleunigerchip transformiert die xyz-Koordinaten, die die Grundelemente definieren, die empfangen wurden, in entsprechende Bildschirmraumkoordinaten, bestimmt Objekt-RGB-Werte und Textur-ST-Werte für die Bildschirmraumkoordinaten, zerlegt die Grundelementvierecke in Dreiecke und berechnet eine Dreiecksebenengleichung zum Definieren jedes Dreiecks. Jeder 3D-Geometriebeschleunigerchip führt auch Ansichtabschneideoperationen durch, um eine genaue Bildschirmanzeige des resultierenden Bildes sicherzustellen, wenn mehrere Fenster angezeigt werden oder wenn sich ein Teil eines Grundelements über das Ansichtsvolumen erstreckt, das auf dem Anzeigebildschirm dargestellt ist. Ausgangsdaten von den 3D-Geometriebeschleunigerchips **32A** und **32B** bzw. **32C** werden über 44-Bit-Busse **42A** und **42B** bei einer Rate von 60 MHz an einen Konzentratorchip **36** geliefert. Der zweidimensionale Geometriebeschleunigerchip **34** liefert auch über einen 46-Bit-Bus **44** Ausgangsdaten an den Konzentratorchip **36**, bei einer Rate von 45 MHz. Der Konzentratorchip **36** kombiniert die 3D-Grundelementausgangsdaten, die von den 3D-Geometriebeschleunigerchips **32A**–**C** empfangen werden, sortiert die Grundelemente neu in der ursprünglichen Reihenfolge, die sie vor der Verteilung durch den Verteilerchip **30** hatten, und liefert die kombinierten Grundelementausgangsdaten über den Bus **18** an die Texturabbildungs- und Rahmenpufferplatinen.

[0053] Die Texturabbildungsplatine **12** umfaßt einen Texturabbildungschip **46** und einen lokalen Speicher **48**, der vorzugsweise als ein Cachespeicher angeordnet ist. Bei einem bevorzugten Ausführungsbeispiel der Erfindung wird der lokale Speicher aus einer Mehrzahl von SDRAM-Chips (SDRAM = synchronous dynamic random access memory = synchroner dynamischer Direktzugriffsspeicher) gebildet, aus nachfolgend erörterten Gründen. Wie es nachfolgend näher beschrieben ist, speichert der Cachespeicher **48** Textur-MIP-Abbildungsdaten, die den Grundelementen zugeordnet sind, die aufbereitet werden sollen, in der Rahmenpufferplatine. Die Textur-MIP-Abbildungsdaten werden von einem Hauptspeicher **17** des Hostcomputers **15** über den Bus **40** durch den 2D-Geometriebeschleunigerchip **34** und über den 24-Bit-Bus **24** heruntergeladen.

[0054] Der Texturabbildungschip **46** empfängt nacheinander Grundelementdaten über den Bus **18**, die die Grundelemente darstellen, die auf dem Anzeigebildschirm aufbereitet werden sollen. Wie es oben erörtert wurde, umfassen die Grundelemente, die von den 3D-Geometriebeschleunigerchips **32A**–**C** geliefert werden, Punkte, Linien und Dreiecke. Die Texturabbildungsplatine führt keine Texturabbildung von Punkten und Linien durch und bearbeitet nur Dreiecksgrundelemente. Die Daten, die die Dreiecksgrundelemente darstellen, umfassen die xyz-Objektpixelkoordinaten für zumindest einen Scheitelpunkt, die Objektfarb-RGB-Werte des zumindest einen Scheitelpunkts, die Koordinaten in ST der Abschnitte der Texturabbildung, die dem zumindest einem Scheitelpunkt entsprechen, und die Ebenengleichung des Dreiecks. Der Texturabbildungschip **46** ignoriert die Objektpixel-z-Koordinate und die Objektfarben-RGB-Werte. Der Chip **46** interpoliert die xy-Pixelkoordinaten und interpoliert die S- und T-Koordinaten, die jedem xy-Bildschirmanzeigepixel entsprechen, das das Grundelement darstellt. Für jedes Pixel greift der Texturabbildungschip auf den Abschnitt der Textur-MIP-Abbildung zu, die demselben von dem Cachespeicher entspricht, und berechnet resultierende Texturdaten für das Pixel, die einen gewichteten Mittelwert der mehreren Texel umfassen können.

[0055] Bei einem beispielhaften Ausführungsbeispiel speichert der Cache **64** Blöcke von 256 × 256 Texeln. Anders als der lokale Speicher, der in der Texturabbildungshardware herkömmlicher Systeme verwendet wird, kann der Cachespeicher der vorliegenden Erfindung nicht die gesamte Reihe von MIP-Abbildungen der Textur speichern, die auf das Grundelement abbildet, das aufbereitet wird, wie z. B. für große Texturen. Statt dessen speichert der Cachespeicher zu jedem Zeitpunkt nur die speziellen Teile der Reihen von MIP-Abbildungen, die tatsächlich beim aktuellen Aufbereiten des Grundelements verwendet werden. Daher wird für die meisten Anwendungen nur ein Teil der vollständigen Texturdaten für das Bild, das aufbereitet wird, zu einem Zeitpunkt in dem Cachespeicher gespeichert.

[0056] Die vollständige Reihe von MIP-Abbildungen für jede Textur ist in dem Hauptspeicher **17** des Hostcomputers **15** angeordnet und gespeichert. Für jedes Pixel des Grundelements das aufbereitet wird, greift der Texturabbildungschip **46** auf ein Verzeichnis des Cachespeichers **48** zu, um zu bestimmen, ob das entsprechende Texel oder die Texel der Textur-MIP-Abbildungen derzeit in dem Cache vorliegen. Falls die entsprechenden Texel zum Zeitpunkt des Zugriffs in dem Cachespeicher gespeichert sind, tritt ein Cachetreffer auf und die Texel werden von dem Cache gelesen und durch den Texturabbildungschip **46** bearbeitet, um die resultierenden Texturdaten zu berechnen, die zu der Rahmenpufferplatine geleitet werden.

[0057] Falls jedoch die entsprechenden Texel für das Grundelementpixel nicht im Cachespeicher gespeichert sind, wenn durch den Texturabbildungschip

46 auf dieselben zugegriffen wird, tritt ein Cachefehlschlag auf. Wenn ein Cachefehlschlag auftritt, wird der Teil der Textur-MIP-Abbildungsdaten, der benötigt wird, um das Grundelement aufzubereiten, von dem Hauptspeicher **17** des Hostcomputers **15** in den Cachespeicher **48** heruntergeladen und ersetzt möglicherweise einige Daten, die vorher in demselben gespeichert waren. Anders als herkömmliche Texturabbildungssysteme, die die gesamte Reihe von MIP-Abbildungen für jedes Grundelement, das aufbereitet wird, herunterladen, lädt die vorliegende Erfindung nur den Teil der Reihe von MIP-Abbildungen herunter, die tatsächlich benötigt werden, um das Grundelement oder den aktuell aufbereiteten Abschnitt desselben aktuell aufzubereiten.

[0058] Wenn ein Cachefehlschlag auftritt, wie es nachfolgend näher beschrieben ist, wird durch den Texturabbildungschip **46** ein Unterbrechungssteuersignal erzeugt, um einen Texturunterbrechungsverwalter in dem Hostcomputer **15** auszulösen. Das Unterbrechungssteuersignal wird über die Leitung **94** an den Verteilerchip **30** geliefert, der wiederum über die Leitung **95** ein Unterbrechungssignal an den Hostcomputer liefert.

[0059] Die angeforderten Texturdaten werden durch den Hostcomputer von seinem Hauptspeicher wiedergewonnen und über den Bus **24** auf die Texturabbildungsplatine **48** heruntergeladen, wodurch die 3D-Grundelementaufbereitungspipeline durch die Vorderendeplatine und den Texturabbildungschip umgangen wird. Wenn somit eine Cachefehlschlagunterbrechung auftritt, kann die Vorderendeplatine weiterhin 3D-Grundelemente bearbeiten, und Ausgangsgrundelementdaten über den Bus **18** an den Texturabbildungschip und die Rahmenpufferplatine über den Bus **18** liefern, während die Texturdaten, die einem Grundelement zugeordnet sind, das den Cachefehlschlag bewirkt hat, von dem Hauptspeicher **17** heruntergeladen werden. Im Gegensatz zu herkömmlichen Texturabbildungssystemen erfordert das Herunterladen von Texturdaten zu der Texturabbildungshardware kein Entleeren der 3D-Grundelementpipeline und erhöht dadurch die Bandbreite und die Leistungsfähigkeit des Systems. Die resultierenden Texturdaten für jedes Pixel werden durch den Texturabbildungschip **46** an die Rahmenpufferplatine über fünf Busse **28** geliefert. Die fünf Busse **28** sind jeweils mit fünf Rahmenpuffersteuerchips **50A**, **50B**, **50C**, **50D** und **50E** gekoppelt, die an der Rahmenpufferplatine vorgesehen sind, und liefern parallel resultierende Texturdaten an die Rahmenpuffersteuerchips. Die Rahmenpuffersteuerchips **50A**–**E** sind jeweils mit Gruppen von zugeordneten VRAM-Chips **51A**–**E** gekoppelt (VRAM = video random access memory = Videodirektzugriffsspeicher). Die Rahmenpufferplatine umfaßt ferner vier Videoformatchips **52A**, **52B**, **52C** und **52D** und einen RAMDAC **54** (RAMDAC = random access memory digital-to-analog converter = Direktzugriffsspeicher-Digital/Analog-Wandler). Die Rahmenpuffersteuerchips steuern unterschiedliche, nichtüberlappende Segmente des Anzeigebildschirms. Jeder Rahmenpuffersteuerchip empfängt Grundelementdaten von der Vorderendeplatine über den Bus **18** und resultierende Texturabbildungsdaten von der Texturabbildungsplatine über den Bus **28**. Die Rahmenpuffersteuerchips interpolieren die Grundelementdaten zum Berechnen der Bildschirmanzeigepixelkoordinaten in ihren jeweiligen Segmenten, die das Grundelement darstellen, und die entsprechenden Objekt-RGB-Farbwerte für jede Pixelkoordinate. Für diese Grundelemente (d. h. Dreiecke), für die resultierende Texturdaten von der Texturabbildungsplatine geliefert werden, kombinieren die Rahmenpuffersteuerchips auf einer Pixel-mal-Pixel-Basis die Objektfarbwerte und die resultierenden Texturdaten zum Erzeugen von End-RGB-Werten für jedes Pixel, das auf dem Anzeigebildschirm angezeigt werden soll.

[0060] Die Art und Weise, wie die Objekt- und Texturfarbwerte kombiniert werden, kann auf eine Anzahl von unterschiedlichen Möglichkeiten gesteuert werden. Beispielsweise können die Objektfarbwerte in einem Austauschmodus einfach durch die Texturfarbwerte ausgetauscht werden, so daß beim Aufbereiten des Pixels nur die Texturfarbwerte verwendet werden. Alternativ können in einem Moduliermodus die Objekt- und Texturfarbwerte miteinander multipliziert werden, um die End-RGB-Werte für das Pixel zu erzeugen. Ferner kann ein Farbsteuerwert für jedes Texel gespeichert werden, das ein Verhältnis spezifiziert, das die Art und Weise definiert, wie die entsprechenden Texturfarbwerte mit den Objektfarbwerten kombiniert werden sollen. Ein resultierendes Farbsteuerwort kann für die resultierenden Texeldaten bestimmt werden, die jedem Pixel entsprechen, und über den Bus **28** an die Rahmenpuffersteuerchips geliefert werden, so daß die Steuerchips das Verhältnis verwenden können, das durch das entsprechende resultierende Steuerwort spezifiziert wird, um die End-RGB-Werte für jedes Pixel zu bestimmen.

[0061] Die resultierenden Bildvideodaten, die durch die Rahmenpuffersteuerchips **50A**–**E** erzeugt werden, einschließlich RGB-Werten für jedes Pixel, sind in den entsprechenden VRAM-Chips **52A**–**E** gespeichert. Jede Gruppe von VRAM-Chips **51A**– **E** umfaßt acht VRAM-Chips, so daß 40 VRAM-Chips auf der Rahmenpufferplatine positioniert sind. Jedes der Videoformatchips **52A**–**D** ist mit einem anderen Satz von zehn VRAM-Chips verbunden und empfängt Daten von demselben. Die Videodaten werden seriell aus den VRAM-Chips verschoben und jeweils über 64-Bit-Busse **58A**, **58B**, **58C** und **58D** zu den vier Videoformatchips **52A**, **52B**, **52C** und **52D** bei einer Rate von 33 MHz geliefert. Die Videoformatchips formatieren die Videodaten, so daß dieselben durch den RAMDAC gehandhabt werden können, und liefern die formatierten Daten über 32-Bit-Busse **60A**, **60B**, **60C** und **60D** an den RAMDAC **54** bei einer Rate von 33 MHz. Der RAMDAC **54** wandelt wiederum die Digitalfarbdaten zu analogen RGB-Farbsteuersignalen

um und liefert die RGB-Steuersignale für jedes Pixel entlang RGB-Steuerleitungen **29** an eine Bildschirmanzeige (nicht gezeigt).

[0062] Bei einem Ausführungsbeispiel der Erfindung wird Hardware auf der Texturabbildungsplatine **12** und der Rahmenpufferplatine **14** reproduziert, so daß bestimmte Grundelementaufbereitungsaufgaben an mehreren Grundelementen parallel durchgeführt werden können, wodurch die Bandbreite des Systems erhöht wird. Ein Beispiel eines solchen alternativen Ausführungsbeispiels der vorliegenden Erfindung ist in **Fig.** 2A gezeigt, die ein Blockdiagramm eines Computergraphiksystems der vorliegenden Erfindung ist, bei dem bestimmte Hardware reproduziert ist. Das System von **Fig.** 2A umfaßt vier 3D-Geometriebeschleunigerchips **32A**, **32B**, **32C** und **32D**, zwei Texturabbildungschips **46A** und **46B**, die jeweils Cachespeichern **48A** und **48B** zugeordnet sind, und zehn Rahmenpufferchips **50A**, **50J**, jeweils mit einer zugeordneten Gruppe von VRAM-Chips. Der Betrieb des Systems von **Fig.** 2A ist ähnlich wie der des Systems von **Fig.** 2, das oben beschrieben ist. Die Reproduktion der Hardware bei dem Ausführungsbeispiel von **Fig.** 2A ermöglicht eine erhöhte Systembandbreite, weil bestimmte Grundelementaufbereitungsoperationen parallel an mehreren Grundelementen durchgeführt werden können.

II. Texturabbildungschipübersicht

[0063] Ein Blockdiagramm des Texturabbildungschips **46** ist in **Fig.** 3 gezeigt. Der Chip **46** umfaßt eine Vorderendepipelineschnittstelle **60**, die Objekt- und Texturgrundelementdaten von der Vorderendeplatine über den 64-Bit-Bus **18** empfängt. Die dreieckigen Grundelemente, die durch den Texturabbildungschip bearbeitet werden, sind durch bis zu **52** 32-Bit-Digitalwörter definiert, können aber durch Wörter mit verschiedenen Längen definiert werden. Die Pipelineschnittstelle umfaßt einen Satz von Masterregistern und einen Satz von entsprechenden Slave-Registern. Während dem Aufbereiten werden die Masterregister nacheinander mit den 52 Digitalwörtern von Daten gefüllt, die das Grundelement definieren. Dann werden die Daten auf den Empfang eines geeigneten Aufbereitungsbefehls hin in die Slave-Register in der Pipelineschnittstelle verschoben, was es auf eine pipelineartige Weise ermöglicht, daß die Masterregister mit Daten gefüllt werden, die ein anderes Grundelement darstellen. Die Grundelementdaten, die über den Bus **18** geliefert werden, umfassen die xyz-Vektorkoordinatendaten, die ST-Texturkoordinaten und die RGB-Objektfarbdaten für zumindest einen dreieckigen Scheitelpunkt, und auch Daten, die die Dreiecksebenengleichung darstellen. Wie es oben erörtert wurde, ignoriert der Texturabbildungschip die Objektpixelz-Koordinate und die Objektfarb-RGB-Werte und speichert nur die anderen Daten in der Vorderendepipelineschnittstelle **60**.

[0064] Die Slave-Register der Pipelineschnittstelle **60** übertragen die Grundelementdaten über den Bus **62** an eine Parameterinterpolatorschaltung **64**. Die Parameterinterpolatorschaltung **64** interpoliert jedes Grundelementdreieck um für jede Anzeigebildschirmpixelkoordinate, die das Dreieck darstellt, die ST-Texturabbildungskoordinaten für die Texturabbildung, die auf das Pixel abbildet, und einen S- und einen T-Gradientenwert (ΔS, ΔT) zu bestimmen. Der S- und der T-Gradient entsprechen jeweils gleichen Änderungen bei den S- und T-Koordinaten zwischen benachbarten Pixeln und werden auf eine nachfolgend erörterte Weise berechnet.

[0065] Die Parameterinterpolatorschaltung **64**, die in **Fig.** 4 näher gezeigt ist, umfaßt einen Kantenschrittgeber **66**, einen FIFO-Puffer **68** (FIFO = first-in, first-out = Zuerst-Hinein-Zuerst-Hinaus), einen Spannweiteschrittgeber **70** und eine Gradienten- und Perspektivenkorrekturschaltung **72**, die alle in Reihe geschaltet sind. Der Kantenschrittgeber beginnt bei der xy-Pixelkoordinate von einem der Dreieckscheitelpunkte und durchläuft schrittweise die Kanten des Dreiecks unter Verwendung der Dreiecksebenengleichung, um die Pixelkoordinaten zu bestimmen, die die Dreieckskanten definieren. Für jede Pixelkoordinate werden Texturabbildungs-S-und T-Koordinaten bestimmt, auf der Basis der ST-Werte der Dreiecksscheitelpunkte, um zu identifizieren, welche Texel in der Texturabbildung jeder Anzeigebildschirmpixelkoordinate entsprechen. Die Pixel- und Texelkoordinaten werden vorübergehend in dem FIFO-Puffer gespeichert und dann an den Spannweiteschrittgeber geliefert. An jeder xy-Pixelposition entlang einer Kante des Dreiecks durchläuft der Spannweiteschrittgeber schrittweise den entsprechenden Umfang des Dreiecks, um die ST-Texelkoordinaten für jede Pixelposition entlang dem Umfang zu bestimmen.

[0066] Jede S- und T-Koordinate für ein Anzeigebildschirmpixel kann einen Ganzzahlabschnitt und einen Bruchteilabschnitt aufweisen, falls das Pixel nicht direkt (in einer Eins-zu-Eins-Entsprechung) auf ein einziges Texel in einer der Reihe von MIP-Abbildungen für die Textur abbildet. Wenn es auf die Texturabbildung abgebildet ist, wie es oben erklärt ist, kann ein Anzeigebildschirmpixel zwischen mehreren Texeln in einer der Reihe von MIP-Abbildungen für die Textur liegen, und kann ferner zwischen (in der Größe) benachbarten MIP-Abbildungen in der Reihe liegen.

[0067] Die Gradienten- und Perspektivenkorrekturschaltung **72** bestimmt die Gradientenwerte von S und T (ΔS, ΔT) für jedes Anzeigebildschirmpixel. Bei einem Ausführungsbeispiel der Erfindung ist der Gradient ΔS ausgewählt, um der Größere der Gradienten ΔSx und ΔSy zu sein, wobei der Gradient ΔSx die Änderung bei der S-Koordinate in der Texturabbildung ist, wenn sich die Koordinate x zwischen benachbarten Pixeln auf dem Anzeigebildschirm ändert, und der Gradient ΔSy die Änderung bei der S-Koordinate ist, wenn sich die Koordinate y zwischen benachbarten Pixeln ändert. Der Gradient ΔT

wird gleichartig berechnet. Die Gradienten ΔS, ΔT für ein Anzeigebildschirmpixel zeigen die Änderungsrate bei der Koordinatenposition in der Texturabbildung für eine Änderung eines Pixels auf dem Anzeigebildschirm in der entsprechenden ST-Dimension an und werden verwendet, um zu bestimmen, auf welche MIP-Abbildung oder Abbildungen zugegriffen werden sollte, um die resultierenden Texturdaten für das Pixel zu liefern. Beispielsweise zeigt ein Gradient gleich 2 für ein Anzeigebildschirmpixel an, daß das Pixel auf vier (d. h. 2^2, wie es nachfolgend erörtert ist) Texel abbildet, so daß auf die MIP-Abbildung, die von der Basisabbildung (z. B. der **Abb.** 102 in **Fig.** 1) in der Größe um 2 reduziert ist, zugegriffen werden sollte, um die resultierenden Texturdaten für das Pixel zu liefern. Während sich somit der Gradient erhöht, ist die Größe der MIP-Abbildung, auf die zugegriffen wird, um die resultierenden Texturdaten für das Pixel zu liefern, reduziert.

[0068] Bei einem Ausführungsbeispiel der Erfindung wird ein einziger Gradient, der gleich dem Größeren von ΔS und ΔT ist, verwendet, um die geeignete MIP-Abbildung für jedes Pixel auszuwählen, so daß der Gradient gleich dem Größten von ΔSx, ΔSy, ΔTx und ΔTy für das Pixel ist. Es sollte jedoch klar sein, daß der Gradient alternativ auf unterschiedliche Weise ausgewählt werden kann, wie z. B. durch Auswählen des Kleinsten dieser Werte, einem Mittelwert dieser Werte oder eine andere Kombination. Da ein einziger Gradient ausgewählt wird, der die Änderungsrate bei nur einer der ST-Koordinaten anzeigt, stellt das Quadrat des Gradienten die Anzahl von Texeln dar, die auf das entsprechende Pixel abbilden.

[0069] Von dem Gradienten bestimmt der Parameterinterpolator die nächstliegende Abbildung, auf die das Pixel abbildet, und einen Wert, der anzeigt, um wieviel das Pixel von der Abbildung direkt zu dieser Abbildung schwankt. Die nächste Abbildung wird durch den Gesamtzahlteil einer Abbildungsnummer identifiziert, der Wert, der anzeigt, um wieviel das Pixel von einer direkten Abbildung abweicht, wird durch eine Bruchteilkomponente der Abbildungsnummer identifiziert.

[0070] Mit erneuter Bezugnahme auf das Blockdiagramm des Texturabbildungschips in **Fig.** 3 werden die Texeldaten, die von der Parameterinterpolatorschaltung **64** ausgegeben werden, über die Leitung **70** an einen Kachel- und Grenzprüfer **72** geliefert, der die Adresse der vier Texel bestimmt, die am nächsten zu der Position in jeder der Texturabbildungen liegen, die durch die Texeldaten bestimmt sind, und prüft, um zu bestimmen, ob jedes innerhalb der Grenze der Textur liegt. Die Texeldaten umfassen die interpolierten ST-Koordinaten (Ganzzahl- und Bruchteilwerte) und auch die Abbildungszahl und den Abbildungsbruchteil. Die Kachelungsvorrichtung verwendet den Ganzzahlabschnitt der S- und T-Koordinaten, die durch den Parameterinterpolator **64** berechnet werden, und fügt Eins zu dem Ganzzahlabschnitt von jedem hinzu, um die Adressen der vier nächstliegenden Texel zu erzeugen. Der Grenzprüfer bestimmt dann, ob die ST-Koordinaten für jedes dieser vier Texel außerhalb der Grenze der Texturabbildung fallen. Falls ein Anzeigebildschirmpixel auf eine ST-Koordinaten-position abbildet, die außerhalb der Grenze der Texturabbildung fällt, wird eines von mehreren Texturabbildungsschemata implementiert, um zu bestimmen, ob resultierende Texturdaten für dieses Pixel erzeugt werden sollen und wie diese Daten erzeugt werden sollen. Beispiele solcher Schemata umfassen Umwickeln (eine Wiederholung der Textur), Spiegeln (eine Wiederholung des Spiegelbildes der Textur), Abschalten der Texturabbildung außerhalb der Grenze und Anzeigen einer durchgehenden Farbe außerhalb der Grenze.

[0071] Die Fähigkeit, es einem Pixel zu ermöglichen, auf eine Position in einer Texturabbildung abzubilden, die über die Grenze desselben hinaus liegt, liefert eine Flexibilität bei der Art und Weise, wie Texturen auf Objektgrundelemente abgebildet werden können. Beispielsweise kann es wünschenswert sein, eine Textur auf eine wiederholende Weise auf ein Objekt abzubilden, so daß die Textur auf mehrere Abschnitte des Objekts abgebildet wird. Falls beispielsweise eine Textur so definiert ist, daß sie ST-Koordinaten aufweist, die von einschließlich [0, 0] bis ausschließlich (10, 10) reicht, könnte ein Benutzer bestimmte Abschnitte des Objekts spezifizieren, um auf die ST-Koordinaten einschließlich [10, 10] bis ausschließlich (20, 20) abzubilden. Die Notation der in eckigen Klammern geschriebenen eingeschlossenen Koordinaten zeigt an, daß diese Koordinaten in dem Abschnitt der Textur enthalten sind, der auf das Objekt abgebildet ist, während das Objekt nur auf die ST-Koordinaten bis zu, aber nicht einschließlich, der ausgeschlossenen Koordinaten in den Klammern abbildet. Falls das Umhüllungsmerkmal für ST-Koordinaten ausgewählt wird, die außerhalb die Grenze der Textur fallen, würden Pixel mit den ST-Koordinaten [10, 10] bis (20, 20) jeweils auf die Texel an den ST-Koordinaten [0, 0] bis (10, 10) abbilden.

[0072] Wie es oben erörtert wurde, können die resultierenden Texturdaten von einer zweidimensionalen Texturabbildung für ein einziges Pixel das Ergebnis einer Kombination von bis zu acht Texeln sein, d. h. den vier nächstliegendsten Texeln in den beiden nächstliegendsten MIP-Abbildungen. Es gibt eine Anzahl von Möglichkeiten, wie die acht Texel kombiniert werden können, um die resultierenden Texeldaten zu erzeugen. Beispielsweise kann das einzige nächstliegendste Texel in der nächstliegendsten Abbildung ausgewählt werden, so daß keine Mittelwertbildung erforderlich ist. Alternativ kann das einzige nächstliegende Texel in jeder. der beiden nächstliegendsten Abbildungen auf der Basis des Werts des Gradienten Bemittelt werden. Solche Schemata bilden die Textur nicht so genau ab, wie wenn die acht nächstliegendsten Texel Bemittelt werden.

[0073] Bei einem Ausführungsbeispiel der Erfindung wird eine trilineare Interpolation unterstützt, wo-

bei die resultie- renden Texturdaten für ein einziges Pixel als gewichteter Mittelwert von bis zu acht Texeln berechnet werden können. Der Gradient, der Änderungsraten von S, T darstellt, wird verwendet, um die beiden nächstliegendsten MIP-Abbildungen zu identifizieren, von denen auf die Texturdaten zugegriffen wird, und auf die vier nächstliegendsten Texel in jede Abbildung wird zugegriffen. Der Mittelwert der vier Texel in jeder Abbildung wird gewichtet auf der Basis, welche Texel am nächsten zu den ST-Koordinaten der Position in des MIP-Abbildung sind, auf die das Anzeigebildschirmpixel abbildet. Der Bruchteilabschnitt der S- und T-Koordinaten für das Pixel wird verwendet, um diese Gewichtung durchzuführen. Der Mittelwert von jeder der beiden nächsten MIP-Abbildungen wird dann auf der Basis des Werts des Gradienten gewichtet. Ein Bruchteilwert wird für die Verwendung bei diesem Gewichtsprozeß von dem Gradienten berechnet. Beispielsweise liegt ein Gradient von 3 auf halbem Weg zwischen den MIP-Abbildungen, die jeweils den Gradienten von 2 und 4 entsprechen.

[0074] Der Texelinterpolationsprozeß wird durch die Texelinterpolatoren **76** durchgeführt. Die Bruchteilabschnitte der S- und T-Koordinaten für jedes Anzeigebildschirmpixel werden von den Parameterinterpolatoren durch den Kachel-/Grenzprüfer, an den Texelinterpolator **76** über die Leitungen **74** gelie- fert. Die Bruchteilabschnitte werden durch den Texelinterpolator verwendet, um das Gewicht zu bestimmen, das für jedes Texel während der Interpolation der mehreren Texel benötigt wird, wenn resultierende Texeldaten berechnet werden.

[0075] Wie es oben erörtert wird, sind Textur-MIP-Abbildungen, die einem Grundelement zugeordnet sind, das aufbereitet wird, lokal in dem Cachespeicher **48** gespeichert (**Fig.** 2). Bei einem Ausführungsbeispiel der Erfindung ist der Cache vollassoziativ. Der Cache umfaßt acht SDRAM-Chips, die in vier Verschachtelungen unterteilt sind, mit zwei SDRAM-Chips in jeder Verschachtelung. Vier getrennte Steuerungen sind vorgesehen, wobei eine Steuerung jeder Verschachtelung entspricht, so daß auf die SDRAM-Chips in jeder Verschachtelung gleichzeitig zugegriffen werden kann. Jeder SDRAM-Chip umfaßt zwei getrennte Speicherbänke, in denen in aufeinanderfolgenden Lesezyklen auf unterschiedliche Speicherseiten zugegriffen werden kann, ohne Seitenumnummerierungsstrafen bzw. Repaging-Strafen auf sich zu laden, die im allgemeinen dem Zugreifen von Daten von zwei unterschiedlichen Seiten (d. h. von zwei unterschiedlichen Zeilenadressen) in einem herkömmlichen DRRM zugeordnet sind.

[0076] Die Texturdaten (d. h. die MIP-Abbildungen) sind in Texelblöcke von Daten unterteilt, die jeweils 256 × 256 Texel umfassen. Der Cachespeicher kann bis zu **64** Datenblöcke zu einem Zeitpunkt speichern. Jeder Block weist ein zugeordnetes Blocktag auf, das den Block eindeutig identifiziert. Der Cache umfaßt ein Cacheverzeichnis **78**, das die Blocktags speichert, die den Datenblöcken entsprechen, die derzeit in dem Cache gespeichert sind. Wie es nachfolgend näher beschrieben wird, umfaßt jedes Blocktag einen Texturidentifizierer(Textur-ID), der die spezielle Textur identifiziert, die der Datenblock darstellt, eine Abbildungsnummer, die die spezielle MIP-Abbildung identifiziert, innerhalb der Reihe von Abbildungen der Textur, die der Datenblock darstellt, und höherwertige S- und T-Koordinaten, die die Position des Datenblocks in der speziellen Abbildung identifizieren. Die physikalische Position des Blocktags in dem Cacheverzeichnis stellt die Position des entsprechenden Datenblocks in dem Cachespeicher dar.

[0077] MIP-Abbildungen von mehr als einer Textur können gleichzeitig in dem Cachespeicher gespeichert werden, wobei der Texturidentifizierer zwischen den unterschiedlichen Texturen unterscheidet. Einige MIP-Abbildungen enthalten weniger als 256 × 256 Texel und verbrauchen daher keinen gesamten Datenblock. Beispielsweise können die kleineren Abbildungen in einer Reihe von MIP-Abbildungen oder selbst die größeren Abbildungen für kleine Texturen 256 × 256 Texel nicht überschreiten. Um Speicherplatz effizient zu nutzen, können Abschnitte von mehreren Abbildungen in einem einzigen Texturdatenblock gespeichert werden, wobei jeder Abbildungsabschnitt einem Teilblock in dem Block zugewiesen ist. Jede der mehreren Abbildungen, die in einem einzigen Block gespeichert ist, weist einen zugeordneten Untertexturidentifizierer (ID) auf, der die Position der Abbildung in dem Block identifiziert.

[0078] Während dem Aufbereiten erzeugt der Kachel-/Grenzprüfer **72** ein Lesecachetag für den Block von Texturdaten, der auf das Pixel abbildet, das aufbereitet werden soll. Die Art und Weise, wie die Tags erzeugt werden, ist nachfolgend näher erklärt. Die Tags sind 23-Bit-Felder, die acht Bits, die den Textur-ID der Texturdaten darstellen, ein Bit, das beim Bestimmen der Abbildungsanzahl der Texturdaten verwendet wird, und die sieben höherwertigen S- und T-Koordinaten der Texturdaten umfassen. Das Cacheverzeichnis **78** vergleicht das Lesecachetag, das von dem Kachel-/Grenzprüfer geliefert wird, mit den Blocktags, die in dem Verzeichnis gespeichert sind, um zu bestimmen, ob sich der Texturdatenblock, der beim Aufbereiten verwendet werden soll, in dem Cachespeicher befindet. Falls das Blocktag der Texturdaten, das auf das Grundelement abbildet, das aufbereitet werden soll, in dem Cacheverzeichnis gespeichert ist (d. h. dasselbe trifft), dann erzeugt das Cacheverzeichnis einen Blockindex, der die physikalische Position des Texturdatenblocks in dem Cache anzeigt, das dem Treffertag entspricht. Die Berechnung des Blockindexes ist nachfolgend näher erörtert. Eine Texeladresse wird auch durch den Kachel-/Grenzprüfer **72** für jedes Texel erzeugt, das von dem Cache gelesen wird, und zeigt die Position des Texels in dem Block an. Die Texeladresse umfaßt niederwertige Adreßbits der interpolierten ST-Koordina-

ten für größere Abbildungen und wird auf der Basis eines Algorithmus berechnet, der nachfolgend für Abbildungen kleinerer Größe beschrieben ist. Der Blockindex und die Texeladresse umfassen zusammen die Cacheadresse, die die Position des Texels in dem Cache anzeigt. Wie es nachfolgend näher beschrieben ist, sind die LSBs der S- und T-Koordinaten für jedes Texel decodiert, um zu bestimmen, in welcher der vier Cacheverschachtelungen das Texel gespeichert ist, und die verbleibenden Bits der Cacheadresse werden an die Texelcachezugriffsschaltung **82** geliefert, zusammen mit einem Befehl über die Leitung **84**, um die Texeldaten zu lesen, die an der adressierten Position in dem Cache gespeichert sind.

[0079] Wenn das Lesecachetag nicht mit einem der Blocktags übereinstimmt, die in dem Cacheverzeichnis **78** gespeichert sind, tritt ein Fehlschlag auf und das Cacheverzeichnis **78** erzeugt ein Unterbrechungssteuersignal über die Leitung **94** (**Fig.** 2) an den Verteilerchip **30** auf der Vorderendeplatine, die eine Unterbrechung über die Leitung **95** zu dem Rostcomputer **15** erzeugt. Ansprechend auf die Unterbrechung führt der Prozessor **19** des Hostcomputers eine Wartungsroutine aus, die nachfolgend näher erörtert ist, die das fehlgeschlagene Blocktag von dem Cacheverzeichnis liest und den entsprechenden Texturdatenblock auf eine Weise in den Cachespeicher herunterlädt, die die 3D-Grundelementpipeline in der Vorderendeplatine und dem Texturabbildungschip **46** umgeht. Die Texturdaten, die von dem Hauptspeicher heruntergeladen werden, werden über den Bus **24** durch das Texeltor **92** (**Fig.** 3) an die Texelcachezugriffsschaltung **82** geliefert, die die Daten an die SDRAMs liefert, die den Cachespeicher bilden.

[0080] Wenn ein Cachefehlschlag auftritt, wartet der Texturabbildungschip auf die neuen Texturdaten, die heruntergeladen werden sollen, bevor mit dem Verarbeiten des Grundelements fortgefahren wird, auf dem der Fehlschlag aufgetreten ist. Die Stufen der Pipeline, die dem Cachelesevorgang folgen, verarbeiten jedoch weiterhin die Grundelemente, die vor dem fehlgeschlagenen Grundelement aufgetreten sind. Die Stufen der Pipeline, die dem Cachelesevorgang vorausgehen, fahren in ähnlicher Weise ebenfalls damit fort, Grundelemente zu verarbeiten, außer und bis sich die Pipeline hinter die Cacheleseoperation auffüllt, während auf das Herunterladen der neuen Texturdaten gewartet wird.

[0081] Während dem Aufbereiten schreiten die späteren Stufen der Pipeline in der Rahmenpufferplatine **14** nicht mit dem Verarbeiten eines Grundelements fort, bis die Texturdaten, die dem Grundelement entsprechen, von der Texturabbildungsplatine empfangen werden. Wenn ein Cachefehlschlag auftritt und der Texturabbildungschip darauf wartet, daß die neuen Texturdaten heruntergeladen werden, wartet die Rahmenpufferplatine **14** daher gleichermaßen auf die resultierenden Texturdaten, die von dem Texturabbildungschip geliefert werden. Wie bei dem Texturabbildungschip fahren die Stufen der Pipeline, die der Stufe folgen, die die Texturabbildungsdaten empfängt, damit fort, die Grundelemente zu verarbeiten, die vor dem fehlgeschlagenen Grundelement empfangen wurden, und die Stufen der Pipeline, die der Stufe vorausgehen, die die Texturabbildungsdaten empfängt, verarbeiten auch weiterhin Grundelemente, außer und bis die Pipeline aufgefüllt ist.

[0082] Wenn die Pipeline der Texturabbildungsplatine oder der Rahmenpufferplatine sichert, wenn dieselbe ansprechend auf einen Cachefehlschlag auf neue Texturdaten wartet, sollte klar sein, daß die Pipeline in der Vorderendeplatine **10** auf gleiche Weise sichern wird. Weil Cachefehlschläge auftreten werden und zu einem Zugriff auf den Hostcomputerhauptspeicher und zu einem Herunterladen von Texturdaten führen, das mehrere Zyklen zum Abschließen benötigt, ist es wünschenswert, sicherzustellen, daß die Pipeline in dem Texturabbildungschip nie warten muß, weil die Pipeline in der Rahmenpufferplatine gesichert wurde. Daher wird bei einem Ausführungsbeispiel der Erfindung die Rahmenpufferplatine mit einer tieferen Grundelementpipeline versehen als die Texturabbildungsplatine, so daß die Texturabbildungspipeline nicht dadurch verzögert werden sollte, daß dieselbe darauf wartet, daß die Rahmenpufferpipeline verfügbar wird.

[0083] Bei einem Ausführungsbeispiel der Erfindung wird die Fähigkeit bereitgestellt, die Texturabbildung auszuschalten. Dies wird durch Software erreicht, die auf dem Prozessor **19** des Hostcomputers arbeitet, um ein Register sowohl in der Texturabbildungsplatine **12** als auch der Rahmenpufferplatine **14** zu setzen. Wenn dieselben eingestellt sind, um die Texturabbildung auszuschalten, hindern diese Register jeweils den Texturabbildungschip **46** daran, Texturdaten an die Rahmenpufferplatine **14** zu liefern, und weisen die Rahmenpufferplatine an, mit dem Aufbereiten von Grundelementen fortzufahren, ohne auf Texturdaten von der Texturabbildungsplatine zu warten.

[0084] Wie es oben beschrieben wurde, kann für jedes Anzeigebildschirmpixel, das mit Texturdaten von einer zweidimensionalen Texturabbildung aufbereitet wird, auf bis zu vier Texel von einer MIP-Abbildung (bilineare Interpolation) oder acht Texel von zwei benachbarten MIP-Abbildungen (trilineare Interpolation) von dem Cachespeicher zugegriffen werden, um die resultierenden Texturdaten für das Pixel zu bestimmen. Die Texel, die von dem Cache gelesen werden, werden über den Bus **86** (**Fig.** 3) an den Texelinterpolator **76** geliefert, der die mehreren Texel interpoliert, um resultierende Texeldaten für jedes Pixel zu berechnen. Die Interpolation kann abhängig von einem Modus variieren, der für das System erstellt wird. Wenn ein Punktabtastungsinterpolationsmodus erstellt wird, sind die resultierenden Texeldaten gleich dem einzelnen Texel, das am nächstliegendsten zu der Position ist, die durch die ST-Koordinaten des Pixels in der Texturabbildung definiert ist. Wenn alternativ eine bilineare oder trilineare Interpolation

verwendet wird, sind die resultierenden Texeldaten jeweils ein gewichteter Mittelwert der vier oder acht nächsten Texel in der einen oder den zwei nächstliegendsten Abbildungen. Das Gewicht, das jedem der mehreren Texel gegeben wird, wird auf der Basis des Werts des Gradienten und der Bruchteilkomponenten der S- und T-Koordinaten bestimmt, die von dem Kachel/Grenzprüfer an den Texelinterpolator **76** geliefert werden.

[0085] Die resultierenden Texeldaten für die Anzeigebildschirmpixel werden nacheinander über den Bus **88** an einen Rahmenpufferschnittstellen-FIFO-Puffer **90** geliefert. Der Rahmenpufferschnittstellen-FIFO-Puffer **90** kann bis zu 64 resultierende Texel speichern.

[0086] Jedes resultierende Texel ist ein 32-Bit-Wort, das acht Bits umfaßt, um R, G, B und α darzustellen. Das α-Byte zeigt der Rahmenpufferplatine **14** (**Fig.** 2) die Art und Weise an, wie die RGB-Werte der resultierenden Texturdaten mit den RGB-Werten der Objektdaten kombiniert werden sollten, die durch die Rahmenpufferplatine beim Berechnen der Endanzeigebildschirm-RGB-Werte für jedes Pixel, das auf das Texel abbildet, kombiniert werden sollten. Die Rahmenpufferschnittstellen-FIFO-Puffer-Ausgangssignale TO–T4 werden über den Bus **28** an die Rahmenpufferplatine **14** (**Fig.** 2) geliefert. Die Rahmenpufferplatine kombiniert die RGB-Werte der resultierenden Texeldaten mit den Objekt-RGB-Werten auf die Weise, die durch α spezifiziert wird, um für jedes Anzeigebildschirmpixel End-RGB-Werte zu erzeugen.

III. Cachespeicherorganisation

[0087] **Fig.** 5 ist ein Blockdiagramm einer Cachespeicherimplementierung gemäß einem darstellenden Ausführungsbeispiel der vorliegenden Erfindung, das mit Abschnitten des Texturabbildungschips gekoppelt ist, einschließlich dem Texeltor **92**, dem Texturinterpolator **76**, dem Cacheverzeichnis **78** und der Texelcachezugriffsschaltung **82**. Bei diesem darstellenden Ausführungsbeispiel umfaßt der Cachespeicher **48** vier Verschachtelungen **204A**, **204B**, **204C** und **204D**. Jede Verschachtelung umfaßt zwei SDRAM-Chips (nicht gezeigt), auf die gleichzeitig zugegriffen werden kann, wobei jedes während einem Lesezyklus acht Datenbits liefert. Daher liefert jede Verschachtelung während einem einzigen Lesezyklus **16** Bits an Texeldaten. Jedes 32-Bit-Wort von Texeldaten wird in dem Cache in einer einzigen Verschachtelung gespeichert, wobei 8 Bits in jeder von zwei aufeinanderfolgenden Positionen in jedem SDRAM in der Verschachtelung gespeichert werden. Um ein Texel von dem Cache zu lesen, werden somit zwei Lesezyklen an aufeinanderfolgenden Positionen in der geeigneten Verschachtelung zum Liefern der 32 Bits von Texeldaten durchgeführt. Wie es nachfolgend erklärt ist, muß nur ein Adreßwort (einschließlich Zeilenund Spaltendaten) zu den SDRAMs in jeder Verschachtelung geliefert werden, um ein Datenburst bzw. Datenbündel in zwei aufeinanderfolgenden Zyklen zu erzielen. Das Bündel umfaßt 16 Bits, die auf einem ersten Zyklus von der gegebenen Adresse geliefert werden, und 16 Bits, die auf einem zweiten Zyklus von einer Adresse geliefert werden, die die gleiche Zeile und eine Spalte, die um 1 inkrementiert ist, aufweist.

[0088] Die Texelcachezugriffsschaltung **82** umfaßt vier getrennte Steuerungen, die mit Steuerung A(200A), Steuerung B (200B), Steuerung C(200C) und Steuerung D(200D) bezeichnet sind. Die vier Steuerungen A, B, C und D können gleichzeitig auf Daten von den vier Verschachtelungen **204A**, **204B**, **204C** und **204D** zugreifen, durch parallele Busse **202A**, **202B**, **202C** und **202D**. Die Steuerungen lesen Texeldaten von dem Speicher **48**, ansprechend auf Befehle und an Adressen, die jeweils über die Busse **48A**, **48B**, **48C** und **48D** empfangen werden.

[0089] Wie es oben beschrieben wurde, kann jedes Pixel potentiell bis zu vier Texel von einer MIP-Abbildung oder acht Texel von mehreren MIP-Abbildungen abbilden. Wie es nachfolgend näher erörtert wird, sind Texeldaten, die in den Cache heruntergeladen werden, in dem Hauptspeicher des Hostcomputers organisiert, so daß je vier benachbarte Texel in jeder MIP-Abbildung in getrennten Verschachtelungen positioniert sind, so daß auf dieselben parallel zugegriffen werden kann. Somit können alle vier benachbarten Texel in einer MIP-Abbildung, die nötig sein können, um resultierende Texeldaten durch bilineare Interpolation zu erzeugen, in einer einzigen Leseoperation gelesen werden. Wenn eine trilineare Interpolation verwendet wird, können die beiden Sätze von vier Texeln von benachbarten MIP-Abbildungen in zwei Leseoperationen gelesen werden.

[0090] **Fig.** 6 stellt ein Beispiel der Art und Weise dar, wie Texturdatenblöcke (nur einige Texel sind gezeigt) organisiert sind, um die Vier-Verschachtelungs-Implementierungen des Cachespeichers auszunutzen, um es zu ermöglichen, daß alle vier benachbarte Texel in einer MIP-Abbildung gleichzeitig gelesen werden. Jedes Texel wird mit A, B, C oder D bezeichnet, um die Verschachtelung in dem Cachespeicher zu identifizieren, wo das Texel gespeichert wird. Die Struktur der A-D-Tags wiederholt sich, so daß jede Position in der Abbildung zwischen vier Texel fällt, die mit A, B, C und D bezeichnet sind. Somit sind für ein Pixel, das auf jede Position in der Abbildung abbildet, die vier nächsten Texel in einer getrennten Verschachtelung A–D, so daß auf dieselben durch die vier unabhängigen Steuerungen **200A**–**D** gleichzeitig zugegriffen werden kann. Beispielsweise bildet das Pixel P0 auf eine Position zwischen vier Texeln ab, die mit A, B, C und D bezeichnet sind, und das Pixel P1 bildet auf eine Position zwischen vier Texeln ab, die mit B, A, D und C bezeichnet sind.

[0091] Es sollte klar sein, daß die oben beschriebene Cacheimplementierung lediglich zu Darstellungszwecken geliefert wird und daß alternative Implementierungen verwendet werden können. Beispiels-

weise kann der Cache in acht getrennten Verschachtelungen implementiert werden, mit acht getrennten Steuerungen, so daß, wenn eine trilineare Interpolation verwendet wird, von dem Cache in einer einzigen Leseoperation gleichzeitig auf die acht Texel zugegriffen werden kann.

[0092] Jeder SDRAM-Chip in dem Cachespeicher ist intern in zwei gleich große Bänke unterteilt, die gleichzeitig getrennte aktive Seiten beibehalten können (d. h. Gruppen von Speicherpositionen, die eine gemeinsame Zeilenadresse haben). Somit kann bei aufeinanderfolgenden Lesezyklen von unterschiedlichen Seiten in den beiden Bänken eines SDRAM-Chips auf Daten zugegriffen werden, ohne die Repaging-Strafe auf sich zu laden, die üblicherweise einem aufeinanderfolgenden Lesen von Daten von unterschiedlichen Seiten in einem herkömmlichen DRAM zugeordnet ist.

[0093] Wie es nachfolgend näher erklärt ist, sind die Texturdaten in dem Cachespeicher organisiert, um dieses Merkmal der SDRAMs auszunutzen, Seitenkreuzungsstrafen zu minimieren, wenn eine trilineare Interpolation durchgeführt wird. Die acht Texel, die für eine trilineare Interpolation erforderlich sind, umfassen Sätze von vier Texeln von zwei MIP-Abbildungen. Jeder Satz von vier benachbarten Texeln in einer einzigen Abbildung ist angeordnet, so daß eine in jeder der Verschachtelungen A, B, C und D auf die Weise gespeichert ist, die oben beschrieben ist, so daß auf die vier Texel gleichzeitig zugegriffen werden kann. Ferner sind gemeinsame Daten von benachbarten MIP-Abbildungen in der Reihe von Abbildungen für jede Textur in dem Cache in unterschiedlichen SDRAM-Bänken gespeichert. Wenn eine trilineare Interpolation durchgeführt wird, werden vier Texel von einer MIP-Abbildung gleichzeitig von einer der SDRAM-Bänke von Verschachtelungen A–D gleichzeitig gelesen, während den zwei Lesezyklen eines ersten Burst, und vier Texel von einer benachbarten MIP-Abbildung werden von der anderen SDRAM-Bank während den beiden Lesezyklen eines nachfolgenden Burst gelesen. Weil zwei Bänke der SDRAM gleichzeitig zeilenaktiv sein können, kann auf die beiden Sätze von vier Texeln in Back-to-Back-Bursts zugegriffen werden, ohne eine Repaging-Strafe auf sich zu laden. Wenn Pixel eines Objekts aufbereitet werden, sollte klar sein, daß benachbarte Pixel häufig auf die gleichen zwei MIP-Abbildungen für die Textur abbilden, was erfordert, daß Lesevorgänge zu dem Cache fortlaufend zwischen den Cacheblöcken hin- und herschalten, die die gemeinsamen Daten in den beiden Abbildungen speichern. Die Cacheorganisation der vorliegenden Erfindung, die es ermöglicht, daß zwei Seiten in jedem SDRAM aktiv bleiben, ist vorteilhaft, weil sie ermöglicht, daß eine trilineare Interpolation durchgeführt wird, ohne eine Repaging-Strafe in jedem Zyklus auf sich zu laden, wenn während dem Aufbereiten von Anzeigebildschirmpixeln zwischen zwei benachbarten MIP-Abbildungen geschaltet wird.

[0094] **Fig.** 7 ist ein detaillierteres Blockdiagramm der oben beschriebenen darstellenden Implementierung des Cachespeichers der vorliegenden Erfindung. Der Cache umfaßt acht SDRAM-Chips, die mit SD1–SD8 bezeichnet sind, die gleichmäßig zwischen den vier Verschachtelungen **204A**–**204D** verteilt sind, wobei jede Verschachtelung zwei SDRAM-Chips umfaßt. Die beiden SDRAM in jeder Verschachtelung verwenden die folgenden gemeinsamen Leitungen gemeinschaftlich: elf Adreßleitungen(ADD), Zeilen- und Spaltenadreßübernahmesignale(RAS und CAS), ein Schreibfreigabesignal(WE), ein Taktfreigabesignal(CKE) und eine Dateneingabe/Ausgabemaske(DQM). Die SDRAMs in jeder Verschachtelung sind mit acht getrennten Datenleitungen gekoppelt, durch die acht Datenbits jeweils während jedem Lese- oder Schreibzyklus gelesen oder geschrieben werden. Jeder SDRAM-Chip umfaßt zwei Speicherbänke, wobei jede Bank bis zu 1.048.576 8-Bit-Wörter von Texturdaten speichert.

[0095] Auf die beiden SDRAM in jeder Verschachtelung kann gleichzeitig zugegriffen werden und dieselben können zusammen 16 Datenbits liefern, wobei einer der SDRAM-Datenbits [15:08] liefert, und der andere Datenbits [07:00] liefert. Wie es oben erörtert wurde, ergeben zwei aufeinanderfolgende Lesezyklen eines einzigen Burst ein volles 32-Bit-Texel von Daten von jeder Verschachtelung, wobei ein getrenntes 8-Bit-Wort jeden der RGB- und α-Werte für das Texel darstellt.

[0096] Die SDRAM-Chips empfangen 20 Adreßbits, die auf den elf Adreßleitungen ADD multiplext sind, um die 1.048.576 8-Bit-Wörter in jeder Bank zu decodieren. Wie es nachfolgend näher erklärt ist, werden ein 6-Bit-Blockindex und eine 16-Bit-Texeladresse für jedes Texel berechnet, auf das von dem Cache zugegriffen werden soll. Der Blockindex zeigt an, in welchem der 64 Datenblöcke das Texel positioniert ist, und die Texeladresse zeigt die genaue ST-Koordinatenadresse des Texels in dem Block an. Acht S-Bits und acht T-Bits umfassen die Texeladresse, unter der Annahme eines quadratischen Datenblocks, der 256 × 256 Texel umfaßt. Eine Cacheadresse ist ein 22-Bit-Wort, das die Kombination von Blockindex (sechs MSB) und Texeladresse(16 LSB) umfaßt. Die Cacheadresse zeigt die genaue Position des Texels in dem Cache an.

[0097] Während dem Aufbereiten decodiert der Kachel-/Grenzprüfer das LSB S-Bit und das LSB T-Bit der Texeladresse (d. h. die LSB S-Koordinate und die LSB T-Koordinate), um zu bestimmen, in welcher der vier Verschachtelungen des Cache das Texel gespeichert ist. Die verbleibenden 20 größeren Adreßbits der Cacheadresse werden entlang den Adreßleitungen ADD an die beiden SDRAM-Chips in der geeigneten Verschachtelung geliefert. Von den 20 Adreßbits, die an die beiden SDRAM geliefert werden, werden 9 Bits verwendet, um die Spalte auszuwählen, und 11 Bits werden verwendet, um die Zeile innerhalb der SDRAM auszuwählen, um auf die Texeldaten zu-

zugreifen. Wie es für einen Fachmann auf dem Gebiet klar sein sollte, sind die Spalten- und Zeilenadreßbits getrennt in den SDRAM gelatcht, in unterschiedlichen Zyklen, und die RAS- und CAS-Übernahmesignale werden herkömmlicherweise verwendet, um auf die Daten zuzugreifen.

[0098] Während einem Zwei-Zyklus-Burst werden 16 Bits von der adressierten Position der beiden SDRAM in der gleichen Verschachtelung während des ersten Zyklus geliefert, und dann, ohne eine weitere Adresse zu liefern, werden 16 Bits von einer anderen Position der beiden SDRAM während des zweiten Zyklus geliefert. Die Adresse in dem zweiten Zyklus umfaßt die gleiche Zeilenadresse und eine Spaltenadresse, die um 1 inkrementiert ist. Es sollte auch klar sein, daß, sobald eine Seite (spezielle Zeilenadresse) aktiviert ist, dieselbe aktiviert bleibt, bis eine andere Zeilenadresse geliefert wird. Falls daher aufeinanderfolgende Texel, auf die von der gleichen Verschachte1ung zugegriffen wird, in der gleichen Seite sind (die gleiche Zeilenadresse umfassen), dann muß die Zeilenadresse nur einmal während dem ersten der aufeinanderfolgenden Bursts geliefert werden.

[0099] Außerdem werden die RAS-, CAS- und WE-Leitungen verwendet, um Daten in dem SDRAM-Chip auf eine herkömmliche Weise zu adressieren und zu schreiben. Wenn das Taktfreigabesignal CKE-Signal deaktiviert ist, ist der interne Takt ausgesetzt. Die SDRAM sprechen auf dieses Signal an durch Intakthalten der Daten an, wodurch beide Bänke im Leerlauf bleiben. Das Dateneingabe-/Ausgabe-Masken-DQM-Signal funktioniert als ein Ausgabefreigabesignal während einem Lesezyklus und als eine Eingabedatenmaske während einem Schreibzyklus.

[0100] SDRAM werden herkömmlicherweise verwendet durch Bestimmen, von welcher zukünftigen Seite auf nachfolgende Daten zugegriffen wird, während auf aktuelle Daten von einer aktuellen Seite zugegriffen wird, und Aktivieren der zukünftigen Seite, bevor der aktuelle Datenlesezyklus abgeschlossen ist. Weil es SDRAMs ermöglichen, daß zwei unterschiedliche Seiten gleichzeitig aktiv sind, vermeidet die herkömmliche SDRAM-Verwendung Repaging-Strafen, die üblicherweise dem Zugreifen auf Daten von unterschiedlichen Seiten bei herkömmlichen DRAM zugeordnet sind. Die herkömmliche SDRAM-Verwendung liefert diesen Vorteil jedoch nicht, wenn Daten, die in mehreren aufeinanderfolgenden Lesezyklen gelesen werden sollen, in unterschiedlichen Seiten positioniert sind, weil mehr als ein Zyklus erforderlich ist, um vorauszuschauen und eine zukünftige Seite zu aktivieren. Das Texturdatenspeicherverfahren der vorliegenden Erfindung liefert einen Vorteil im Vergleich zu der herkömmlichen SDRAM-Verwendung, indem ermöglicht wird, daß mehrere aufeinanderfolgende SDRAM-Lesezyklen von unterschiedlichen Seiten auftreten können, ohne eine Strafe auf sich zu laden. Genauer gesagt, durch Speichern gemeinsamer Daten von benachbarten MIP-Abbildungen einer Textur (die das Zugreifen während aufeinanderfolgender Lesezyklen erfordern, wenn eine trilineare Interpolation ausgeführt wird) in getrennten Bänken der SDRAM, kann auf die Daten von den getrennten Bänken in aufeinanderfolgenden Lesezyklen ohne Strafe zugegriffen werden. Obwohl das Verfahren der vorliegenden Erfindung für eine Datenspeicherzuweisung zum Verbessern der SDRAM-Leistungsfähigkeit mit Bezug auf die Speicherung von Texturabbildungsdaten gezeigt und beschrieben wurde, ist klar, daß das Verfahren der vorliegenden Erfindung nicht so beschränkt ist. Insbesondere ist das Verfahren anwendbar zum Zuweisen jedes Datentyps, bei dem mehrere aufeinanderfolgende Lesezyklen auf Daten von unterschiedlichen Speicherpositionen zugreifen.

IV. Cache-Steuer-FIFOs

[0101] **Fig.** 8 ist ein detaillierteres Blockdiagramm eines Teils des Texturabbildungschips, der den Grenzprüfer **72**, das Cacheverzeichnis **78**, die Cachezugriffsschaltung **82**, den Cachespeicher **48** und den Texelinterpolator **76** umfaßt. Die Texelcachezugriffseinheit **82** umfaßt vier Cachezugriffsbefehl-FIFOs **206A**, **206B**, **206C** und **206D**. Die Cachezugriffsbefehl-FIFOs **206A**–**D** speichern Cachezugriffsbefehle, die jeweils von dem Grenzprüfer über 16-Bit-Busse **84A**, **84B**, **84C** und **84D** empfangen werden. Die Cachezugriffsbefehl-FIFOs **206A**–**D** entsprechen jeweils den in **Fig.** 6 gezeigten Steuerungen **200A**–**D**. Beispielsweise rufen die Befehle in dem FIFO **206A** einen Cachezugriff der SDRAMs in der Verschachtelung **204A** auf. Bei diesem Ausführungsbeispiel ist jeder Cachezugriffsbefehl-FIFO in der Lage, vorübergehend acht 16-Bit-Befehle zu speichern. Um somit die Pipelinefähigkeit des Systems zu verbessern, können acht Befehle in jedem der Cachezugriffsbefehl-FIFOs gespeichert werden, bevor die Cachezugriffseinheit wirkt.

[0102] Wie es oben erörtert wurde, vergleicht der Grenzprüfer **72** während dem Aufbereiten das Lesecachetag für jeden Texturdatenblock, der auf das Pixel abbildet, das bearbeitet wird, mit jedem der Blocktags, das in dem Cacheverzeichnis **78** gespeichert ist, um zu bestimmen, ob das Texel in dem Cache ist. Falls ein Treffer auftritt, wird der Blockindex erzeugt, der die Position des entsprechenden Texturdatenblocks in dem Cache darstellt. Der Kachel-/Grenzprüfer implementiert gleichzeitig eine Routine zum Bestimmen der Texeladresse von den interpolierten ST-Koordinaten, dem Textur-ID und dem Untertextur-ID des speziellen Texels, und auch der Abbildungsnummer der Abbildung, von der auf das Texel zugegriffen werden soll, und die Größe der Basisabbildung der Textur, wie es nachfolgend näher erklärt wird. Von dem Blockindex und der Texeladresse (die zusammen die Cacheadresse umfassen) bestimmt der Optimierer dann die spezielle Verschach-

telung des Cache, in der das Texel gespeichert wird, und die Spalten- und Zeilenadreßbits der SDRAM-Chips dieser Verschachtelung, wie es oben erklärt ist. Die Adreßinformationen werden an den entsprechenden Cachezugriffsbefehl-FIFO geliefert, zusammen mit einem Befehl zum Lesen des Cache.

[0103] Der Texelinterpolator **76** umfaßt acht Texeldaten-FIFOs, die mit **214A0**, **214A1**, **214B0**, **214B1**, **214C0**, **214C1**, **214D0** und **214D1** bezeichnet sind. Die Texeldaten-FIFOs **214A0** und **214A1** entsprechen der Verschachtelung **204A** des Cachespeichers, die FIFOs **214B0** und **214B1** entsprechen der Verschachtelung **204B**, die FIFOs **214C0** und **214C1** entsprechen der Verschachtelung **204C** und die FIFO **214D0** und **214D1** entsprechen der Verschachtelung **204D**.

[0104] Wie es oben beschrieben wurde, kann auf jede der vier Verschachtelungen des Cachespeichers gleichzeitig durch getrennte Cachezugriffswege zugegriffen werden. Wenn die Texelcachezugriffseinheit **82** auf Texeldaten von dem Cachespeicher **48** zugreift, werden während dem Aufbereiten Texelzugriffssteuerwörter über die Busse **208A**, **208B**, **208C** und **208D** an den Cachespeicher **48** geliefert. Während den zwei Back-to-Back-l6-Bit-Lesezyklen wird auf vier Texel gleichzeitig von den vier Verschachtelungen zugegriffen. Die vier Texel werden jeweils über die Busse **210A**, **210B**, **210C** und **210D** an einen der Texeldaten-A-FIFO (**214A0** oder **214A1**), einen der Texeldaten-B-FIFO (**214B0** oder **214B1**), einen der Texeldaten-C-FIFO (**214C0** oder **214C1**) oder einen der Texeldaten-D-FIFO (**214D0** oder **214D1**) geliefert. Das Paar von Texeldaten-FIFO (d. h. Null und Eins), die jeder Verschachtelung A–D entsprechen, werden in abwechselnder Weise geladen. Beispielsweise wird ein erster Texellesevorgang von der Verschachtelung A in dem Texeldaten-FIFO **214A0** gespeichert, ein zweiter Texellesevorgang von der Verschachtelung A wird in dem FIFO **214A1** gespeichert, ein drittes Texel von der Verschachtelung A wird in dem FIFO **214A0** gespeichert, usw. Dieses abwechselnde Schema wird aus nachfolgend erörterten Gründen verwendet.

[0105] Jedes der Texeldaten-FIFO ist 32 Bits breit und acht Stufen tief. In Kombination speichern die acht FIFO **214** acht pipelineartige Stufen, wobei jede Stufe die acht Texel umfaßt, die verwendet werden, um während einer trilinearen Interpolation die resultierenden Texeldaten zu bestimmen. Die Busse **210A**, **210B**, **210C** und **210D** sind 16 Bits breit. Jedes SDRAM-Paar in jeder Verschachtelung liefert 16 Datenbits während jedem Lesezyklus. Während jedem Burst werden die ersten 16 Bits von jedem SDRAM-Paar in ein erstes 16-Bit-Register (nicht gezeigt) geliefert, und die nächsten 16 Bits werden von jedem SDRAM-Paar in ein zweites 16-Bit-Register (ebenfalls nicht gezeigt) geliefert. Am Ende des zweiten Zyklus des Burst werden die Daten von beiden Registern auf dem entsprechenden 32-Bit-Bus **212A**, **212B**, **212C** oder **212D** geliefert. Um die resultierenden Texeldaten für jedes Pixel zu bestimmen, greift der Texelinterpolator **76** auf die FIFOs zu, um die nächste Stufe von acht Texeln zu lesen, und interpoliert diese Texel auf die oben beschriebene Weise. Die resultierenden Texeldaten werden dann über den Bus **28** an die Rahmenpufferplatine **14** (**Fig.** 2) geliefert, wo dieselben beim Aufbereiten der Anzeigebildschirmpixel in der oben erörterten Weise verwendet werden.

[0106] Wenn eine trilineare Interpolation durchgeführt wird, werden die resultierenden Texeldaten für jedes Pixel von vier Texeln in einer MIP-Abbildung unter vier Texeln in einer benachbarten MIP-Abbildung interpoliert. Benachbarte Anzeigebildschirmpixel werden im allgemeinen aufeinanderfolgend aufbereitet. Häufig bilden benachbarte Anzeigebildschirmpixel auf benachbarte Positionen in einer Textur-MIP-Abbildung ab. Folglich ist es üblich, daß einige gemeinsame Texeldaten beim Interpolieren resultierender Texeldaten für aufeinanderfolgend aufbereitete Grundelemente verwendet werden können. Wenn bei einem Ausführungsbeispiel der Erfindung auf gemeinsame Texeldaten mehrmals zugegriffen wird, in einer Anzahl von nahe beabstandeten Lesezyklen, wird auf den Cache nur für den ersten Lesevorgang zugegriffen, wodurch Cachelesezyklen für jeden nachfolgenden Lesevorgang gespart werden. Die aktuellsten Lesetexel werden in den Texeldaten-FIFO gespeichert. Somit werden nachfolgende Zugriffe auf diese Texel von den FIFO durchgeführt anstatt von dem Cache. Dies reduziert die Anzahl erforderlicher Cachezugriffe und erhöht dadurch die Systembandbreite.

[0107] Falls für jeden der Texeldatenwege A, B, C und D die Texeldaten, die als letztes in einen der Texeldaten-FIFO 0 oder **1** für ein vorhergehendes Pixel geschrieben wurden, mit den Texeldaten für ein Pixel übereinstimmen, das derzeit in der Pipelineposition zum Zugreifen auf den Cache ist, dann wird kein Cachezugriffsbefehl an den entsprechenden Cachezugriffs-FIFO **206A**, **B**, **C** oder **D** geliefert. Statt dessen wird ein Befehl an den Texelinterpolator gesendet, um anzuzeigen, daß die Texeldaten in der als letztes geschriebenen Position der entsprechenden Texeldaten-FIFO **214A**, **B**, C oder D gespeichert sind. Für jeden der Wege A, B, C und D, bei dem die Texeldaten, die dem Pixel entsprechen, das derzeit in der Pipelineposition zum Zugreifen auf den Cache ist, nicht mit den Daten übereinstimmt, die in der als letztes geschriebenen Position des entsprechenden Texeldaten-FIFO sind, wird ein Texelcachezugriffsbefehl an den entsprechenden Texelcachezugriffsbefehl-FIFO geliefert, um die Texeldaten von dem Cachespeicher **48** zu lesen.

[0108] Es sollte klar sein, daß für einige der Verschachtelungen A –D für jedes Pixel, das aktuell in der Pipelineposition ist, für die ein Cachezugriff in Betracht gezogen werden muß, ein anderes Ergebnis auftreten kann. Beispielsweise können gemeinsame Texeldaten aufeinanderfolgende Pixel für die Ver-

schachtelung A, aber nicht für die Verschachtelungen B–D existieren. In einem solchen Fall werden Texeldaten von den Verschachtelungen B–D für das zweite der aufeinanderfolgenden Pixel in der Pipelineposition zum Zugreifen auf Texeldaten von dem Cache gelesen, aber die Texeldaten von der Verschachtelung A für dieses zweite Pixel werden von der gleichen Position von einem der Texeldaten-FIFO **214A0** oder **214A1** gelesen. Das vorliegende Schema liefert Bandbreiteneinsparungen, wenn Texel für mehrere Pixel von den Texeldaten-FIFO erneut gelesen werden, ohne auf den Cache zuzugreifen.

[0109] Der Texelinterpolator **76** umfaßt einen Texelinterpolatorbefehls-FIFO **216**, der 53-Bit-Befehle von dem Grenzprüfer **72** über den 53-Bit-Bus **218** empfängt. Der Texelinterpolatorbefehls-FIFO kann bis zu 16 Befehle speichern, die dem Interpolator anzeigen, welche Texeldaten-FIFO-Positionen die Texeldaten enthalten, die beim Interpolieren der resultierenden Texeldaten während jedem Zyklus verwendet werden sollen. Die Interpolatorbefehle zeigen auch den Interpolationsmodus an (d. h. Punktabtastung, bilinear oder trilinear) und umfassen den Gradient und die Bruchteilwerte der S- und T-Koordinaten, die die Art und Weise spezifizieren, wie jedes Texel in der Interpolation gewichtet werden sollte. Die Befehle umfassen Daten, die anzeigen, von welchen Texeldaten-FIFOs **214A0**, A1, B0, B1, C0, C1, D0 oder D1 jedes der vier (bilinear) oder acht (trilinear) Texel gelesen werden sollen, und ob die Texeldaten neu oder alt sind. Texeldaten sind neu, wenn sie sich von den Texeldaten unterscheiden, die in der als letztes geschriebenen Position irgendeines Texeldaten-FIFOs dieses Wegs geschrieben sind. Wenn dieselben neu sind, ist ein Cachelesevorgang erforderlich. Texeldaten sind alt, wenn sie dieselben sind, wie diejenigen, die in der als letztes beschriebenen Position eines der Texeldaten-FIFO gespeichert sind. Wenn dieselben alt sind, ist kein Cachelesevorgang erforderlich. Wenn die Texeldaten neu sind, muß der FIFO-Lesezeiger zu einer nächsten Position in dem FIFO bewegt werden, während, wenn die Texeldaten alt sind, die gleichen Daten von der gleichen FIFO-Position gelesen werden und der Lesezeiger nicht bewegt werden muß.

[0110] Das folgende Beispiel, das mit Bezugnahme auf **Fig.** 9 und 10 erklärt ist, stellt ferner den Betrieb der in **Fig.** 8 gezeigten Texelzugriffsschaltung dar. **Fig.** 9 zeigt mehrere Texel einer oberen MIP-Abbildung und mehrere Texel einer unteren (kleineren) MIP-Abbildung. Die Texel sind mit An, Bn, Cn und Dn bezeichnet (wobei n eine Ganzzahl darstellt) gemäß dem Bezeichnungsschema, das vorher mit Bezugnahme auf **Fig.** 7 beschrieben wurde. Sieben Pixel, die aufbereitet werden sollen, sind mit P0, P1,..., P6 bezeichnet. Wie es gezeigt ist, bilden die Pixel, die aufbereitet werden sollen, nicht direkt auf die Texel der MIP-Abbildungen ab. Bei diesem Beispiel wird eine trilineare Interpolation durchgeführt, so daß auf vier Texel von der oberen Abbildung und vier Texel von der unteren Abbildung zugegriffen werden soll und dieselben für jedes Pixel interpoliert werden. Die Schrittrichtung ist die Richtung des Aufbereitens und entspricht der numerischen Numerierung der Pixel.

[0111] **Fig.** 10 stellt den Cachezugriffsbefehl-FIFO (**206A**), den Texeldaten-FIFO A0 (214A0), den Texeldaten-FIFO A1 (214A1) und den Texelinterpolatorbefehls-FIFO **216** dar. Aus Zweckmäßigkeitsgründen sind nur die FIFOs, die dem Texeldatenweg A zugeordnet sind, gezeigt, weil die FIFOs für jeden der anderen Texeldatenwege B, C und D auf gleiche Weise arbeiten. Jeder FIFO-Puffer umfaßt einen Schreibzeiger und einen Lesezeiger, die jeweils zu einzelnen Positionen in dem FIFO zeigen, an die Daten geschrieben werden sollten und von denen Daten gelesen werden sollten. Bei diesem darstellenden Ausführungsbeispiel können die Zeiger sich zu einem Zeitpunkt um eine Position bewegen.

[0112] Das Pixel P0 bildet auf die Texel A0, B0, C0 und D0 in der oberen Abbildung und auf die Texel A0, B0, C0 und D0 in der unteren Abbildung ab, daher sind diese acht Texel interpoliert, um die resultierenden Texeldaten für das Pixel P0 zu erzeugen. Für das Pixel P0 wird die Adresse des Texels A0 der oberen Abbildung (d. h. uA0) an eine erste Position in dem Cachezugriffsbefehl-FIFO **206A** geschrieben, zusammen mit einer Adresse, die anzeigt, daß der Texeldaten-FIFO **214A0** mit den Texeldaten beschrieben werden sollte, die von dem Cache an dieser Adresse gelesen werden. Nachfolgend wird der Schreibzeiger des Cachezugriffsbefehl-FIFO **206A** um eine Position bewegt, und die Adresse des Texels A0 in der unteren Abbildung (d. h. 1A0) wird an diese nächste FIFO-Position geschrieben, zusammen mit einer Adresse, die anzeigt, daß der Texeldaten-FIFO **214A1** mit den Texeldaten beschrieben werden sollte, die von dem Cache an dieser Adresse gelesen werden. Auf diese Weise werden die Texeldaten-FIFOs **0** und **1** aus den oben erörterten Gründen abgewechselt. Die Cachezugriffsbefehl-FIFO **206B**–**D** werden auf ähnliche Weise aktualisiert, die sich auf die Texel B0, C0 und D0 in der oberen und unteren Abbildung bezieht.

[0113] Für das Pixel P1 müssen die Texel A1 in der oberen und unteren Abbildung, die an den Adressen uA1 bzw. 1A1 gespeichert sind, interpoliert werden. Da die Texel A1 in der oberen und unteren Abbildung neue Texel sind und nicht Texeln von dem vorhergehenden Pixel P0 entsprechen, wird auf dieselben von dem Cache zugegriffen. Somit werden die Texeladressen für diese Texel den nächsten beiden Positionen des Cachezugriffsbefehl-FIFO **206A** hinzugefügt, zusammen mit den entsprechenden Adressen, die jeweils anzeigen, daß die Texeldaten, die von diesen Adressen gelesen werden, in den Texeldaten-FIFOs **214A0** und **214A1** gespeichert werden sollen. **Fig.** 10 stellt den Cachezugriffsbefehl-FIFO **206A** dar, nachdem er mit diesen Informationen aktualisiert wurde.

[0114] Weil es keinen gemeinsamen A-adressierten

Texel für die ersten beiden Pixel P0 und P1 gibt, wird auf den Cachespeicher zugegriffen, um die Texeldaten für beide wiederzugewinnen. Der erste Befehl wird von dem Cachezugriffsbefehl-FIFO **206A** gelesen, wodurch bewirkt wird, daß die Texeldaten an der Adresse uA0 von dem Cachespeicher gelesen werden und an die erste Position des Texeldaten-FIFO **214A0** geschrieben werden. Dann wird der nächste Befehl von dem Cachezugriffsbefehl-FIFO gelesen und auf Texeldaten an der Adresse **1A0** wird von dem Cache zugegriffen und dieselben an die erste Position des Texeldaten-FIFO **214A1** geschrieben. Der nächste Befehl wird dann von dem Cachezugriffsbefehl-FIFO gelesen und auf die Texeldaten an der Adresse uA1 wird von dem Cache zugegriffen und dieselben werden an die nächste Position in dem Texeldaten-FIFO **214A0** geschrieben. Schließlich wird der vierte Befehl von dem Cachezugriffsspeicher-FIFO gelesen und auf die Texeldaten an der Adresse **1A1** wird von dem Cache zugegriffen und dieselben werden an die nächste Position des Texeldaten-FIFO **214A1** geschrieben.

[0115] Für das nächste Pixel P2, das aufbereitet werden soll, müssen die Texel an den Adressen uA1 und 1A1 interpoliert werden. Weil auf diese Texel für das vorher aufbereitete Pixel P1 zugegriffen wurde, werden dieselben in den als letztes geschriebenen Einträgen in den Texeldaten-FIFOs **214A0** bzw. 214A1 gespeichert. Somit werden keine neuen Cachezugriffsbefehle für diese Texel an den Cachezugriffsbefehl-FIFO **206A** geliefert. Nachdem die resultierenden Texeldaten für das Pixel P1 interpoliert sind, kann statt dessen jeweils auf die Texeldaten, die an den Adressen uA1 und 1A1 gespeichert sind, zugegriffen werden, durch den Texelinterpolator von den als letztes gelesenen Positionen der Texeldaten-FIFOs **214A0** und **214A1**, ohne Zugriff auf den Cache zu haben. Das Lesen von Daten direkt von einem FIFO-Puffer ist weniger zeitaufwendig als das Zugreifen auf Daten von einem Cachespeicher. Daher erhöhen die FIFO-Puffer der vorliegenden Erfindung, die die Cachezugriffe reduzieren, die Systembandbreite.

[0116] Wie es oben erörtert wurde, umfassen die Texeldaten-FIFOs **214**, die jeder der Verschachtelungen A – D entsprechen, getrennt gesteuerte FIFOs Null und Eins. Die FIFOs sind auf diese Weise unterteilt, um eine trilineare Interpolation effizient zu implementieren. Wie es aus dem Vorhergehenden offensichtlich ist, liefern bei dem oben beschriebenen Ausführungsbeispiel die Texeldaten-FIFOs **214** jeweils Zugriff zu dem als letztes gelesenen Eintrag durch Beibehalten des Lesezeigers zum Zeigen zu dem gleichen Eintrag für nachfolgende Lesevorgänge. Obwohl jede Verschachtelung zwischen Lesevorgängen von zwei Abbildungen während aufeinanderfolgenden Lesezyklen abwechselt, können somit die getrennten FIFOs aufeinanderfolgende Lesevorgänge innerhalb einer einzigen Abbildung durchführen, was es dem Lesezeiger ermöglicht, in aufeinanderfolgenden Zugriffen auf den FIFO zu den gleichen Texeldaten zu zeigen.

[0117] Während jedes Pixel durch dem Kachel-/Grenzprüfer 72 bearbeitet wird und Befehle an den Cachezugriffsbefehl-FIFO geliefert werden, werden auch Befehle in den Texelinterpolatorbefehls-FIFO **216** geschrieben. Wenn beispielsweise der Befehl zum Zugreifen auf das Texel an die Adresse uA0 an den Cachezugriffsbefehl-FIFO für das Pixel P0 geliefert wird, wird der Befehl NewO an die erste Position des Texelinterpolatorbefehls-FIFO **216** geliefert. Der Befehl NewO zeigt dem Texelinterpolator an, daß auf die nächsten Texeldaten von der Verschachtelung A von dem Cache zugegriffen wird und dieselben an den Texeldaten-FIFO **214A0** geliefert werden, was anzeigt, daß bei der Reihenfolge zum Lesen der Texeldaten von dem FIFO der Texelinterpolator den FIFO-Lesezeiger von der Position, die als letztes gelesen wurde, um eine Position bewegen sollte.

[0118] Für den nächsten Befehl, der an den Cachezugriffsbefehl-FIFO geliefert wurde, der der Texeladresse **1A0** entspricht, wird der Befehl New1 an die nächste Position des Texelinterpolatorbefehls-FIFO geliefert. Der Befehl New1 zeigt dem Texelinterpolator an, daß die nächsten Texeldaten von der Verschachtelung A ebenfalls neu sind und von dem Texeldateninterpolator **214A1** gelesen werden sollten. Gleichartig dazu werden für die Befehle, die den Texeladressen uA1 und 1A1 zugeordnet sind, die dem Pixel P1 entsprechen, die Befehle NewO und New1 jeweils auf die nächsten beiden Positionen des Texelinterpolatorbefehls-FIFO **216** geschrieben.

[0119] Da für das Pixel P2 die Texeldaten an den Adressen uA1 und 1A1 identisch sind mit den Daten, die für das vorhergehende Pixel P1 in den FIFO geschrieben wurden, sind die Befehle, die an die nächsten beiden Positionen des Texelinterpolatorbefehls-FIFO **216** geschrieben werden, Old0 und Old1, die dem Texelinterpolator jeweils anzeigen, daß die nächsten Texeldaten von den als letzten gelesenen Positionen der Texeldaten-FIFO **214A0** und **214A1** erneut gelesen werden sollten. Die Befehle Old0 und Old1 zeigen an, daß der Texelinterpolator, um die nächsten Pixeldaten von dem FIFO zu lesen, den FIFO-Lesezeiger nicht von der Position bewegen sollte, die als letztes gelesen wurde.

[0120] **Fig.** 9 zeigt drei Tabellen: die erste Tabelle zeigt die Texel an, die für jedes der Pixel interpoliert werden müssen, die zweite Tabelle listet die getrennten Texeldatenwerte auf, die in den Texeldaten-FIFO A0, B0, C0 und D0 gespeichert werden müssen, und die dritte Tabelle listet die getrennten Texeldatenwerte auf, die in den Texeldaten-FIFOs A1, B1, C1 und D1 gespeichert werden müssen. Die Leerstellen zeigen gemeinschaftlich verwendete Texeldaten an, die vorher von dem Cache gelesen wurden, die nicht erneut von dem Cache gelesen werden müssen und auf die statt dessen von den FIFOs zugegriffen werden kann. Wenn resultierende Texeldaten für mehre-

re Pixel interpoliert werden, wie es dieses Diagramm anzeigt, kann eine große Anzahl von Cachezugriffen durch das FIFO-Schema der vorliegenden Erfindung eingespart werden, was zu einer Erhöhung bei der Systembandbreite führt.

[0121] **Fig.** 11 ist ein Blockdiagramm einer Schaltung, die durch den Texturabbildungschip verwendet wird, um zu bestimmen, ob bei jeder Verschachtelung Texeldaten, die für ein Pixel gelesen werden sollen, für das als letztes aufbereitete Pixel gelesen wurden. Diese Schaltung wird verwendet, um zu bestimmen, ob ein neuer Befehl in einen der Cachezugriffsspeicher-FIFO geschrieben wird, um zu bewirken, daß neue Daten von dem Cache gelesen werden, oder ob ein Befehl an den Texelinterpolatorbefehls-FIFO geschrieben wird, der anzeigt, daß die Texeldaten alt sind und von einem der Texeldaten-FIFO gelesen werden sollten. **Fig.** 11 zeigt nur eine einzige Schaltung, die der Verschachtelung A entspricht. Ähnliche Schaltungen werden ebenfalls für die Verschachtelungen B, C und D geliefert. Die Schaltung ist in dem Optimiererelement des Kachel-/Grenzprüfers positioniert. Von dem interpolierten ST-Wert, der von dem Kachel/Grenzprüfer für jedes Texel empfangen wird, das interpoliert werden soll, liefert der Optimierer eine Texeladresse (einschließlich des Blocktags und der Texeladresse) auf dem Bus **220A**. Die Adresse des als letztes verarbeiteten Texels, die den Texeldaten-FIFO **214A0** und **214A1** zugeordnet sind, werden jeweils in den Adreßregistern **222A0** und **222A1** gespeichert. Die aktuelle Texeladresse wird jeweils mit den Texeladressen verglichen, die in den Registern **222A0** und **222A1** gespeichert sind, durch die Komparatoren **224A0** und **224A1**.

[0122] Wenn die vorliegende Texeladresse mit keiner der Adressen, die in den Registern **222A0** und **222A1** gespeichert sind, übereinstimmt, muß auf Texeldaten, die dieser Texeladresse entsprechen, von dem Cachespeicher zugegriffen werden, und der entsprechende Befehl wird in den Cachezugriffsbefehl-FIFO geschrieben. Wenn jedoch die Texeladresse mit der Adresse übereinstimmt, die in dem Adreßregister **222A0** oder **222A1** gespeichert wird, werden die Texeldaten jeweils in dem Texeldaten-FIFO **212A0** oder **212A1** an der Position gespeichert, die durch den Texelinterpolator unmittelbar vor dem Zugreifen auf die Texeldaten gelesen wird, die der Adresse entsprechen. Daher wird kein Cachezugriffsbefehl in den Cachezugriffsbefehl-FIFO geschrieben, und in den entsprechenden Texelinterpolatorbefehls-FIFO wird ein Befehl geschrieben, der anzeigt, daß die Texeldaten alt sind, und daß auf dieselben von der als letzten gelesenen FIFO-Position zugegriffen werden sollte, ohne den Lesezeiger zu bewegen.

V. Organisation von Texturdatenblöcken

[0123] **Fig.** 1 zeigt eine Reihe von quadratischen Textur-MIP-Abbildungen, die eine Basisabbildung **100** von 8 × 8 Texeln umfassen. Von der Basisabbildung wird jede nachfolgende Abbildung in der Größe gefiltert, auf eine **Abb.** 108 kleinster Größe (d. h. die nur ein Texel umfaßt). Der **Abb.** 108 kleinster Größe wird eine Abbildungsnummer von 0 zugewiesen, und die Abbildungsnummer für jede nachfolgende größere Abbildung wird um Eins inkrementiert, so daß die Basisabbildung **100** bei diesem Beispiel eine Abbildungsnummer von Drei hat. Die Abbildungsnummer wird beim Bestimmen des Blocktags für jeden Block von Texturdaten auf eine Weise verwendet, die nachfolgend beschrieben wird. Gemäß diesem Abbildungsnumerierungsschema bei der Annahme einer quadratischen Texturbasisabbildung entspricht eine Abbildungsnummer von 10 einer Abbildung von 1.024 × 1.024 Texeln, eine Abbildungsnummer von 9 stellt eine 512 × 512-Texelabbildung dar, eine Abbildungsnummer von 8 stellt eine 256 × 256-Texelabbildung dar usw. Falls die Texturbasisabbildung nicht quadratisch ist, dann entspricht eine Abbildungsnummer von 10 einer Abbildung mit einer größeren Abmessung von 1.024 Texeln. Obwohl diese Erörterung von einer quadratischen Texturbasisabbildung ausgeht, sind rechteckige Abbildungen ebenfalls möglich. Falls sie rechteckig sind, wird die Abbildungsnummer durch die Anzahl von Texeln der längeren Abmessung der Abbildung bestimmt. Beispielsweise hat eine rechteckige Abbildung mit einer Abbildungsnummer von 10 eine längere Abmessung mit 1.024 Texeln. Es sollte klar sein, daß alternativ auch andere Abbildungsnumerierungsschemata verwendet werden können.

[0124] Eine quadratische 1.024 × 1.024-Texelabbildung, die eine Abbildungsnummer von 10 aufweist, erfordert zehn Bits von S-Koordinaten S[9:0] und zehn Bits von T-Koordinaten T[9:0] zum eindeutigen Identifizieren der Position jedes Texels innerhalb der Abbildung. Gleichartig dazu erfordert eine Abbildung mit einer Abbildungsnummer von 9 neun Bits sowohl von S- als auch den T-Koordinaten zum Identifizieren der Position jedes Texels, eine Abbildung mit einer Abbildungsnummer von 8 erfordert acht Bits sowohl von S- als auch T-Koordinaten zum Identifizieren der Position jedes Pixels, usw. Die S- und T-Koordinaten, die die Position eines Texels in einer MIP-Abbildung eindeutig identifizieren, die einem Pixel entspricht, werden auf die oben beschriebene Weise interpoliert.

[0125] Wie es nachfolgend näher beschrieben ist, werden Texturdaten in dem Hauptspeicher **17** des Rostcomputers **15** (**Fig.** 2) in Blöcken von 256 × 256 Texeln gespeichert. Wenn ein Cachefehlschlag auftritt, wird ein Lesecachetag, das den Texturdatenblock identifiziert, der an dem Cache fehlgeschlagen hat, durch den Hostcomputer gelesen, und dieser Texturdatenblock wird dann zu dem Cachespeicher **48** der Texturabbildungsplatine heruntergeladen. Bei dem darstellenden Ausführungsbeispiel der vorliegenden Erfindung können 64 Texturdatenblöcke zu einem Zeitpunkt in dem Cachespeicher gespeichert sein. Diese **64** Texturdatenblöcke können Daten von

mehreren MIP-Abbildungen von einer oder mehreren Texturen umfassen. Jeder Block weist ein zugeordnetes Blocktag auf, das dasselbe eindeutig identifiziert. MIP-Abbildungen mit einer Abbildungsnummer von 9 oder mehr umfassen mehr als 256 × 256 Texel und sind daher in mehreren Blöcken gespeichert. Die höherwertigen ST-Koordinaten für jede Abbildung, die in mehreren Blöcken gespeichert ist, sind in dem Blocktag für die Datenblöcke enthalten, die die Abbildung speichern.

[0126] Beispielsweise haben MIP-Abbildungen mit einer Abbildungsnummer von 9 eine Abmessung gleich 512 Texeln und, falls dieselben quadratisch sind, sind sie 512 × 512 Texel groß. Die Abbildung ist in vier Blöcke von 256 × 256 Texel unterteilt (bei einer quadratischen Texturabbildung). Daher umfaßt das Blocktag für jeden dieser Blöcke ein höherwertiges S-Koordinatenbit und ein höherwertiges T-Koordinatenbit (d. h. S [8] und T [8]), die die Position des Blocks in der Abbildung identifizieren. Gleichartig dazu sind MIP-Abbildungen mit einer Abbildungsnummer von 10 1.024 × 1.024 Texel groß und sind in 16 Datenblöcke unterteilt. Daher umfassen die Blocktags für jeden dieser Blöcke zwei höherwertige S-Koordinatenbits und zwei höherwertige T-Koordinatenbits (d. h. S[9:8] und T[9:8]), die die Position des Blocks innerhalb der Abbildung identifizieren.

[0127] Um die Systembandbreite während einer trilinearen Interpolation zu reduzieren, wie es oben beschrieben wurde, sind die Textur-MIP-Abbildungen unterteilt und in einem Speicher gespeichert, so daß die gleichen Abschnitte benachbarter MIP-Abbildungen in gegenüberliegenden SDRAM-Bänken gespeichert sind. Um außerdem eine effiziente Verwendung von Speicherplatz innerhalb des Cachespeichers zu liefern, können mehrere Abbildungen, die kleiner als 256 × 256 Texel sind, in einem einzigen Cachespeicherblock gespeichert werden.

[0128] **Fig.** 12 zeigt einen Satz von Textur-MIP-Abbildungen für eine spezielle Textur, die das Oberflächenbild umfaßt:

LA

95

[0129] Wie es in **Fig.** 12 gezeigt ist, ist jede MIP-Abbildung in der Reihe von MIP-Abbildungen für eine Textur in vier Quadranten unterteilt, die für eine quadratische Texturabbildung die gleiche Größe aufweisen. Bei dem in **Fig.** 12 gezeigten Beispiel hat die Basisabbildung eine Abbildungsnummer von 9 und ist in Quadranten **9Q1** (einschließlich Bild L), 9Q2(einschließlich Bild A), **9Q3**(einschließlich Bild 9) und 9Q4(einschließlich Bild 5) unterteilt. Gleichartig dazu ist die Abbildungsnummer **8** in Quadranten **8Q1**, **8Q2**, **8Q3**, **8Q4** unterteilt, die jeweils die Bilder L, A, 9 und 5 umfassen. Gleichartig dazu ist die Abbildungsnummer **7** in Quadranten **7Q1**, **7Q2**, **7Q3**, **7Q4** unterteilt, die jeweils die Bilder L, A, 9 und 5 umfassen. Die kleineren Abbildungen sind gleichartig dazu in Quadranten unterteilt.

[0130] Zwei Quadranten jeder MIP-Abbildung sind in einer Bank der SDRAMs gespeichert, die den Cache bilden, während die anderen beiden Quadranten in der gegenüberliegenden Bank gespeichert sind. Gemäß dem Texturdatenzuweisungsschema der Erfindung sind für Texturen mit einer Basisabbildung mit einer Nummer größer oder gleich 8 (die größer oder gleich 256 × 256 Texel groß ist) die Speicherpositionen in den Blöcken von Speicherplatz für alle Quadranten aller MIP-Abbildungen dieser Textur vordefiniert. Beispielsweise sind die Quadranten **9Q1** und **9Q4** der Abbildungsnummer **9** in getrennten Blöcken in der Cachebank **1** gespeichert, und die Quadranten **9Q2** und **9Q3** sind in getrennten Blöcken der Cachebank **0** gespeichert, wie es in **Fig.** 13 gezeigt ist. Die entsprechenden Quadranten benachbarter MIP-Abbildungen sind in Blöcken in gegenüberliegenden Bänken gespeichert. Somit sind bei diesem Beispiel die Quadranten **8Q1** und **8Q4**, die jeweils die boxgefilterten Texturdaten der Quadranten **9Q1** und **9Q4** umfassen, in dem gleichen Block in der Cachebank 0 gespeichert. Gleichartig dazu sind die Quadranten **8Q2** und **8Q3**, die jeweils die boxgefilterten Texturdaten der Quadranten **9Q2** und **9Q3** umfassen, in dem gleichen Block in der Cachebank **1** gespeichert. **Fig.** 13 ist bezüglich **Fig.** 12 nicht maßstabsgerecht gezeichnet. Es sollte klar sein, daß die Abbildungsquadranten von **Fig.** 12 gleich groß sind wie diejenigen von **Fig.** 13, da dieselben identisch sind.

[0131] Aufgrund der jeweiligen Größen der Abbildungen besetzt jeder Quadrant der Abbildungsnummer **9** einen vollständigen Block von 256 × 256 Texeln, während die Quadranten der Abbildungsnummer **8** jeweils nur ein Viertel eines Blocks besetzen. Daher besetzen die Quadranten **8Q2** und **8Q3** zusammen die Hälfte des gleichen Blocks und die Quadranten **8Q1** und **8Q4** besetzen eine Hälfte des anderen Blocks in der gegenüberliegenden Bank. Um den Cachespeicherplatz effizient zuzuweisen, werden die unbesetzten Positionen in jedem dieser Blöcke durch geeignete Quadranten von Abbildungen besetzt, die eine Abbildungsnummer von 7 oder weniger aufweisen. Daher besetzen alle der Abbildungen mit den Nummern **0** bis **8** zusammen zwei Blöcke, wobei jeder der beiden Blöcke in einer getrennten Bank ist.

[0132] Die Positionen der Quadranten für die Abbildungen mit einer Abbildungsnummer von 8 oder weniger (bei einer Basisabbildung mit einer Abbildungsnummer von 8 oder mehr) sind auf die Weise vordefiniert, die in **Fig.** 13 gezeigt ist. Wie es gezeigt ist, behalten der obere rechte Quadrant **8Q2** und der untere linke Quadrant **8Q3** die gleiche physikalische Beziehung bei und besetzen jeweils den oberen rechten und unteren linken Quadrant eines ersten Blocks und der obere linke Quadrant **8Q1** und der untere rechte Quadrant **8Q4** behalten auch die gleiche physikalische Beziehung bei und besetzen entsprechend den oberen linken und unteren rechten Quadrant eines zweiten Blocks, der sich in einer anderen Bank als der erste Block befindet. Außerdem behal-

ten die Quadranten **7Q1** und **7Q4** die gleiche physikalische Beziehung bei und besetzen entsprechend den oberen linken Quadranten des ersten Blocks und die Quadranten **7Q2** und **7Q3** behalten die gleiche physikalische Beziehung bei und besetzen entsprechend den oberen rechten Quadranten des zweiten Blocks.

[0133] Falls während einer trilinearen Interpolation ein Pixel auf eine Position in der Texturabbildung abbildet, die zwischen vier Texeln in einer MIP-Abbildung und vier Texeln in einer benachbarten MIP-Abbildung ist, dann wird auf alle acht Texel von dem Cache zugegriffen. Die Texel, auf die von beiden MIP-Abbildungen zugegriffen wird, umfassen allgemeine Texturdaten, wobei die Daten von der kleineren Abbildung eine gefilterte Version der Daten von der größeren Abbildung sind. Wenn Pixel eines Objekts aufbereitet werden, wie es oben erörtert wird, bilden benachbarte Pixel regelmäßig auf die gleichen zwei MIP-Abbildungen für die Textur ab, was erfordert, daß Lesevorgänge zu dem Cache fortlaufend zwischen den Cacheblöcken hin- und herschalten, die die beiden Abbildungen speichern. Durch Speichern gemeinsamer Daten von benachbarten MIP-Abbildungen in unterschiedlichen Bänken der Cache-SDRAM-Chips werden keine Repaging-Strafen bewirkt, wenn Cachelesevorgänge während aufeinanderfolgender Lesezyklen zwischen den beiden MIP-Abbildungen hin- und herschalten. Dies liefert eine effiziente Implementierung der trilinearen Interpolation.

[0134] Wie es aus dem Vorhergehenden erkannt werden sollte, wenn eine Textur eine Basisabbildung umfaßt, die eine Abbildungsnummer von 8 oder höher umfaßt, ist die Zuweisung der MIP-Abbildungen unter den Blöcken für diese Textur gemäß dem beschriebenen darstellenden Ausführungsbeispiel der Erfindung vordefiniert. Dies liegt daran, daß zwei Quadranten einer Abbildung mit einer Abbildungsnummer von 8 a bestimmte vordefinierte Positionen eines ersten Blocks innerhalb einer der Bänke besetzen, und die anderen beiden Quadranten der Abbildung mit einer Abbildungsnummer von 8 bestimmte vordefinierte Positionen in einem anderen Block der gegenüberliegenden Bank besetzen, wie es oben erörtert wurde und in **Fig.** 13 gezeigt ist. Für Texturen mit einer Basisabbildung mit einer Abbildungsnummer von 7 oder weniger sind jedoch mehrere Positionen in den beiden Speicherblöcken verfügbar (ein Block in jeder Bank), um die Abbildung zu speichern, und werden durch den Hostcomputer ausgewählt. Wenn Abschnitte von mehreren Abbildungen einen einzelnen Datenblock gemeinschaftlich verwenden, wird eine Untertexturidentifikation (ID) auf eine Weise zugewiesen, die nachfolgend beschrieben ist, um die Position jeder Abbildung in dem gemeinschaftlich verwendeten Block zu identifizieren.

[0135] Zusätzlich zu der Organisation der Reihen von MIP-Abbildungen von **Fig.** 12 zeigt **Fig.** 13 auch die Art und Weise, wie eine zweite Reihe von MIP-Abbildungen von einer anderen Textur (d, h. ein Schachbrettmuster) zwischen den Speicherblöcken zugewiesen wird. Die MIP-Abbildungen dieser zweiten Textur sind unterteilt und in getrennten Datenblökken gespeichert, auf gleiche Weise wie die erste Textur. Obwohl die Organisation von **Fig.** 13 die MIP-Abbildungen der unterschiedlichen Texturen als in getrennten Blöcken organisiert zeigt, sollte klar sein, daß die Texturdaten von zwei unterschiedlichen Texturen in dem gleichen Block gespeichert werden können.

[0136] Wie es oben erörtert wurde, kann der Cachespeicher bei einem darstellenden Ausführungsbeispiel bis zu 64 Texturabbildungsdatenblöcke speichern, wobei jeder Block 256 × 256 Texel umfaßt. Der Cachespeicher ist in zwei Bänke unterteilt, wobei die Blöcke **0**–**31** in der Bank Null liegen und die Blöcke **32**–**63** in der Bank Eins liegen. Das Cacheverzeichnis umfaßt bis zu 64 Blocktageinträge, die den Blöcken in dem Cache entsprechen. Die physikalische Position jedes Blocktags in dem Cacheverzeichnis identifiziert die physikalische Position des entsprechenden Texturdatenblocks in dem Cachespeicher. Ein Blockindex wird von dem Blocktag erzeugt, das die Position des Blocks anzeigt. Die Cacheadresse für jedes Texel in dem Cache wird von dem Blockindex für den Block und der Texeladresse in dem Block gebildet. Die Texeladresse umfaßt die interpolierten niederwertigen ST-Koordinaten für das Texel und kann auch Bits von der Untertextur-ID umfassen, wie es nachfolgend erörtert wird.

[0137] **Fig.** 14 zeigt ein Beispiel einer Textur-MIP-Abbildung mit einer Abbildungsnummer von 9, die in Quadranten unterteilt ist. Die MIP-Abbildung ist 512 × 512 Texel groß, und daher ist jeder Quadrant 256 × 256 Texel groß und entspricht einem einzelnen Speicherblock. Gemäß einem Ausführungsbeispiel der vorliegenden Erfindung wird durch den Rostcomputer ein einfaches Schema implementiert, um die Bank in dem Cache zu bestimmen, der jeder Quadrant der MIP-Abbildung zugewiesen werden sollte. Wie es nachfolgend erklärt wird, geben für jeden MIP-Abbildungsquadranten die Ergebnisse einer logischen Exklusiv-ODER-Operation auf den Werten der höchstwertigen Bits der S- und T-Koordinaten für den Quadrant die SDRAM-Bank in dem Cache vor, der der Quadrant zugewiesen wird.

[0138] Für eine Abbildung von 512 × 512 Texeln spezifizieren neun S-Koordinatenbits S[8:0] und neun T-Koordinatenbits T[8:0] die Position jedes Texels in der Abbildung. Die Quadrantengrenzen werden auf halber Strecke der Abbildung sowohl in der S- als auch der T-Abmessung erstellt, die durch die höchstwertigsten S- und T-Koordinatenbits 5[8] und T[8] dargestellt sind. Um daher die Cachebänke für jeden der vier Quadranten einer MIP-Abbildung mit einer Abbildungsnummer von 9 zu bestimmen, wird für jeden Quadranten auf den Werten seiner entsprechenden höchstwertigsten S- und T-Koordinatenbits S[8] und T[8] eine Exklusiv-ODER-Operation durch-

geführt. Gleichartig dazu wird für eine MIP-Abbildung mit einer Abbildungsnummer von 10 die Cachebank für jeden Quadranten durch eine Exklusiv-ODER-Operation auf den entsprechenden Werten seiner höchstwertigsten S- und T-Koordinatenbits S[9] und T[9] bestimmt. Für MIP-Abbildungen mit einer ungeraden Abbildungszahl wird das Ergebnis der exklusiven ODER-Operation umgekehrt, so daß gemeinsame Daten von benachbarten Abbildungen in unterschiedlichen Bänken gespeichert werden.

[0139] Bei dem in **Fig.** 14 gezeigten Beispiel entsprechen die Blöcke, die mit Block **1**–**4** bezeichnet sind, jeweils dem oberen linken Quadranten, dem oberen rechten Quadranten, dem unteren linken Quadranten und dem unteren rechten Quadranten der 512 × 512-Texelabbildung. Für Block **1**–**4** sind die Bits S[8], T[8] jeweils gleich [0, 0], [1, 0], [0, 1] und [1, 1]. Daher ist für den Block **1** das Ergebnis der XOR-Operation S[8] XOR T[8] gleich Null. Weil die Abbildung eine ungerade Abbildungszahl aufweist (d. h. 9), wird das Inverse dieses Ergebnisses (gleich 1) verwendet, um anzuzeigen, daß Block **1** in Bank Eins des Cache gespeichert werden soll. Für Block **2** ist das Inverse des Ergebnisses der XOR-Operation S[8] XOR T[8] gleich Null, was anzeigt, daß der Block **2** in der Bank Null in dem Cache gespeichert werden soll. Für Block **3** und Block **4** ist das Inverse des Ergebnisses der XOR-Operation S[8] XOR T[8] jeweils gleich Eins und Null, was anzeigt, daß der Block **3** in der Bank Eins gespeichert werden soll und der Block **4** in der Bank Null gespeichert werden soll.

[0140] Für eine Abbildung mit einer Abbildungsnummer von 10 für die gleiche Textur, wie es in dem Beispiel von **Fig.** 14 gezeigt ist, würde die Abbildung in 16 Blöcke von jeweils 256 × 256 Texeln aufgeteilt, weil die 1.024 × 1.024 Texel groß ist. Für jeden Block würden die Ergebnisse von S[9] XOR T[9] die Banknummer für diesen speziellen Block anzeigen. Es ist anzumerken, daß die Ergebnisse der XOR-Operationen für jeden Block der Abbildung mit einer Abbildungsnummer von 10 nicht invertiert sind, da dieselben für die benachbarte Abbildung mit der Abbildungsnummer **9** waren, so daß die entsprechenden Quadranten in den beiden Abbildungen in unterschiedlichen Cachebänken gespeichert werden.

[0141] Abhängig von der Größe der Abbildung kann das Blocktag für Texturdatenblöcke, die die Abbildung darstellen, zumindest ein höherwertiges S-Koordinatenbit und ein höherwertiges T-Koordinatenbit umfassen, das die Position des Blocks in der speziellen MIP-Abbildung anzeigt. Für eine 512 × 512-Texel-MIP-Abbildung mit einer Abbildungsnummer von 9 wäre nur ein 5-Koordinatenbit und ein T-Koordinatenbit in dem Blocktag erforderlich, um die Position jedes Blocks in der MIP-Abbildung anzuzeigen. Für eine 1.024 × 1.024-Texe1-MIP-Abbildung mit einer Abbildungsnummer von 10, die 16 Datenblöcke umfaßt, wären zwei S-Koordinatenbits und zwei T-Koordinatenbits in dem Blocktag erforderlich, um die Position jedes Blocks in der MIP-Abbildung anzuzeigen. Für Abbildungen mit einer Abbildungsnummer von 8 oder weniger sind keine S- und T-Bits in dem Blocktag erforderlich. Wenn die Textur-MIP-Abbildungsdaten von dem Hauptspeicher des Rostcomputers zu dem Cachespeicher heruntergeladen werden, decodiert der Rostcomputer die höherwertige S- und T-Koordinatenbits des Blocktags unter Verwendung des oben erörterten exklusiven ODER-Schemas zum Bestimmen der speziellen Bank, in die jeder Datenblock geschrieben werden sollte.

[0142] Um Texturdaten zuzuweisen, so daß nicht genutzter Speicherplatz minimiert wird, kann jeder Datenblock weiter unterteilt werden in 16 Unterblöcke von 64 × 64 Texeln. Jeder Unterblock von Texturdaten umfaßt einen Untertextur-ID, der die Position des speziellen Unterblocks in dem Block identifiziert. Der Untertextur-ID umfaßt zwei S-Bits S[1:0] und zwei T-Bits Tj1:0]. Mehrere Untertexturen von einer oder mehreren MIP-Abbildungen von einer oder mehreren Texturen können in einem einzigen Block gespeichert werden.

[0143] **Fig.** 15 stellt Block **1** und Block **2** dar, die jeweils den Bänken Null und Eins des Caches zugewiesen sind, wobei jeder in 16 Untertexturen von 64 × 64 Texel Größe unterteilt ist. Die Untertexturen jedes Blocks sind mit ST0–ST15 bezeichnet und durch einen Untertextur-ID identifiziert, der zwei S-Koordinatenbits und zwei T-Koordinatenbits umfaßt. Die Untertexturen haben eine einheitliche Bezeichnung, aber spiegeln Positionen in den beiden Cachebanken, damit dieselben mit dem oben beschriebenen Speicherzuweisungsschema übereinstimmen. Die Größe der Untertexturen von 64 × 64 Texeln ist beispielhaft gewählt und kann geändert werden. Beispielsweise würde es eine kleinere Untertextur ermöglichen, daß mehr Texturen in die gleichen Blöcke gepackt werden. Es sollte klar sein, daß der Untertextur-ID mehr Bits umfassen muß, wenn die Größe der Untertextur verringert wird.

[0144] Während dem Aufbereiten werden für jeden Texelstrom, der interpoliert wird, der Textur-ID, der Untertextur-ID und das 8-Bit-Wort, das die Größe der Basisabbildung für diese Textur darstellt, die diesen Texeln zugeordnet ist, durch die 3D-Pipeline an den Kachel-/Grenzprüfer geliefert, der die Daten vorübergehend in einem 20-Bit-Register (nicht gezeigt) speichert. Wenn ein Texel, das interpoliert werden soll, einen anderen Untertextur-ID oder Textur-ID aufweist, werden die neuen Daten an den Kachel-/Grenzprüfer geliefert und in dem Register gespeichert. Der Untertextur-ID kann als Teil der Texeladresse verwendet werden, wie es nachfolgend erklärt wird.

[0145] Ob die Texeladresse ST-Koordinatenbits eines Untertextur-ID umfaßt, hängt von der Größe der Abbildung ab, die adressiert wird, und von der Größe der Basisabbildung dieser Textur. Falls die Abbildung, die adressiert wird, eine Abbildungsgröße von 7 oder kleiner aufweist, und die entsprechende Basisabbildung derselben ebenfalls die Größe **7** oder kleiner aufweist, dann umfassen bestimmte obere

Adreßbits der Texeladresse Bits von dem Untertextur-ID, um die Position der Untertextur in dem Block zu adressieren, wie es nachfolgend näher erklärt wird. Wenn die Basisabbildung eine Abbildungsnummer von 8 oder mehr aufweist, sind die Positionen aller der MIP-Abbildungsquadranten für diese Textur in ihren jeweiligen Datenblöcken vordefiniert, wie es oben erklärt ist. Wenn daher von einer der Abbildungen für diese Textur auf ein Texel zugegriffen wird, die eine Abbildungsnummer von 7 oder weniger aufweist, sind diese vordefinierten Positionen bekannt und werden verwendet, um die oberen Bits der Texeladresse für jeden Quadranten zu erzeugen, ohne den Untertextur-ID zu verwenden. Wenn die Basisabbildung einer Textur jedoch eine Abbildungsnummer von 7 oder weniger aufweist, sind die Positionen der MIP-Abbildungsquadranten nicht vordefiniert und die Untertextur-ID-Bits werden als obere Bits der Texeladresse verwendet, um die Position der Untertextur zu bestimmen.

[0146] Wie es oben angemerkt wurde, können mehrere Abbildungen von unterschiedlichen Texturen in unterschiedlichen Untertexturen eines einzelnen Datenblocks gespeichert werden, solange die Basisabbildung von dieser Textur klein genug ist. Wenn dies auftritt, umfaßt die Texturadresse für jede Abbildung Untertextur-ID-Bits. Falls beispielsweise vier unterschiedliche Abbildungen mit der Abbildungsnummer von 7 von vier unterschiedlichen Texturen zwischen unterschiedlichen Untertexturen in einem Block zugewiesen werden und die Abbildungsnummer für die Basisabbildung jeder Textur **7** ist, dann wären ein S-Koordinatenbit und ein T-Koordinatenbit von dem Untertextur-ID Teil der Texeladresse zum Unterscheiden zwischen den Texturen. Die Routine, mit der der Kachel-/Grenzprüfer die Texeladresse berechnet, ist nachfolgend mit Bezugnahme auf **Fig.** 17 beschrieben.

[0147] Bei dem dargestellten Ausführungsbeispiel der Erfindung werden Textur-MIP-Abbildungsdaten einen Block nach dem anderen Block heruntergeladen. Es sollte jedoch klar sein, daß alternativ ein Untertextur-ID in dem Blocktag enthalten sein kann, so daß Untertexturen von dem Hauptspeicher heruntergeladen werden könnten. Außerdem sollen die Größen der Blöcke und Untertexturen, die bei diesem Ausführungsbeispiel beschrieben sind, lediglich beispielhaft sein und können geändert werden, um an jede Anwendung angepaßt zu werden.

VI. Cacheblockta und Blockindex

[0148] Das Cacheverzeichnis umfaßt einen Blocktag für jeden seiner 64 Einträge und identifiziert einen entsprechenden Blockindex für jeden Eintrag. Der Blockindex identifiziert die physikalische Position in dem Cache, wo der Anfang des entsprechenden Texturdatenblocks gespeichert ist. Das Blocktag ist ein 23-Bit-Identifizierer, der jeden Texturdatenblock auf die in **Fig.** 16 gezeigte Weise eindeutig identifiziert.

[0149] Um jedes Texel von Texturdaten eindeutig zu identifizieren, muß die Textur, der derselbe entspricht, identifiziert werden. Bei einem Ausführungsbeispiel der Erfindung unterstützt die Texturabbildungshardware einen 8-Bit-Textur-ID, der eine Textur eindeutig identifiziert. Außerdem wird für Texturdaten von unterschiedlichen Texturen, die in dem gleichen Block gespeichert sind, ein zusätzlicher 4-Bit-Untertextur-ID durch die Hardware unterstützt, um die Texturen zu identifizieren. Somit unterstützt die Texturabbildungshardware der vorliegenden Erfindung **212** oder 4.096 eindeutige Texturen, die zu jeweils einem Zeitpunkt aktiv sein können.

[0150] Wie es oben erörtert wurde, ist jede Textur durch eine Reihe von MIP-Abbildungen dargestellt, und bei einem Ausführungsbeispiel der Erfindung ist jede der MIP-Abbildungen mit einer Abbildungsnummer versehen, die die Position derselben in der Reihe von MIP-Abbildungen anzeigt. Somit ist jedes Datentexel nicht nur durch den Textur-ID, den Untertextur-ID und die Größe der Basisabbildung für diese Textur identifiziert, sondern auch durch die Abbildungsnummer der MIP-Abbildung, der dasselbe entspricht. Schließlich ist das Texel innerhalb der MIP-Abbildung durch seine S-und T-Koordinaten (d. h. seinen interpolierten ST-Werten) eindeutig identifiziert.

[0151] Außer dem Untertextur-ID und der Texturabbildungsbasisgröße werden die oben beschriebenen Parameter, die ein Texel eindeutig identifizieren, verwendet, um das 23-Bit-Blocktag zu erzeugen. Bezüglich der Abbildungsnummer und den S- und T-Koordinaten ist bei einem Ausführungsbeispiel der vorliegenden Erfindung die Hardware, die verwendet wird, um die S- und T-Koordinaten zu erzeugen, auf 15 Bits beschränkt. Daher hat bei diesem Ausführungsbeispiel die größte Texturabbildung, die durch die Hardware unterstützt wird, ein 15-Bit-S-Feld [14:0] und ein 15-Bit-T-Feld [14:0], was zu einer maximalen Texturabbildung führt, die 32K × 32K Texel groß ist. Wie es oben erörtert wurde, umfaßt jeder Texeldatenblock 256 × 256 Texel. Somit werden die niederwertigen Sund T-Bits (d. h. T[7:0] und S[7:0]) verwendet, um ein spezielles Texel in einem Texeldatenblock zu identifizieren. Nur die höherwertigen S- und T-Bits (T(14:8] und S[14:8]) werden in dem Blocktag verwendet, um einen speziellen Texeldatenblock zu identifizieren.

[0152] Wie es oben angemerkt wurde, ist jeder MIP-Abbildung eine Abbildungsnummer zugewiesen, die dieselbe eindeutig identifiziert innerhalb der Reihe von Abbildungen für seine entsprechende Textur. Unabhängig von der Nummer der MIP-Abbildungen in der Reihe von Abbildungen für eine Textur ist die kleinste MIP-Abbildung in der Reihe (d. h. ein Texel groß) der Abbildungsnummer 0 zugewiesen. Da die größte Reihe von MIP-Abbildungen für eine 32K × 32K-Textur 16 MIP-Abbildungen umfaßt, ist die größte unterstützte Abbildungsnummer **15**.

[0153] Die Art und Weise, wie das Blocktag gebildet

ist, ist in der Tabelle von **Fig.** 16 gezeigt. Die höherwertigen acht Bits des Blocktags [22:15] entsprechen dem Textur-ID der Textur, die durch den Texturdatenblock dargestellt ist. Die niederwertigen Bit des Blocktags [13:00] entsprechen den höherwertigen T- und S-Koordinaten, T[14:08] und S[14:08]. Das Blocktag [**14]** entspricht einem Abbildungsbit, das in Verbindung mit den Werten in dem höherwertigen T-Koordinatenfeld die Identifikation der Abbildungsnummer ermöglicht. Es sollte klar sein, daß Abbildungen, die kleiner sind als die maximalen 32K × 32K nicht die vollständigen S- und T-Adreßfelder verwenden, so daß um so mehr höherwertige S- und T-Adreßbits nicht verwendet werden, je kleiner die Abbildung ist. Wie es in **Fig.** 16 gezeigt ist, wird für Abbildungen mit einer Abbildungsnummer größer als 8 das Blocktagbit, das dem niedrigstwertigen nicht verwendeten T-Koordinatenbit entspricht, auf eine logische „0" gesetzt, und die Blocktagbits, die den verbleibenden höherwertigen T-Koordinatenbits entsprechen, und das Abbildungsbit werden auf eine logische „1" gesetzt. Für die Abbildungsnummer **15**, die alle T-Koordinatenbits verwendet, ist das Abbildungsbit auf eine logische „0" gesetzt. Durch Lesen von Blocktagbits [14:07], die dem Abbildungsbit und den höherwertigen T-Koordinatenbits [14:0] entspricht, zeigt die Position der ersten logischen „0", die beim Lesen von links nach rechts angetroffen wird, die Abbildungsnummer an, die durch das Blocktag dargestellt wird. Falls in allen Blocktagbits [14:08] eine logische „1" enthalten ist, sind die Abbildungsnummern **8** und weniger dargestellt.

[0154] Wie es oben beschrieben wurde, sind alle der Abbildungen einer speziellen Textur mit einer Abbildungsnummer von 8 oder weniger in zwei Datenblöcken gespeichert, wobei jeder Block in einer anderen Bank des Cache positioniert ist. Zwei Quadranten oder die Hälfte von jeder der Abbildungen mit Abbildungsnummern von 8 oder weniger sind in jedem der beiden Blöcke gespeichert. Das Blocktagbit [**07]** stellt dar, in welchem der beiden Blöcke jeder halbe Abschnitt der Abbildungen mit den Abbildungsnummern von 8 oder weniger gespeichert ist. Somit hat für jede der Abbildungen mit einer Abbildungsnummer von 8 oder weniger das Blocktagbit [**07]** einen Wert von „0" für die Hälfte (zwei Quadranten) dieser Abbildung (die in dem Block in Bank 0 gespeichert ist) und hat einen Wert von „1" für die andere Hälfte (zwei Quadranten) dieser Abbildung (die in dem Block in Bank **1** gespeichert ist). Da alle Abbildungen von einer speziellen Textur mit einer Abbildungsnummer von 8 oder weniger in zwei Blöcken gespeichert sind, sollte klar sein, daß dann nur ein Blocktagbit verwendet wird, um diese beiden Blöcke zu identifizieren. Die spezielle Abbildungsnummer für jede der Abbildungen mit einer Nummer von 8 und niedriger ist daher nicht als Teil des Blocktagfelds gespeichert.

[0155] Der Wert des Blocktagbits [07] für jeden Quadranten von jeder der Abbildungen mit einer Abbildungsnummer von 8 oder weniger wird auf der Basis des Schemas zum Bestimmen der Bank, in der der Quadrant gespeichert sein sollte, berechnet. Dieses Schema umfaßt die logische Exklusiv-ODER-Operation der Werte der MSB-Bits für jeden Quadranten von geradzahligen Abbildungen und das Inverse der Operation für jeden Quadranten von ungeradzahligen Abbildungen.

[0156] Wie es in **Fig.** 16 gezeigt ist, sind die Blocktagbits [6:0], die den höherwertigen S-Adreßbits entsprechen, für kleine Abbildungen auf eine logische „0" gesetzt, wenn die S-Adreßbits nicht verwendet werden, so daß, falls eines dieser Bits als eine logische „1" erfaßt wird, in Verbindung mit einer Abbildungsnummer, die anzeigt, daß dieselben gleich einer logischen „0" sein sollten, dasselbe verwendet werden kann, um anzuzeigen, daß es keine gültigen Daten in dem Cacheverzeichniseintrag gibt.

[0157] Wie es oben erörtert wurde, geben für jeden MIP-Abbildungsquadranten die Ergebnisse einer logischen exklusiven ODER-Operation (XOR-Operation) an den Werten der höchstwertigsten S- und T-Koordinaten für den Quadranten die SDRAM-Bank in dem Cache vor, dem der Quadrant zugewiesen ist. Die Banknummer ist gleich dieser XOR-Operation für Abbildungen mit einer geraden Abbildungsnummer und ist gleich dem logischen Inversen der XOR-Operation für Abbildungen mit einer ungeraden Abbildungsnummer. Dies ist in der rechten Spalte der Tabelle von **Fig.** 23 gezeigt, wo das Symbol „^" eine XOR-Operation anzeigt und das Symbol „!" ein logisches Inverses anzeigt. Für Abbildungen mit einer Abbildungsnummer von 9 oder höher verbraucht jeder Quadrant zumindest einen vollen Datenblock und jeder Block ist in der Bank gespeichert, die durch die XOR-Operation vorgegeben ist, die in der letzten Spalte der Tabelle von **Fig.** 16 gezeigt ist.

[0158] Für Abbildungen mit einer Abbildungsnummer von 8 oder weniger besetzen alle dieser Abbildungen zwei Datenblöcke, einen Block in jeder Bank. Die letzten zwei Zeilen der Tabelle von **Fig.** 16 entsprechen unterschiedlichen Hälften (zwei Quadranten) von jeder Abbildung mit einer Abbildungsnummer von 8 oder weniger. Das Blocktagbit [07] stellt dar, in welchem Block der Bank 0 oder in welchem Block der Bank **1** die halbe Abbildung gespeichert ist. Der Wert dieses Bits [**0**7] wird auf der Basis der beschriebenen XOR-Operation berechnet. Beispielsweise wäre für eine Abbildung mit einer Abbildungsnummer von 8 für jeden Quadranten der Abbildung das Blocktagbit [07] gleich S[7] XOR T[7]. Für jeden Quadranten einer Abbildung mit einer Abbildungsnummer **7** wäre das Blocktagbit [07] gleich dem Inversen von S[6] XOR T[6]. Das Blocktagbit [07] wird für jeden Quadranten kleiner Abbildungen gleichartig berechnet, wobei das Ergebnis der XOR-Operation nur für ungerade Abbildungen invertiert wird. Weil zwei Quadranten jeder Abbildung (mit einer Abbildungsnummer von 8 oder weniger) in dem gleichen Block gespeichert sind, sollte aus dem Vorhergehenden klar sein, daß diese beiden Quadranten jeder Ab-

bildung das gleiche Blocktagbit [07] aufweisen würden.

[0159] Wenn zwischen interpolierten ST-Koordinaten (die ein Texel adressieren, auf das zugegriffen werden soll) und einem der 23-Bit-Blocktags in dem Cacheverzeichnis ein Treffer auftritt, erzeugt das Cacheverzeichnis einen Blockindex, der die physikalische Position in dem Cachespeicher identifiziert, wo der Cacheblock gespeichert ist, der dieses Texel enthält. Der Cache speichert 64 Texeldatenblöcke zu einem Zeitpunkt. Um daher eine Blockadresse in dem Cachespeicher zu identifizieren, ist ein 6-Bit-Blockindex ($2^6 = 64$) vorgesehen, der als die höherwertigen Adreßbits für den Cache dient, wie es oben beschrieben ist.

[0160] Die Texeladresse ist ein 16-Bit-Wort, das die Bits S[7:0] und T[7:0] enthält, und das die Position des Texels, auf das zugegriffen werden soll, in dem 256 × 256-Texelblock anzeigt. Die Texeladresse wird von den interpolierten ST-Koordinaten, der Abbildungsnummer der Abbildung, auf die zugegriffen werden soll, der Textur- und Untertextur-IDs, und der Basisabbildungsgröße der Textur gemäß einer Routine berechnet, die nachfolgend mit Bezugnahme auf **Fig.** 17 erörtert wird. Wie es oben erörtert wurde, werden das LSB-S-Bit und das LSB-T-Bit der Texeladresse decodiert, um die geeignete Verschachtelung zu bestimmen, in der das Texel gespeichert ist. Die verbleibenden 14 Bits der Texeladresse in Verbindung mit den sechs Blockindexbits dienen als die Cacheadresse (wobei die sechs Bits des Blockindex die sechs MSB der Cacheadresse sind), die an das SDRAM-Paar in der decodierten Verschachtelung des Cache geliefert wird.

VII. Texeladreßberechnung

[0161] Während dem Aufbereiten empfängt das Kachel-/Grenzprüferelement **72** von dem Interpolator **64** den interpolierten ST-Wert des Texels, auf das zugegriffen werden soll, und auch ein 4-Bit-Wort, das die Abbildungszahl der Abbildung darstellt, von der aus auf das Texel zugegriffen werden soll. Jeder der interpolierten S- und T-Koordinatenwerte, der von dem Parameterinterpolator empfangen wird, umfaßt 16 Ganzzahlbits und 8 Bruchteilbits. Das 4-Bit-Wort, das die Abbildungsnummer darstellt, umfaßt Abbildungen, die von der Abbildungsnummer 0 (ein Texel groß) zu einer Abbildungsnummer **15** (32K × 32K Texel groß) reicht und von dem Gradienten berechnet wird, wie es oben beschrieben ist. Ein Vergleich des interpolierten ST-Werts mit den Blocktageinträgen in dem Cacheverzeichnis wird dann durchgeführt. Falls ein Treffer mit einem der Blocktags auftritt, wird der Blockindex erzeugt. Gleichzeitig zur Durchführung der Cacheverzeichnissuche wird die Texeladresse berechnet, gemäß der Routine, die nachfolgend mit Bezugnahme auf das Flußdiagramm von **Fig.** 17 beschrieben ist.

[0162] Die Texeladresse wird durch den Kachel-/Grenzprüfer auf der Basis des Textur-ID, des Untertextur-ID, der Abbildungsnummer, der Basisabbildungsnummer und den interpolierten ST-Koordinaten des Texels berechnet. Der Kachel-/Grenzprüfer hat alle diese Informationen. Für jedes einzelne Texel, auf das zugegriffen werden soll, empfängt der Kachel/Grenzprüfer von dem Parameterinterpolator die interpolierten ST-Koordinaten (einschließlich 16 Ganzzahl- und 8 Bruchteil-Bits sowohl für S als auch T), und ein 4-Bit-Wort, das die Abbildungsnummer darstellt, von der aus auf das Texel zugegriffen werden soll. Außerdem wird durch die 3D-Pipeline (die auch durch den Parameterinterpolator kommt) ein Befehl empfangen, der den 8-Bit-Textur-ID, den 4-Bit-Untertextur-ID und ein 8-Bit-Wort umfaßt, das die Größe der Basisabbildung für diese Textur darstellt. Das 8-Bit-Wort, das die Größe der Basisabbildung darstellt, umfaßt vier S-Bits und vier T-Bits, die dem Abbildungsnumerierungsschema der Erfindung entsprechen und jeweils die Größe der S-Abmessung und T-Abmessung der Basisabbildung definieren. Beispielsweise kann jedes der 4-Bit-S- und T-Wörter einen Wert aufweisen, der von 0 (was einer Abmessung von einem Texel entspricht) bis 15 reicht (was einer Abmessung von 32K Texeln entspricht). Die 20 Datenbits, die den Textur-ID, den Untertextur-ID und die Basisnummer umfassen, werden vorübergehend in einem 20-Bit-Register (nicht gezeigt) in dem Kachel-/Grenzprüfer gespeichert, bis dieselben mit neuen und unterschiedlichen Daten für ein nachfolgendes Texel ersetzt werden, auf das von dem Cache zugegriffen werden soll. Mit diesen Informationen berechnet der Kachel-/Grenzprüfer die Texeladresse für jedes Texel.

[0163] Wie es oben erklärt wurde, haben die Quadranten jeder Abbildung für Texturen mit einer Basisabbildung mit einer Abbildungsnummer von größer oder gleich 8 (entspricht einer Basisabbildung von 256 × 256 Texeln oder mehr), in dieser Textur eine vordefinierte Position in den Texturdatenblökken und Cachespeicherbänken. Somit kann jedes Bit der Texeladresse für jedes Texel einer solchen Textur gemäß diesem bekannten vordefinierten Zuweisungsschema berechnet werden. Für Texturen mit einer Basisabbildung mit einer Abbildungsnummer von 7 oder weniger (was einer Basisabbildung von 128 × 128 Texeln oder weniger entspricht) sind jedoch eine Anzahl von bestimmten Speicherpositionen für jeden Quadranten der Abbildung der Textur verfügbar, und daher werden bestimmte höherwertige Bits der Texeladresse einige oder alle Bits (oder das Inverse dieser Bits) des Untertextur-ID umfassen.

[0164] Die Routine, die durch den Kachel-/Grenzprüfer implementiert wird, um die Texeladresse zu berechnen, ist durch das Flußdiagramm von **Fig.** 17 dargestellt. Zum Vervollständigen erfordert die Routine nur einen Zyklus. Die Routine kann durch einen Satz von Logikgattern (nicht gezeigt) implementiert werden, die den Grenzprüferabschnitt des Texturabbildungschips bilden. Für einen Fachmann auf die-

sem Gebiet sollte klar sein, wie Logikgatter zu implementieren sind, um die Routine durchzuführen, die durch das Flußdiagramm von **Fig.** 17 dargestellt ist. Beispielsweise kann die Routine in einer Softwaresimulationssprache geschrieben werden, wie z. B. Verflog, und dann in eine Logikgatterschaltung umgewandelt werden durch ein Synthesewerkzeug, wie z. B. Synopsys™, das auf einem Universalprozessor arbeitet. Die Routine kann alternativ in Software geschrieben werden und durch einen Prozessor durchgeführt werden.

[0165] Die Routine beginnt bei Schritt 250, wo die Texeladreßbits S[7:0], T[7:0] voreingestellt sind, um gleich den interpolierten ST-Koordinatenbits S[7:0], T[7:0] zu sein. Jedes der Bits der Texeladresse bleibt in diesem Schritt bei dem Wert, auf das es voreingestellt ist (gleich der entsprechenden S- oder T-Koordinate), außer derselbe wird später in der Routine neu gesetzt. Dann schreitet die Routine zu Schritt 252 fort, wo bestimmt wird, ob die spezielle Abbildung, in der das Texel, das interpoliert wird, gespeichert ist, eine Abbildungsnummer von mehr oder gleich 8 aufweist. Falls dies der Fall ist, dann endet die Routine für ein solches Texel und die Bitwerte für die Texeladresse bleiben wie voreingestellt gleich den interpolierten ST-Koordinaten.

[0166] Falls die Abbildungsnummer nicht größer oder gleich 8 ist, dann schreitet die Routine zu Schritt 254 fort, wo bestimmt wird, ob das Texel in der Bank Nr. 1 oder in der Bank Nr. 0 gespeichert ist. Wie es oben beschrieben ist, ist es durch Untersuchen des Werts des Blocktagbits [07] bekannt, ob das Texel in der Bank Nr. 1 oder der Bank Nr. 0 gespeichert ist.

[0167] Falls das Texel in der Bank Nr. 1 gespeichert ist, schreitet die Routine zu Schritt 256 fort, wo bestimmte Texeladreßbits von ihren voreingestellten Werten zurückgesetzt werden. Für Abbildungen mit Abbildungsnummern 1–4 ist das Texeladreßbit S[4] = 1 und für Abbildungen mit Abbildungsnummern 1 und 2 ist das Texeladreßbit S[2] = 1. Falls das Texel in der Bank 0 gespeichert ist, schreitet die Routine zu Schritt 258 fort, wo für die Abbildungen mit den Abbildungsnummern 0–5 das Texeladreßbit S[5] = 1 ist, für Abbildungen mit den Abbildungsnummern 0–3 das Texeladreßbit S[3] = 1 ist und für Abbildungen mit einer Abbildungsnummer 0 und 1 das Texeladreßbit S[1] = 1 ist.

[0168] Von jedem der Schritte 256 und 258 schreitet die Routine zu Schritt 260 fort, wo es bestimmt wird, ob die Basisabbildung eine Abbildungsnummer aufweist, die größer oder gleich 8 ist. Falls dies der Fall ist, schreitet die Routine zum Schritt 262 fort, wo bestimmt wird, ob das Texel in der Bank 0 oder der Bank 1 gespeichert ist. Falls das Texel in der Bank 1 gespeichert ist, schreitet die Routine zu Schritt 264 fort, wo für eine Abbildung mit einer Abbildungsnummer von 7 das Texeladreßbit S[7] = 0 ist und für Abbildungen mit den Abbildungsnummern 0–6 die Texeladreßbits S [7 : 6] = 0: 1 sind. Die Routine ist dann für ein solches Texel beendet. Für ein Texel, das in der Bank 0 gespeichert ist, schreitet die Routine zu Schritt 266 fort, wo für eine Abbildung mit einer Abbildungsnummer von 7 das Texeladreßbit S[7] = 1 ist und für Abbildungen mit den Abbildungsnummern 0–6 die Texeladreßbits S[7:6] = 1:0 sind. Die Routine ist dann für ein solches Texel beendet.

[0169] Falls die Basisabbildung keine Abbildungsnummer größer oder gleich 8 aufweist, schreitet die Routine zu Schritt 268 fort, wo bestimmt wird, ob die Basisabbildung eine Abbildungsnummer gleich 7 aufweist. Falls dies der Fall ist, schreitet die Routine zu Schritt 270 fort, wo bestimmt wird, ob das Texel in der Bank 0 oder 1 gespeichert ist. Falls das Texel in der Bank **1** gespeichert ist, schreitet die Routine zu Schritt 272 fort, wo für die Abbildungsnummer **7** das Texeladreßbit S[7] gleich dem Inversen des Untertextur-ID-Bits S[1] ist, und das Texeladreßbit T[7] gleich dem Untertextur-ID-Bit T[1] ist, und für Abbildungen mit einer Abbildungsnummer von 0 – 6 die Texeladreßbits S[7:6] gleich dem Inversen des Untertextur-ID-Bits S[1] bzw. 1 sind, und das Texeladreßbit T[7] gleich dem Untertextur-ID-Bit T[1] ist. Die Routine endet dann für ein solches Texel. Falls das Texel in der Bank 0 gespeichert ist, schreitet die Routine zu Schritt **274** fort, wo für eine Abbildung mit einer Abbildungsnummer von 7 das Texeladreßbit S[7] gleich dem Untertextur-ID-Bit S[1] ist, und das Texeladreßbit T[7] gleich dem Untertextur-ID-Bit T[1] ist, und für Abbildungen mit einer Abbildungsnummer von 0–6 die Texeladreßbits S[7:6] gleich dem Untertextur-ID-Bit S[1] bzw. 0 sind, und das Texeladreßbit T[7] gleich dem Untertextur-ID-Bit T[1] ist. Die Routine endet dann für ein solches Texel.

[0170] Falls die Basisabbildung der Textur weder eine Abbildungsnummer von mehr oder gleich 8 (bestimmt bei Schritt **260**), noch eine Abbildungsnummer gleich 7 (bestimmt bei Schritt **268**) aufweist, dann ist selbstverständlich bekannt, daß die Basisabbildung der Textur eine Abbildungsnummer von weniger °der gleich 6 aufweist und die Routine schreitet zu Schritt 276 fort, wo bestimmt wird, ob das Texel in Bank **1** oder Bank 0 gespeichert ist. Falls das Texel in Bank **1** gespeichert ist, schreitet die Routine zu Schritt **278** fort, wo die Texeladreßbits S[7:6] gleich dem Inversen der Untertextur-ID-Bits S[1:0] gesetzt werden, und die Texeladreßbits T[7:6] gleich den Untertextur-ID-Bits T[1:0] gesetzt werden. Die Routine ist dann für ein solches Texel abgeschlossen. Falls das Texel in der Bank 0 gespeichert ist, schreitet die Routine zu Schritt **280** fort, wo die Texeladreßbits S[7:6] gleich den Untertextur-ID-Bits S[1:0] sind, und die Texeladreßbits T[7:6] gleich den Untertextur-ID-Bits T[1:0] sind. Die Routine ist dann für ein solches Texel abgeschlossen.

VIII. Texturdatenorganisationsbeispiele

[0171] Das folgende Beispiel beschreibt die Prozedur, durch die der Hostcomputer Texturdaten gemäß dem oben beschriebenen Ausführungsbeispiel der

Erfindung organisiert. Für eine spezielle Anwendung kann ein Grundelement A, das aufbereitet werden soll, auf eine Textur A abbilden, und ein Grundelement B kann auf eine Textur B abbilden. Eine Möglichkeit wäre, daß der Hostcomputer die Textur A in eine Mehrzahl von Texturdatenblöcken organisiert, und dann die Textur B innerhalb des gleichen Blocks wie die Textur A in unterschiedliche Untertexturen organisiert. Der Hostcomputer würde die Texturdatenblöcke, die die Texturen A und B umfassen, in den Cachespeicher herunterladen, bevor die Grundelemente A und B aufbereitet werden.

[0172] Alternativ kann der Host die Textur A in eine Mehrzahl von Texturdatenblöcken organisieren und dann die Blöcke, die die Textur A umfassen, in den Cachespeicher herunterladen. Der Hostcomputer könnte dann die Textur B in dem Hauptspeicher innerhalb der unterschiedlichen Untertexturen in den gleichen Blöcken wie die Textur A organisieren. In dieser Situation würde der Hostcomputer einen Befehl ausgeben, den Betrieb des Texturabbildungschips **46** anzuhalten (**Fig.** 2), und würde die neu organisierten Texturdatenblöcke (einschließlich den Texturen A und B in den gleichen Blöcken zu dem Cachespeicher des Texturabbildungssystems herunterladen. Falls die HALT-Bedingung nicht implementiert ist und die neu organisierten Daten von dem Hauptspeicher nicht in den Cachespeicher des Texturabbildungssystems heruntergeladen werden, ist klar, daß während dem Aufbereiten des Grundelements B auf falsche Texturabbildungsdaten zugegriffen werden könnte. Dies liegt daran, daß ein Treffer in dem Cacheverzeichnis auftreten würde, wenn das Grundelement B aufbereitet wird, weil das Lesecachetag für den Datenblock, der die Textur B umfaßt, mit dem Blocktag übereinstimmen würde, das den Datenblöcken in dem Cache entspricht, die die Textur A speichern. Die Datenblöcke in dem Cache speichern jedoch nur Texturdaten, die sich auf die Textur A beziehen und nicht auf die Textur B.

IX. Umgehung einer dreidimensionalen Grundelementpipeline und Unterbrechungsschema zum Herunterladen von Texturabbildungen

[0173] Wie es oben erörtert wurde, ermöglicht es ein Merkmal der vorliegenden Erfindung, daß eine MIP-Abbildung für eine neue Textur durch einen Datenweg in den lokalen Speicher in der Texturabbildungshardware heruntergeladen wird, der von der Pipeline zum Handhaben von 3D-Grundelementdaten getrennt ist. Mit Bezugnahme auf das in den Figuren offenbarte darstellende Ausführungsbeispiel haben die Texturabbildungsplatine **12** (**Fig.** 2) und der Texturabbildungschip **46** (**Fig.** 3) jeweils getrennte Tore zum jeweiligen Empfangen von 3D-Grundelementdaten und Texturdaten. Die 3D-Grundelementdaten werden über den Bus **18** von dem Konzentratorchip **36** empfangen, während die Texturdaten über den Bus **24** von dem 2D-Geometriebeschleunigerchip **34** empfangen werden. Wenn daher neue Texturdaten von dem Hostcomputer **15** zu dem Texturabbildungschip **46** heruntergeladen werden, muß die 3D-Grundelementpipeline durch die Vorderendeplatine **10** und den Texturabbildungschip **46** nicht entleert werden, wodurch eine erhöhte Bandbreite geliefert wird im Vergleich zu herkömmlichen Texturabbildungssystemen, die das Entleeren der 3D-Grundelementpipeline jedesmal erfordern, wenn neue Texturdaten zu dem lokalen Speicher in der Texturabbildungshardware heruntergeladen werden.

[0174] Der getrennte Datenweg zum Herunterladen von Texturdaten, der die 3D-Grundelementpipeline umgeht, ist besonders vorteilhaft in Verbindung mit dem oben beschriebenen Ausführungsbeispiel der vorliegenden Erfindung, bei dem der lokale Speicher in der Texturabbildungsplatine **12** als ein Cache implementiert ist. Wenn neue Texturdaten zu dem Cache heruntergeladen werden, wird, wie es oben erörtert ist, nur der Teil der MIP-Abbildung, der erforderlich ist, heruntergeladen, anstatt der gesamten Reihe von MIP-Abbildungen für die Textur. Somit ermöglicht die 3D-Pipelineumgehung, daß Cachefehlschläge ohne Entleeren der Pipeline gehandhabt werden.

[0175] Wie es oben erörtert wurde, werden bei einem Ausführungsbeispiel der Erfindung, das in **Fig.** 2A gezeigt ist, Abschnitte des Graphiksystems dupliziert, um die Systembandbreite zu erhöhen. Die Texturabbildungsplatine **12** ist mit zwei Texturabbildungschips **46A** und **46B** und mit zwei Cachespeichern **48A** und **48B** versehen. Bei diesem Ausführungsbeispiel behalten beide Cachespeicher **48** immer die gleichen Texturdaten bei, weil beide Texturabbildungschips typischerweise gleichzeitig Grundelemente bearbeiten, unter Verwendung der gleichen Texturdaten. Daher bewahrt dieses Ausführungsbeispiel der vorliegenden Erfindung durch Aktualisieren beider Caches jedesmal, wenn ein Fehlschlag von einem empfangen wird, die Systembandbreite durch Sicherstellen, daß die gleichen Texturdaten nicht in getrennten Operationen zu den beiden Caches heruntergeladen werden.

[0176] Bei dem Dualtexturabbildungschipausführungsbeispiel von **Fig.** 2A wird jeder Cachespeicher nur mit Texturdaten aktualisiert, die von dem Hostcomputer heruntergeladen werden, und wird nicht lokal von der Texturabbildungshardware beschrieben. Daher wird eine Übereinstimmung zwischen den beiden Cachespeichern beibehalten durch Sicherstellen, daß jedesmal, wenn ansprechend auf einen Fehlschlag von einem der Caches Texturdaten von dem Hostcomputer heruntergeladen werden, beide Caches mit den neuen Texturdaten aktualisiert werden. Wenn ein Cachefehlschlag von einem der Texturabbildungschips **56** auftritt und eine Unterbrechung erzeugt wird, werden beide Texturabbildungschips **46** angehalten, so daß beide Cachespeicher mit den heruntergeladenen Texturdaten aktualisiert werden können. Somit spricht jeder Texturabbildungschip auf die Erzeugung eines Cachefehl-

schlagsignals von jedem der Texturabbildungschips an, um den Betrieb anzuhalten. Zusätzlich unterstützt die vorliegende Erfindung gleichzeitige Cachefehlschläge von den beiden Texturabbildungschips **46** auf unterschiedliche Cacheblöcke und antwortet durch Herunterladen beider neuen Blöcke von Texturdaten zu beiden Caches, ansprechend auf die Fehlschläge.

[0177] Bei dem in **Fig.** 2 gezeigten dargestellten Ausführungsbeispiel wird das Umgehen der 3D-Grundelementpipeline erreicht durch Verwenden der 2D-Grundelementpipeline durch den 2D-Geometriebeschleunigerchip 34, um Texturdaten herunterzuladen. Somit wird bei diesem Ausführungsbeispiel ein Teil des 2D-Grundelementdatenwegs gemeinschaftlich verwendet mit und bildet einen Teil des Wegs zum Herunterladen von Texturdaten. Es sollte klar sein, daß der Datenweg zum Herunterladen von Texturdaten zu dem Texturabbildungschip **46** auf eine Anzahl von verschiedenen Möglichkeiten implementiert werden kann, während nach wie vor die 3D-Grundelementpipeline umgangen wird. Beispielsweise kann ein speziell zugewiesener Datenweg von dem Hostcomputer zu der Texturabbildungsplatine vorgesehen sein.

[0178] Der Hostcomputer des Graphiksystems der vorliegenden Erfindung kann ein Betriebssystem, wie z. B. UNIX, verwenden, bei dem mehrere Prozesse gleichzeitig arbeiten können, und das ein Schema liefert, das es einem Prozeß ermöglicht, bestimmte Systemressourcen zu sperren, wie z. B. daß ein Prozeß nicht unterbrochen werden kann, wenn er gesperrt ist. Durch Verwenden des Sperrschemas kann ein Prozeß, der bestimmte Hardwareressourcen verwendet, sicherstellen, daß der Prozeß nicht ausgetauscht wird, bis er diese Ressourcen entsperrt.

[0179] Bei einem Ausführungsbeispiel der Erfindung sind zwei Sperrtypen für die Verwendung durch Prozesse vorgesehen, d. h. eine schnelle Sperre und eine langsame Sperre. Wenn eine schnelle Sperre verwendet wird, prüft ein Prozeß, der eingetauscht wird, die geeigneten Hardwareressourcen zum Bestimmen, ob derselbe der letzte Prozeß ist, der diese Ressourcen verwendet. Falls er dies war, setzt der Prozeß einfach fort ohne den Zustand der Hardwareressourcen wiederherzustellen. Falls der Prozeß jedoch nicht der letzte war, der die Ressourcen verwendet, ist eine langsame Sperre erforderlich, was zur Wiederherstellung der Hardwareressourcen zu dem Zustand führt, in dem sie waren, als der Prozeß als letztes ausgetauscht wurde. Es sollte klar sein, daß eine Anzahl alternativer Techniken verwendet werden kann, um die gleichen Ergebnisse zu erreichen.

[0180] Bei dem Ausführungsbeispiel der vorliegenden Erfindung, bei dem die 2D-Grundelementpipeline verwendet wird, um Texturdaten herunterzuladen, während 3D-Grundelemente aufbereitet werden, werden 2D- und 3D-Prozesse nicht gleichzeitig betrieben. Diese Beschränkung wird erfüllt indem durch die Verwendung des Sperrschemas, das durch das Betriebssystem des Rostcomputers geliefert wird, sichergestellt wird, daß kein 2D-Prozeß beginnt, außer wenn die 3D-Pipeline leer ist, und daß kein 3D-Prozeß beginnt, außer wenn die 2D-Pipeline leer ist. Wenn ein 3D-Prozeß beginnt, aktiviert er eine Sperre, und wenn der vorhergehende Prozeß ein 2D-Prozeß war, wartet derselbe, bis die 2D-Pipeline leer ist, bevor er beginnt. Gleichartig dazu, wenn ein 2D-Prozeß beginnt, aktiviert er eine Sperre, und wenn der vorhergehende Prozeß ein 3D-Prozeß war, wartet derselbe, bis die 3D-Pipeline leer ist, bevor er beginnt.

[0181] Einige Prozesse führen sowohl 3D- als auch 2D-Operationen durch und können zwischen 3D-Grundelementen und 2D-Grundelementen schalten, ohne die langsame Sperre aufzugeben. Solche Prozesse implementieren auch ein Schema zum Sicherstellen, daß die 3D-Pipeline leer ist, bevor 2D-Grundelementdaten zu der Hardware heruntergeladen werden, um gleichartig dazu zum Sicherzustellen, daß die 2D-Pipeline leer ist, bevor 3D-Grundelementdaten heruntergeladen werden. Um dieses Ergebnis zu erreichen, können Registerstatusbits vorgesehen sein, die anzeigen, ob jede der 2D- als auch 3D-Grundelementpipelines leer ist. Jeder Prozeß, der sowohl 2D- als auch 3D-Grundelementdaten verwendet, liest dieses Statusregister um sicherzustellen, daß die Pipelines leer sind, bevor zwischen 2D- und 3D-Grundelementdaten geschaltet wird.

[0182] Obwohl das darstellende Ausführungsbeispiel der Erfindung, das in den Figuren dargestellt ist, einen lokalen Speicher auf der Texturabbildungsplatine umfaßt, der als ein Cache implementiert ist, sollte klar sein, daß die Erfindung nicht darauf beschränkt ist. Das Texturabbildungssystem kann alternativ implementiert werden, so daß der lokale Speicher auf der Texturabbildungsplatine kein Cache ist, und andere Techniken werden verwendet, um sicherzustellen, daß jeder Block von Texturabbildungsdaten, der benötigt wird, um ein Grundelement aufzubereiten, heruntergeladen wird, durch einen Weg, der von der 3D-Grundelementpipeline getrennt ist, bevor das Grundelement aufbereitet wird, so daß die Texturabbildungsdaten von dem lokalen Speicher verfügbar sind, wenn das Grundelement aufbereitet wird.

[0183] Ferner sollte klar sein, daß das Schema der vorliegenden Erfindung zum Erzeugen einer Unterbrechung zu einem Hostcomputer zum Aktualisieren von Datenblöcken in einem lokalen Speicher mit vielen anderen Anwendungen verwendet werden kann und nicht auf die Verwendung in einem Texturabbildungshardwaresystem beschränkt ist. Dieses Schema ist bei jedem Datenverarbeitungssystem sinnvoll, das einen Hostcomputer mit einem Hauptspeicher umfaßt, der Datenblökke speichert, die verarbeitet werden sollen, und einer Datenverarbeitungshardware mit einem lokalen Speicher, der Datenblöcke speichert, die verarbeitet werden.

X. Cacheblockaustauschschema

[0184] Wenn ein Fehlschlag für einen Texturdatenblock auftritt, der nicht in dem Cache ist, lädt der Hostcomputer, wie es oben erörtert wurde, den angeforderten Texturdatenblock zu dem Cache **48** herunter (**Fig.** 2). Falls der Cache voll war, wenn der Fehlschlag auftrat, wird einer der Cacheblöcke durch den neu heruntergeladenen Texturdatenblock ersetzt. Bei einem Ausführungsbeispiel der Erfindung wird eine Bestimmung durchgeführt, welcher Cacheblock am längsten nicht verwendet wurde und dieser Block wird für den Austausch ausgewählt, um aktive Blöcke in dem Cache zu halten. Die Bestimmung, welcher Cacheblock ausgetauscht werden soll, wird durch eine Softwareroutine durchgeführt, die in dem Speicher **17** in dem Hostcomputer **15** gespeichert ist, und arbeitet auf einem Prozessor **19** in dem Hostcomputer. Der Texturabbildungschip **46** umfaßt zwei Registersätze, die die Softwareroutine unterstützen beim Bestimmen, welcher Cacheblock ausgetauscht werden soll. Wenn ein Cachefehlschlag auftritt, werden diese Register durch den Hostcomputer durch den 3D-Umgehungsdatenweg gelesen und beim Bestimmen verwendet, welcher Cacheblock ausgetauscht werden soll.

[0185] Der erste Satz von Registern umfaßt zwei als letztes verwendete 32-Bit-Register MRU0 und MRU1 (zusammen MRU), die jeweils den Bänken 0 und 1 des Caches **48** entsprechen. Jedes Bit in diesen Registern entspricht einem der **32** Cacheblökke, die in der entsprechenden Cachebank enthalten sind. Jedesmal, wenn ein Treffer zu einem Block in dem Cache auftritt, wird das entsprechende Bit in MRUO oder MRU1 gesetzt, so daß die als letztes verwendeten Register Treffer für den Cache sammeln.

[0186] Der zweite Satz von Registern umfaßt zwei derzeit verwendete 32-Bit-Register CU0 und CU1 (zusammen CU), die auch jeweils den Bänken 0 und 1 des Cache entsprechen. Wenn ein Bit entweder in CU0 oder CU1 gesetzt ist, zeigt es an, daß der entsprechende Cacheblock derzeit in einem Miniverzeichnis des Cache ist und nicht ausgetauscht werden sollte. Das Cacheminiverzeichnis ist nachfolgend näher beschrieben.

[0187] Wenn ein Cachefehlschlag auftritt und den Hostcomputer unterbricht, wird die Softwareroutine, die in dem Flußdiagramm von **Fig.** 18 dargestellt ist, durch den Prozessor **19** des Hostcomputers ausgeführt, um zu bestimmen, welcher Cacheblock durch denjenigen ersetzt werden sollte, der die angeforderten Texturdaten enthält, die heruntergeladen werden sollen. Die Softwareroutine behält zwei 64-Bit-Statuswörter bei (d. h. BLOCKS TO USE (BLÖCKE_ZUM_VER-WENDEN) und BLOCKS_ BUSY(BLÖCKE_BESETZT)), die beim Implementieren der Austauschroutine verwendet werden. Jedes der 64 Statusbits in diesen Statuswörtern entspricht einem der 64 Cacheblöcke.

[0188] Wie es bei Schritt 300 gezeigt ist, wird BLOCKS_TO_USE initialisiert, so daß jedes Bit desselben aktiviert ist, was anzeigt, daß jedes anfangs für einen Austausch verfügbar ist. Bei Schritt **302** prüft das Verfahren fortlaufend, um zu bestimmen, ob eine Cachefehlschlagunterbrechung empfangen wurde, und wenn eine erfaßt wird, schreitet das Verfahren zu Schritt **304** fort. Bei Schritt **304** liest das Verfahren die Register MRU und CU durch den 3D-Umgehungsdatenweg. Bei dem Ausführungsbeispiel der Erfindung, bei dem zwei Texturabbildungschips verwendet werden, wie es oben erörtert wurde, behalten die Cachespeicher in den beiden Chips immer die gleichen Texturdaten bei. Falls das System somit zwei Texturabbildungschips umfaßt, werden die Register MRU und CU von beiden gelesen, so daß das Verfahren einen Cacheblock, der an jedem der Texturabbildungschips am längsten nicht verwendet wurde, für den Austausch auswählen kann. Bei Schritt **306** deaktiviert das Verfahren die Bits in BLOCKS_TO_USE, die den Bits entsprechen, die entweder in MRU oder CU aktiviert sind. Bei dem Ausführungsbeispiel, bei dem zwei oder mehr Texturabbildungschips verwendet werden, wird eine logische ODER-Operation der MRUs und CUs verwendet, um zu bestimmen, welche Bits in BLOCKS TO USE deaktiviert sind.

[0189] Bei Schritt 308 wird eine Bestimmung durchgeführt, ob in BLOCKS_TO_USE irgendwelche Bits aktiviert sind, und wenn zumindest eines aktiviert ist, schreitet das Verfahren zu Schritt **310** fort, wobei eine Bestimmung durchgeführt wird, ob die Anzahl aktivierter Bits in BLOCKS_TO_USE unter einem vorbestimmten Schwellenwert liegt. Dieser Schritt wird durchgeführt zum Unterstützen der Beibehaltung einer Historie der Cacheblockverwendung über mehrere Cachefehlschläge und zum Sicherstellen der ordnungsgemäßen Handhabung weiterer Cachefehlschlagunterbrechungen auf die nachfolgend beschriebene Weise. Wenn die Anzahl aktivierter Bits in BLOCKS_BUSY unter dem vorbestimmten Schwellenwert liegt, schreitet das Verfahren zu Schritt **312** fort, wobei alle Bits in den MRUs deaktiviert werden. Als Folge beginnen die MRUs damit, Treffer in dem Cache zu sammeln, die nur nach dem Cachefehlschlag auftreten, der aktuell durch das Verfahren verarbeitet wird. Bei einem Ausführungsbeispiel der Erfindung wird der Schwellenwert bei 11 Bits erstellt, die in BLOCKS_TO_USE aktiviert sind, was anzeigt, daß 11 Cacheblöcke für den Austausch verfügbar sind.

[0190] Nachdem bei Schritt **312** die MRUs entleert sind oder wenn es bei Schritt 310 bestimmt wird, daß die Anzahl aktivierter Bits in BLOCKS_TO_USE nicht unter den vorbestimmten Schwellenwert gefallen ist, schreitet das Verfahren zu Schritt **314** fort, wobei eines der Bits, das in BLOCKS_TO_USE aktiviert wird, für den Austausch mit dem neuen Texturdatenblock ausgewählt wird, der heruntergeladen werden soll. Der Block, der bei Schritt **314** für den Austausch ausgewählt wird, wird durch den neuen Block von Textur-

daten auf eine Weise ersetzt, die nachfolgend in Verbindung mit dem Verfahren von **Fig.** 20 erörtert wird. Nachdem der Block, der ausgetauscht werden soll, bei Schritt **314** ausgewählt wird, kehrt das Verfahren zu Schritt **302** fort, um auf eine weitere Cachefehlschlagunterbrechung zu warten.

[0191] Wenn bei Schritt **308** bestimmt wird, daß in BLOCKS_TO_USE keine Bits aktiviert sind, schreitet das Verfahren zu Schritt 316 fort, wobei BLOCKS_BUSY gleich einem logischen ODER der MRUs und CUs gesetzt ist. Somit entsprechen nur die Bits, die in BLOCKS BUSY aktiviert sind, denjenigen, die in jedem der MRU- oder CU-Register aktiviert sind. Danach wird BLOCKS_TO_USE gleich dem Komplementärwert von BLOCKS_BUSY gesetzt. Auf diese Weise ist jedes Bit in BLOCKS_TO_USE aktiviert, außer denjenigen, die den Bits entsprechen, die in den MRUs und CUs aktiviert sind, was anzeigt, daß diese Blöcke nicht für einen Austausch ausgewählt werden sollten.

[0192] Nachdem BLOCKS_TO_USE bei Schritt **316** gleich dem Komplementärwert von BLOCKS BUSY gesetzt wurde, schreitet das Verfahren zu Schritt **318** fort, wobei eine Bestimmung durchgeführt wird, ob in BLOCKS_TO_USE irgendwelche Bits aktiviert sind. Wenn zumindest ein Bit in BLOCKS_TO_USE aktiviert ist, schreitet das Verfahren zu Schritt **310**–**314** fort, wobei die MRU gelöscht werden, falls die Anzahl aktivierter Bits in BLOCKS_TO_USE unter den Löschschwellenwert gefallen ist, und eines der aktivierten Bits in BLOCKS_TO_USE für einen Austausch auf die oben beschriebene Weise ausgewählt wird.

[0193] Wenn bei Schritt **318** bestimmt wird, daß in BLOCKS_TO_USE keine Bits aktiviert sind, schreitet das Verfahren zu Schritt **320** fort, wobei drei Aktionen durchgeführt werden. Zunächst werden die MRU gelöscht, weil die Anzahl von Bits, die in BLOCKS_TO_USE aktiviert werden, notwendigerweise unter den vorbestimmten Schwellenwert gefallen ist. Nachfolgend wird BLOCKS_BUSY gleich den CU-Registern gesetzt. Wie es oben angemerkt wurde, zeigt jedes CU-Register die Cacheblöcke an, die derzeit in dem entsprechenden Cacheminiverzeichnis beibehalten werden, und daher nicht ausgetauscht werden sollten. Wenn zwei oder mehr Texturabbildungschips verwendet werden, wird BLOCKS BUSY gleich dem logischen ODER der CU-Register gesetzt. Schließlich wird BLOCKS_TO_USE gleich dem Komplementärwert von BLOCKS_BUSY gesetzt. Als Folge wird jedes Bit von BLOCKS_TO_USE aktiviert, außer denjenigen, die den Datenblöcken entsprechen, die aktuell in dem Cacheminiverzeichnis von einem der Texturabbildungschips beibehalten werden. Das Verfahren schreitet dann zu Schritt **314** fort, wobei eines der zugewiesenen Bits in BLOCKS_TO_USE für den Austausch ausgewählt wird. Auf diese Weise kann jeder der Blöcke in dem Cache, außer denjenigen in dem Miniverzeichnis, für einen Austausch ausgewählt werden.

[0194] Das in **Fig.** 18 gezeigte Ausführungsbeispiel der vorliegenden Erfindung verwendet ein Austauschschema, das einen am längsten nicht verwendeten Cacheblock ausgetauscht, wenn ein Cachefehlschlag auftritt. Es sollte klar sein, daß verschiedene Modifikationen an diesem Schema durchgeführt werden können, ohne von dem Schutzbereich der vorliegenden Erfindung abzuweichen. Beispielsweise wird bei dem in **Fig.** 18 gezeigten Ausführungsbeispiel das MRU-Hardwareregister verwendet, um Treffer in dem Cache über eine Zeitperiode zu sammeln, die potentiell mehrere Cachefehlschläge umfassen kann, und das MRU-Register wird nur entleert, sobald die Anzahl von Bits, die in BLOCKS_TO_USE aktiviert sind, unter den vorbestimmten Schwellenwert gefallen ist. Außerdem wird das Softwarestatuswort BLOCKS_BUSY nur bei Schritt **316** oder **320** aktualisiert, wenn bestimmt wird, daß in BLOCKS_TO_USE keine Bits aktiviert sind. Das Austauschschema kann alternativ durch Aktualisieren von BLOCKS_BUSY von dem MRU-Register implementiert werden, jedesmal, wenn ein Cache eine Cachefehlschlagunterbrechung empfangen wird und dann das MRU-Register entleert wird. Auf diese Weise kann das Softwarestatuswort BLOCKS_BUSY verwendet werden, um die Historie von Treffern in dem Cache über eine Zeitperiode zu sammeln, die potentiell mehrere Cachefehlschläge umfassen kann, und das Hardwareregister MRU kann verwendet werden, um nur Treffer zwischen Fehlschlägen zu sammeln.

[0195] Obwohl der Schwellenwert von aktivierten Bits in BLOCKS_TO_USE, der zum Löschen der MRU führt, bei dem oben beschriebenen darstellenden Ausführungsbeispiel bei 11 verfügbaren Blöcken eingestellt ist, sollte ferner klar sein, daß diese Nummer offensichtlich geändert werden kann. Dieser Schwellenwert beeinträchtigt die Anzahl von Malen, die die Routine bei Schritt 308 auf eine Situation trifft, bei der keines der Bits in BLOCKS_TO_USE aktiviert ist. Es ist wünschenswert, diese Situation zu vermeiden, weil dieselbe zum Aktualisieren von BLOCKS_TO_USE führt (bei Schritt **316** oder **320**), mit nur der aktuellsten Historie der Cacheblockverwendung, d. h. der Historie nach dem vorher verarbeiteten Cachefehlschlag. Es wird bevorzugt, einen höheren Auflösungsgrad zu liefern, so daß die Bits, die in BLOCKS_TO_USE aktiviert werden, Blöcke reflektieren, die durch das Verarbeiten mehrerer Cachefehlschläge nicht verwendet wurden, falls es solche Blöcke gibt. Somit kann durch Steuern des Schwellenwerts aktivierter Bits in BLOCKS_TO_USE, das zum Löschen der MRU führt, die Anzahl von Durchläufen durch das Verfahren, bei dem keines der Bits von BLOCKS_TO_USE bei Schritt 308 aktiviert wird, minimiert werden, was einen erwünschten Auflösungspegel beim Bestimmen eines am längsten nicht verwendeten Cacheblocks liefert.

[0196] Es sollte klar sein, daß das oben beschriebene Blockaustauschschema, das durch eine Softwareroutine implementiert wird, die auf einem Hostcomputer ausgeführt wird, nicht auf die Verwendung mit einem Cachespeicher begrenzt ist. Diese Austauschroutine kann bei jedem Datenverarbeitungssystem verwendet werden, bei dem ein lokaler Speicher Datenblöcke umfaßt, die verarbeitet werden, und bei dem Datenblöcke in dem lokalen Speicher ausgetauscht werden, wenn zusätzliche Datenblöcke von einem Hostcomputer zu dem lokalen Speicher heruntergeladen werden.

XII. Cache-Deaktivieren-Operation

[0197] Bei einem Ausführungsbeispiel der Erfindung ist eine Fähigkeit vorgesehen, die Cacheoperation des lokalen Speichers auf der Texturabbildungsplatine zu deaktivieren, durch Deaktivieren von Cachefehlschlägen, so daß Texturdaten für jedes 3D-Grundelement in den Speicher **48** heruntergeladen werden, bevor dieselben während dem Aufbereiten des Grundelements erforderlich sind. Jeder Texturabbildungschip **46** umfaßt ein Statusbit, das anzeigt, daß die Operation des lokalen Speichers als ein Cache aktiviert ist. Wenn dieses Statusbit aktiviert ist, führen Cachefehlschläge zu einer Unterbrechung des Hostcomputers und zum Anhalten des Texturabbildungschips. Wenn das Statusbit deaktiviert ist, arbeitet der lokale Speicher **48** auf der Texturabbildungsplatine jedoch nicht als ein Cache, und die Texturdaten für jedes Grundelement werden in den Speicher **48** heruntergeladen, bevor dieselben durch das Grundelement benötigt werden, so daß keine Fehlschläge in dem Speicher auftreten. Wenn bei einem Ausführungsbeispiel der Erfindung der Betrieb des lokalen Speichers als ein Cache deaktiviert ist, werden Texturdaten durch die 3D-Grundelementpipeline auf den lokalen Speicher auf der Texturabbildungsplatine heruntergeladen, um die Synchronisation der Texturdaten mit den entsprechenden 3D-Grundelementdaten zu ermöglichen.

XII. Texeltorregister, die das Schema zum Herunterladen von Texturdaten ansprechend auf einen Cachefehlschlag unterstützen

[0198] Wie es oben erörtert wurde, umfaßt der Texturabbildungschip **46** (**Fig.** 2) ein Texeltor **92** (**Fig.** 3), das verwendet wird, um Texturdaten zu empfangen, die von dem Hostcomputer **15** heruntergeladen werden. Das Texeltor umfaßt eine Anzahl von Registern, die das Herunterladen von Texturdaten unterstützen. Einige dieser Register wurden oben erörtert, einschließlich den Registern MRU und CU. Die anderen Texeltorregister umfassen ein Befehlsregister, ein Statusregister, ein Texeldatenregister, ein Verzeichnistagregister, ein Cacheadreßregister und ein Rohretikett- bzw. Pipetagregister, von denen jedes nachfolgend erörterte Funktionen durchführt.

[0199] Auf die Texeltorregister wird ein Zugriff geliefert, um es denselben zu ermöglichen, durch die 3D-Grundelementpipeline beschrieben zu werden. Selbst wenn die 3D-Pipeline besetzt ist, können die Texeltorregister beschrieben werden, wobei die Daten zum Schreiben der Register einfach in die Pipeline plaziert werden. Ferner kann auch durch die 3D-Pipelineumgehung auf die Texeltorregister zugegriffen werden, die über den 24-Bit-Bus **24** (**Fig.** 2) geliefert wird. Wenn auf die Texeltorregister zugegriffen wird, werden acht Bits des Busses **24** als eine Regist eradresse verwendet, um zu spezifizieren, welches Texeltorregister gelesen oder geschrieben werden soll, und wenn Daten auf ein Texeltorregister geschrieben werden, liefern die anderen 16 Bits des Busses die Daten.

[0200] Der Aufbau der Texeltorregister ist in **Fig.** 19 gezeigt. Bei einem Ausführungsbeispiel der Erfindung umfaßt jedes der Texeltorregister **32** Bits, selbst wenn eine Anzahl der Bits in einigen der Register nicht verwendet wird.

A. Texelbefehlsregister

[0201] Das Texelbefehlsregister umfaßt eine Anzahl von Bits, die durch die Hostcomputersoftwareroutine verwendet wird, die Cachefehlschläge bedient, und die nachfolgend näher erörtert wird. Ein Haltebit **350** wird durch die Softwareunterbrechungshandhabungsroutine gesetzt und weist den Texturabbildungschip an, seinen Betrieb anzuhalten. Wie es oben angemerkt ist, werden bei dem Ausführungsbeispiel der Erfindung, bei dem zwei Texturabbildungschips vorgesehen sind, beide Texturabbildungschips mit den gleichen Texturdaten aktualisiert, ansprechend auf einen Cachefehlschlag von einem derselben, so daß die Caches übereinstimmend bleiben. Wenn somit ein Fehlschlag von einem der Texturabbildungschips empfangen wird, werden beide durch Setzen des Haltebits **350** in ihren jeweilige Texelbefehlsregister angehalten. Das Haltebit wird durch die Softwareroutine gelöscht, die den Cachefehlschlag handhabt, durch Schreiben auf das Befehlsregister zum Löschen des Bits, nachdem neue Texturdaten von dem Hostcomputer heruntergeladen wurden, ansprechend auf den Cachefehlschlag.

[0202] Ein Unterbrechungsfreigabebit **352** gibt, wenn es aktiviert ist, Unterbrechungen von dem Texeltor frei, wenn ein Cachefehlschlag auftritt. Dieses Bit ist deaktiviert, um die oben beschriebene Fähigkeit zu liefern, daß der lokale Speicher **48** auf der Texturabbildungsplatine **12** (**Fig.** 2) nicht als ein Cache arbeitet.

[0203] Die Schreib-Loki0- und Schreib-Loki1-Bits **354** und **356** sind Schreibfreigabesignale für die Texeltorregister. Loki ist ein Abkürzungsname, der verwendet wird, um den Texturabbildungschip **46** zu identifizieren. Bei dem Ausführungsbeispiel der Erfindung, wo zwei solche Chips verwendet werden, werden die Chips als Loki0 bzw. Loki1 bezeichnet. Wenn

nur ein einziger Texturabbildungschip verwendet wird, wird er als Loki0 bezeichnet. Wenn über den Texeltorbus **24** ein Befehl empfangen wird, auf eines der Texelbefehlsregister zu schreiben, prüft jeder Texturabbildungschip (d. h. Loki0 und Loki1) sein Befehlregister, um zu bestimmen, ob sein Schreibbit freigegeben ist, und falls dies der Fall ist, aktualisiert derselbe seine Texeltorregister gemäß dem empfangenen Schreibbefehl. Somit kann eine Softwareroutine, die auf dem Hostcomputer arbeitet, durch Steuern der Werte der Schreib-Loki0- und Schreib-Loki1-Bits **354** und **356** entweder getrennt oder kombiniert auf die Texeltorregister in den beiden Texturabbildungschips schreiben.

[0204] Das Loki-Lesebit **358** gibt Lesevorgänge des Texeltorregisters von einem der Texturabbildungschips frei. Wenn über den Texelbus **24** ein Befehl empfangen wird, ein Texeltorregister zu lesen, spricht nur einer der Texturabbildungschips zu einem Zeitpunkt an, um die Inhalte seines Texeltorregisters auf dem Bus bereitzustellen. Bei dem Ausführungsbeispiel, bei dem zwei Texturabbildungschips vorgesehen sind, kann jedes mit einem Stift versehen sein, der fest verdrahtet ist, um anzuzeigen, ob der Chip Loki0 oder Loki1 ist. Wenn das Loki-Lesebit durch Software gesetzt ist, zeigt es an, daß Lesevorgänge von Loki1 freigegeben sind, und wenn das Lesebit deaktiviert ist, zeigt es an, daß Lesevorgänge für Loki0 aktiviert sind. Aus dem Vorhergehenden sollte klar sein, daß es das Format der Texelbefehlsregister ermöglicht, daß dasselbe auf beide Texturabbildungschips (Loki0 und Loki1) gleichzeitig geschrieben wird, mit den gleichen Daten, wodurch nur ein einziger Schreibzyklus erforderlich ist, um beide Befehlsregister zu beschreiben.

B. Texelstatusregister

[0205] Das Texeltorstatusregister umfaßt ein Dual-Loki-Bit **360**, das, wenn es aktiviert ist, anzeigt, daß das System zwei Texturabbildungschips umfaßt. Ein Unterbrechungsfreigabebit **362** ist jedesmal aktiviert, wenn das Bit **352** in dem Befehlsregister aktiviert ist, und zeigt an, daß der lokale Speicher in dem Texturabbildungschip als ein Cache arbeitet, der Fehlschläge erzeugt, die den Hostcomputer unterbrechen, wenn Texturdaten benötigt werden, die nicht in dem Cache sind. Dieses Bit ist in dem Statusregister enthalten und auch in dem Befehlsregister, so daß der Status des Texeltors durch einfaches Lesen des Statusregisters gelesen werden kann.

[0206] Ein Unterbrechung-Gültig-Bit **364** ist aktiviert, wenn eine Unterbrechung von dem Texturabbildungschip aufgetreten ist und der Chip darauf wartet, daß neue Texturdaten heruntergeladen werden. Dieses Bit wird gelöscht, wenn das Cacheverzeichnistagregister (nachfolgend erörtert) mit einem Cachetag beschrieben wird, das mit dem Cachelesetag übereinstimmt, das in dem Pipetagregister (nachfolgend erörtert) gespeichert ist, welches das Tag ist, das in dem Cache fehlgeschlagen hat.

[0207] Das Statusregister umfaßt zwei Bits, die das Anhalten des Texturabbildungschips unterstützen, wenn ein Cachefehlschlag auftritt. Das Haltefreigabebit **368** wird durch die Softwareroutine, die auf dem Hostcomputer arbeitet, jedesmal gesetzt und gelöscht, wenn das Haltebit **350** jeweils in dem Befehlsregister gesetzt und gelöscht ist, und weist den Texturabbildungschip an, sich selbst anzuhalten, wenn das Bit aktiviert ist. Dieses Bit ist in dem Statusregister und auch in dem Befehlsregister vorgesehen, so daß der Status des Texturabbildungschips in einem einzigen Register gespeichert ist. Das Unterbrechung-Gültig-Bit **368** wird durch Hardware in dem Texturabbildungschip gesetzt, wenn ein Cachefehlschlag aufgetreten ist und das Cacheverzeichnis darauf wartet, daß Daten heruntergeladen werden. Dieses Bit wird gelöscht, wenn das Cacheverzeichnistagregister (nachfolgend erörtert) mit einem Cachetag beschrieben wird, das mit dem Blocktag übereinstimmt, das in dem Cache fehlgeschlagen hat.

C. Pipetagregister

[0208] Das Pipetagregister speichert das letzte Blocktag, das durch die Pipeline in dem Texturabbildungschip indexiert wurde. Wenn ein Cachefehlschlag auftritt, speichert das Pipetagregister das Blocktag **370**, das in dem Cache fehlgeschlagen hat. Somit kann durch Lesen des Pipetagregisters über den Texeltorbus **24** die Software, die auf die Cachefehlschlagunterbrechung anspricht, das Tag für den Cacheblock bestimmen, das ansprechend auf den Fehlschlag auf den Cache heruntergeladen werden sollte.

D. Texeldatenregister

[0209] Das Texeldatenregister wird verwendet, um Texturdaten zu dem Cache **48** herunterzuladen, wenn ein Cachefehlschlag auftritt. Wie es oben angemerkt wurde, wird jedes Texel durch 32 Datenbits dargestellt, wobei ein Byte **372** Alpha darstellt, ein Byte **364** den Rotwert darstellt, ein Byte **376** den Grünwert darstellt und ein Byte **378** den Blauwert darstellt.

E. Texelcacheadreßregister

[0210] Das Texelcacheadreßregister wird verwendet, um Texeldaten in den Cache zu schreiben und Blocktags in das Cacheverzeichnis. Wie es oben erörtert wurde, speichert der Cache **64** Texturdatenblöcke, wobei jeder Block ein Array von 256 × 256 Texeln umfaßt. Das Texelcacheadreßregister umfaßt ein 6-Bit-Blockindexfeld **380**, das den bestimmten der 64 Blöcke in dem Cache identifiziert, der gelesen oder geschrieben werden soll. Außerdem umfaßt das Register ein 16-Bit-Blockadreßfeld **382**, das die bestimmte Texeladresse identifiziert, die in dem Block

gelesen oder geschrieben wird, der in dem Blockindexfeld identifiziert ist. Wenn Daten ansprechend auf einen Cachefehlschlag zu dem Texturspeicher heruntergeladen werden, wird der Blockindex durch die Softwareroutine gesetzt, das das Am-Längsten-Nicht-Verwendet-Austauschschema verwendet, das oben erörtert wurde, und das Blockadreßfeld **382** wird auf Nullen initialisiert, um das erste Texel in dem Block zu schreiben. Das Cacheadreßregister inkrementiert das Blockadreßfeld **382**, jedesmal automatisch, wenn auf das Texeldatenregister zugegriffen wird. Somit kann das Blockadreßfeld durch alle Blockadressen in dem Cacheblock inkrementiert werden, um den neuen Texeldatenblock in den Cache zu schreiben.

F. Texelverzeichnistagregister

[0211] Das Texelverzeichnistagregister umfaßt ein 23-Bit-Blocktagfeld **384**, das das Cacheblocktag darstellt, und verwendet wird, um den Cacheverzeichniseintrag, der durch das Blockindexfeld **380** definiert ist, in das Cacheadreßregister zu schreiben. Wie es oben erörtert wurde, stellen die 23 Bits des Cacheblocktags acht Bits von Textur-ID, sieben Bits von S-Koordinaten, sieben Bits von T-Koordinaten und ein zusätzliches Bit dar, das die Abbildungsnummer in der Reihe von MIP-Abbildungen der Abbildung identifiziert, die durch den Texturdatenblock dargestellt wird, der dem Blocktag entspricht. Wenn ansprechend auf einen Cachefehlschlag ein neuer Texturdatenblock von dem Hostcomputer heruntergeladen wird, wird sein Blocktag über den Texelbus **24** in das Verzeichnistagregister geladen. Von dem Verzeichnistagregister wird das Blocktag in das Cacheverzeichnis geschrieben, in dem Eintrag, der durch das Blockindexfeld **380** des Cacheadreßregisters identifiziert ist. Wie es oben angemerkt wurde, wenn ein Blocktag in das Verzeichnistagregister geschrieben wird, das mit dem Tag in dem Pipetagregister übereinstimmt (welches dasjenige ist, dessen Lesevorgang zu einem Cachefehlschlag geführt hat), wird die Cachefehlschlagunterbrechung gelöscht.

XIII. Softwareroutine zum Warten von Cachefehlschla unterbrechungen

[0212] Wie es aus dem Vorhergehenden offensichtlich ist, werden die Texeltorregister durch eine Softwareroutine verwendet, die auf dem Hostcomputer **15** arbeitet, die Cachefehlschlagunterbrechungen wartet, um die notwendigen Texturdaten herunterzuladen. Ein Flußdiagramm dieser Softwareroutine ist in **Fig.** 20 gezeigt. Bei Schritt **400** wird das Texelbefehlsregister sowohl für Loki0 als auch Loki1 geschrieben, um das Haltebit **350** in beiden zu setzen. Das Verfahren schreitet dann zu Schritt **402** fort, um das angehaltene Bit **368** in den Texelstatusregistern zu lesen, um zu bestimmen, ob beide Lokis angehalten wurden. Das Verfahren liest fortlaufend die Statusregister von Loki0 und Loki1, bis bestimmt wird, daß beide angehalten wurden, und schreitet dann zu Schritt 404 fort. Wenn das System nur einen einzigen Texturabbildungschip **46** umfaßt (d. h. Loki0), spricht Loki0 auch auf Anforderungen an, zum Lesen der Texeltorregister von Loki1, durch Bereitstellen der Inhalte seiner Register auf dem Texelbus **24**. Wenn somit die Softwareroutine bei Schritt **402** prüft, um zu bestimmen, ob beide Lokis angehalten haben, spricht Loki0 auf die Lesevorgänge von Loki1 an, so daß, wenn Loki0 angehalten wurde, das Verfahren zu Schritt **404** fortschreitet.

[0213] Bei Schritt **404** wird das Unterbrechung-Gültig-Bit **364** in dem Texelstatusregister von Loki0 gelesen, um zu bestimmen, ob Loki0 unterbrochen hat, um den Cachefehlschlag zu bewirken, und wenn dasselbe das getan hat, schreitet das Verfahren zu Schritt **406** fort, indem das Pipetagregister von Loki0 gelesen wird, um das Blocktag des Texturdatenblocks zu identifizieren, das in dem Cache fehlgeschlagen hat. Die Softwareroutine verwendet dieses Blocktag zum Zugreifen auf den entsprechenden Texturdatenblock in dem Speicher **17** (**Fig.** 2) des Rostcomputers und schreitet zu Schritt **408** fort, um zu bestimmen, welcher Block in dem Cache durch den neuen Texturdatenblock ausgetauscht werden sollte, der heruntergeladen werden soll. Diese Bestimmung wird unter Verwendung des Am-Längsten-Nicht-Verwendet-Schemas durchgeführt, das oben in Verbindung ist **Fig.** 18 beschrieben wurde.

[0214] Wenn das System, wie es oben angemerkt wurde, zwei Texturabbildungschips umfaßt, werden die Caches in jedem derselben beibehalten, um identische Einträge aufzuweisen. Daher werden Texturdaten, die von dem Hostcomputer heruntergeladen werden, ansprechend auf einen Cachefehlschlag von einem der Texturabbildungschips, auf die Caches in beiden Chips geschrieben. Sobald somit der Cacheblock, der ausgetauscht werden soll, identifiziert wurde, schreitet das Verfahren zu Schritt **410** fort, in dem das Cacheadreßregister in Loki0 und Loki1 (falls Loki1 existiert) mit dem Blockindex beschrieben wird, der während Schritt **408** bestimmt wird. Bei Schritt **412** wird das Verzeichnistagregister mit dem Blocktag des Texturdatenblocks geschrieben, der ansprechend auf den Cachefehlschlag auf den Texturcache heruntergeladen werden soll, und bei Schritt **414** werden die Texturdaten in das Texeldatenregister geschrieben. Auf diese Weise spricht das Verfahren auf den Cachefehlschlag an, durch Herunterladen des Texturdatenblocks, der in dem Cache fehlgeschlagen hat, und durch Schreiben dieses Datenblocks in den Cache.

[0215] Nachdem der Texturdatenblock in den Schritten **406**–**414** zu Loki0 und Loki1 heruntergeladen wird, oder falls bei Schritt **404** bestimmt wird, daß Loki1 nicht unterbrochen wurde, schreitet das Verfahren zu Schritt **416** fort, in dem eine Bestimmung durchgeführt wird, ob das Unterbrechung-Gültig-Bit **364** in dem Loki1-Statusregister gesetzt wurde, was

anzeigt, daß in Loki1 ein Cachefehlschlag aufgetreten ist. Falls das System nur einen einzigen Texturabbildungschip umfaßt, wie es oben erörtert wurde, spricht Loki0 auf Lesevorgänge der Loki1-Texeltorregister an. Wenn Loki0 auf einen Lesevorgang des Statusregisters von Loki1 anspricht, maskiert es das Unterbrechung-Gültig-Bit **364**, so daß die Softwareroutine bei Schritt **416** bestimmt, daß Loki1 nicht unterbrochen wurde. Diese Maskierung wird durchgeführt, damit die Softwareroutine die Unterbrechung von Loki1 nicht neu verarbeitet, indem es erneut den Texturdatenblock herunterlädt, der in den Schritten **406**–**414** heruntergeladen wurde. Daher wird das Verfahren bei einem System, in dem nur ein einziger Texturabbildungschip vorgesehen ist, bei Schritt **416** bestimmen, daß Loki1 nicht unterbrochen wurde, und wird bei Schritt **418** fortfahren, bei dem das Befehlsregister in Loki0 geschrieben wird, um das Haltebit **350** zu deaktivieren, wodurch der Texturabbildungschip freigegeben wird zum Fortsetzen der Verarbeitung der Grundelemente in der Pipeline desselben.

[0216] Wenn das System zwei Texturabbildungschips umfaßt, wird das Verfahren bei Schritt **416** bestimmen, ob Loki1 unterbrochen wurde, und falls es dies nicht wurde, schreitet es ebenfalls direkt zu Schritt **418** fort, bei dem das Haltebit in beiden Texturabbildungschips deaktiviert wird und es denselben ermöglicht, mit dem Verarbeiten von Grundelementen fortzufahren. Wenn jedoch bei Schritt **416** bestimmt wird, daß Loki1 ansprechend auf einen Cachefehlschlag unterbrochen wurde, schreitet das Verfahren durch die Schritte **420** –**424** fort, um die Unterbrechung auf die gleiche Weise zu verarbeiten, wie es in Verbindung mit den Schritten **406**– **414** zum Handhaben der Unterbrechung von Loki0 erörtert wurde. Das Verfahren schreitet dann zu Schritt **418** fort, bei dem die Haltebits in beiden Texturabbildungschips deaktiviert werden.

[0217] Es sollte klar sein, daß bei einem System, bei dem zwei Texturabbildungschips vorgesehen sind, beide Chips gleichzeitig eine Cachefehlschlagunterbrechung für das gleiche Blocktag oder für unterschiedliche Blocktags erzeugen können. Wenn beide Texturabbildungschips Cachefehlschlagunterbrechungen für das gleiche Blocktag erzeugen, wird die Unterbrechung in den Schritten **400**–**414** verarbeitet. Daher wird das Verfahren bei Schritt **416** keine Unterbrechung von Loki1 erfassen, weil die Unterbrechung von Loki1 durch das Schreiben des fehlgeschlagenen Blocktags in das Verzeichnistagregister beider Lokis bei Schritt **412** gelöscht wird. Somit ist das in **Fig.** 20 gezeigte Verfahren in der Lage, auf eine Unterbrechung von jedem der Texturabbildungschips einzeln oder von beiden gleichzeitig anzusprechen.

[0218] XIV. Cacheminiverzeichnis und Hauptverzeichnis Wie es oben erörtert wurde, umfaßt der Cache bei einem Ausführungsbeispiel der Erfindung **64** Blöcke von 256 × 256 Texeln an Daten und ein vollassoziatives Cacheverzeichnis, das 64 Einträge von 23-Bit-Blocktags umfaßt. Wenn die vorliegende Erfindung in einem trilinearen Interpolationsmodus arbeitet, werden acht Texellesevorgänge durchgeführt, um die resultierenden Texeldaten für ein Pixel zu bestimmen, wobei vier Texel in einer Abbildung gleichzeitig in einer Leseoperation gelesen werden und vier Texel in der anderen Abbildung gleichzeitig in einer zweiten Leseoperation gelesen werden. Falls das Pixel, das bearbeitet wird, auf eine Position in einer Abbildung abbildet, die benachbart zu einer Cacheblockgrenze ist, können die vier Texel, die von dem Cache gelesen werden, um die resultierenden Texeldaten in einer Abbildung zu erzeugen, jeweils in einem anderen Cacheblock sein. Somit könnte das gleichzeitige Lesen von vier Texeln von dem Cache für jedes Pixel vier getrennte Vergleiche mit den 64 Blocktageinträgen in dem Cacheverzeichnis erfordern.

[0219] Herkömmliche vollassoziative Caches arbeiten auf eine von zwei Möglichkeiten. Erstens stellen einige getrennte Hardwarekomparatoren für jeden Cachetageintrag bereit, so daß ein Lesetag mit jedem Cachetageintrag in einem einzigen Zyklus verglichen werden kann. Eine solche Technik würde bei der vorliegenden Erfindung, bei der vier Lesevorgänge gleichzeitig durchgeführt werden, hohe Hardwarekosten nach sich ziehen, und würde 265 (d. h. 4 × 64) 23-Bit-Komparatoren erfordern. Eine zweite Technik, die durch herkömmliche vollassoziative Caches verwendet wird, verwendet einen einzelnen Cachetagkomparator, und jeder Cacheeintrag wird seriell mit dem Lesetag verglichen. Eine solche Technik würde die Systembandbreite bei der vorliegenden Erfindung negativ beeinträchtigen, bei der möglicherweise 256 Lesezyklen des Cacheverzeichnisses erforderlich wären, um zu bestimmen, ob jedes der vier Texel, die während einer einzigen Leseoperation gelesen werden, in dem Cache vorliegt.

[0220] Um diese Probleme zu überwinden, umfaßt das Cachesystem der vorliegenden Erfindung sowohl ein Miniverzeichnis (**Fig.** 21) als auch ein Hauptverzeichnis (**Fig.** 22). Das Miniverzeichnis ist vollassoziativ und umfaßt die fünf als letztes gelesenen Cacheblocktags und auch einen entsprechenden Blockindex für jeden. Wie es in **Fig.** 21 gezeigt ist, umfaßt das Miniverzeichnis **500** fünf Einträge, die jeweils über die Ausgänge **501**–**505** von dem Miniverzeichnis ausgegeben werden, von denen jedes mit vier Gruppen von Tagkomparatoren **507**–**510** gekoppelt ist. Jede Gruppe von Tagkomparatoren **507**–**510** umfaßt fünf 23-Bit-Komparatoren (nicht gezeigt) und entspricht einem der vier Cachelesetags, die in einer einzelnen Leseoperation durchgeführt werden, wenn eine bilineare oder trilineare Interpolation durchgeführt wird. Somit ist die vollassoziative Art des Miniverzeichnisses mit 20 23-Bit-Komparatoren implementiert, gleich der Anzahl von Tags, die gleichzeitig gelesen wird, multipliziert mit der Anzahl von Einträgen in dem Miniverzeichnis.

[0221] Die vier Cachelesetags, die gleichzeitig für ein Pixel gelesen werden, identifizieren die Cacheblocks, die die vier Texel umfassen, die am nächsten

zu der Position in der Abbildung sind, auf die das Pixel abbildet, und werden als ein oberes linkes (UL = upper left) Tag, ein oberes rechtes (UR = upper right) Tag, ein unteres linkes (LL = lower left) Tag und ein unteres rechtes (LR = lower right) Tag. Die Cachelesetags für das obere linke, obere rechte, untere linke und untere rechte Texel sind jeweils mit Gruppen von oberen linken, oberen rechten, unteren linken und unteren rechten Tagkomparatoren **507**–**510** verbunden. Jede Gruppe von Tagkomparatoren **507**–**510** vergleicht ihr entsprechendes Cachelesetag mit den fünf Blocktags, die in dem Miniverzeichnis gespeichert sind, und erzeugt eine Trefferausgabe, die anzeigt, ob das Tag mit einem der Miniverzeichniseinträge übereinstimmt, und wenn es dies tut, gibt dasselbe auch einen Blockindex aus, der die Positionen im Cache anzeigt, in dem der entsprechende Texeldatenblock gespeichert ist.

[0222] Wie es aus dem Vorhergehenden ersichtlich ist, falls jedes der vier Cachelesetags (UL, UR, LL, LR) in dem Miniverzeichnis ist, ist nur ein einzelner Verzeichniszugriff erforderlich, um die Blockindizes zu bestimmen, die die Positionen in dem Cache identifizieren, in denen die entsprechenden vier Texeldatenblöcke gespeichert sind. Auf das Hauptcacheverzeichnis wird nur zugegriffen, falls ein oder mehrere der Lesetags nicht in dem Miniverzeichnis sind. Das Miniverzeichnis **500** wird jedesmal aktualisiert, wenn ein Lesetag in dem Miniverzeichnis fehlschlägt, so daß das Miniverzeichnis **500** immer die Blocktags der fünf als letztes zugegriffenen Texturdatenblöcke umfaßt.

[0223] Falls eines oder mehrere der vier Cachelesetags in dem Miniverzeichnis nicht trifft, wird auf das Hauptcacheverzeichnis **520** zugegriffen (**Fig.** 22). Wie es oben angemerkt wurde, umfaßt das Hauptverzeichnis **64** Einträge, wobei jeder ein Blocktag umfaßt. Das Hauptverzeichnis ist mit 64 23-Bit-Komparatoren **522** versehen, so daß ein Cachelesetag in einem einzigen Zyklus mit dem gesamten Hauptverzeichnis verglichen werden kann. Die Komparatoren **522** liefern ein Signal, das anzeigt, ob das Cachelesetag einen der Einträge in dem Hauptverzeichnis getroffen hat, und wenn es dies getan hat, wird die Position des Komparators, die mit dem Lesetag übereinstimmt, auch verwendet, um einen Blockindex zu erzeugen, der identifiziert, wo sich der entsprechende Texeldatenblock in dem Cache befindet. Falls das Lesetag mit keinem der Einträge in dem Hauptcacheverzeichnis übereinstimmt, wird ein Cachefehlschlag erzeugt, der bewirkt, daß der Hostcomputer unterbrochen wird, um den angeforderten Texturdatenblock auf die oben beschriebene Weise herunterzuladen.

[0224] Wie es oben angemerkt wurde, wird auf das Hauptcacheverzeichnis **520** nur zugegriffen, wenn eines oder mehrere der vier Cachelesetags (UL, UR, LL, LR) das Miniverzeichnis nicht trifft. Falls zwei oder mehr der Cachelesetags in dem Miniverzeichnis fehlschlagen, ist es wünschenswert, die Leistungsstrafe zu reduzieren, die auftreten würde, falls für jedes Cachelesetag in getrennten Zyklen auf das Hauptverzeichnis zugegriffen werden müßte. Um dieses Ergebnis zu erzielen, ist bei einem Ausführungsbeispiel der Erfindung eine Gruppe von sechs zusätzlichen Komparatoren **526**–**530** vorgesehen, wie es in **Fig.** 23 gezeigt ist. Die sechs Komparatoren vergleichen jedes der vier Cachelesetags, auf die gleichzeitig zugegriffen wird, mit den anderen, um zu bestimmen, ob irgendwelche identisch sind. Die Komparatoren umfassen den Komparator **526**, der das UL-Tag mit dem UR-Tag vergleicht, den Komparator **527**, der das UL- und das LL-Tag vergleicht, den Komparator **528**, der das UL- und das LR-Tag vergleicht, den Komparator **529**, der das UR- und das LL-Tag vergleicht, den Komparator **530**, der das UR- und das LR-Tag vergleicht, und den Komparator **532**, der das LL- und das LR-Tag vergleicht.

[0225] Die Vergleiche, die durch die Komparatoren **526**–**532** durchgeführt werden, können parallel mit anderen Vergleichen durchgeführt werden, um keine Leistungsstrafe zu bewirken. Beispielsweise können diese Vergleiche während dem Zyklus durchgeführt werden, wenn die Cachelesetags mit dem Miniverzeichnis verglichen werden, oder während dem Zyklus, wenn ein erstes Cachelesetag, das in dem Miniverzeichnis fehlgeschlagen hat, mit dem Hauptverzeichnis verglichen wird. Wenn bestimmt wird, daß zumindest zwei Cachelesetags das Hauptverzeichnis nicht treffen und gleich sind, werden die Ausgaben der Komparatoren **526**–**532** verwendet, um anzuzeigen, daß für diese zumindest zwei Cachelesetags nur einmal auf das Hauptverzeichnis zugegriffen werden muß. Auf diese Weise müssen beim Zugreifen auf das Hauptverzeichnis für Tags, die identisch sind, nicht mehrere Zyklen verwendet werden, wodurch die Beeinträchtigung der Systembandbreite minimiert wird, wenn zwei oder mehr Cachelesetags in dem Miniverzeichnis fehlschlagen.

[0226] Wie es aus dem Vorhergehenden ersichtlich ist, gleicht das Ausführungsbeispiel der vorliegenden Erfindung, das das Cacheminiverzeichnis verwendet, die widersprüchlichen Ziele der Verwendung einer relativ kleinen Hardwaremenge zum Implementieren des Cacheverzeichnisses, während gleichzeitig eine hohe Systembandbreite erreicht wird, effektiv aus. Die Leistungsstrafen, die auftreten, wenn zwei oder mehr Cachelesetags in dem Miniverzeichnis fehlschlagen, sind anwendungsabhängig. Obwohl es möglich ist, daß zwei eindeutige Sätze von vier Cachelesetags alle zwei Zyklen durch das Miniverzeichnis verarbeitet werden können, wird davon ausgegangen, daß typischerweise nur ein oder zwei eindeutige Blocktags in jedem Satz von vier Cachelesetags erscheinen. Wie es oben erörtert wurde, wenn Pixel eines Objekts aufbereitet werden und eine trilineare Interpolation verwendet wird, bilden benachbarte Pixel häufig auf die gleichen zwei Abbildungen für die MIP-Abbildung ab, was es erfordert, daß Lesevorgänge in den Cache fortlaufend zwischen den Cacheblöcken wechseln, die die beiden Abbildungen

speichern. Bei dem in **Fig.** 21 gezeigten darstellenden Ausführungsbeispiel speichert das Miniverzeichnis fünf Blocktags, um sicherzustellen, daß, selbst wenn sich vier eindeutige Cachetags für einen aktuell verarbeiteten Satz von Lesetags in dem Minicache befinden, zumindest ein Tag, auf das in dem vorhergehenden Satz von Lesetags zugegriffen wurde, in dem Miniverzeichnis bleibt. Selbst wenn zwischen zwei Sätzen von vier eindeutigen Cachetags gewechselt wird, bleibt somit während einer trilinearen Interpolation zumindest eines der Lesecachetags für jeden Satz in dem Miniverzeichnis, so daß vier Cachetags nicht auf serielle Weise mit dem Hauptverzeichnis verglichen werden müssen.

[0227] Wenn eine trilineare Interpolation verwendet wird, lesen aufeinanderfolgende Lesevorgänge des Caches während dem Aufbereiten von Texeln einen ersten Satz von vier Texeln in einer Abbildung und einen zweiten Satz von vier Texeln in einer anderen. Während ein Grundelement aufbereitet wird, wird auf benachbarte Texel in jeder der zwei Abbildungen in jedem zweiten Zyklus zugegriffen, und zwei oder mehr der Texel sind im allgemeinen in einem einzigen Cacheblock positioniert. Falls daher nur ein oder zwei eindeutige Tags in jedem Satz von vier Cachelesetags erscheinen, kann eine große Anzahl von Pixeln aufbereitet werden, wenn jedes Cachelesetag das Hauptverzeichnis **500** trifft. Falls nur ein Cachelesetag in jedem Satz von vier in dem Miniverzeichnis fehlschlägt, wird keine Leistungsstrafe bewirkt, weil das Tag mit dem Hauptverzeichnis verglichen werden kann, während der nächste Satz von vier Lesetags mit dem Miniverzeichnis verglichen wird.

[0228] Es sollte klar sein, daß das Cacheverzeichnis der vorliegenden Erfindung, das sowohl ein Hauptverzeichnis als auch ein kleineres Miniverzeichnis umfaßt, mit vielen anderen Anwendungen verwendet werden kann, und nicht auf die Verwendung bei einem Texturabbildungshardwaresystem beschränkt ist. Das Minicacheverzeichnisschema der vorliegenden Erfindung ist insbesondere sinnvoll beim Implementieren eines vollassoziativen Cache und beim Reduzieren der Kosten von Verzeichnistagvergleichen, wenn mehrere Cachelesetags gleichzeitig verarbeitet werden, und wenn Cachelesetags mit nachfolgend zugegriffenen, vorher verwendeten Tags korreliert werden. Beispielsweise ist es für ein Cacheverzeichnis, das X Tags zu jedem Zeitpunkt speichert und bei dem N Cachelesetags gleichzeitig mit den Verzeichnisblocktags verglichen werden, ausreichend, um ein Miniverzeichnis beizubehalten, das M Tags umfaßt, wobei M größer oder gleich N ist. Jedes der M Miniverzeichnistags wird in einer einzigen Leseoperation mit den N Cachelesetags verglichen. Auf das Hauptverzeichnis wird für jedes Cachelesetag, das in dem Miniverzeichnis fehlschlägt, seriell zugegriffen. Solche Lesetags werden in einem einzigen Zyklus mit den Hauptverzeichnistags verglichen. Die Hardwareeinsparungen bezüglich der Komparatoren von einem System, bei dem jedes der X Tags in dem Hauptverzeichnis mit den N Lesetags in einer einzigen Leseoperation verglichen wird, hängt von dem Verhältnis von (X + M*N)/(X*N) ab.

[0229] Die Leistungsstrafe, die bewirkt wird, um diese Hardwareeinsparungen zu erreichen, ist anwendungsabhängig, auf der Basis des Verhaltens der Sequenz von Tags, auf die in aufeinanderfolgenden Leseoperationen zugegriffen wird. Falls nicht mehr als ein Tag in jedem Lesesatz in dem Miniverzeichnis fehlschlägt, wird keine Strafe bewirkt, da das fehlgeschlagene Tag parallel mit dem nächsten Satz von Lesetags, die mit dem Miniverzeichnis verglichen werden, mit dem Hauptverzeichnis verglichen werden kann.

[0230] Bezüglich den oben beschriebenen Komparatoren **526**–**530**, die verwendet werden, um Leistungsstrafen zu reduzieren, wenn zwei oder mehr Cachelesetags in dem Miniverzeichnis fehlschlagen, werden sechs verwendet, weil auf vier Lesetags gleichzeitig zugegriffen wird. Die Anzahl von Komparatoren, die verwendet wird, um jedes Cachelesetag mit den anderen zu vergleichen, hängt von der Anzahl N von Lesetags ab, auf die gleichzeitig zugegriffen wird, und ist gleich der Summe von Ganzzahlen von 1 bis N-1.

[0231] Eine darstellende Implementierung eines Cacheverzeichnisses, das das Miniverzeichnis und das Hauptverzeichnis von **Fig.** 21–23 zeigt, ist in **Fig.** 24 gezeigt. Es sollte klar sein, daß die in **Fig.** 24 gezeigte Implementierung nur zu Darstellungszwecken geliefert ist, und daß auch andere Implementierungen verwendet werden können.

[0232] Die Miniverzeichniseinträge **501**–**505** (**Fig.** 21) sind in eine Tagkomponente, die in den Tagregistern **501T**–**505T** gespeichert ist, und eine Indexkomponente, die in den Indexregistern **501I**–**505I** gespeichert ist, aufgeteilt. Wie es oben erörtert wurde, empfängt das Cacheverzeichnis einen Satz von vier Lesecachetags, die den vier Texeln (d. h. UL, UR, LL und LR) entsprechen, die am nächsten zu der Position in einer MIP-Abbildung sind, auf die ein Pixel abbildet, das bearbeitet wird. Jedes der vier Lesetags wird an sechs Tagkomparatoren **541**–**546** geliefert. Fünf der Komparatoren (d. h. **542**–**546**) werden jeweils ebenfalls mit einem der fünf Miniverzeichnistagregister **501T**–**505T** gekoppelt. Beispielsweise ist der Komparator **542** für den Miniverzeichniseintrag **1** mit dem Tagregister **501T** gekoppelt und erzeugt eine Ausgabe, die anzeigt, ob das Tag in diesem Eintrag des Miniverzeichnisses mit dem Tag von einem der Lesecachetags UL, UR, LL oder LR übereinstimmt. Die Komparatoren **543**– **546** arbeiten auf ähnliche Weise und vergleichen jeweils die Lesecachetags UL, UR, LL und LR mit den Tagregistern **502T**– **505T**, die jeweils die Tags für den Miniverzeichniseintrag **2** – Eintrag **5** speichern. Jeder neue Satz von vier Lesecachetags wird in einem einzigen Zyklus mit dem Miniverzeichnis verglichen. Am Ende des Zyklus werden die vier Tags UL, UR, LL und LR jeweils in den Registern **550**–**553** gespeichert. Wie es in **Fig.** 24 ge-

zeigt ist, ist jedes der Register **550**– **553** auch mit einer Steuerschaltung **559** gekoppelt, die die Ausgaben der Miniverzeichnistagkomparatoren **542**–**546** empfängt. Am Ende des Zyklus, in dem ein neuer Satz von vier Lesetags mit den Miniverzeichnistags verglichen wird, wird jedes der Register **550**–**553** auch mit Daten geladen, die identifizieren, ob das entsprechendes Tag (d. h. UL, UR, LL, LR) mit einem der Miniverzeichniseinträge übereinstimmte, und falls dies der Fall ist, welcher Eintrag übereinstimmte.

[0233] Falls nur ein einzelnes Cachelesetag in dem Miniverzeichnis fehlschlägt, wird dieses Tag, wie es oben erörtert wurde, mit dem Hauptverzeichnis verglichen, während ein nächster Satz von vier Texellesetags mit dem Miniverzeichnis verglichen wird. Wenn in dem Miniverzeichnis ein Fehlschlag auftritt, wir das Miniverzeichnis aktualisiert, um das Tag aufzunehmen, das fehlgeschlagen hat, für das das Miniverzeichnis immer die fünf als letztes zugegriffenen Cachetags reflektiert. Während des Zyklus, in dem ein Lesecachetag, das in dem Miniverzeichnis fehlgeschlagen hat, mit dem Hauptverzeichnis verglichen wird, während ein nächster Satz von vier Lesetags mit dem Miniverzeichnis verglichen wird, wurden die Miniverzeichnistagregister **501T**–**505T** noch nicht aktualisiert, um den Cachetag zu umfassen, der in dem vorhergehenden Zyklus in dem Miniverzeichnis fehlgeschlagen hat. Wenn daher der nächste Satz von Lesecachetags mit dem Miniverzeichnis verglichen wird, wird ein sechster Komparator **541** verwendet, um die vier Lesetags (UL, UR, LL und LR) mit dem Tag zu vergleichen, das in dem vorhergehenden Zyklus in dem Miniverzeichnis fehlgeschlagen hat und mit dem Hauptverzeichnis verglichen wird. Falls mehr als ein eindeutiges Tag in dem Satz von vier Cachelesetags (UL, UR, LL und LR) in dem Miniverzeichnis fehlschlägt, wird die Pipeline durch das Cacheverzeichnis angehalten, weil mehrere Vergleiche mit dem Hauptverzeichnis auftreten werden. Falls nur ein eindeutiges Tag in dem Miniverzeichnis fehlschlägt, fährt die Pipeline auf folgende Weise fort, so daß das Cacheverzeichnis in jedem Zyklus einen neuen Satz von vier Cachelesetags empfängt.

[0234] Wie es oben angemerkt wurde, werden die Lesetags, die in dem vorhergehenden Zyklus mit dem Miniverzeichnis verglichen wurden, in den Registern **550**–**553** gespeichert. Die Ausgänge dieser Register sind mit einem Vier-zu-Eins-Multiplexer **555** gekoppelt, der eines dieser Register zu einem Zeitpunkt auswählt, damit es mit dem Hauptverzeichnis verglichen wird und, an dem Ende des Zyklus in das Miniverzeichnis geladen wird, so daß das Miniverzeichnis mit den als letztes empfangenen Lesecachetags aktualisiert wird. Der Ausgang des Multiplexers **555** ist auch mit dem sechsten Komparator **541** gekoppelt, so daß das Cachelesetag, das in dem vorhergehenden Zyklus in dem Miniverzeichnis fehlgeschlagen hat, mit jedem des neuen Satzes von Lesetags UL, UR, LL und LR verglichen wird. In Kombination mit den Komparatoren **542**–**546** stellt der Komparator **541** sicher, daß das Miniverzeichnis jeden Satz von vier Cachelesetags, der durch das Cacheverzeichnis empfangen wird, mit den fünf als letztes empfangenen Lesetags vergleicht.

[0235] Wie es oben angemerkt wurde, wird das Cachelesetag, das von dem Multiplexer **555** ausgegeben wird, auch in eines der Miniverzeichnistagregister **501T**–**505T** geladen, an dem Ende des Zyklus, in dem es mit dem Hauptverzeichnis verglichen wird. Somit wird das Miniverzeichnis aktualisiert, um die als letztes zugegriffene Cachetag zu umfassen. Die Bestimmung darüber, welcher Eintrag mit dem neuen Cachetag von dem Multiplexer **555** beschrieben wird, wird durch das nachfolgend erörterte Austauschschema getroffen.

[0236] Der Satz von sechs Komparatoren **526**–**532**, der oben in Verbindung mit **Fig.** 23 erörtert ist, ist der Zweckmäßigkeit halber in **Fig.** 24 als ein einzelner Komparatorblock gezeigt. Die Ausgaben dieser Komparatoren und auch die Ausgaben der Komparatoren **541**–**546** werden jeweils an die Steuerschaltung **559** geliefert, die mehrere Funktionen durchführt. Wenn ein Fehlschlag zu dem Miniverzeichnis auftritt, bestimmt die Steuerschaltung **559**, welcher Eintrag in dem Miniverzeichnis mit dem neuen Lesecachetag ausgetauscht werden soll. Die Steuerschaltung **559** ersetzt keinen Eintrag, der durch eines der vier neu empfangenen Lesecachetags getroffen wurde, die mit dem Miniverzeichnis verglichen wurden, oder den als letztes gelesenen Cachetag, der mit dem Hauptverzeichnis verglichen wurde, und weist diesen Einträgen eine höchste Priorität zum Beibehaltenwerden in dem Miniverzeichnis zu. Außerdem speichert die Steuerschaltung **559** Zustandsinformationen bezügliche dessen, welche Miniverzeichniseinträge durch den vorhergehenden Satz von vier Lesetags getroffen wurden und weist denselben die nächsthöchste Priorität zum Beibehaltenwerden in dem Miniverzeichnis zu. Den verbleibenden Einträgen wird eine geringere Priorität zugewiesen.

[0237] Die Steuerschaltung **559** wählt einen Eintrag für den Austausch aus, der in der Gruppe niedrigster Priorität ist, die zumindest einen Eintrag umfaßt. Falls es somit zumindest einen Eintrag in der Gruppe niedrigerer Priorität gibt, der nicht durch eines der vier neu empfangenen Lesecachetags getroffen wurde, die mit dem Miniverzeichnis verglichen werden, das nicht das letzte Lesecachetag war, das mit dem Hauptverzeichnis verglichen wurde, und nicht in dem vorhergehenden Satz von vier Lesetags war, wird einer der Einträge in der Gruppe niedrigerer Priorität für den Austausch ausgewählt. Falls es jedoch keine Einträge in der Gruppe niedrigerer Priorität gibt, wird eine größere Gruppe von Einträgen ausgewählt, die nur die Einträge höchster Priorität ausschließt (d. h. diejenigen, die durch eines der vier neu empfangenen Lesecachetags getroffen wurden, und das letzte Lesecachetag, das mit dem Hauptverzeichnis verglichen wurde), und ein Eintrag von dieser Gruppe wird für den Austausch ausgewählt.

[0238] Sobald die Gruppe von verfügbaren Miniverzeichniseinträgen niedrigster Priorität identifiziert ist, wird eine Bestimmung, welcher Eintrag in der Gruppe ersetzt werden sollte, gemäß einem Austauschschema durchgeführt, das zyklisch durch jeden der fünf Miniverzeichniseinträge verläuft, jedesmal, wenn einer ausgetauscht wird. Dies kann auf eine Vielzahl von Arten durchgeführt werden. Bei einem Ausführungsbeispiel der Erfindung werden die fünf Miniverzeichniseinträge mit 1 bis 5 bezeichnet. Der Eintrag, der ausgetauscht werden soll, wird von der Gruppe niedrigster Priorität ausgewählt, indem zunächst der Eintrag mit höchster Nummer identifiziert wird, der nicht in der Gruppe ist, und dann der Eintrag mit nächsthöchster Nummer, der in der Gruppe ist, für den Austausch ausgewählt wird. Wenn der Eintrag **5** nicht in der Gruppe mit niedrigster Priorität ist, dreht sich das Schema um, so daß der Eintrag **1** als der Eintrag mit der nächsthöchsten Nummer behandelt wird. Durch dieses Austauschschema verläuft die Steuerschaltung **559** zyklisch durch die Miniverzeichniseinträge, jedesmal, wenn einer ausgetauscht werden muß, und steuert das Laden der ausgewählten Miniverzeichnistagregister **501T** – 505T.

[0239] Die Steuerschaltung **559** decodiert auch die Ausgaben der Komparatoren **541**–**546** zum Erzeugen von Daten für jedes der vier Lesetags (UL, UR, LL und LR), die anzeigen, ob das Lesetag mit einem Eintrag in dem Miniverzeichnis übereinstimmt, und falls dies der Fall ist, welcher Eintrag übereingestimmt hat. Diese Daten werden für jedes der Lesetags UL, UR, LL und LR in dem entsprechenden Register **550**–**553** gespeichert. Falls beispielsweise das Lesetag UL mit dem Miniverzeichniseintrag **3** übereinstimmt, würden die Daten, die durch die Steuerschaltung **559** decodiert werden, in dem UL-Register **550** gespeichert, um anzuzeigen, daß das Lesetag mit dem Miniverzeichniseintrag **3** übereinstimmt. Wie es nachfolgend erörtert wird, werden die Daten durch die Cacheverzeichnispipeline geleitet und zeigen an, daß der Blockindex für das UL-Texel in dem Register **503l** gespeichert ist, das den Blockindex für den Miniverzeichniseintrag **3** hält.

[0240] Wenn nur ein eindeutiges Tag für den Satz von Lesetags UL, UR, LL und LR in dem Miniverzeichnis fehlschlägt, wird jedes der Register **550**–**553**, das das Lesetag speichert, mit Daten geladen, die anzeigen, daß der Blockindex für die entsprechenden Texturdaten nicht in dem Miniverzeichnis ist. Während dem nächsten Zyklus wird die Ausgabe von einem der Register **550**–**553**, das das fehlgeschlagene Tag speichert, mit dem Hauptverzeichnis **520** verglichen, und der Blockindex für das Lesetag wird von dem Hauptverzeichnis in ein Register **561** geladen, das den Hauptverzeichnisblockindex speichert. Die Daten, die anzeigen, daß der Blockindex keinem Eintrag in dem Miniverzeichnis entspricht, wird auch in dem Register **561** gespeichert, von der Eingabe **562**, die von dem Ausgang des Multiplexers **555** geliefert wird.

[0241] Wie es oben beschrieben wurde, umfaßt der Cachespeicher vier Verschachtelungen A–D, so daß auf vier Texel gleichzeitig zugegriffen werden kann. Der Satz von vier Texellesetags UL, UR, LL und LR kann auf jede Weise den Verschachtelungen A–D entsprechen. Die Daten, die in den Registern **550**–**553** gespeichert sind, die identifizieren, welcher Miniverzeichniseintrag den Blockindex speichert, der jedem der Texel UL, UR, LL und LR entspricht, werden durch einen Bitstellenverschieber **563** geleitet, der gesteuert wird, um jedes der Texel UL, UR, LL und LR mit seiner entsprechenden Verschachtelung A–D zu korrelieren. Die Ausgaben des Bitstellenverschiebers werden in die Verschachtelungsindexsteuerregister **565**–**568** geladen, die jeweils den Verschachtelungen A–D entsprechen, und die jeweils den Miniverzeichniseintrag identifizieren, falls es einen gibt, der den Blockindex für die Verschachtelung speichert. Wenn nur ein einziger eindeutiger Lesecachetag in dem Miniverzeichnis fehlschlägt, tritt das Verschieben der Ausgaben von dem Register **550**–**553** und das Laden der Register **565**– 568 parallel zu dem Zugriff auf das Hauptverzeichnis **520** auf.

[0242] Wie es oben angemerkt wurde, identifizieren die Daten, die in die Register **565**–**568** geladen sind, welcher Miniverzeichniseintrag, falls es einen gibt, den Blockindex für die entsprechende Verschachtelung speichert. Diese Daten werden verwendet, um eine Mehrzahl von Verschachtelungsindexmultiplexern zu steuern, die bei **571** identifiziert werden, die den entsprechenden Blockindex für jede Verschachtelung von einem der Miniverzeichnisindexregister **501l**–**505l** und von dem Hauptverzeichnisblockindexregister **561** auswählen. Die Mehrzahl von Verschachtelungsindexmultiplexern **571** stellt vier unabhängige Sechs-zu-Eins-Multiplexer dar. Ein Multiplexer entspricht jeder Verschachtelung und wählt zwischen fünf Miniverzeichnisindexregistern **501l**–**505l** und dem Hauptverzeichnisblockindexregister **561** aus. Jeder Verschachtelungsindexmultiplexer wird durch das eine der Register **565**–**568** gesteuert, das der gleichen Verschachtelung entspricht und identifiziert, welcher Miniverzeichniseintrag den Blockindex für die Verschachtelung speichert. Wenn diese Daten anzeigen, daß sich der Blockindex für eine Verschachtelung nicht in einem Miniverzeichniseintrag befindet, wählt der entsprechende Multiplexer den Index aus, der von dem Hauptverzeichnisblockindexregister **561** geliefert wird, der einen Blockindex speichert, der von dem Hauptverzeichnis gelesen wird, nach einem Fehlschlag in dem Miniverzeichnis. Der Blockindex für jede der Verschachtelungen A–D wird über die Leitungen **580**–**583** geliefert und wird verwendet, um die Cache-SDRAMs auf die oben beschriebene Weise zu adressieren.

[0243] Wenn, wie es oben erörtert wurde, mehr als eines des Satzes von Lesecachetags UL, UR, LL und LR in dem Miniverzeichnis fehlschlägt, aber nur ein einziges eindeutiges Cachetag umfaßt, wird auf das

Hauptverzeichnis **520** nur einmal zugegriffen, um den Blockindex für dieses Lesetag zu liefern. Dieser Prozeß wird auch durch die Steuerschaltung **559** gesteuert, die die Ausgaben der Komparatoren **526**–**532** verwendet, um zu identifizieren, ob irgendwelche zwei der vier Lesetags übereinstimmen. Falls zwei oder mehr des Satzes von vier Lesetags in dem Miniverzeichnis fehlschlagen, aber das gleiche Cachetag umfassen, wird jedes der entsprechenden Register **550**–**553** durch die Steuerschaltung **559** eingestellt, um anzuzeigen, daß der Blockindex in keinem Miniverzeichniseintrag enthalten ist. Wenn somit die Daten, die diesen Lesetags entsprechen, in die Verschachtelungsindexregister **565**–**568** geleitet werden, wird jedes das Hauptverzeichnisblockindexsteuerregister **561** zum Leiten durch seinen entsprechenden Verschachtelungsindexmultiplexer **571** auswählen.

[0244] Die Steuerschaltung **559** stellt auch ein Verzeichnissteuerregister **573** ein, das steuert, welches der Lesetagregister **550**–**553** mit dem Hauptverzeichnis verglichen werden soll. Das Register **573** steuert den Multiplexer **555** zum Auswählen von einem der Register **550**–**553**, das zu einem Zeitpunkt mit dem Hauptverzeichnis verglichen werden soll. Falls mehr als eines der Lesetags UL, UR, LL, LR in dem Hauptverzeichnis fehlschlägt, aber ein gemeinsames Tag gemeinschaftlich verwendet, wird das Steuerregister **573** eingestellt, um anzuzeigen, daß nur eines der Register mit dem Hauptverzeichnis verglichen werden sollte. Auf diese Weise wird nur einmal auf das Hauptverzeichnis zugegriffen, wenn der Satz von vier Lesecachetags nur ein einziges eindeutiges Tag umfaßt, das in dem Miniverzeichnis fehlschlägt.

[0245] Falls der Satz von vier Lesecachetags (UL, UR, LL, LR) mehr als ein eindeutiges Tag umfaßt, das in dem Miniverzeichnis fehlschlägt, wird der oben beschriebene Fluß durch die Cacheverzeichnispipeline geändert, und das Cacheverzeichnis wird besetzt und empfängt keinen neuen Satz von Lesetags in dem nächsten Zyklus. Das Verzeichnis zeigt an, daß es besetzt ist, so daß jedes der Register **550**–**553**, das ein Lesetag umfaßt, das in dem Miniverzeichnis fehlgeschlagen hat, mit dem Hauptverzeichnis verglichen werden kann, und nicht mit einem neuen Lesetag überschrieben wird. Ferner wird der Fluß durch die Verzeichnispipeline geändert, so daß für jedes Lesetag auf das Hauptverzeichnis zugegriffen werden kann, das in dem Hauptverzeichnis fehlgeschlagen ist, und der Blockindex, der diesen entspricht, kann von dem Hauptverzeichnis in eines der Register **501I**–**505I** oder 561 geladen werden. Die Pipeline ist angeordnet, um die Daten in jedem der Register **550**–**553** daran zu hindern, durch den Bitstellenverschieber **563** geleitet zu werden, bis alle der Blockindizes für den Satz von Lesetags (UL, UR, LL, LR) entweder von dem Hauptverzeichnis gelesen wurden oder sich bereits in dem Miniverzeichnis befinden. Somit wird der Satz von Texeln UL, UR, LL und LR als eine Gruppe mit ihren entsprechenden Verschachtelungen korreliert.

[0246] Wenn mehr als ein eindeutiges Tag in einem Satz von Lesetags in dem Miniverzeichnis fehlschlägt, werden die fehlgeschlagenen Tags seriell verarbeitet. Während dem ersten Zyklus (d. h, wenn der Satz von Tags mit dem Miniverzeichnis verglichen wird) bestimmt die Steuerschaltung **559**, welcher Eintrag in dem Miniverzeichnis durch ein erstes fehlgeschlagenes Lesetag ausgetauscht werden soll, und das entsprechende Register **550**–**553** wird mit Daten geladen, die anzeigen, daß ein Blockindex in diesem Miniverzeichniseintrag gespeichert wird. Wenn die Ausgabe des Registers **550**–**553**, das das erste verarbeitete Fehlschlagtag speichert, während einem zweiten Zyklus mit dem Hauptverzeichnis **520** verglichen wird, wird das Hauptverzeichnisblockindexregister **561** mit den Daten aktualisiert, die anzeigen, welches Miniverzeichnisindexregister **501I**–**505I** ausgetauscht werden soll. Während einem dritten Zyklus wird der entsprechende Blockindex von dem Register **561** in das Register **501I**–**505I** geladen, das dem Hauptverzeichniseintrag entspricht, der für einen Austausch ausgewählt ist.

[0247] Jedes der nachfolgend verarbeiteten eindeutigen Tags, das in dem Hauptverzeichnis fehlgeschlagen hat, wird auf gleiche Weise gehandhabt, bis zu dem letzten Fehlschlagtag, das verarbeitet werden soll, das ein zweites Fehlschlagtag sein kann, falls nur zwei eindeutige Tags in dem Miniverzeichnis fehlgeschlagen sind, oder ein drittes oder viertes Fehlschlagtag sein kann. Das letzte Fehlschlagtag, das durch das Cacheverzeichnis verarbeitet wurde, wird gehandhabt, als ob es das einzige eindeutige Tag in dem Satz von Lesetags wäre, die in dem Miniverzeichnis fehlschlagen. Wenn das Verarbeiten des letzten Fehlschlagtags beginnt, deaktiviert das Verzeichnis das Signal, das anzeigt, daß dasselbe besetzt ist, so daß es einen neuen Satz von Lesetags empfangen kann.

[0248] Für das letzte verarbeitete Fehlschlagtag lädt die Steuerschaltung **559** ihr entsprechendes Register **550**–**553** mit Daten, die anzeigen, daß der Blockindex für das Tag nicht in einem Miniverzeichniseintrag gespeichert ist. Dies kann während dem ersten Zyklus durchgeführt werden, bei dem alle Lesetags mit dem Miniverzeichnis verglichen werden, oder parallel mit der Verarbeitung der anderen Fehlschlagtags. Während dem Zyklus, bei dem das letzte Fehlschlagtag mit dem Hauptverzeichnis verglichen wird, werden die Daten in den Registern **550**–**553** durch den Bitstellenverschieber **563** geleitet und in die Verschachtelungssteuerregister **565**– **568** geladen, und der Blockindex für das Fehlschlagtag wird von dem Hauptverzeichnis in das Hauptverzeichnisblockindexregister **561** geladen. Schließlich werden in der letzten Pipelinestufe des Verzeichnisses die Ausgänge der Verschachtelungsindexsteuerregister **565**–**568** verwendet, um ihre entsprechenden Verschachtelungsindexmultiplexer **571** zu steuern, so daß der Index für das letzte verarbeitete Fehlschlag-

tag von dem Hauptverzeichnisblockindexregister **561** geliefert wird, und der Blockindex für jedes der anderen Lesetags in dem Satz von seinem entsprechenden Miniverzeichnisindexregister **501l–505l** geliefert wird. Es sollte klar sein, daß durch Zugreifen auf den Blockindex für das als letztes verarbeitete Fehlschlagtag von dem Hauptverzeichnisblockindexregister **561** ein Zyklus eingespart wird, indem nicht darauf gewartet wird, daß der Blockindex für dieses Tag in das Miniverzeichnisindexregister desselben geladen wird.

Patentansprüche

1. Ein Verfahren zum Verwalten von Texturabbildungsdaten in einem Computergraphiksystem, wobei das Computergraphiksystem einen Hostcomputer (**15**), eine Grundelementaufbereitungshardware (**12**, **14**) und einen texturiertes-Grundelement-Datenweg (**30**, **32A**, **32B**, **32C**, **36**, **18**) umfaßt, der sich zwischen dem Hostcomputer (**15**) und der Grundelementaufbereitungshardware (**12**, **14**) erstreckt, wobei der Hostcomputer (**15**) texturierte Grundelemente, die durch das System unter Verwendung von entsprechenden Texturabbildungsdaten aufbereitet werden sollen, über den texturiertes-Grundelement-Datenweg (**30**, **32A**, **32B**, **32C**, **36**, **18**) an die Grundelementaufbereitungshardware leitet, wobei der Hostcomputer (**15**) einen Hauptspeicher (**17**) umfaßt, der Texturabbildungsdaten speichert, die den texturierten Grundelementen entsprechen, die aufbereitet werden sollen, wobei die Grundelementaufbereitungshardware einen lokalen Texturspeicher (**48**) umfaßt, der die Texturabbildungsdaten, die zumindest einem der texturierten Grundelemente, die aufbereitet werden sollen, entsprechen, lokal speichert, wobei das Verfahren folgenden Schritt umfaßt:
(a) wenn Texturabbildungsdaten, die einem der texturierten Grundelemente, die aufbereitet werden sollen, entsprechen, in dem Hauptspeicher (**17**) des Hostcomputers (**15**), aber nicht in dem lokalen Texturabbildungsspeicher (**48**) gespeichert sind, vor dem Abschluß der Aufbereitung des einen der texturierten Grundelemente, Herunterladen der Texturabbildungsdaten, die dem einem der texturierten Grundelemente von dem Hauptspeicher (**17**) des Hostcomputers (**15**) entsprechen, zu dem lokalen Texturabbildungsspeicher (**48**) durch einen Texturabbildungsdatenweg (**40**, **34**, **24**), der von dem texturiertes-Grundelement-Datenweg (**30**, **32A**, **32B**, **32C**, **36**, **18**) getrennt ist;
wobei die Grundelementaufbereitungshardware (**12**, **14**) eine Grundelementpipeline (**14**) umfaßt, durch die Grundelemente verarbeitet werden, und eine Texturabbildungspipeline (**12**), durch die Texturabbildungsdaten verarbeitet werden, wobei die Grundelementpipeline (**14**) tiefer ist als die Texturabbildungspipeline (**12**), so daß die Texturabbildungspipeline aufgrund der Nichtverfügbarkeit der Grundelementpipeline (**14**) nicht verzögert werden sollte.

2. Das Verfahren zum Verwalten von Texturabbildungsdaten gemäß Anspruch 1, bei dem das Computergraphiksystem einen nichttexturiertes-Grundelement-Datenweg (**24**) umfaßt, der sich zwischen dem Hostcomputer und der Grundelementaufbereitungshardware erstreckt, wobei der Hostcomputer zumindest nichttexturierte Grundelemente, die durch das System aufbereitet werden sollen, über den nichttexturiertes-Grundelement-Datenweg an die Grundelementaufbereitungshardware ohne entsprechende Texturabbildungsdaten leitet, und bei dem der Schritt (a) gekennzeichnet ist durch einen Schritt des Verwendens zumindest eines Teils des nichttexturiertes-Grundelement-Datenwegs in dem Texturabbildungsdatenweg.

3. Das Verfahren zum Verwalten von Texturabbildungsdaten gemäß Anspruch 1, bei dem das eine der texturierten Grundelemente, die aufbereitet werden sollen, ein erstes texturiertes Grundelement ist, wobei die Texturabbildungspipeline angeordnet ist, um ein zweites texturiertes Grundelement, das aufbereitet werden soll, zu speichern, und wobei der Schritt (a) gekennzeichnet ist durch einen Schritt des Herunterladens der Texturabbildungsdaten, die dem ersten texturierten Grundele- ment entsprechen, während das zweite texturierte Grundelement in der Pipeline beibehalten wird.

4. Das Verfahren zum Verwalten von Texturabbildungsdaten gemäß Anspruch 3, bei dem die Texturabbildungspipeline zumindest eine erste und eine zweite Stufe umfaßt, wobei die erste Stufe texturierte Grundelemente verarbeitet, die aufbereitet werden sollen, und die verarbeiteten texturierten Grundelemente an die zweite Stufe leitet, und bei dem das Verfahren ferner durch den Schritt des Verarbeitens des zweiten texturierten Grundelements in der ersten Stufe und Leiten des zweiten texturierten Grundelements in der zweiten Stufe während dem Herunterladen der Texturabbildungsdaten, die dem ersten texturierten Grundelement entsprechen, gekennzeichnet ist.

5. Das Verfahren zum Verwalten von Texturabbildungsdaten gemäß Anspruch 1, bei dem Schritt (a) gekennzeichnet ist durch einen Schritt des Unterbrechens des Hostcomputers mit einer Anforderung, die Texturabbildungsdaten, die dem einen der texturierten Grundelemente entsprechen, herunterzuladen.

6. Das Verfahren zum Verwalten von Texturabbildungsdaten gemäß Anspruch 1, bei dem die Texturabbildungsdaten, die in dem lokalen Texturspeicher gespeichert sind, eine Mehrzahl von Texturen darstellen, wobei die Texturabbildungsdaten, die dem einen der texturierten Grundelemente entsprechen, einer ersten Textur entsprechen, und bei dem Schritt (a) gekennzeichnet ist durch einen Schritt des Beibehaltens von Texturabbildungsdaten, die einer zweiten

Textur in dem lokalen Texturspeicher entsprechen, wenn die heruntergeladenen Texturabbildungsdaten, die der ersten Textur entsprechen, in den lokalen Texturspeicher geladen werden.

7. Das Verfahren zum Verwalten von Texturabbildungsdaten gemäß Anspruch 1, bei dem die Texturabbildungsdaten, die in dem Hauptspeicher gespeichert sind, eine Mehrzahl von Texturen darstellen, wobei jede Textur durch eine Reihe von MIP-Abbildungen dargestellt ist, wobei die Texturabbildungsdaten, die dem einen der texturierten Grundelemente entspricht, eine erste Textur darstellt, und bei dem Schritt (a) gekennzeichnet ist durch einen Schritt des Herunterladens von weniger als der gesamten Serie von MIP-Abbildungen für die erste Textur.

8. Ein Computergraphiksystem, das durch folgende Merkmale gekennzeichnet ist:
einen Hostcomputer (**15**), der Grundelemente liefert, die durch das System aufbereitet werden sollen, wobei der Hostcomputer einen Hauptspeicher (**17**) umfaßt, der Texturabbildungsdaten speichert, die texturierten Grundelementen entsprechen, die durch das System aufbereitet werden sollen;
eine Grundelementaufbereitungshardware (**12**, **14**), die Grundelemente aufbereitet, die von dem Hostcomputer geliefert werden, wobei die Grundelementaufbereitungshardware einen lokalen Texturspeicher (**48**) umfaßt, der Texturabbildungsdaten speichert, die zumindest einem texturierten Grundelement entsprechen, das aufbereitet werden soll;
einen texturiertes-Grundelement-Datenweg (**30**, **32A**, **32B**, **32C**, **36**, **18**), der den Hostcomputer mit der Grundelementaufbereitungshardware (**12**, **14**) koppelt, über den texturierte Grundelemente, die unter Verwendung von entsprechenden Texturaufbereitungsdaten aufbereitet werden sollen, von dem Hostcomputer an die Grundelementaufbereitungshardware geliefert werden; und
einen Texturabbildungsdatenweg (**40**, **34**, **24**), der den Hostcomputer und die Grundelementaufbereitungshardware (**12**) koppelt, über den Texturabbildungsdaten von dem Hauptspeicher (**14**) in dem Hostcomputer (**15**) zu dem lokalen Texturspeicher (**48**) heruntergeladen werden, wobei der Texturdatenweg von dem texturiertes-Grundelement-Datenweg getrennt ist,
dadurch gekennzeichnet, daß die Grundelementaufbereitungshardware (**12**, **14**) eine Grundelementaufbereitungspipeline (**14**) umfaßt, durch die texturierte Grundelemente verarbeitet werden, und eine Texturabbildungspipeline (**12**), durch die Texturabbildungsdaten verarbeitet werden, wobei die Grundelementaufbereitungspipeline (**14**) tiefer ist als die Texturabbildungspipeline (**12**), so daß die Texturabbildungspipeline (**12**) aufgrund einer Nichtverfügbarkeit der Grundelementaufbereitungspipeline (**14**) nicht verzögert werden sollte.

9. Das Computergraphiksystem gemäß Anspruch 8, das ferner dadurch gekennzeichnet ist, daß der lokale Texturspeicher ein Cachespeicher ist.

Es folgen 25 Blatt Zeichnungen

Anhängende Zeichnungen

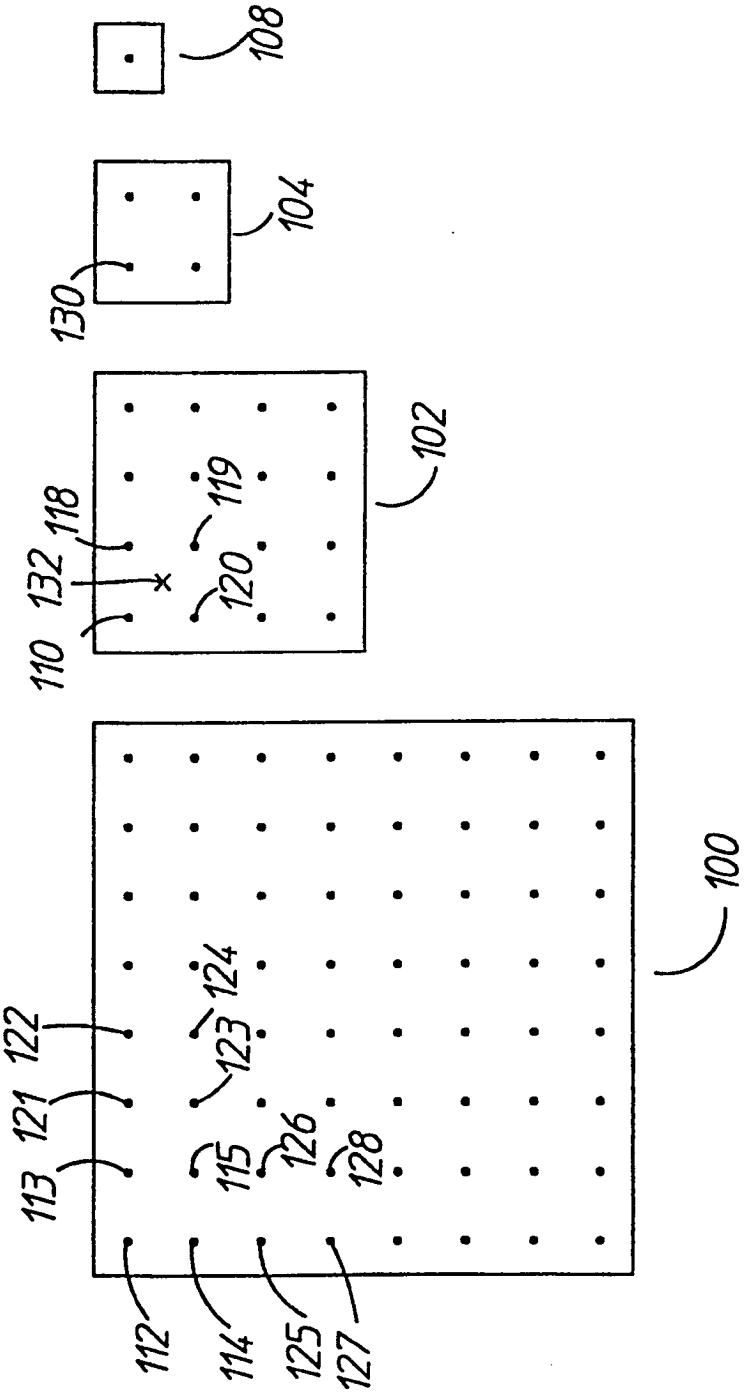


FIG 1

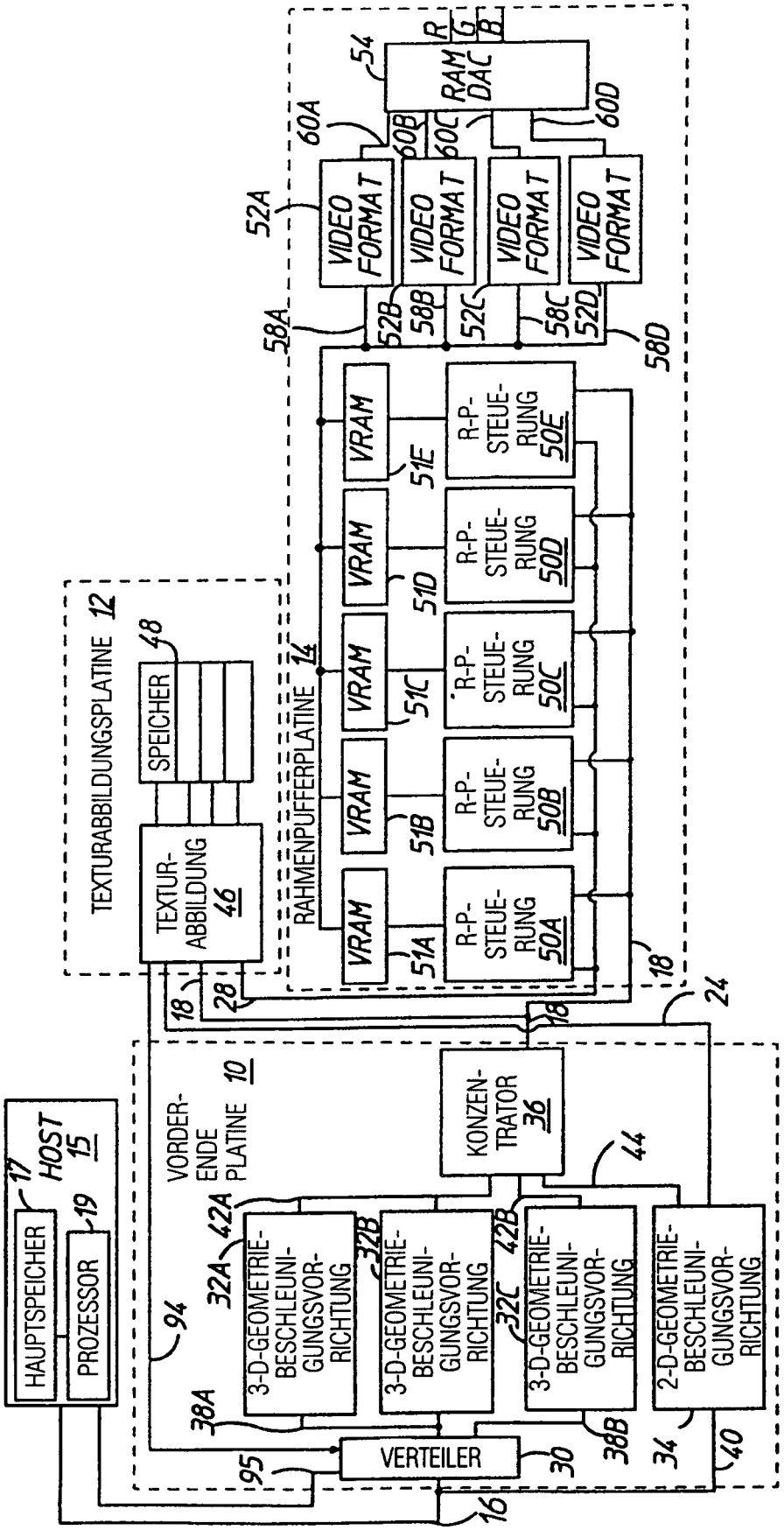
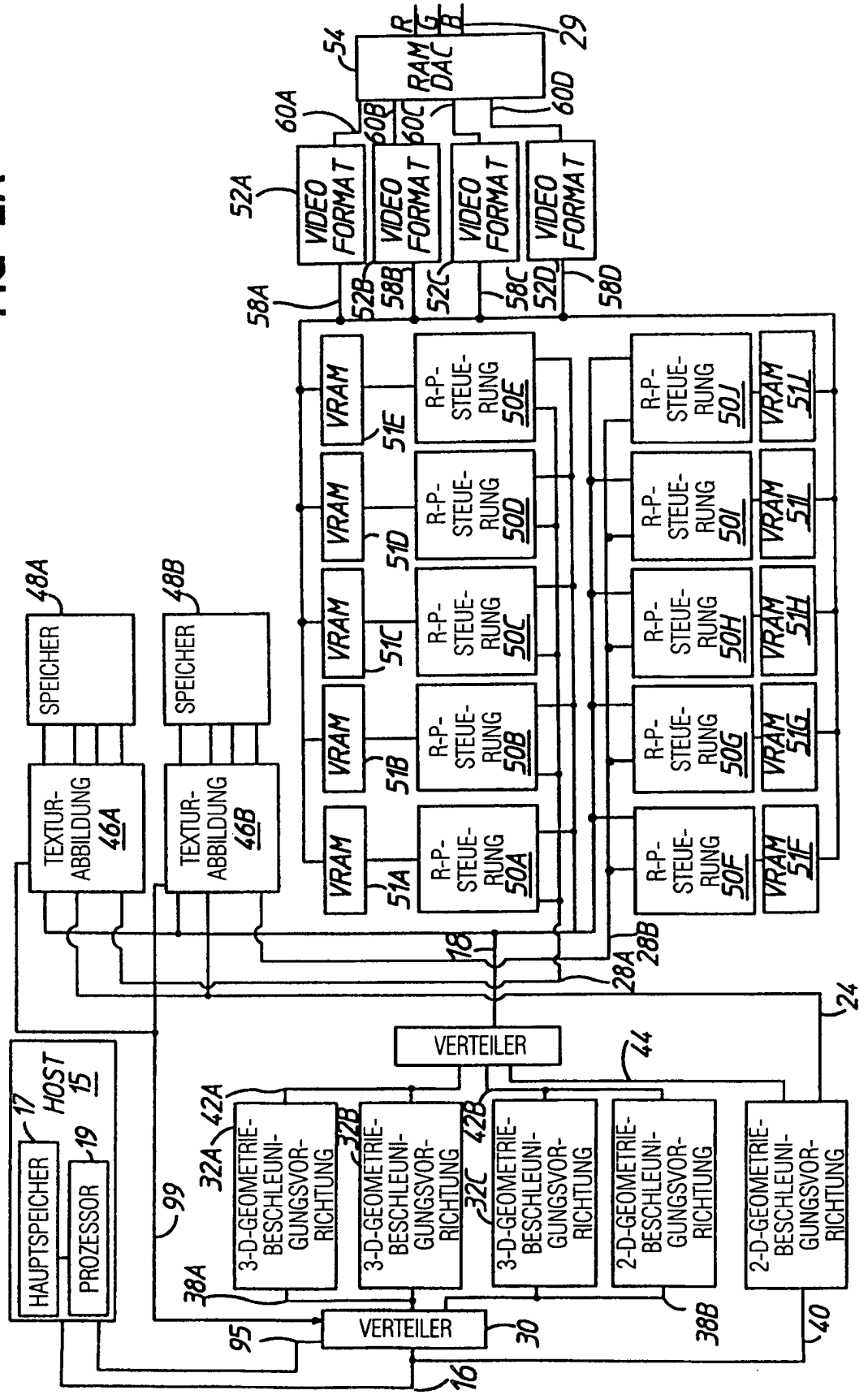

HOST 15
HAUPTSPEICHER
PROZESSOR
VORDER-ENDE PLATINE 10
3-D-GEOMETRIE-BESCHLEUNI-GUNGSVOR-RICHTUNG
KONZEN-TRATOR 36
VERTEILER
2-D-GEOMETRIE-BESCHLEUNI-GUNGSVOR-RICHTUNG
TEXTURABBILDUNGSPLATINE 12
SPEICHER
TEXTUR-ABBILDUNG 46
RAHMENPUFFERPLATINE 14
VRAM
R-P-STEUE-RUNG
VIDEO FORMAT
RAM DAC
R
G
B

FIG 2

FIG 2A


HAUPTSPEICHER
PROZESSOR
HOST 15
17
19
SPEICHER
48A
48B
TEXTUR-ABBILDUNG 46A
TEXTUR-ABBILDUNG 46B
VERTEILER
VERTEILER
3-D-GEOMETRIE-BESCHLEUNI-GUNGSVOR-RICHTUNG
2-D-GEOMETRIE-BESCHLEUNI-GUNGSVOR-RICHTUNG
VRAM
R-P-STEUE-RUNG 50A
R-P-STEUE-RUNG 50B
R-P-STEUE-RUNG 50C
R-P-STEUE-RUNG 50D
R-P-STEUE-RUNG 50E
R-P-STEUE-RUNG 50F
R-P-STEUE-RUNG 50G
R-P-STEUE-RUNG 50H
R-P-STEUE-RUNG 50I
R-P-STEUE-RUNG 50J
VRAM 51A
VRAM 51B
VRAM 51C
VRAM 51D
VRAM 51E
VRAM 51F
VRAM 51G
VRAM 51H
VRAM 51I
VRAM 51J
VIDEO FORMAT
RAM DAC
R
G
B
54
29
60A
60B
60C
60D
52A
52B
52C
52D
58A
58B
58C
58D
18
28A
28B
24
44
42A
42B
32A
32B
32C
38A
38B
99
95
30
40
16

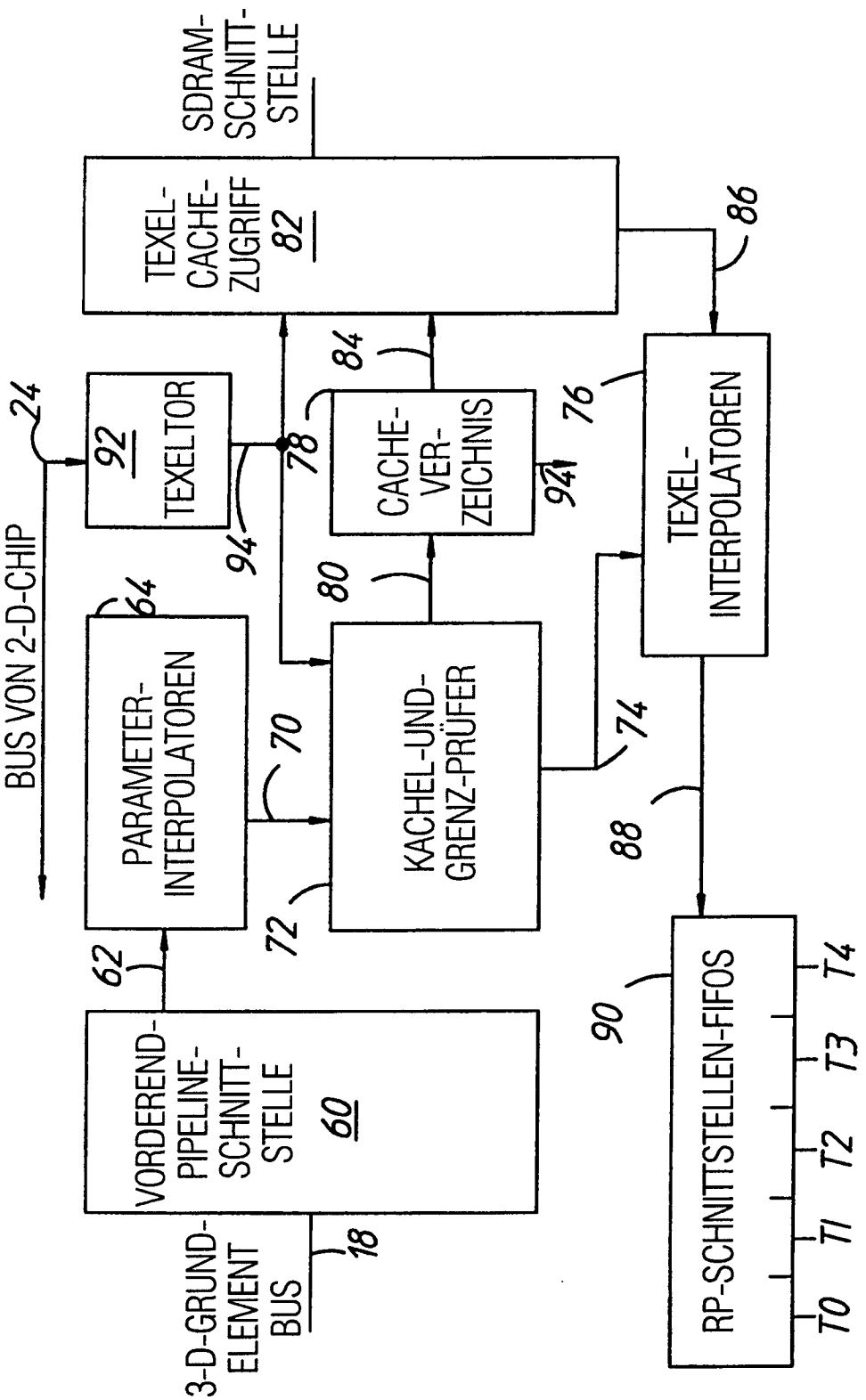

SDRAM-SCHNITT-STELLE
TEXEL-CACHE-ZUGRIFF 82
86
84
24
92 TEXELTOR
78
CACHE-VER-ZEICHNIS
94
94
76
TEXEL-INTERPOLATOREN
BUS VON 2-D-CHIP
64
80
PARAMETER-INTERPOLATOREN
70
KACHEL-UND-GRENZ-PRÜFER
74
72
88
62
VORDEREND-PIPELINE-SCHNITT-STELLE 60
90
RP-SCHNITTSTELLEN-FIFOS
T0
T1
T2
T3
T4
3-D-GRUND-ELEMENT BUS
18

FIG 3

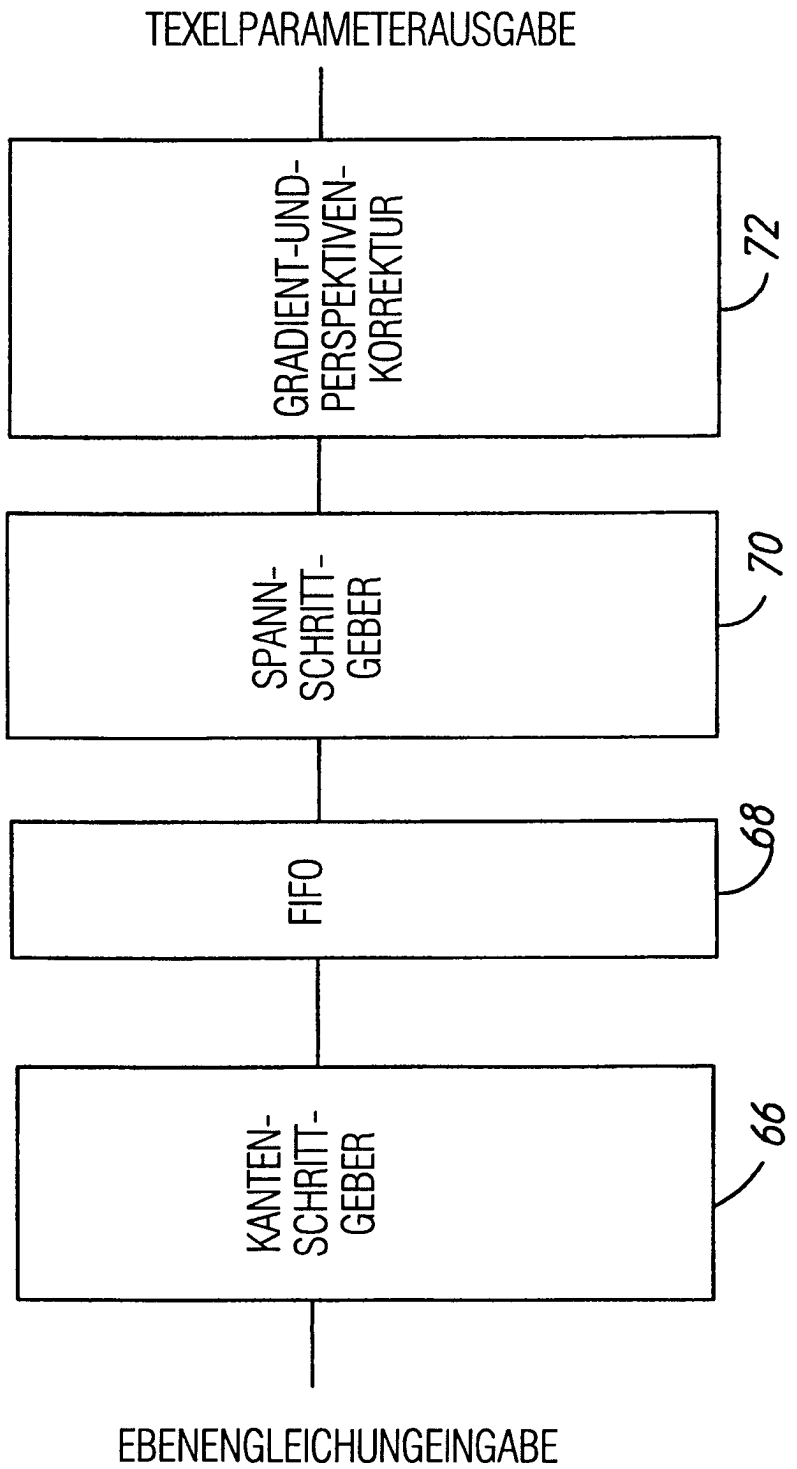

TEXELPARAMETERAUSGABE
GRADIENT-UND-PERSPEKTIVEN-KORREKTUR
72
SPANN-SCHRITT-GEBER
70
FIFO
68
KANTEN-SCHRITT-GEBER
66
EBENENGLEICHUNGEINGABE

FIG 4

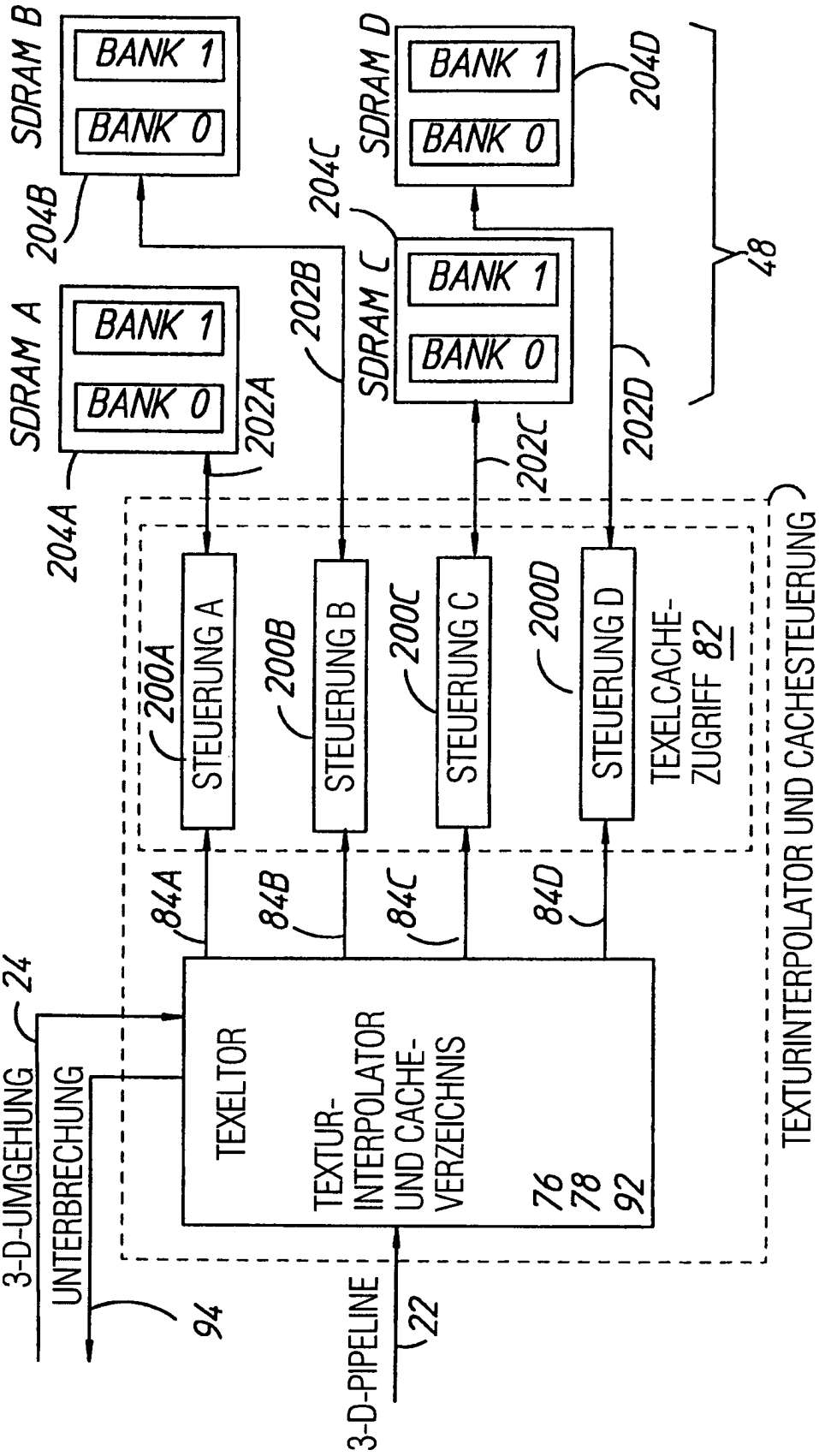
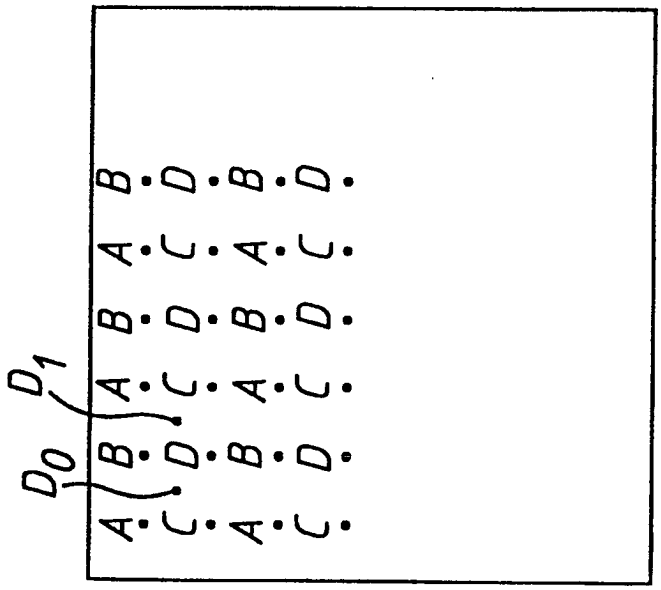

SDRAM B
BANK 1
BANK 0
204B
SDRAM D
BANK 1
BANK 0
204D
204C
SDRAM A
BANK 1
BANK 0
202A
202B
SDRAM C
BANK 1
BANK 0
48
204A
202C
202D
200A
STEUERUNG A
200B
STEUERUNG B
200C
STEUERUNG C
200D
STEUERUNG D
TEXELCACHE-ZUGRIFF 82
84A
84B
84C
84D
24
3-D-UMGEHUNG
UNTERBRECHUNG
94
TEXELTOR
TEXTUR-INTERPOLATOR UND CACHE-VERZEICHNIS
76
78
92
3-D-PIPELINE
22
TEXTURINTERPOLATOR UND CACHESTEUERUNG

FIG 5

FIG 6



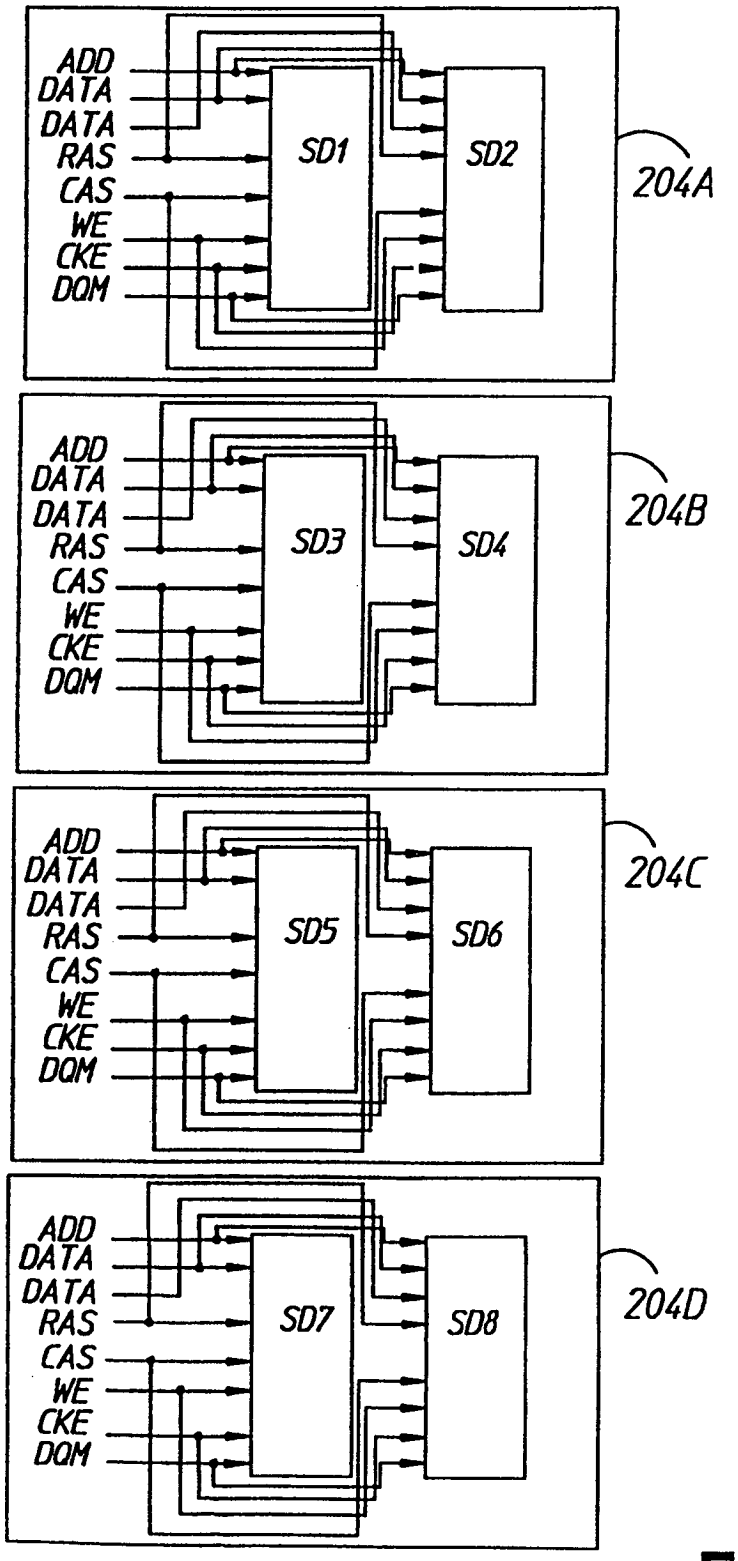

ADD
DATA
DATA
RAS
CAS
WE
CKE
DQM
SD1
SD2
204A
SD3
SD4
204B
SD5
SD6
204C
SD7
SD8
204D

FIG 7

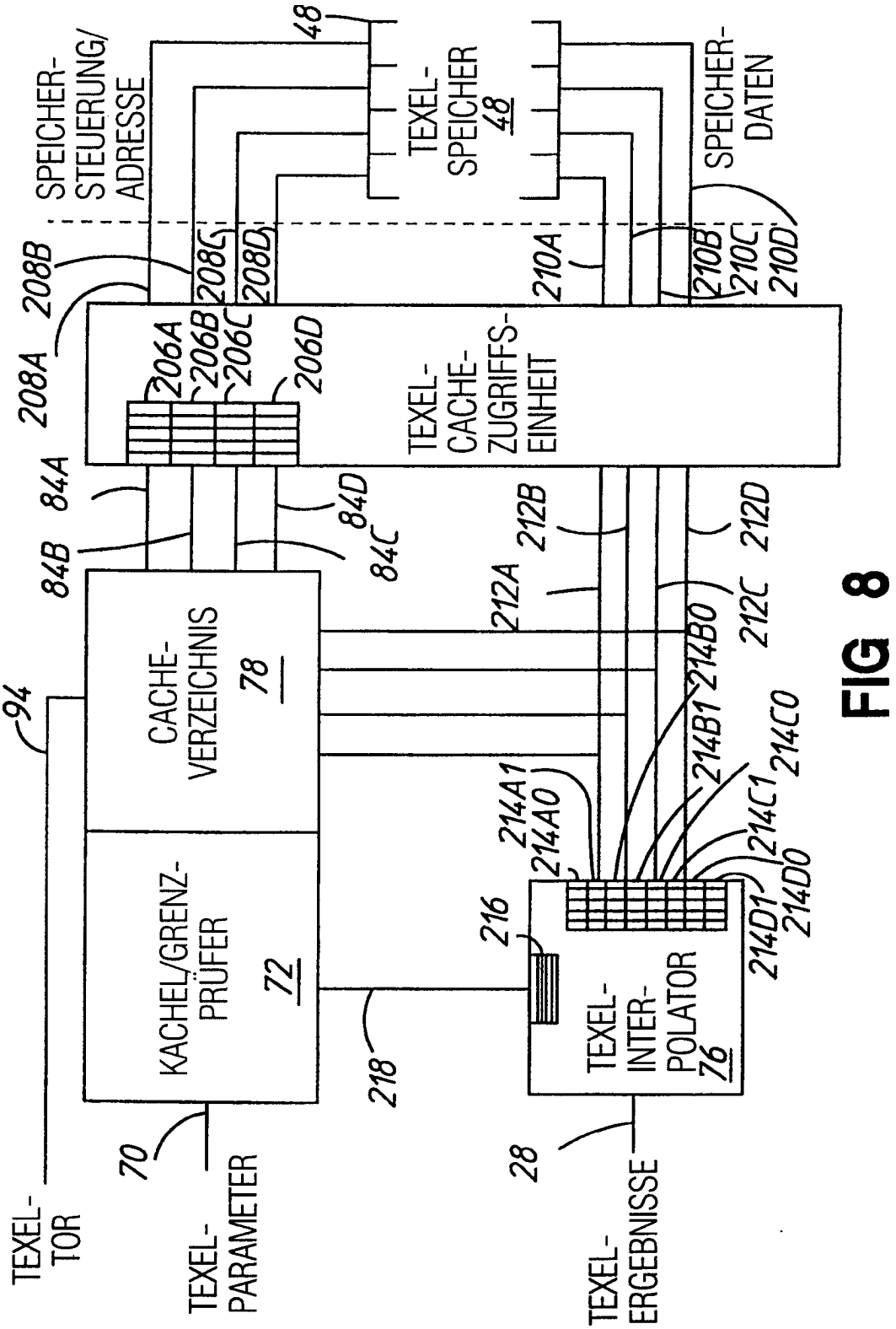

SPEICHER-
STEUERUNG/
ADRESSE
48
TEXEL-
SPEICHER
48
SPEICHER-
DATEN
208B
208C
208D
210A
210B
210C
210D
208A
206A
206B
206C
206D
TEXEL-
CACHE-
ZUGRIFFS-
EINHEIT
84A
84B
84C
84D
212A
212B
212C
212D
214B0
214B1
214C0
214C1
214D0
214D1
214A0
214A1
94
CACHE-
VERZEICHNIS
78
KACHEL/GRENZ-
PRÜFER
72
216
218
TEXEL-
INTER-
POLATOR
76
70
28
TEXEL-
TOR
TEXEL-
PARAMETER
TEXEL-
ERGEBNISSE

FIG 8

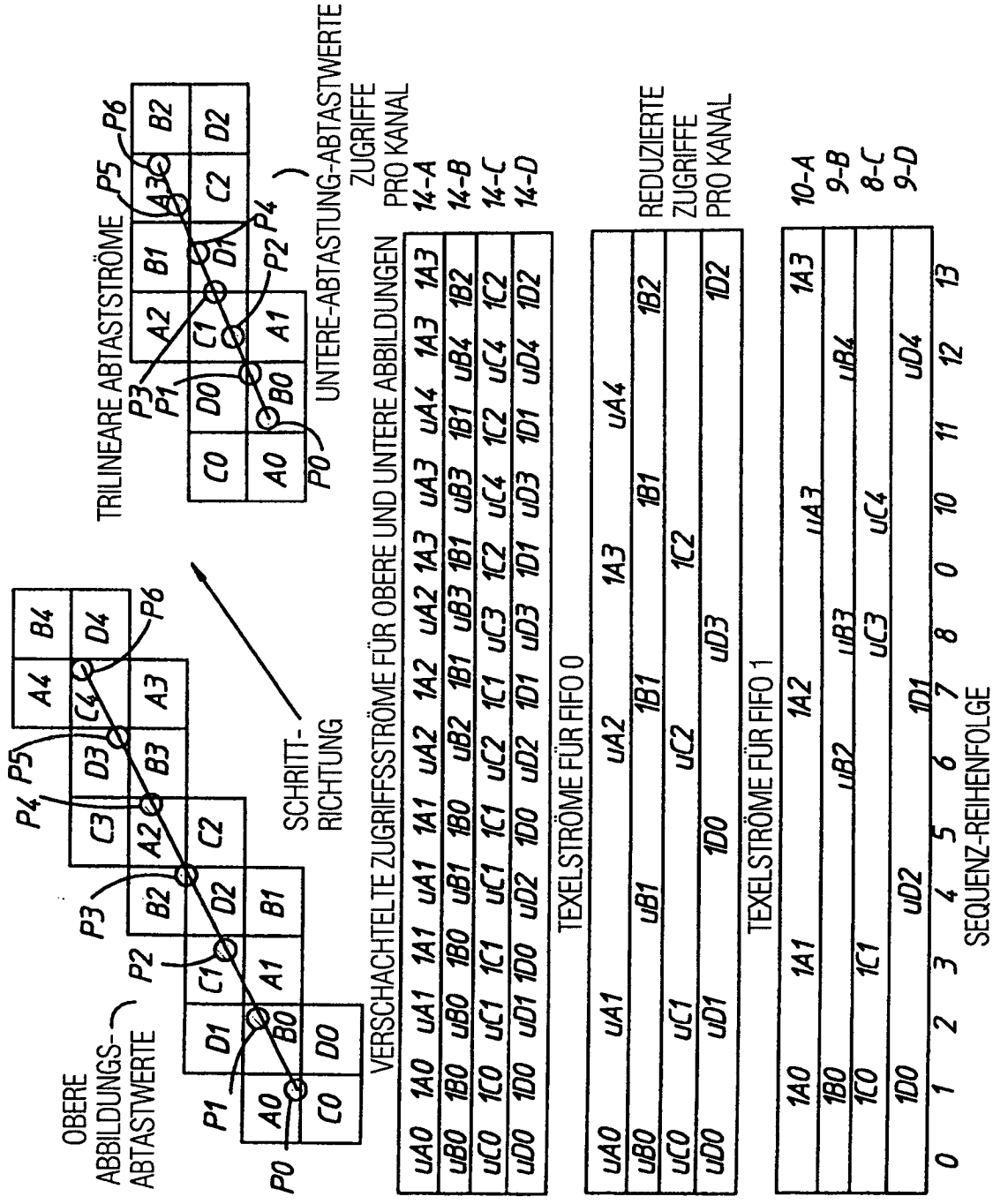

OBERE ABBILDUNGS-ABTASTWERTE
TRILINEARE ABTASTSTRÖME
UNTERE-ABTASTUNG-ABTASTWERTE
SCHRITT-RICHTUNG
VERSCHACHTELTE ZUGRIFFSSTRÖME FÜR OBERE UND UNTERE ABBILDUNGEN
TEXELSTRÖME FÜR FIFO 0
TEXELSTRÖME FÜR FIFO 1
ZUGRIFFE PRO KANAL
REDUZIERTE ZUGRIFFE PRO KANAL
14-A
14-B
14-C
14-D
10-A
9-B
8-C
9-D
SEQUENZ-REIHENFOLGE

FIG 9

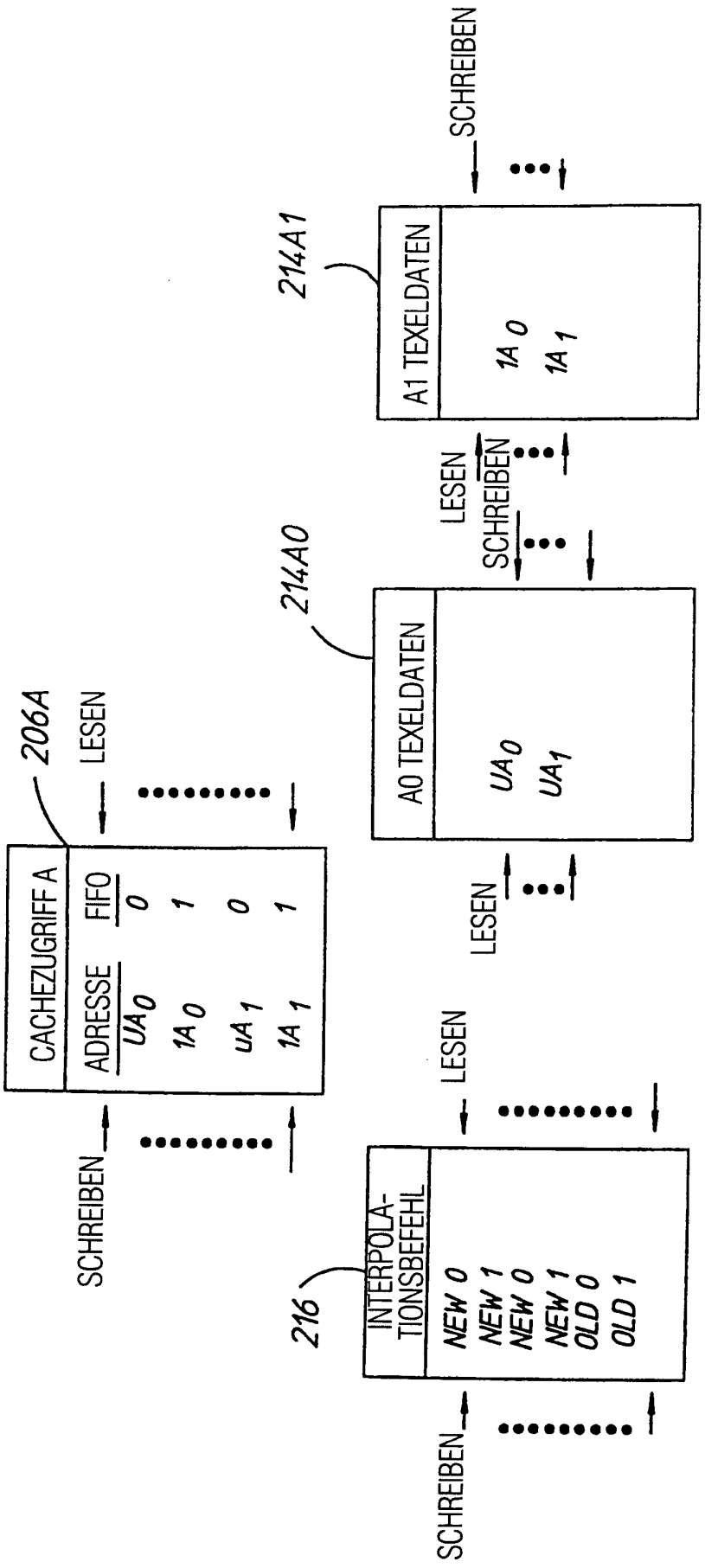

206A
CACHEZUGRIFF A
ADRESSE
FIFO
UA0
0
1A0
1
UA1
0
1A1
1
LESEN
SCHREIBEN
214A0
A0 TEXELDATEN
UA0
UA1
LESEN
SCHREIBEN
214A1
A1 TEXELDATEN
1A0
1A1
LESEN
SCHREIBEN
216
INTERPOLA-TIONSBEFEHL
NEW 0
NEW 1
NEW 0
NEW 1
OLD 0
OLD 1
LESEN
SCHREIBEN

FIG 10

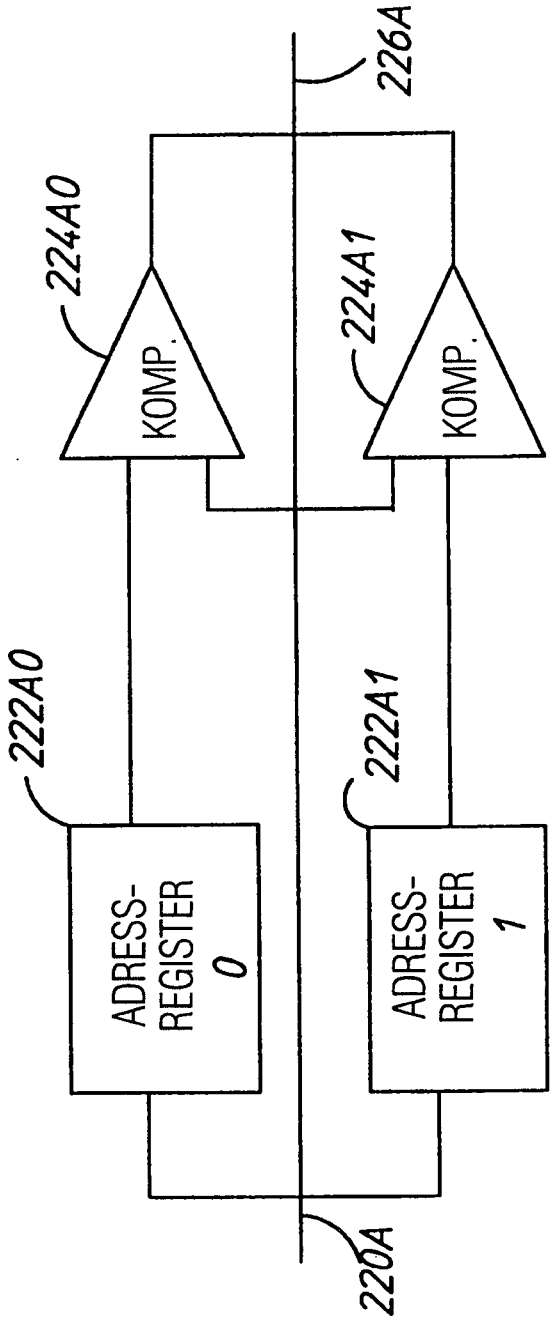

226A
224A0
KOMP.
224A1
KOMP.
222A0
ADRESS-
REGISTER
0
222A1
ADRESS-
REGISTER
1
220A

FIG 11

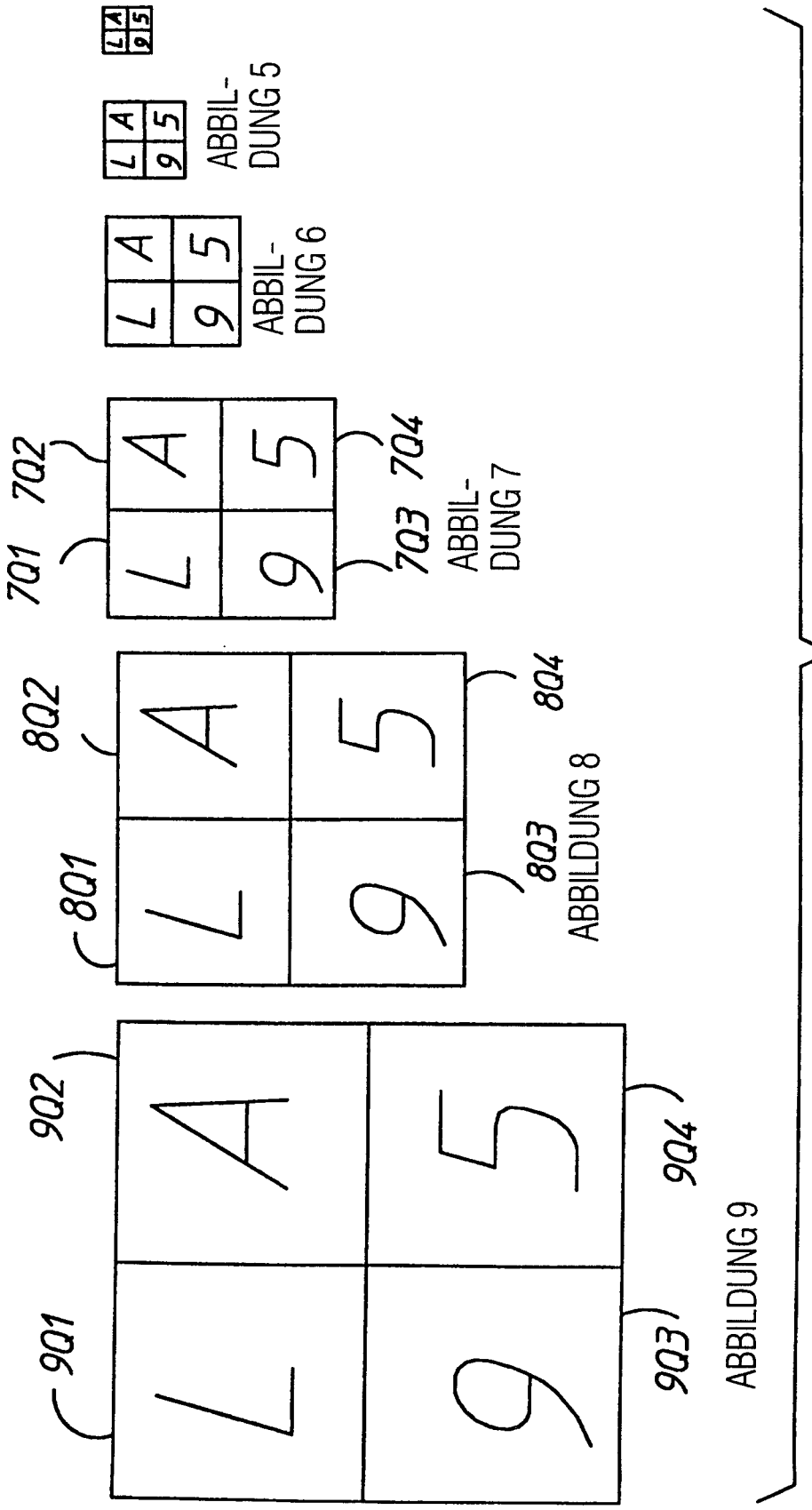

L A
9 5
ABBIL-
DUNG 5
L A
9 5
ABBIL-
DUNG 6
701
702
703
704
L A
9 5
ABBIL-
DUNG 7
801
802
803
804
L A
9 5
ABBILDUNG 8
901
902
903
904
L A
9 5
ABBILDUNG 9

FIG 12

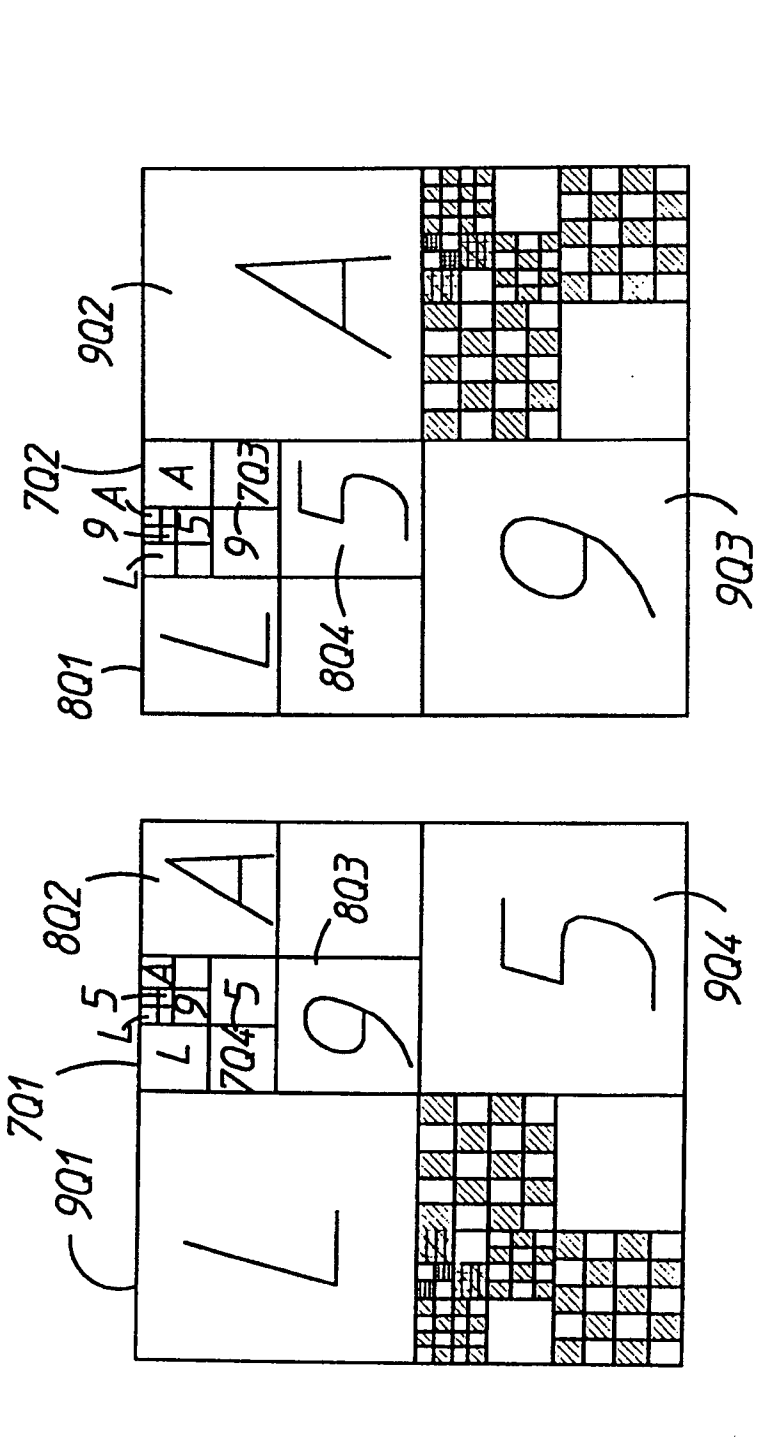


FIG 13

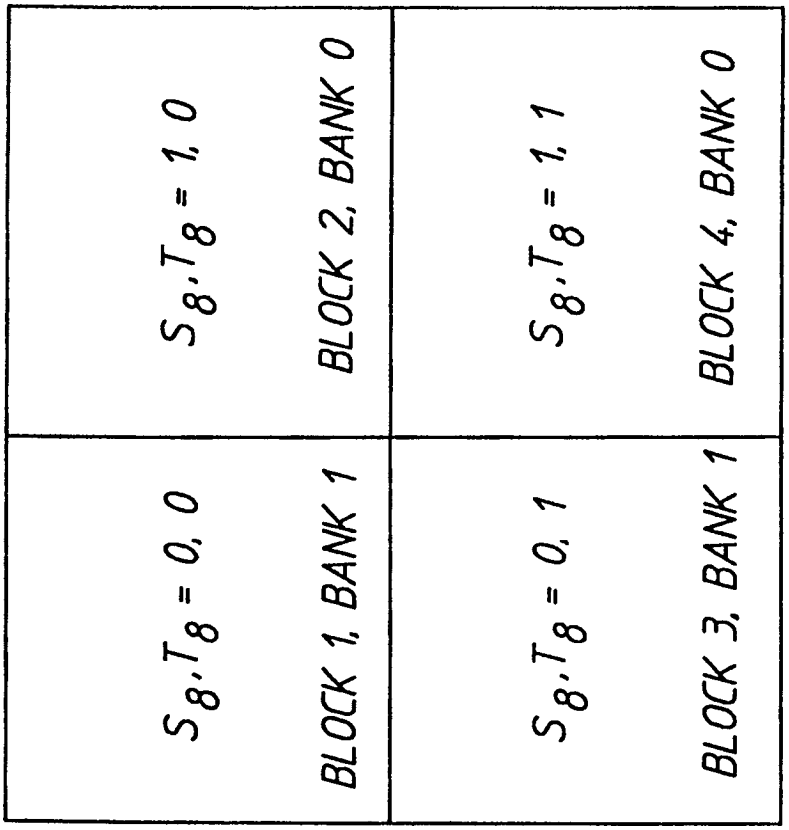
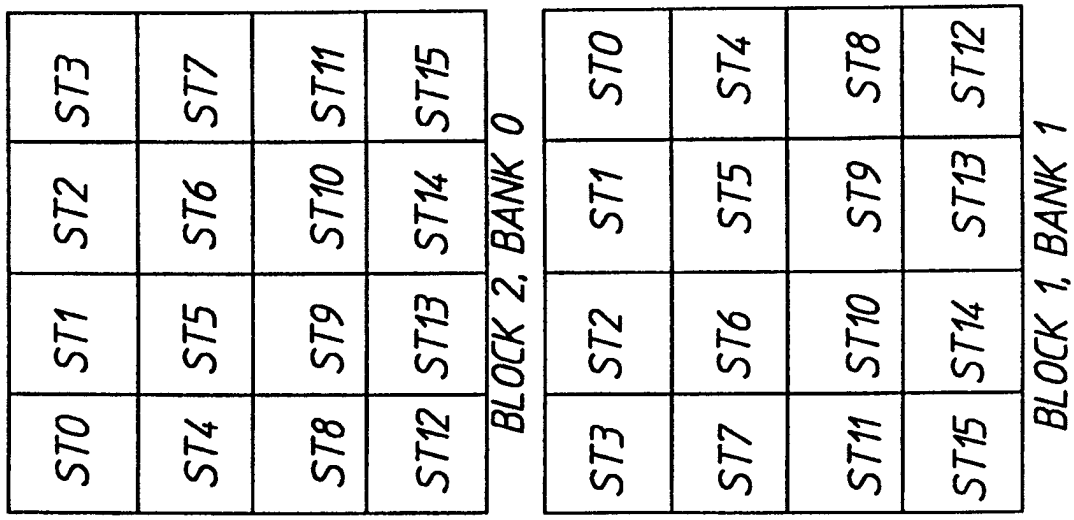
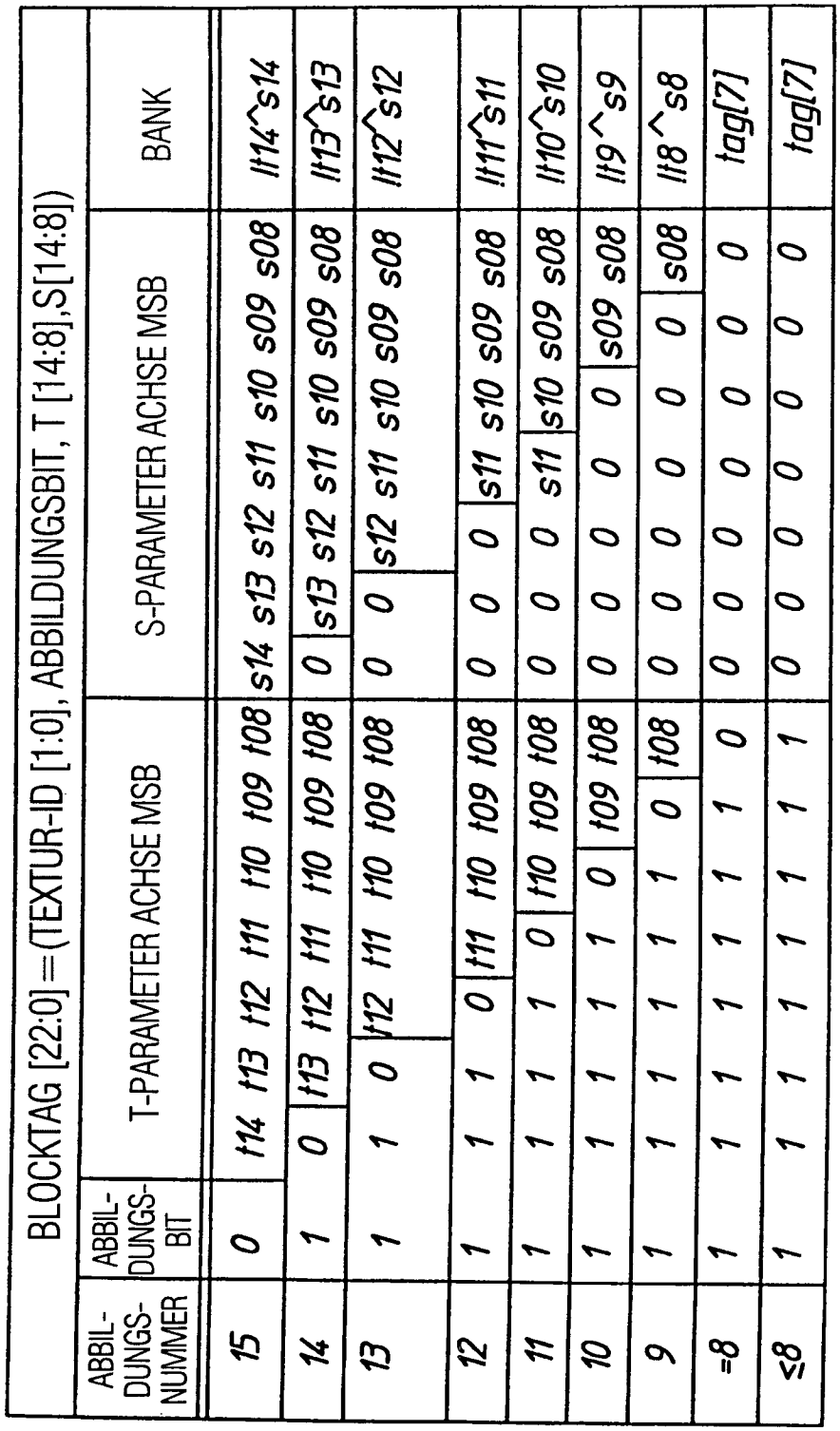


ABBILDUNG 9

FIG 14

FIG 15


ST0 ST1 ST2 ST3
ST4 ST5 ST6 ST7
ST8 ST9 ST10 ST11
ST12 ST13 ST14 ST15
BLOCK 2, BANK 0
ST3 ST2 ST1 ST0
ST7 ST6 ST5 ST4
ST11 ST10 ST9 ST8
ST15 ST14 ST13 ST12
BLOCK 1, BANK 1



BLOCKTAG [22:0] = (TEXTUR-ID [1:0], ABBILDUNGSBIT, T [14:8], S[14:8])				
ABBILDUNGS-NUMMER	ABBILDUNGS-BIT	T-PARAMETER ACHSE MSB	S-PARAMETER ACHSE MSB	BANK
15	0	t14 t13 t12 t11 t10 t09 t08	s14 s13 s12 s11 s10 s09 s08	l[14]^s14
14	1 0	t13 t12 t11 t10 t09 t08	0 s13 s12 s11 s10 s09 s08	l[13]^s13
13	1 1 0	t12 t11 t10 t09 t08	0 0 s12 s11 s10 s09 s08	l[12]^s12
12	1 1 1 0	t11 t10 t09 t08	0 0 0 s11 s10 s09 s08	l[11]^s11
11	1 1 1 1 0	t10 t09 t08	0 0 0 s11 s10 s09 s08	l[10]^s10
10	1 1 1 1 1 0	t09 t08	0 0 0 0 s09 s08	l[9]^s9
9	1 1 1 1 1 1 0	t08	0 0 0 0 0 s08	l[8]^s8
=8	1 1 1 1 1 1 1 0		0 0 0 0 0 0 0	tag[7]
≤8	1 1 1 1 1 1 1 1		0 0 0 0 0 0 0	tag[7]

FIG 16

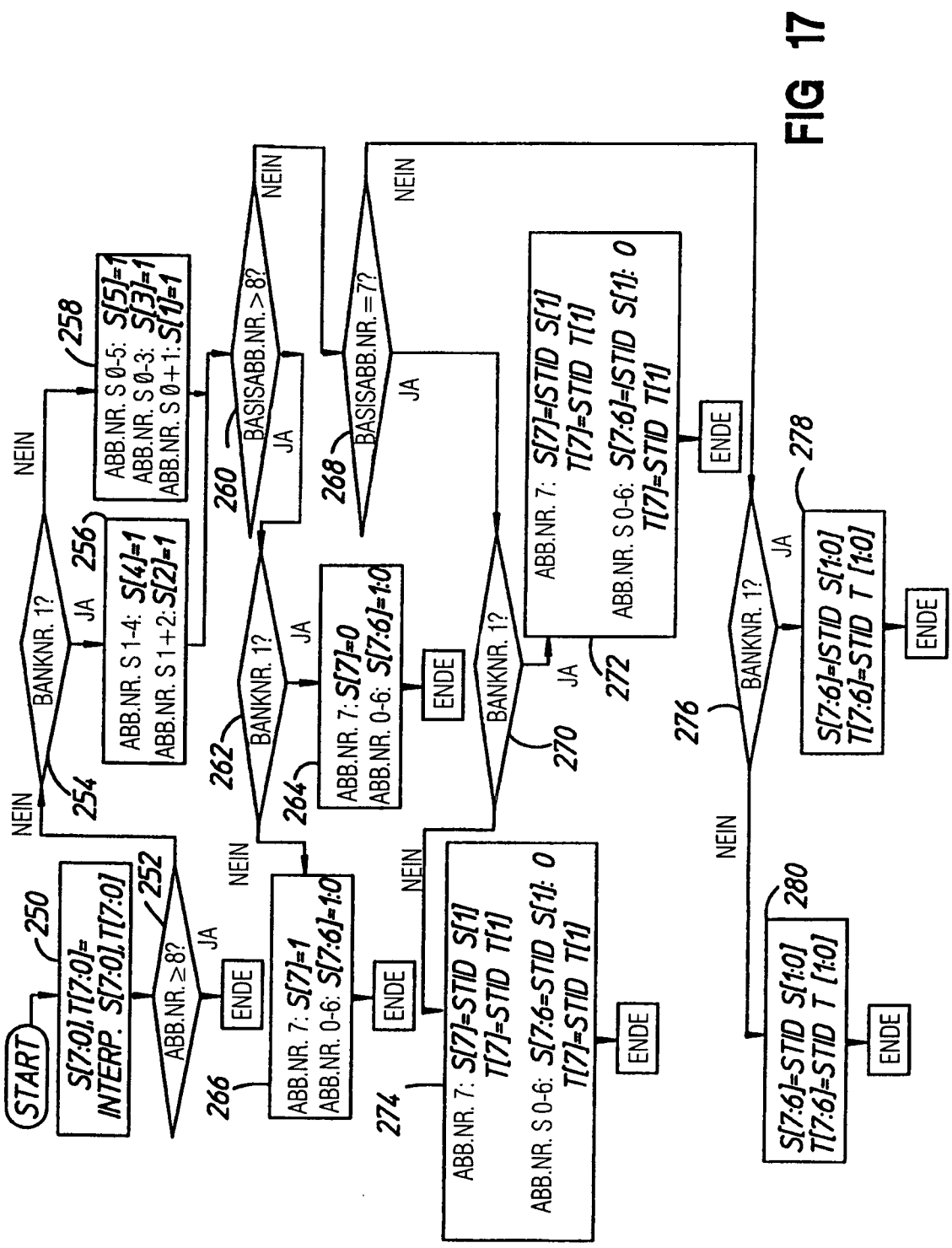

START
S[7:0],T[7:0]= INTERP. S[7:0],T[7:0]
250
ABB.NR. ≥ 8?
252
JA
ENDE
NEIN
BANKNR. 1?
254
JA
ABB.NR. S 1-4: S[4]=1
ABB.NR. S 1+2: S[2]=1
256
NEIN
ABB.NR. S Ø-5: S[5]=1
ABB.NR. S Ø-3: S[3]=1
ABB.NR. S Ø+1: S[1]=1
258
BASISABB.NR. > 8?
260
JA
NEIN
BANKNR. 1?
262
JA
ABB.NR. 7: S[7]=0
ABB.NR. 0-6: S[7:6]=1:0
264
ENDE
NEIN
ABB.NR. 7: S[7]=1
ABB.NR. 0-6: S[7:6]=1:0
266
ENDE
BASISABB.NR. = 7?
268
JA
NEIN
BANKNR. 1?
270
JA
ABB.NR. 7: S[7]=STID S[1]
T[7]=STID T[1]
ABB.NR. S 0-6: S[7:6]=STID S[1]: 0
T[7]=STID T[1]
272
ENDE
NEIN
ABB.NR. 7: S[7]=STID S[1]
T[7]=STID T[1]
ABB.NR. S 0-6: S[7:6]=STID S[1]: 0
T[7]=STID T[1]
274
ENDE
BANKNR. 1?
276
JA
S[7:6]=STID S[1:0]
T[7:6]=STID T[1:0]
278
ENDE
NEIN
S[7:6]=STID S[1:0]
T[7:6]=STID T[1:0]
280
ENDE

FIG 17

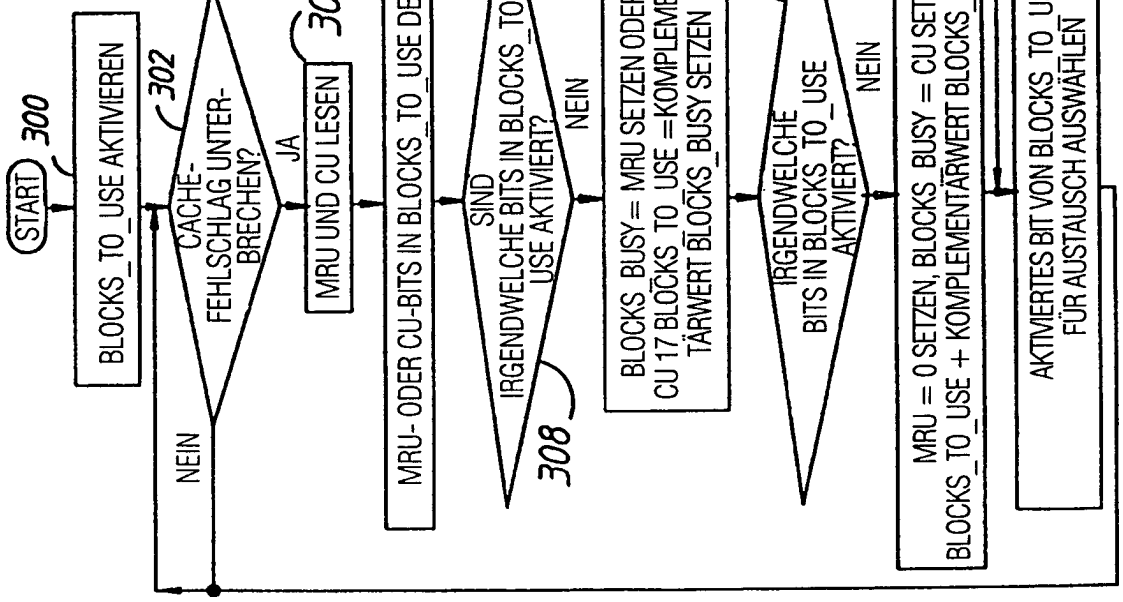

START
300
BLOCKS_TO_USE AKTIVIEREN
302
CACHE-FEHLSCHLAG UNTERBRECHEN?
NEIN
JA
304
MRU UND CU LESEN
306
MRU- ODER CU-BITS IN BLOCKS_TO_USE DEAKTIVIEREN
JA
SIND IRGENDWELCHE BITS IN BLOCKS_TO_USE AKTIVIERT?
308
NEIN
316
BLOCKS_BUSY = MRU SETZEN ODER CU 17 BLOCKS_TO_USE = KOMPLEMENTÄRWERT BLOCKS_BUSY SETZEN
318
JA
IRGENDWELCHE BITS IN BLOCKS_TO_USE AKTIVIERT?
NEIN
310
AKTIVIERTE BITS IN BLOCKS_TO_USE UNTERHALB DEM SCHWELLENWERT?
NEIN
310
JA
312
MRU = 0 SETZEN
320
MRU = 0 SETZEN, BLOCKS_BUSY = CU SETZEN, BLOCKS_TO_USE + KOMPLEMENTÄRWERT BLOCKS_BUSY SETZEN
314
AKTIVIERTES BIT VON BLOCKS_TO_USE FÜR AUSTAUSCH AUSWÄHLEN

FIG 18

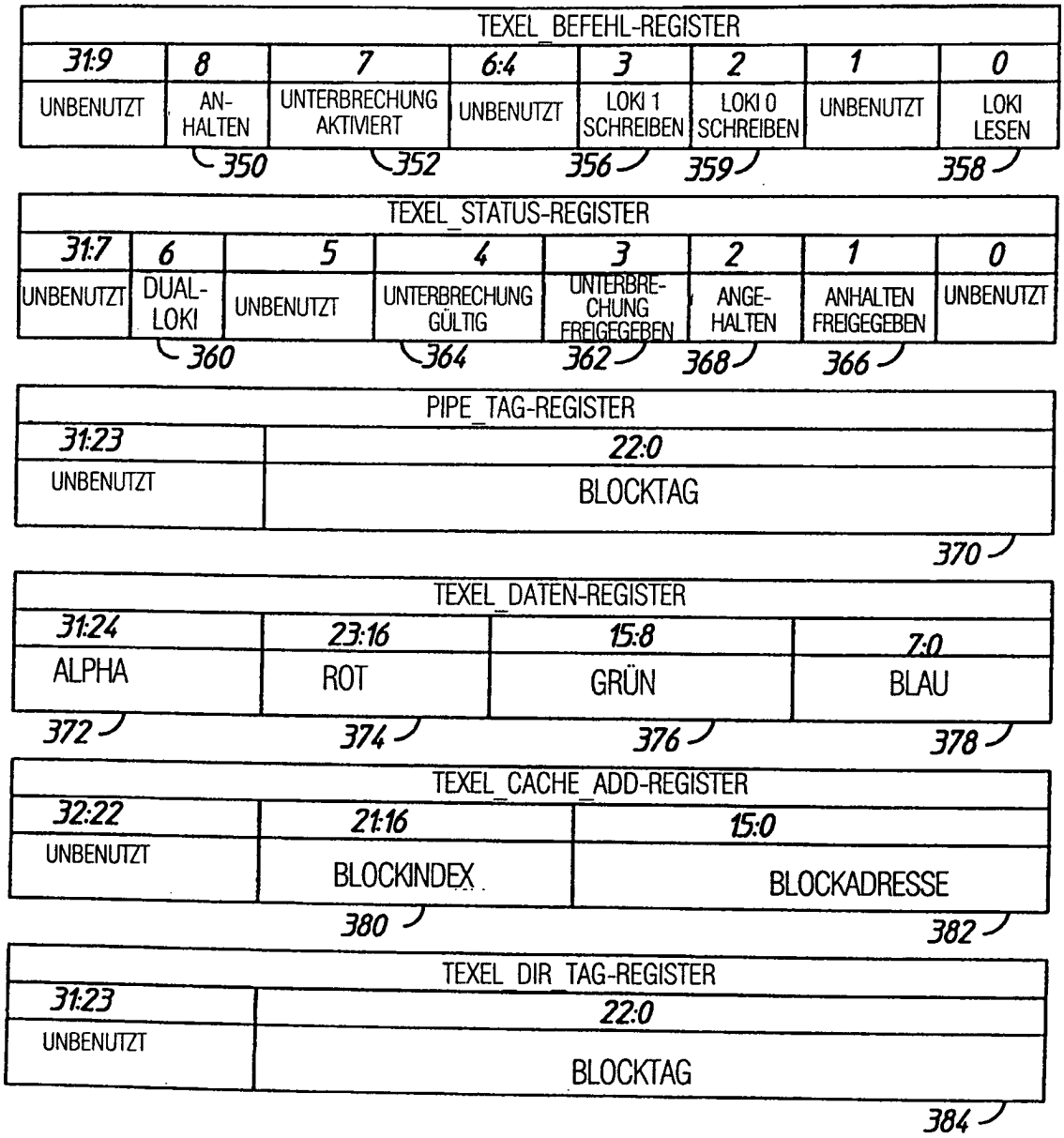

TEXEL_BEFEHL-REGISTER
31:9
UNBENUTZT
8
AN-HALTEN
7
UNTERBRECHUNG AKTIVIERT
6:4
UNBENUTZT
3
LOKI 1 SCHREIBEN
2
LOKI 0 SCHREIBEN
1
UNBENUTZT
0
LOKI LESEN
350
352
356
359
358
TEXEL_STATUS-REGISTER
31:7
UNBENUTZT
6
DUAL-LOKI
5
UNBENUTZT
4
UNTERBRECHUNG GÜLTIG
3
UNTERBRE-CHUNG FREIGEGEBEN
2
ANGE-HALTEN
1
ANHALTEN FREIGEGEBEN
0
UNBENUTZT
360
364
362
368
366
PIPE_TAG-REGISTER
31:23
UNBENUTZT
22:0
BLOCKTAG
370
TEXEL_DATEN-REGISTER
31:24
ALPHA
23:16
ROT
15:8
GRÜN
7:0
BLAU
372
374
376
378
TEXEL_CACHE_ADD-REGISTER
32:22
UNBENUTZT
21:16
BLOCKINDEX
15:0
BLOCKADRESSE
380
382
TEXEL_DIR_TAG-REGISTER
31:23
UNBENUTZT
22:0
BLOCKTAG
384

FIG 19

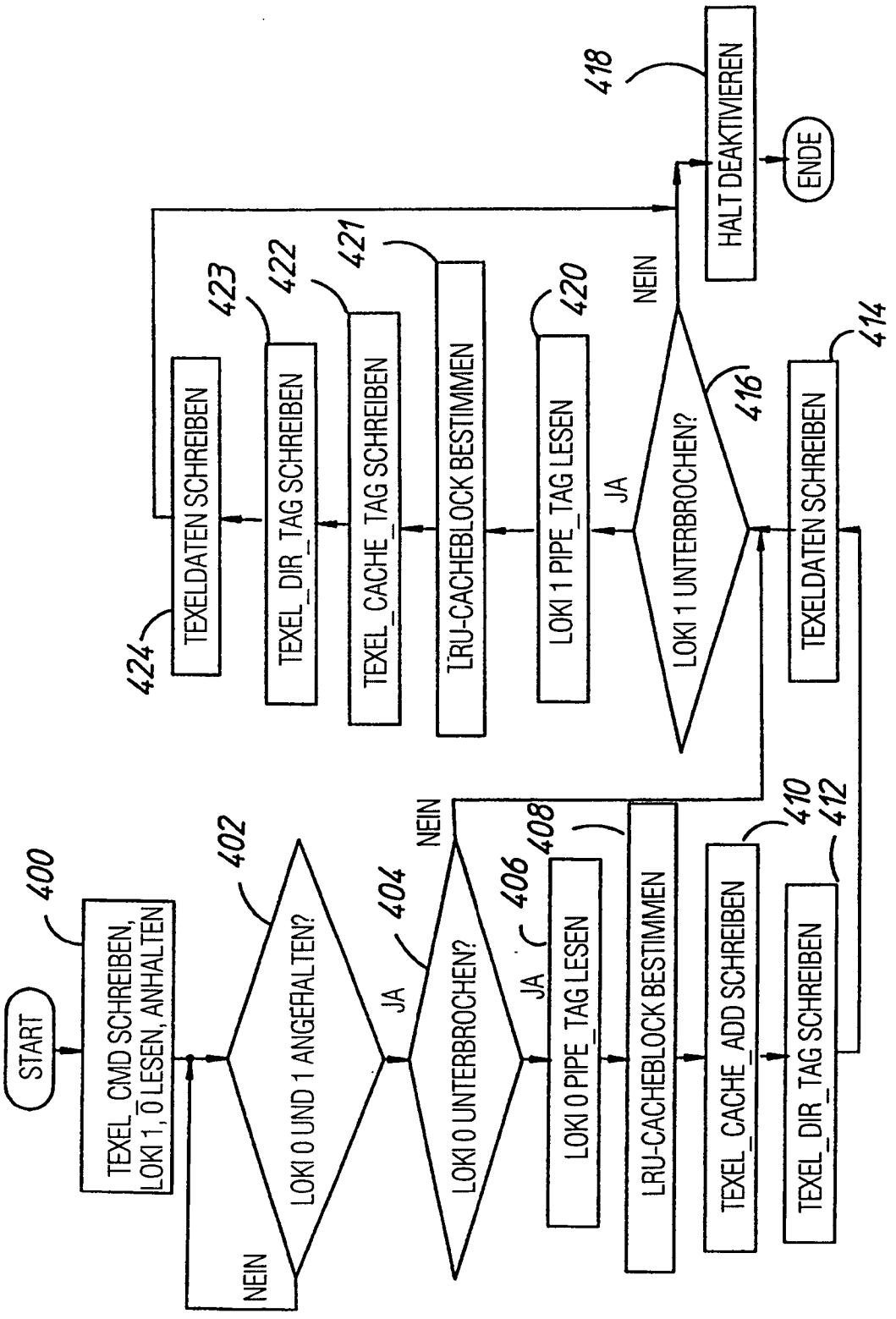

START
400
TEXEL_CMD SCHREIBEN, LOKI 1, 0 LESEN, ANHALTEN
402
LOKI 0 UND 1 ANGEHALTEN?
NEIN
JA
404
LOKI 0 UNTERBROCHEN?
NEIN
JA
406
408
LOKI 0 PIPE_TAG LESEN
LRU-CACHEBLOCK BESTIMMEN
410
TEXEL_CACHE_ADD SCHREIBEN
412
TEXEL_DIR_TAG SCHREIBEN
414
TEXELDATEN SCHREIBEN
416
LOKI 1 UNTERBROCHEN?
NEIN
JA
420
LOKI 1 PIPE_TAG LESEN
421
LRU-CACHEBLOCK BESTIMMEN
422
TEXEL_CACHE_TAG SCHREIBEN
423
TEXEL_DIR_TAG SCHREIBEN
424
TEXELDATEN SCHREIBEN
418
HALT DEAKTIVIEREN
ENDE

FIG 20

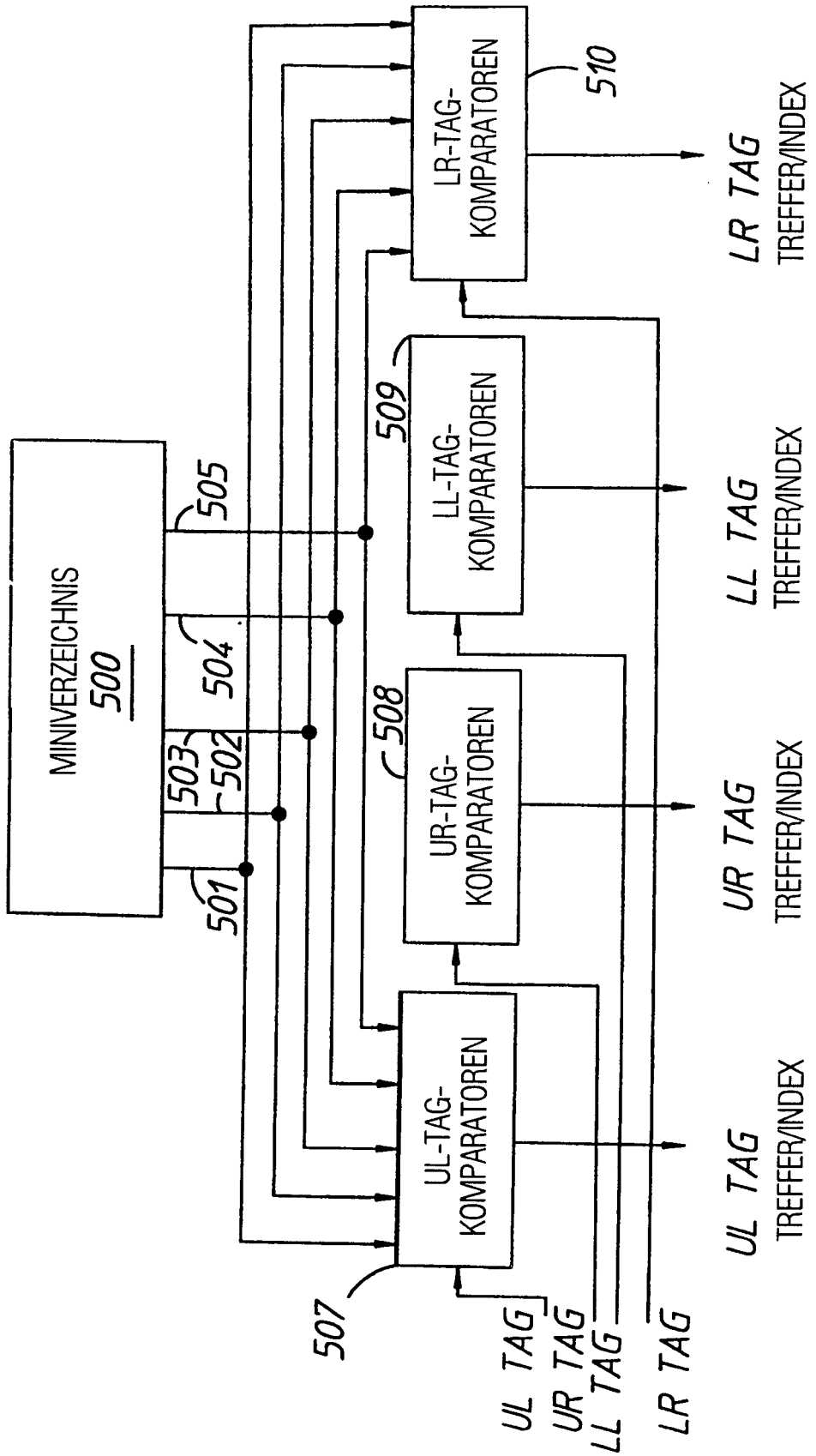

MINIVERZEICHNIS
500
501
502
503
504
505
507
508
509
510
UL-TAG-KOMPARATOREN
UR-TAG-KOMPARATOREN
LL-TAG-KOMPARATOREN
LR-TAG-KOMPARATOREN
UL TAG
UR TAG
LL TAG
LR TAG
UL TAG TREFFER/INDEX
UR TAG TREFFER/INDEX
LL TAG TREFFER/INDEX
LR TAG TREFFER/INDEX

FIG 21

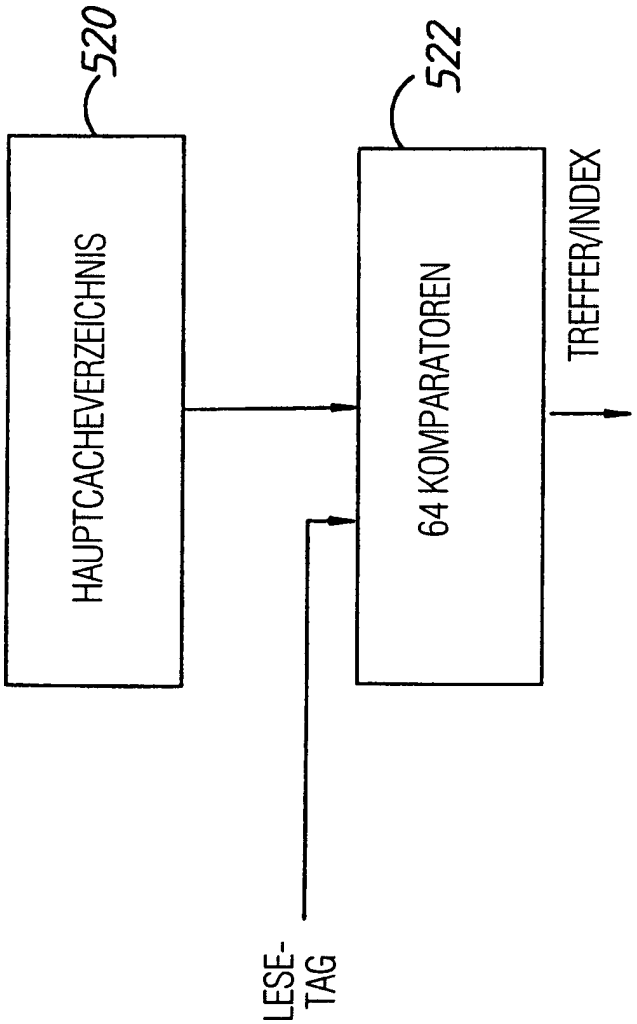

520
HAUPTCACHEVERZEICHNIS
522
64 KOMPARATOREN
TREFFER/INDEX
LESE-
TAG

FIG 22

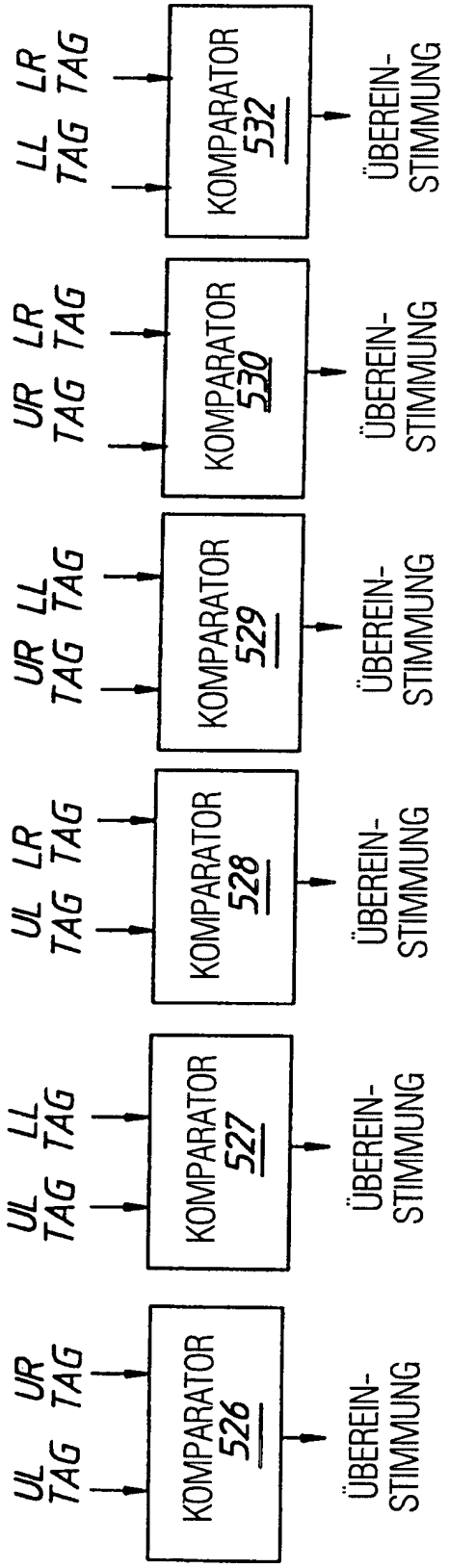

UL TAG
UR TAG
KOMPARATOR 526
ÜBEREIN-STIMMUNG
UL TAG
LL TAG
KOMPARATOR 527
ÜBEREIN-STIMMUNG
UL TAG
LR TAG
KOMPARATOR 528
ÜBEREIN-STIMMUNG
UR TAG
LL TAG
KOMPARATOR 529
ÜBEREIN-STIMMUNG
UR TAG
LR TAG
KOMPARATOR 530
ÜBEREIN-STIMMUNG
LL TAG
LR TAG
KOMPARATOR 532
ÜBEREIN-STIMMUNG

FIG 23

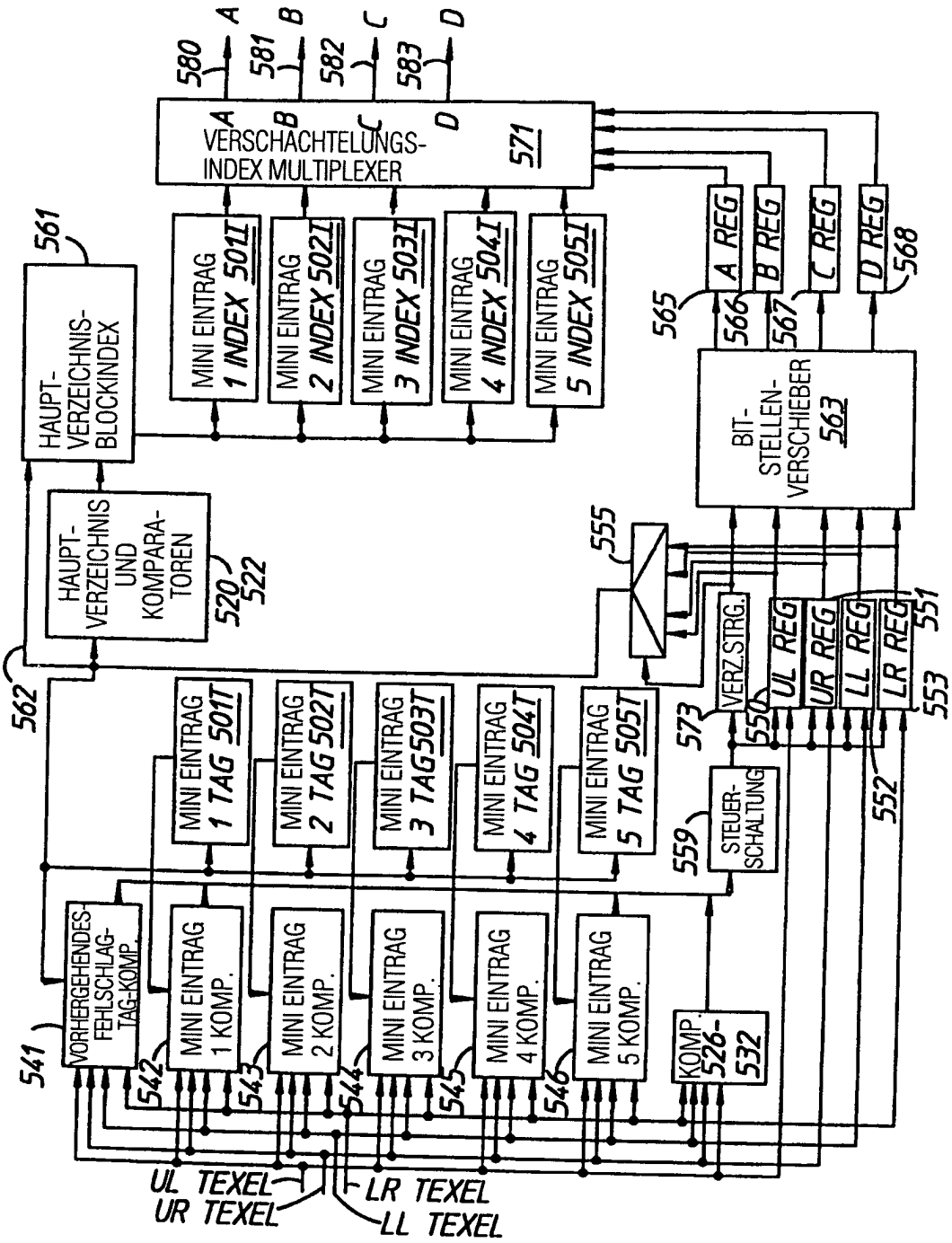

VERSCHACHTELUNGS-INDEX MULTIPLEXER
571
HAUPT-VERZEICHNIS-BLOCKINDEX
561
HAUPT-VERZEICHNIS UND KOMPARA-TOREN
520
522
MINI EINTRAG 1 INDEX 501I
MINI EINTRAG 2 INDEX 502I
MINI EINTRAG 3 INDEX 503I
MINI EINTRAG 4 INDEX 504I
MINI EINTRAG 5 INDEX 505I
A REG
B REG
C REG
D REG
568
BIT-STELLEN-VERSCHIEBER 563
MINI EINTRAG 1 TAG 501T
MINI EINTRAG 2 TAG 502T
MINI EINTRAG 3 TAG 503T
MINI EINTRAG 4 TAG 504T
MINI EINTRAG 5 TAG 505T
VERZ.STRG.
UL REG
UR REG
LL REG
LR REG
551
553
552
550
573
559
STEUER-SCHALTUNG
VORHERGEHENDES FEHLSCHLAG-TAG-KOMP.
MINI EINTRAG 1 KOMP.
MINI EINTRAG 2 KOMP.
MINI EINTRAG 3 KOMP.
MINI EINTRAG 4 KOMP.
MINI EINTRAG 5 KOMP.
KOMP. 526-532
UL TEXEL
UR TEXEL
LR TEXEL
LL TEXEL

FIG 24